



Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo
Odsjek za računarstvo i informatiku



Implementation Documentation

Embedded systems






Name and surname: *Nejla Subašić*
Index number: *18620*

Name and surname: *Matej Marinčić*
Index number: *18729*

Implementation guide

The best way to guide you through our implementation process and the soul structure of our project is to begin from the smallest (*and the most simple*) part and climb our way to the biggest (*complex*) parts.

The simplest parts in our project are these functions:








```
339
340 ▶ void pointToPlayer(int player){}
369
370 ▶ void make1PlayerView(){}
449
450 ▶ void make2PlayerView(){}
526
527 ▶ void makeHomeView(){}
585
586 ▶ void onPausePressed(){}
```

You can see that they are big in code but inside you will find all the basic m-bed functions for drawing on the screen. All these functions do is draw different views for our users. For example:

```
i26
i27 ▼ void makeHomeView(){
i28     // pozadina, elipsa i text u njoj
i29     BSP_LCD_SetTextColor(LCD_COLOR_DARKBLUE);
i30     BSP_LCD_FillRect(0, 0, BSP_LCD_GetXSize(), BSP_LCD_GetYSize());
i31     BSP_LCD_SetTextColor(LCD_COLOR_WHITE);
i32     BSP_LCD_FillEllipse(BSP_LCD_GetXSize()/2, BSP_LCD_GetYSize()/8 + 5, 53, 23);
i33     BSP_LCD_SetTextColor(LCD_COLOR_BLUE);
i34     BSP_LCD_FillEllipse(BSP_LCD_GetXSize()/2, BSP_LCD_GetYSize()/8 + 5, 50, 20);
i35     BSP_LCD_SetTextColor(LCD_COLOR_WHITE);
i36     BSP_LCD_SetBackColor(LCD_COLOR_BLUE);
i37     BSP_LCD_SetFont(&Font12);
i38     BSP_LCD_DisplayStringAt(BSP_LCD_GetXSize()/2 - 33, BSP_LCD_GetYSize()/8,
i39         (uint8_t *)"4 IN A ROW", LEFT_MODE);
i40     // kraj pozadine, elipse i texta u njoj
i41
i42     // prvi covjeculjak
i43     BSP_LCD_SetTextColor(LCD_COLOR_YELLOW);
i44     BSP_LCD_FillCircle(BSP_LCD_GetXSize()/5, BSP_LCD_GetYSize()/3 + 4, 5);
i45     BSP_LCD_FillRect(BSP_LCD_GetXSize()/5 - 3, BSP_LCD_GetYSize()/3 + 4, 7, 20);
i46     BSP_LCD_FillRect(BSP_LCD_GetXSize()/5 + 3, BSP_LCD_GetYSize()/3 + 8, 4, 10);
i47     BSP_LCD_FillRect(BSP_LCD_GetXSize()/5 - 6, BSP_LCD_GetYSize()/3 + 8, 4, 10);
...
```

This is just a small part of the “home screen” because we don’t want to make this all about that.

Now when we have all our screen designed, we must somehow navigate through them. We created functions for pressing elements on our screen.

```
616
617 ▶ bool single_pressed(int x1, int y1){}
624
625 ▶ bool multi_pressed(int x1, int y1){}
630
631 ▶ bool pausePressed(int x1, int y1){}
636
637 ▶ bool housePressed(int x1, int y1){}
642
643 ▶ bool homePressed(int x1, int y1){}
649
650 ▶ bool ContinuePressed(int x1, int y1){}
656
657 ▶ bool RestartPressed(int x1, int y1){}
663
```

As you can see they are all very simple and short, but very important and precise in giving result even to the smallest pixel.

For example here we you can see a function that will determine did the user press the single player option (“1 Player” on the screen already shown in the project specification)

```
617 ▶ bool single_pressed(int x1, int y1){
618     if(x1 > BSP_LCD_GetXSize()/5 - 3 + 40 && x1 < BSP_LCD_GetXSize()/5 - 3 + 130
619     && y1 > BSP_LCD_GetYSize()/3 && y1 < BSP_LCD_GetYSize()/3 + 25){
620         return true;
621     }
622     return false;
623 }
```

Now when we combine the 2 elements described above in our main() we can easily navigate through our project using a variable “viewNow” that can variate from 0 to 4 (*that is the number of different screens we designed*), that way it is very easy to add new screens, just add a new function for screen design and another “if” in main.

We created 2 classes in our project that made our “backend” implementation relatively easy.

Our first class is “**Slot**”

Slot represents a slot in our 6x7 game board, each Slot has its coordinates and an Enum object StateOfTheSlot{ NOTTAKEN, YELLOW, RED }

This Enum object helps us determine which slots are already taken and which player did put a “coin” where.

Everything said above you can see here:

```
30
31 enum StateOfSlot { NOTTAKEN, YELLOW, RED }; |
32 class Slot{
33     int xKoord, yKoord;
34     StateOfSlot state;
35 public:
36     Slot(int x, int y, StateOfSlot s){}
41     StateOfSlot getState(){return state;}
44     void changeState(StateOfSlot s){state = s;}
47     int getXKoord(){return xKoord;}
48     int getYKoord(){return yKoord;}
49 };
50
```

We implemented all the methods we needed in our project, they helped us to work out a very sleek solution for our project.

Now because we have 6x7 (42) slots on the board we created another class that would represent that board. That is “**PlayingBoard**”. And this class has the most complex methods in our project. (*VectorOfSlots is an alias for vector<vector<Slot>>*)

```
52 class PlayingBoard{
53     VectorOfSlots slots;
54 public:
55     PlayingBoard(){}
56     PlayingBoard(VectorOfSlots s){}
59     void makeTheBoard(){}
72     int findFirstAvailableSlot(int column){}
83     void insertCoin(int column, int index){}
118    void updateView(){}
140    bool endGame(){}
248    bool drawGame(){}
255    void winnerView(int player){}
329    void playBot(){}
338 };
```

We will go through all the methods one by one to explain why we used each one in our system.

- makeTheBoard – creates a new VectorOfSlots with 42 Slots each one has its own coordinates and in the beginning the state object NOTTAKEN
- findFirstAvailableSlot(int column) – returns index of the first available slot where a user can put his “coin” or -1 if the whole column is already full
- insertCoin(int column, int index) – the most complex method here, creates an animation of a coin drop from the first to to index where the user can

drop a coin, sets the state object in Slot to YELLOW or RED depending which player put the coin (*that information is held in an global variable **player***)

- `updateView()` – goes through all the Slots on the board and puts red or yellow coins where they already were. We use this method because we implemented a pause button, so when we go back to the game we will have the function for drawing our view (*already described above*), then this function will fill any coins if the game was already started.
- `endgame()` – checks if there is a winner, goes through all the slots searching for “4 in a row” with the same state {YELLOW or RED}, we call this method right after inserting a new coin.
- `drawGame()` – checks if the game is a tie, the game is a tie if there is no “4 in a row” and no more slots available. Also we call it right after inserting a new coin.
- `winnerView(int player)` – displays a new window with an animation of fireworks and a big sign of the winner. From this screen you can go back to the home screen
- `playBot()` – when the single player mode is selected, we call this method right after the first player (*only player*) inserts his coin. This method than generates a random number from 0 – 6 and places the new coin as an opponent to the user.

This kind of implementation will allow us to easily add new screens and new functionalities to our app, because we can simply add a new method in our class and apply it to our system.

Now when we went through our app thoroughly, when we combine everything in our main we will get a full functioning app of the game “**4 in a row**”.