



EEET2505

Introduction to Embedded Systems

Assignment

Battleship Game with AVR

Microcontroller

Lecturer: Mr. Thanh Pham - thanh.pham@rmit.edu.vn

Team 44

Student 1: Amila Alexander s3766009

Contents

Introduction	3
Importance of Embedded Systems	3
AVR Microcontroller for the Design of “Battleship” Game	3
Requirement Analysis	4
Design Approach	5
Hardware Analysis & Implementation	6
User Input	6
LCD Display.....	7
8x8 LED Array	8
Additional Hardware Required	9
Software Implementation.....	9
Serial Communication.....	10
EEPROM Persistence.....	13
Game Logic.....	14
8x8 LED Matrix Logic.....	18
How LED matrix responds to Hits and Misses	18
How LED display persistence is Sustained Even with Cursor Movements.....	20
Other Important Methods etc Used in the Program	20
Demonstration of 2 Player Mode	24
Code Running on User A (For setting the map/stage)	24
Results.....	27
References	28

Introduction

Modern computing is the cornerstone of all modern technology. Whether it's a supercomputer or a grain-sized microchip, they are all working on the same fundamental principles of logic and design. In the past, computers were referred to as centers for advanced computational tasks; however, with the invention of the modern transistor, it has shrunk and now is available everywhere. Computers are not just on tables and desks anymore; they are in people's hands, all around them in every corner of the modern world. The reason for this massive burst in computing is mainly due to new microcontrollers being produced at an ever-changing, more complex rate. The age of the all-in-one powerful computer in the GHz range has now been replaced by smaller, more cheaper MCUs which are designed to implement a specific task but extremely well and cost effectively.

Importance of Embedded Systems

CPUs, regardless of their make and design, essentially do the same underlying fundamental task of computation, which is taking data and converting it into useful information via processes. However, CPUs vary in power quite considerably. The processors used in modern computers, for instance, have to cope with advanced, highly sophisticated tasks which run upon an operating system. They have to deal with multiple inputs from users and also have to do multiple tasks at the same time.

For more generalized electronic scenarios, however, like in washing machines, air conditioners, microwaves, remote controls, traffic lights, etc., advanced computational functionality is not required. The main goal of these systems is to carry out a certain, very specific task repeatedly, reliably, in the long term. To satisfy these features, they must be small, cost-effective, energy-efficient, and easy to program. This is the reason microcontrollers have gained so much popularity. For every laptop or PC with a powerful processor, there are at least 10 devices around it with small computational processors around it.

Apart from embedded systems being reliable, they are difficult to penetrate and do not run into common errors like in higher computer systems, like viruses, etc. These microsystems can be used in control systems, as monitoring devices, as repeated simple computational centers, and in devices which require simple logic-based functions (like calculators). Embedded systems need not have high computational power because they are not expected to run multiple programs or deal with many windows, operating systems, etc. Hence, they are extremely useful for low power, computationally simple tasks.

AVR Microcontroller for the Design of "Battleship" Game







The game battleship is a classic among people and especially gained traction in miniaturized handheld devices with the development of computing in the early days. The users could see a map where they had to shoot ships and look for hits. The game would have sounds, animations, and other miscellaneous features and can be thought of as a prime example for the wonders of computing back then. The game relied on simple logic design principles but also took into account gaming design rules like, user-friendliness, etc. For instance, animations and sounds are not necessary to implement, but they create a more sense of immersion when playing this game. Small graphics and images rendered in these mini-displays would give the user a richer feeling as to what they are playing.

This project scopes some of the above features, and the task assigned is to develop a fully functional system which is capable of playing battleship on.

Requirement Analysis

The requirements for the single player version of battleship can be summarized into the following table below. However, since additional requirements will also be covered (requirements meant for groups of 2 & 3) they will also be included.

Legend

Individual	Group of 2	Group of 3
	 	  

Requirement Type	Requirement	Satisfied	Remark
Game Functionality	Ability to shoot, miss, hit	✓	Shoot push button
	Ability to move cursor	✓	4 buttons for XY movement
	Ability to load map from PC	✓	RS232 USB interface
	Check for ship “sunk”	✓	Individual ship by ship checking
	Map Persistence	✓	Loading map from EEPROM
	“GAME OVER” and “WINNER”	✓	Check if max shots reached/ check is any ships left
	Two player mode	✓	Via Serial interface
	Detect same position hit	✓	Keep track of shots/hits & display “already hit” on LCD
	Count maximum shots	✓	-
	Check for actual sinking	✓	Ships are specifically defined
User Experience	Display total shots, hits, total sunk and cursor position as coordinates on LCD	✓	-
	Flash LED when “hit” and turn off when “miss”	✓	LED on array blinks for hit
	“Loading...” Screens	✓	0 ~ 100 loading displayed
	Game Mode displayed to user	✓	PLAY, LOAD, P2P
	Misc. info displays, i.e: “writing to EEPROM please wait” etc	✓	Flashing operations require waiting
	LED array to show ships	✓	Proper display of ships etc

	Keypad functionality	✗	Normal push buttons used
Miscellaneous	Write to EEPROM	✓	Memory writing
	Check to see if proper map encoding provided	✓	Error checking
Debugging	Mirror data sent back to serial monitor	✓	-
	‘Cheat’ mode to see if the map shows on the LED array by lighting places where ships are located	✓	View map to see if properly loaded
	Print EEPROM map data to Serial Monitor	✓	View contents of EEPROM
	Print in memory map data to serial monitor	✓	-

Table 1 - Requirement Analysis

Design Approach

For programming the game in C, initially the design was conceptualized, and technical analysis was performed to a certain extent. Then throughout the project rigorous testing and trialing each subcomponent enabled to choose between sticking to the initially proposed solution or moving to a new one. Listed below are the initial ideas/concepts that were proposed to be implemented to solve the problem. The semi-technical requirement analysis and conceptualization is given below.

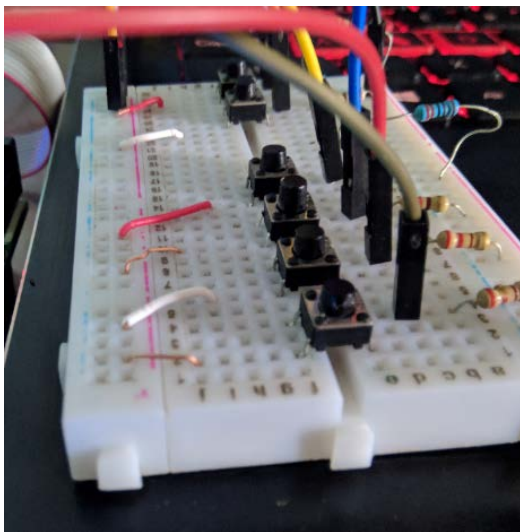
1. From the initial analysis of the requirements it was clear that the game had to set up step by step beginning from the one player version and then progressing to more advanced features and game modes
2. Since map loading is a crucial function one of the primary steps was being able to read a text file sent via PC. To accomplish this the option “send text” was proposed to be used in the software terminal.exe.
3. This software reads the file byte by byte and transmits the data to the ATMEGA32 MCU over UART. To verify integrity of received data can be printed back onto the serial monitor.
4. Since the received data is in the form of a long input stream it has to be converted to a matrix to apply matrix operations and enable compatibility with the LED array. A method to convert an input stream of bytes to an array was needed
5. Since serial communication is not required (for single player mode) after the map is loaded from the PC an FSM was conceptualized to simplify game modes
6. One state proposed for Play mode and another for Load Mode in FSM
7. To switch between these two states an ISR can be used which has a volatile state variable inside the routine.

8. For cursor movements two variables could be kept track of. External buttons can be used to increment/decrement these variables.
9. Once the user is satisfied with a set position, they need to have the ability of shooting. A separate button is needed for this function.
10. Separate variables need be updated to keep track of total number of shots/hit and other information
11. Since the 8x8 LED array is being used a framework needs to be setup to display map on intuitive display. Cursor is also need be set on the 8x8 array
12. For two player mode the TX, RX pins of two boards need be connected to transmit serial data from one board to the other
13. The software for two player mode is identical in both cases meaning the same code is uploaded to two boards and they can be used interchanged.
14. Map Persistence need be included to properly save the data even when power has been disconnected (use EEPROM)

Hardware Analysis & Implementation

To implement the hardware required for the system certain components were required other than the OpenUSBIO board (Atmega32). This section attempts to clarify the hardware required and explains how each component was used for the production of the battlefield game.

User Input

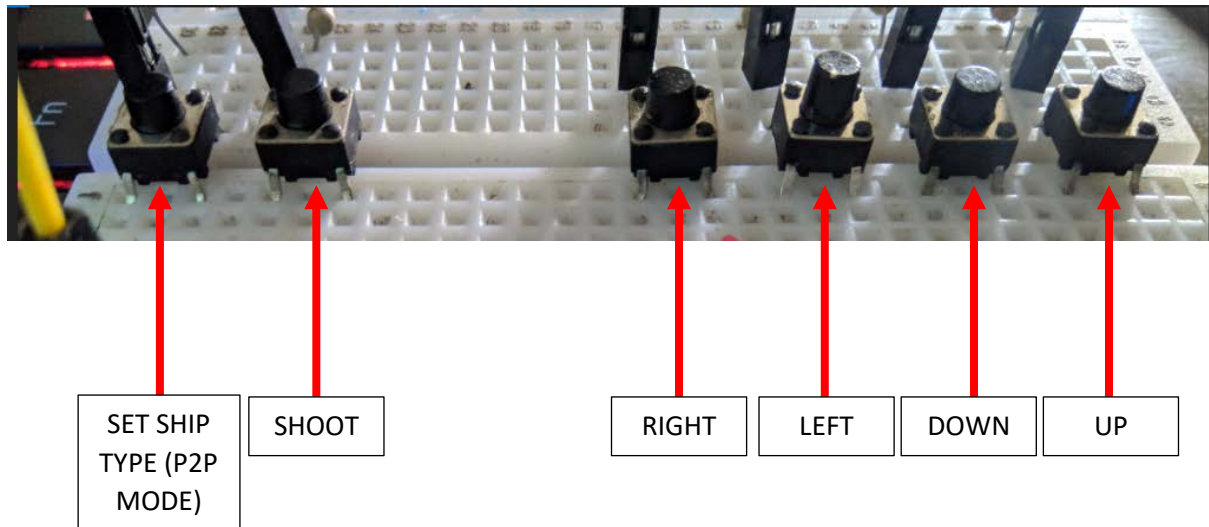


- The most crucial aspect of the game is the ability to control a cursor to move around the map, hence external input from the user was required. To implement this feature simple push button switches were set to PULL-UP and the board was configured to receive inputs.
- Additionally, other controls are required to switch the game mode and use other functions such as “shoot” and “set stage” when creating a new map in P2P Mode.
- ✓ Another important aspect that should be considered when applying push buttons is switch bouncing. A debouncing mechanism should be implemented in to mitigate it's effects. For the current setup software debouncing is used.

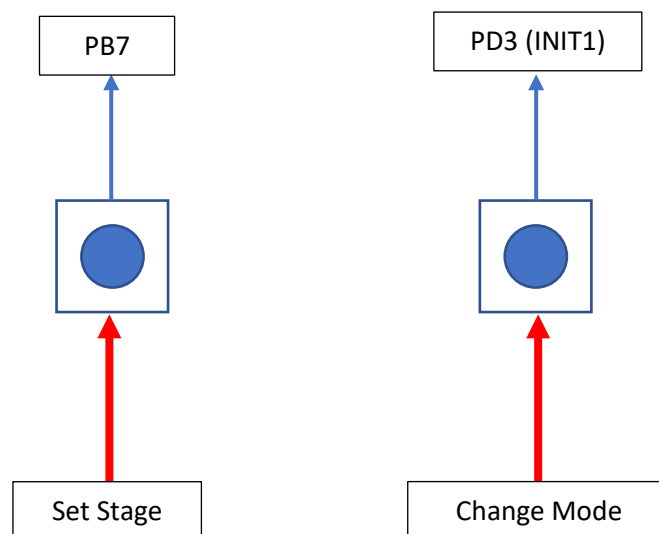
Figure 1 - Controller Mechanism

Board 1

Figure 2 - Controller Mechanism Explanation



Additionally, a Second Board is Used (*no picture available*)



LCD Display

The LCD display provides all the user's current stats and provide insight into how many shots they have taken, how many ships they have sunk and notifies them if they have hit water or a ship. To connect the LCD, display a library is included to simplify numerous operations. This library includes functions to

1. Setting the cursor position
2. Writing a string or single char to the position

Writing multiple digit numbers etc. requires custom built methods and functions which will discussed in the software section.



Figure 3 - Welcome Screen

The LCD display shown above runs with the aid of the Hitachi HD44780 driver. To manipulate letters and numbers on it a library provided by electroSome is used. This library contains a header file which we use to specify the connected pins to the board. Once properly connect this library can be included to the main c file and it's functions/methods called.

8x8 LED Array

The primary goal of the LED matrix is to show the user where the cursor position is exactly on the map. They can clearly get an idea of

1. Where they are on the map
2. Where the ships they have hit/sunk are located
3. Places where they have not shot yet

Thus, they do not have to rely on remembering a complex coordinate system and remember all the places they have hit before. Applying this method enables better use experience. More so, it is much easier for the creator to set the map in P2P mode because they can clearly see where the ships, they placed are in the LED matrix.

The challenge of the LED matrix is persistence. That is, due to it's nature it only allows writing row by row as a byte. This means that whenever another light on the same row is tried to be lit the former light will turn off unless proper functions are written. More will be discussed in software section

The 8x8 LED array relies on the MAX7219 chip for its function. This greatly reduces the total number of pins required to connect it. The pins go from 16 to only 3. The three pins are CS, DIN and CLK

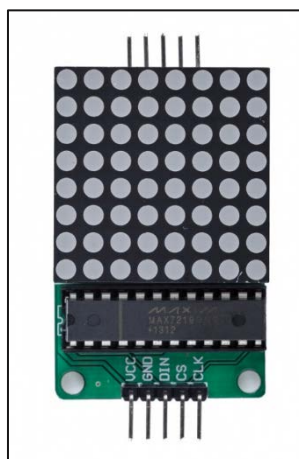


Figure 4 - LED matrix with MAX7219

Additional Hardware Required

Apart from the basic connectors (jumpers) and main board a serial to USB interface is required to read UART data from the computer. For 2 player mode a simple jumper cables are sufficient to transfer data from board to board.



Figure 5 - RS232 to USB connector

Software Implementation

This section of the project is the most important and most notable feature of this system that integrates all the above hardware and make the game a reality. By taking in consideration the different limitations and features of the aforementioned hardware software has to be methodically developed to satisfy the user goals and work under any conditions.

Each step of the process will be methodically covered under a dedicated subsection which will attempt to explain how each task problem has been tackled and ultimately, solved. The software architecture breakdown of the game is as follows;

1. Serial Communication - This is a crucial aspect for both single player and multiplayer to load the map from the PC via a notepad document. Additionally, 2 player mode also requires a board-to-board connection
2. Game Logic This is the framework which allow the user to play and enjoy a game of battleship where they feel the excitement of a game and anticipate a WIN. The user must feel like they have to play wisely to win the game otherwise they can lose. Applying smart proper game logic ensures the game is not boring and always challenging and engaging.
3. EEPROM Persistence This topic mainly covers memory writing functions and how memory writing has been attempted taking into consideration the different requirements while reading or writing from memory (waiting functions)

- | | |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| 4. 8x8 Array Logic | Writing to the LED matrix was one of the challenging tasks during this project. The methods and functions used for that will be discussed here |
| 5. Miscellaneous | Other important methods and functions used in the game |

Serial Communication

In this project Serial communication is primarily used via two different techniques.

1. Serial Receive Interrupt Enable
 2. Incoming Serial Data Wait
- The first mode is used primarily for detecting incoming map data. During “load Map...” mode the system waits for a trigger to determine if a map is about to be loaded. This trigger is received in the RXCIE Interrupt. A look at the text file being uploaded from Terminal.exe shows clearly how this mode will operate.

<pre> b00110000 00220000 00003330 44400000 50000660 50000770 88000000 00099900 </pre>	<ul style="list-style-type: none"> ✓ The character ‘b’ is the trigger that signals the program to start reading a incoming map ✓ By using a trigger mechanism, the program can properly be set to read a map without reading false data ✓ The map shown on the left displays how ships are arranged. They can be placed vertically, horizontally or in any other orientation imaginable. ✓ The key feature here is that each sink is defined specifically, meaning for it to be sunk all its adjacent point must have to be hit
----------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 6 - Example of a Map with trigger recognition

- ✓ For example, to sink ship type ‘6’ all location displaying ‘6’ will have to be shot.

The map loading code is briefly explained in the snippet below.

```
void loadMap(){

    if(Receivedbyte=='b'){
        PORTB ^= (1<<PB4);           //toggle LED
        UCSRB &= ~(1<<RXCIE);        //turn off Interrupt for normal read of
                                     //serial data
        state = READ;                //set state to READ
    };

}
```

Code Snippet 1 - Trigger recognition of letter 'b' to set state change

Code Explanation

1. The above code runs in the LOAD state and waits for an incoming trigger to begin reading a map.
2. The trigger is important both for false map data read prevention and during two player mode a user might send irrelevant serial data via their controls (up, down, left, right, shoot etc)
3. Once the trigger has been set the system moves into a new state known as **READ**
4. The interrupt for RX is turned off to enable reading data in a normal byte by byte “wait” fashion.

Reading Map Data

```
void readMap(){

    while(( UCSRA & (1 << RXC )) == 0){if(state!=READ){break;}}; //wait for incoming
bytes, break if state is not equal READ
    Receivedbyte = UDR ;           //set received data to receivedbyte char

    if(isdigit(Receivedbyte)){      //check if incoming data is a number 1 ~ 9
        count11++;                //increment index counting variable
        arr1[count11] = Receivedbyte; //write received data to array
        writeNumber(count11*1.6,2,12); //write loading progress to LCD
        if(count11==63){          //signals 64 chars have been received

            Lcd4_Set_Cursor(1,0);
            Lcd4_Write_String("Loading...");
            printdata();

        }

    }

    PORTB &= ~(1<<PB3);
```

Code Snippet 2 - read map data from txt to array

Code explanation

1. The above code snippet runs once the system has been triggered via 'b'. The state of the system is now READ.
2. The above code is responsible for storing the map temporarily into running program memory. It takes each byte sent via serial and first checks if it's a number before storing it into a `uint8_t` array.
3. This array will then be converted to a 2D array via a inbuilt function called `memcpy()`

```
uint8_t newarray31[8][8];  
memcpy(newarray31, arr1, 64*sizeof(uint8_t)); //copies contents of arr1 to newarray31
```

`memcpy()` takes as the input destination array, the source array and the size of the array to be copied (in bytes) and copies the memory from one location to another.

4. Additionally, progress data is also displayed to the user during this process (0 ~ 100 loading)

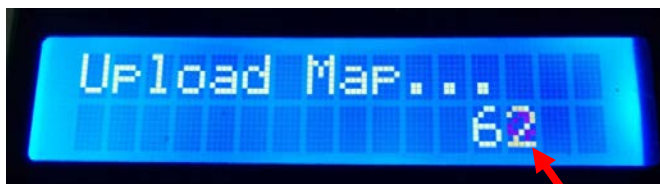


Figure 9 - Loading Progress

Loading Progress

For verification this data is also mirrored onto the Serial Monitor (after it has been written to the 2D array)

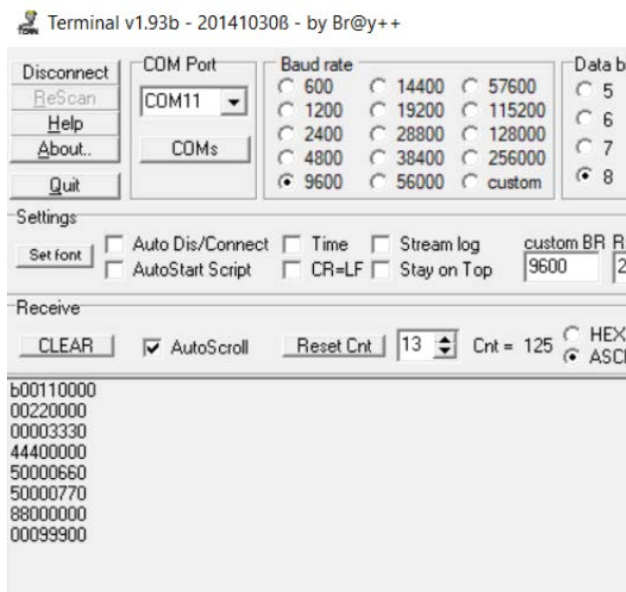


Figure 10 - Output to serial

EEPROM Persistence

Writing to the EEPROM mainly occurs in two different instances.

1. After a map has been loaded
2. After a map has been staged in 2 player mode

During this process the programmer has to write data into the nonvolatile section of the memory of the AVR microcontroller. The atmega32 has 1024 bytes of nonvolatile memory that can be written to. For writing to the EEPROM the command `eeprom_write_block()` is used to write bytes of an given length to the EEPROM.

```
//write 2D array to 60th position of EEPROM  
eeprom_write_block((const void*)newarray31,(void*)60,sizeof(newarray31)/sizeof(uint8_t));  
    eeprom_busy_wait();    //wait for write to complete
```

The above snippet shows how data is written to the EEPROM. The method takes 3 parameters of which the first is the name of the array/variable to be written (the source) the location of the byte in the EEPROM to start writing at and finally the size of the array as bytes (64).

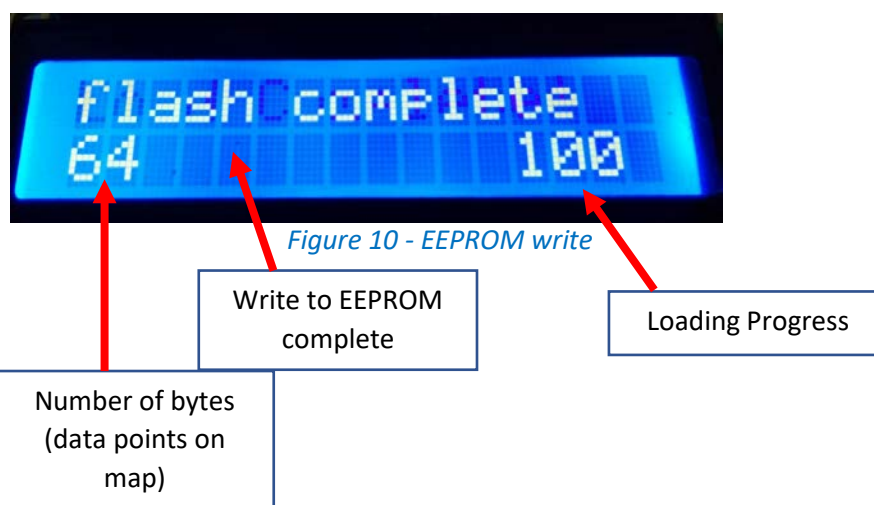
Additionally, since no other processes can occur during EEPROM reading and writing the program has to wait until the write is completed to continue to the next stage.

Reading from EEPROM

Reading from the EEPROM involves a similar process where an array has to be specified (the destination array) an the location of where the read has to start and the number of bytes of data from the EEPROM to be read.

```
eeprom_read_block((void*)checkArray, (const void*)60,(sizeof(checkArray)/sizeof(uint8_t)));  
eeprom_busy_wait();
```

Here *checkArray* is a temporary array which stores the EEPROM map information for verification and debugging purposes. (mirroring onto serial monitor)



Game Logic

This is the most important aspect of program which determines how the player interacts and responds to the system. The software used in this system has to properly identify the player's moves and display relevant data once they have taken an action.

The basic idea of the system is to capture an input from the player. This input is in the form of cursor location. The user can move up and down the grid of points and their location will be live updated to them on both the LED matrix and the LCD display



Figure 11 - Cursor Movements

Index for rows and columns start from zero. The cursor can be seen at the (0,2) position in the image above.

For incrementing rows and columns the buttons are simply set to increment a given row or column variable every time a user presses a button. The rows and columns increment up to a value of 7 before returning to 0 and looping over.

Sample code:

```
if(UP){  
    row_select--;           //decrements cursor position by 1 row  
    myDelay(200);           //custom timer 200ms, debouncing  
}
```

The rather simple action of moving the cursor is covered in the topic above. The most complex part of the game logic is when the user decides to shoot a given location. In addition to the LED matrix writing functions and methods the following methods are used to check for hits/ misses/ already hits or if the user expended the total shots. Other forms of game logic like displaying the number of ships sunk is also calculated during the moment user presses SHOOT

Identifying a Hit or Miss

```
if(isdigit(checkArray[row][col])==true &&checkArray[row][col]!='0')
```

To identify a hit the program checks whether passed coordinates during shoot correspond with a number in the position of the array where the user has shot. If the number is zero the player **misses**. Else if the number is 1~9 the player **hits**.

Identifying an Already Hit

To identify an already hit location first the program has to know if the location has been shot or not. For this it relies on an array named *shotsArray* which keeps track of the users shot locations. Whenever the user hits it sets the index of the row and col in this element position to '1'.

Therefore, by checking this value every time to see if it is '1' or '0' the user can know if they already hit a location twice. (the number of hits count does not increase in this scenario, total # shots increase)

```
if(shotsArray[row][col]=='1'){  
    //if shot location already shot location, don't run rest of current method  
    Lcd4_Set_Cursor(1,0);  
    Lcd4_Write_String("already hit");  
    return;  
}
```



Figure 11a - LCD display users real time stats

Finding out if a ship has been Sunk

Since the 'advanced feature' of detecting whether an adjacent spot has been hit or not has been implemented the program has to check if a ship has just been hit once or has it been hit completely (in this case it would sink)

Since each ship has been defined individually the program can identify if a ship has been sunk or not by counting all the remaining positions which the ship type exists. After each successful hit the location of the map where the ship was is set to '0'. Hence by counting the remaining number of digits of the ship's type it can be calculated if the ship has really been sunk.

For example.

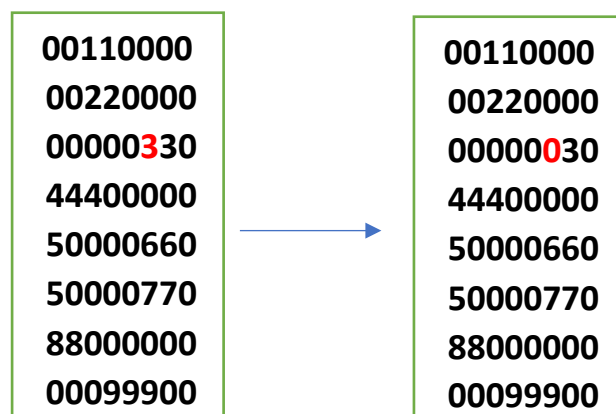


Figure 12 - how map arrays in program process a hit

1. If the following location has been hit. The number '3's index will be set to zero.
2. By counting the remaining number of '3' in the map the program can identify if there are any remaining parts in the ship and if so, the ship is not sunk. If there are absolutely no remaining parts for the ship it will be marked as sunk.

One interesting feature that can be easily implemented with this technique is to display to the user which ship they exactly hit. For example, they can get a message “destroyer hit”, “aircraft carrier hit”, “fishing boat hit” etc. (For this a “dictionary” must be defined for the each individual ship type) *(Please note that this feature has not been implemented in the current iteration of the game, however can be included in updates very easily)*

Code

```
if(checkforOthers(shipType,shipsArray)){ //method returns true if ship not fully sunk, false
                                         if ship has been fully sunk
}

else{

    //writeToLoc(1,0,"others not there!    ");

    shipKillCount++;    //increment kill count

    writeNumber(shipKillCount,2,13); //if ship fully sunk

    writeToLoc(2,8,"Kills");
```

Code Snippet 3 - how ship sunk count is incremented

More insight into the method called from the above image is given below.


```

int shipPartcounter=0;

//method to detect if ship is sunk or not
bool checkforOthers(uint8_t shipPart, uint8_t arr[8][8]){
    for(int j=0; j<8; j++){

        for(int i =0;i<8; i++){           //counts through entire map
            if(arr[j][i]==shipPart){
                shipPartcounter++; //counts if ship has any remaining parts
            }
        }

        if(shipPartcounter==0){           //returns false is ship is not yet sunk
            shipPartcounter=0;
            return false;
        }
        else{
            shipPartcounter=0;           //returns true if ship is sunk
            return true;
        }
    }
}

```

Code Snippet 4 - method to check if ship has been fully sunk

Check for WINNER or GAME OVER

To check for winner or loser a similar system is employed in the program where it counts the total number of ship parts and if the number of parts is zero the user WINS. The state changes to the WIN state so the user cannot accidentally trigger any other function. The user will have to reset the game to start over again. (since EEPROM memory is stored they can replay the game again or reload map using PC or via 2 player P2P mode)

To check for looser the program simply checks if the total number of shots is higher than a given shot threshold. (say 15 shots MAX) Then if the user takes more than 15 shots, they will lose the game and GAME OVER will be displayed. This value can be adjusted easily based on difficulty. Similar to WINNING the user is then transferred into a state called GAMEOVER which means they cannot do any other functions unless they reset the system.

LCD Displays WINNER or GAME OVER in trailing fashion across screen.



Figure 13 -GAME OVER and WINNER STATE

8x8 LED Matrix Logic

This section is covered separately as it encompasses a certain degree of complexity when writing to the LED matrix. The operation of the matrix can be explained below.

The LED matrix controlled by the MAX7219 chip consists of 8 rows and 8 columns. Each row can be considered as a byte and each column in a given row can be considered as a bit. This means that when writing to the display the complete byte has to be addressed at once. This means the user cannot write new data to the byte without the previous data erasing.

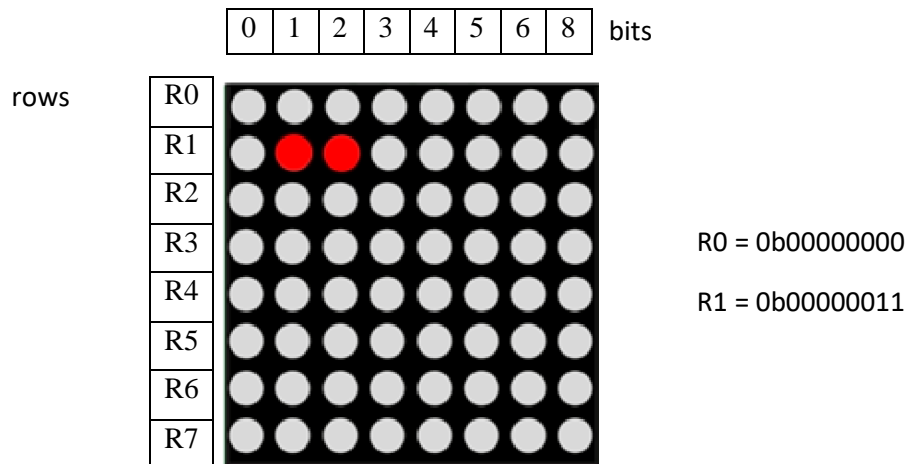


Figure 14 - Explanation of how data is written to Matrix

How LED matrix responds to Hits and Misses

The user navigates through the map and can see their movements via the cursor.

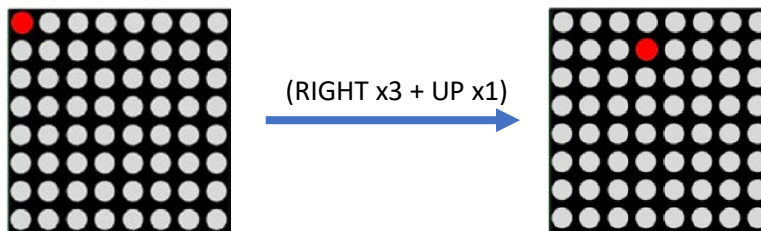


Figure 15 - Cursor movements

Whenever the user selects a location and hits shoot the program checks if a ship is there. If a ship is there the LCD will display hit while simultaneously blinking the corresponding LED light on the matrix array. This feature immediately notifies the user clearly that they have in fact hit a ship. If the user hits water no such indication is shown. Furthermore, if the user hits an already shot location, they will **not** get a blinking light (because it's already hit)

When a user successfully hits a ship the LED display will continue to display the hit location even as the user moves around the map.

Traditional Approach

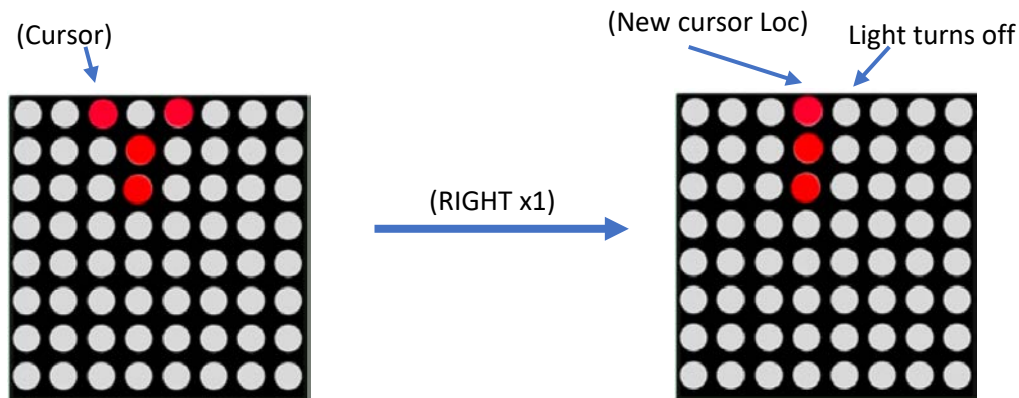


Figure 16 - Cursor movements without special methods

This phenomenon occurs because each byte has to be addressed properly when writing to the LED array. It does not effect other columns but effects other LEDs in the same row.

This undesirable phenomenon was however fixed according to the approach given below

New Improved Approach

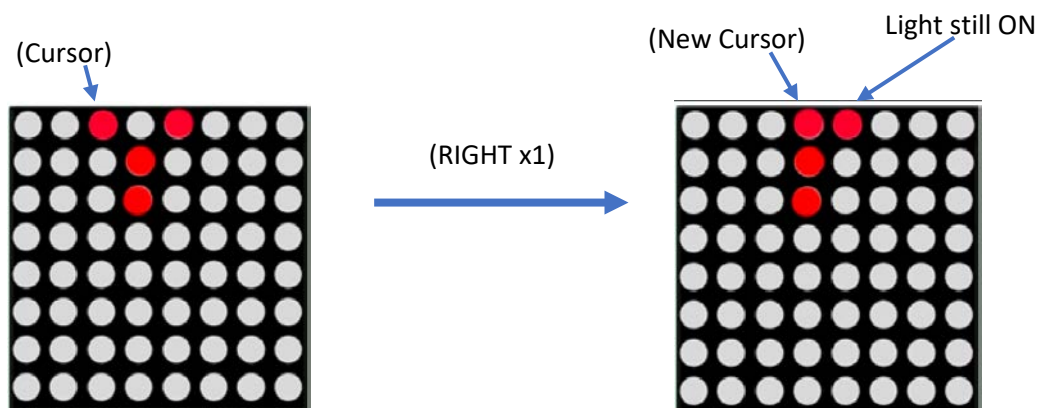


Figure 15 - Cursor movements with special methods

As can be seen in the figures above this approach sustains the map layout properly even as the cursor moves throughout the map. To implement this approach special methods were written to solve this problem.

How LED display persistence is Sustained Even with Cursor Movements

The following code snippets will better help understand how map persistence was sustained in the LED matrix while the cursor was moving and while new ships were hit or even when the blinking function was implemented.

Code Explanation

1. The above code snippet begins by taking in 3 parameters which are the row and column of the cursor and the array which contains the already hit ship data.
2. This array is then processed row by row and if a ship is already hit in a row it saves the hit location as a bit in byte named *writeByte*
3. *writeByte* is then stored temporarily until the for loop moves to the next row. Before moving to the next row however it writes the value to the LED matrix. Since column by column writing does not invoke a clear command in the LED matrix each column can be written to one by one. This enables proper persistence and ideal game experience
4. A similar method *blinkForHit* is also used for blinking an LED for a hit, even when the LED blinks the map stays persistent and does not affect any other LEDs which are already displayed on the matrix.

Please refer code for *blinkForHit* method

Other Important Methods etc Used in the Program

```
void loadMaptoLED(int row, int col, uint8_t arr[8][8]){ //special method to write to LED array

    //enables proper cursor movement, blinking, persistence
    for(int i = 0; i<8;i++){
        //reads an array row by row and saves each row to a byte
        uint8_t writeByte=0;
        for (int j =0;j<8;j++){

            if(isdigit(arr[i][7-j])==true && arr[i][7-j] != '0' ){           //if ship is present on map (from shots array etc) in
                                                                                   row, write row index as bit into byte
                writeByte |=(1<<(7-j));

            }
            else{writeByte&=~(1<<(7-j));} //if no data present write 0 to bit in writebyte

        }

        if(row==(i)){
            writeByte |=(1<<(col));           //add cursor alongside with shots etc data
        }

        max7219_digit(ic,7-i,writeByte);    //writes to LED array row by row (happens
        extremely quick)

    }
}
```

Custom Timer

One special feature of this current revision of battleship is the fact that no `_delay_ms` function has been used throughout the main program. Instead a custom timer was developed with the intentions of mitigating the need for an external library. This custom timer works using CTC mode and has a max resolution of 1ms.

The implementation of the custom timer can be seen below.

Prior to implementation all the necessary headers must be set like the activation of `TIMER1` and setting of value of `OCR1A` to a suitable value.

To calculate `OCR1A` value the following set of equations were used to generate a interrupt every 1ms.

$$\begin{aligned} \text{Timer Counts} &= \frac{12 \times 10^6}{2 \times 500} - 1 \\ &= 11999 \end{aligned}$$

500 Hz signal generates an interrupt every 1ms.

This interrupt is then used to increment a variable inside a while loop. The while loop waits until this variable reaches a threshold before released the program and allowing the rest of the code to function. The implementation of the process can be seen in the code below.

```
void myDelay(int delay1){                                //custom delay timer

    activated=true;                                       //variable which enables the counter to function
    TCNT1=0;                                             //sets TCNT1 register to zero
    TIMSK |= (1 << OCIE1A);                             //enables output compare interrupt

    while(timerCount<delay1){
//performs count operation/ timerCount increments inside ISR of output compare interrupt
    }
    activated=0;                                         //turn off timer, enables one time use every time timer is called
    timerCount=0;                                       //sets timerCount back to zero
    TIMSK &= ~(1 << OCIE1A);                          //turns of output compare match interrupt to prevent
                                                         unnecessary CPU bandwidth usage
}
```

Code Snippet 6 - custom Timer

The above method works side by side with the output compare interrupt to perform a methodized delay function for the program. In the above code snippet initially

1. The variable *activated* is set to true, this activates counting inside the output compare interrupt
2. The output compare interrupt is enabled and the TCNT1 is set to zero to begin counting
3. The while loop stalls the code until the variable (which is being incremented) in the interrupt reaches a certain threshold
4. After reaching this threshold the rest of the code is allowed to run which sets *activated* as false and turns off the output compare interrupt to save resources.

The output compare ISR is shown below

```
ISR (TIMER1_COMPA_vect)    //fires every time OCR1A is equal to value of TCNT1
{
    if(activated){          //activated variable enables or disables counting
        timerCount++;       //increment volatile timerCount variable
    }
}
```

Code Snippet 7 - custom Timer ISR

The advantage of this methodized timer is the ability to set any delay value in ms the user requires. For example, when the code is running the programmer can set a custom delay of 200ms simply by calling from any function in the main program

```
myDeLay(200);
```

Usage of Interrupts

In this program one interrupt has been used to change the state of the FSM. An interrupt was used due to the fact that it has the ability to override all other functions in the program and execute a certain task. In this program Interrupt 1 is used in the following manner,

```
ISR(INT1_vect){
    //transitions through the states on every press
    if(state==PLAY){state =LOAD;}
    else if(state==LOAD){state = CREATE;}
    else if(state==CREATE){state=PLAY; }
    else{ state=PLAY;}

    row_select=0;
    col_select=0;
}
```

Finite State Machine

A finite state machine was implemented to simplify each mode and to allow the same program to run in 2 player mode. The exact code can be uploaded to both systems and they can play without the use of a computer then after.

```
        switch(state){  
  
case LOAD:loadMap();PORTB|=(1<<PB5);writeLCD(LOAD,"Upload Map...")  
        ;break; //state to enable map receiving  
  
case PLAY:PORTB&=~(1<<PB5);play(); writeLCD(PLAY,"Play Game!");UCSRB |= (1<<RXCIF)  
;break; //state for playing game  
  
case READ:PORTB|=(1<<PB7); readMap()  
;break; //state which reads map received from Serial  
  
case CREATE:create(); writeLCD(CREATE,"P2P Share")  
;break; //state for on board map creation  
  
case OVER: Lcd4_Write_String("GAME OVER    ");writeLCD(OVER,"GAME OVER")           ;break;  
//end game state  
  
case WIN: Lcd4_Write_String("WINNER    "); writeLCD(WIN,"WINNER!");break; //winner game  
state  
default: state=PLAY;break; //default game state  
  
        }
```

The functions of each state of the finite state machine are indicated as follows

1. PLAY: State for playing the game
2. LOAD: State to anticipate a map sent from PC or even USER 2
3. READ: Processes the map received and saves as an array inside board
4. CREATE: State used to create new map in 2 player mode
5. WIN: State entered when game is won
6. OVER: State entered when GAME OVER

Demonstration of 2 Player Mode

For two player mode the main goal is to remove the task done by the PC. The user A can set the map for user B and vice versa. The user A loads P2P mode to set the map for user B and user B loads P2P mode and plays the game as usual. Results are displayed on User A side.

As the user B plays the game, they are sending cursor information to user A via Serial commands. Inside the program running on user A's board there are certain conditions which detect signals received from user B, which run the game logic as usual (hit,miss,WIN etc)

An additional feature that has been implemented is the ability of user A to even send the entire map to user B as if it was uploaded from a text file from a PC.

Code Running on User A (For setting the map/stage)

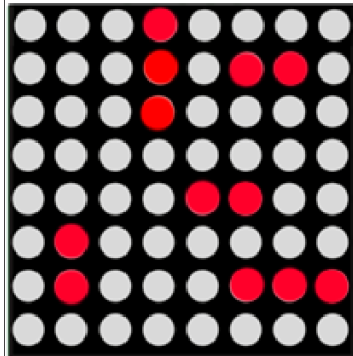
A partial snippet of the code running as a user sets up the map is given below. (This runs on transmitting side) (*note UDR is not necessary here and is for debugging*)

```
void create(){           //method used to create map on-board
//alternatively can be used for 2 player mode (send cursor locations and shoot)
    if(UP){
        row_select--; //decrements row selection for cursor
        UDR='w'; //write to serial monitor to signal cursor movements made by
second player
        myDelay(200);
    }
    if(DOWN){
        row_select++;
        UDR='s';
        myDelay(200);
    }
    if(LEFT){
        col_select--;
        UDR='a';
        myDelay(200);
    }
    ...
}
```

The above code is very similar to normal game functionality as it keeps track of the users' cursor information and allows them to set ship part location by pressing the button "shoot". The user can change the ship type (destroyer, aircraft carrier, dinghy boat.etc) via a separate button. Once the user is done setting the map they can "SET THE STAGE" of the map allow the other user to play.



Figure 15a - P2P share mode (2 player mode)



Each ship here could have a type “destroyer, aircraft carrier, fishing boat..etc” up to 9 ship types are available. The user changes the ships types via a separate button.

The user defines each ship one dot at a time.

Figure 16 - example of user set stage

The other board receives these signal via the it's RX interrupt and interprets them using a special method called *serialButtons()*;

A partial snippet of the method serialbuttons is given below. (This runs of receiving side)

```
void SerialButtons(){
    if(Receivedbyte=='a'){
        col_select--;
        //loadMaptoLED();
        //char_current==Receivedbyte;
    }
    else if(Receivedbyte=='d'){
        col_select++;
        //char_current==Receivedbyte;
    }
    else if(Receivedbyte=='w'){
        row_select--;
        //char_current==Receivedbyte;
    }
    ...
}
```

This code runs on the receiving side. The variable ReceivedByte is set by the RX interrupt and checked in the method serialButtons() to see which button the other player has pressed. They can move around the screen and shoot as usual. The board will detect these commands and run the normal functions like hit or miss as usual.

To demo this concept the serial monitor was used.

Macros were used for simulating button pressed sent from player 2. (*please note actual code has also been implemented within the program to allow physical button presses from other user by two boards are required*)

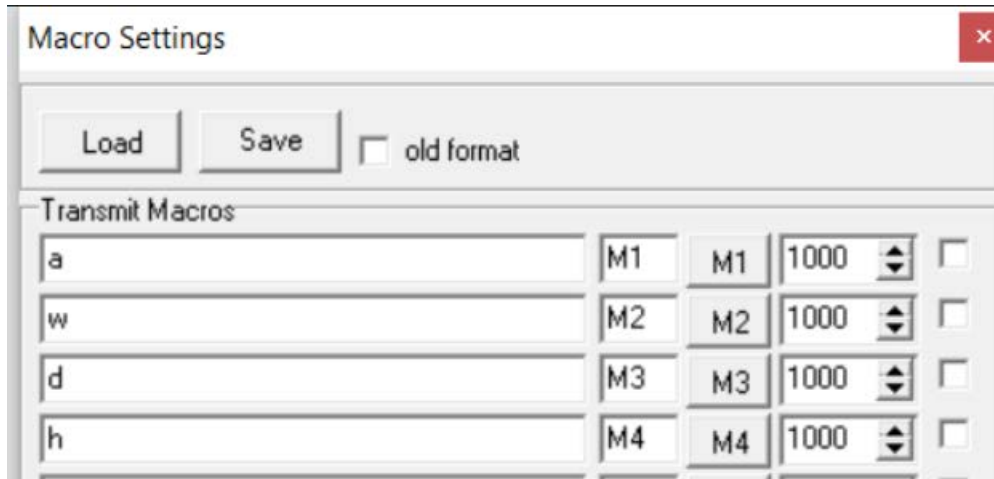


Figure 17 -Macros settings in Serial Monitor

These commands are



Figure 18 -Macros buttons

M1 → LEFT

M2 → UP

M3 → RIGHT

M14 → DOWN

M15→ SHOOT

M4 → CHEAT MODE (Displays the visible map on the LED Array; for debugging)

Please refer to attached video files for further clarification.

Results

Test Case	Pass/Fail	Remark
Proper hit, miss, already hit functionality	✓	Each case was tested and verified with LCD display
Blink light on LED matrix when hit	✓	Was verified with matrix
Displays hits and total shots	✓	Each case was tested and verified with LCD display
Proper incrementing of columns and variables	✓	Was checked with serial monitor and LCD
Proper transmission of map data to board	✓	Checked via serial monitor by mirroring data back
Proper writing of data to EEPROM	✓	Checked via serial monitor by mirroring data back
Proper SHOOT functionality to check for WIN or GAME OVER	✓	Checked by shooting all ships or exhausting shot limit
Check for proper sunks via adjacent hits	✓	Checked via LCD display by displaying "others there" or "sunk" if ship is fully sunk
Proper writing to LCD display	✓	Checked (obviously)
Proper writing to LED array	✓	Tried initially to observe functionality and tested throughout the use cases
Simulated 2 player mode	✓	Via Serial Monitor
Actual 2 player mode using 2 boards	*✓ (simulated via serial monitor)	Actual implementation of code is already done, just need to connect another board. Current method has been tested on another board (just the full game has not; connecting another LCD etc)

Table 2 - Test Cases

References

- [1] Atmel Corporation “Atmel-8155-8-bit-Microcontroller-AVR-ATmega32A_Datasheet”
- [2] Max7219 - Maxim Integrated “Max7219” Datasheet
- [3] LED Matrix- “Driving Dot Matrices with MAX7219” [Online]
<https://docs.labs.mediatek.io/resource/linkit7697-arduino/en/tutorial/driving-8x8-dot-matrices-with-max7219>
- [4] Writing to EEPROM - “EEPROM Read and Write” [Online}
<https://teslabs.com/openplayer/docs/docs/prognotes/EEPROM%20Tutorial.pdf>