**Screen Driver Subsystem**

In the development phase of the Battleship Game Console, using NOKIA 5110 LCD Screen is one of the most challenging task we had faced. In development, we aim to obtain fast and glitch free visual data in the screen. In the following part of this section we would explain how we initialized and used Tiva Board for proper operation and provide necessary algorithms and codes for clear explanation.

In order to have a functioning screen, 3 separate configurations are required [1]. For the sake of simplicity corresponding initialization codes are provided in Appendix 2. In the first step initialization of corresponding GPIO ports are done. For GPIO initialization we have 5 pin connections, with two different characteristics. First 3 pins (PA2, PA3, PA5) are SPI pins which governed by the SPI module on the TM4C123. Other two pins are external control pins of the screen (RESET and Data/Command). The initialization configuration for corresponding pins can be observed in Table 1.

Table 1: Initialize configuration of screen connections

|  |  |  |
| --- | --- | --- |
| SPI Pins  PA2 PA3 PA5 | GPIO\_PORTA\_DIR | All pins are set to output |
| GPIO\_PORTA\_DEN | All pins are digitally enabled |
| GPIO\_PORTA\_AMSEL | Analog function disabled |
| GPIO\_PORTA\_AFSEL | Alternate function enabled |
| GPIO\_PORTA\_PCTL | Pins connected to SSI interface |
| External  control Pins  PB0 & PB1 | GPIO\_PORTB\_DIR | All pins are set to output |
| GPIO\_PORTB\_DEN | All pins are digitally enabled |
| GPIO\_PORTB\_AFSEL | Alternate functions are disabled |

In the second step the internal SPI module of TM4C123 must be configured. In this configuration TM4C123 is used as master. In the connection between screen there is no acknowledge signal so there is no need to configure Rx port. The SPI configuration we used in this console can be observed in Table 2.

Table 2: SPI configuration for NOKIA 5110 Screen

|  |  |
| --- | --- |
| SSI0\_CC | System clock is chosen (80MHz) |
| SSI0\_CPSR | Clock prescale is set to 2 (80MHz => 40MHz) |
| SSI0\_CR0 | Freescale SPI Frame Format  8-bit data  Serial Clock Rate = 10 (40Mhz => 4MHz) |

In the final phase, after all the hardwire connections are done and ports are initialized, the display is configured by following the steps provided in datasheet of PCD8544 NOKIA 5110 display [2]. The configuration steps are provided below.

* 100ms RESET (Active low) signal is send
* Command mode is activated by setting D/C pin to low
* Power ON, Vertical operation mode activated, and Extended command mode (H) activated, with (0x21) signal
* Vop Value is set to 0xBF
* Temperature value is set to 0x05
* Voltage bias value is set to 0x13
* Extended command mode is deactivated (0x20)
* Normal display mode is select (0x0C)

After the initialization phase of the screen, which shown in previous part, is completed, the screen is ready to receive commands, text, and images to be displayed. To send our visual information we wrote subroutine “DATA\_WRITE” and to adjust the location of the data going to be written we wrote subroutine “ADDRESS\_CHANGE”. Since, in most parts of the main code these subroutines are widely used, it is necessary to write these routines with minimum impact to the branched code. Hence both subroutines take their command from register R4 and returns no value to main code.

To briefly explain the operation of the “DATA\_WRITE”; when called, this subroutine takes least significant byte of the R4. After receiving R4, controls status register of the SSI (SSI\_SR), as if there is any empty location in Transmit FIFO. If FIFO is full, waits until any empty spot is present. If FIFO is not full, sends least significant byte of R4 to the data register of SSI. After writing the corresponding value to FIFO, ends its operation and returns to the address stated by link register.

The “ADDRESS\_CHANGE” subroutine is written for adjusting the memory address of the screen. In NOKIA screen 84 horizontal (X) and 48 vertical (Y) pixels are present. On the other hand, vertical pixel bits are organized as 8bit pixel blocks which means there are only 6 vertical address. In normal operation the screen increments its address after every received data. However, this addressing mode is very low and has no flexibility. Hence for our operation we need to control these 84x6 addresses. For that in “ADDRESS\_CHANGE” subroutine we take address data from the R4 register in a special configuration. The least significant byte of the R4 is reserved for X data and second byte is reserved for Y data. This organization is shown in Figure 4.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 24:23 | | 16:15 | | 8:7 | | 0 |
| 0000.0000 | | 0000.0000 | | YYYY.YYYY | | XXXX.XXXX | |

Figure 4: Bitmap configuration of R4 data input

To briefly explain the operation of the “ADDRESS\_CHANGE”; when called, this subroutine first makes Data/Command pin low to indicate it will send a command. Then takes the R4 value and extracts Y and X values from the R4. Before sending these values, controls status register of the SSI (SSI\_SR), as if there is any empty location in Transmit FIFO. After FIFO became available, sends corresponding X and Y address values one by one to the screen. Before finishing its operation, this block needs to convert its D/C pin to high for forthcoming data write commands. Hence after predetermined delay, sets D/C pin to high and ends its operation and returns to the address stated by link register.

MEMORY MAP

Design of a multiple input and multiple scenario game forces us to follow various data and their real time comparison. This kind of task is impossible if we only use the 13 register in the core of the ARM. To cope with this amount of data we used SRAM of TM4C123. This memory block provides 32Kbyte of memory area for us. As shown in Table 3, we used this memory area for various operations. The corresponding code of “MEMORY\_MAP” is provided in Appendix 4.

Table 3: Corresponding locations and purposes of memory blocks

|  |  |  |
| --- | --- | --- |
| Memory Name | Address | Purpose |
| EMPTY\_FIELD | 0x20000400-  0x200005F7 | Holds an empty playfield map for clear screen cases |
| PLAYFIELD | 0x20000600-  0x200007F7 | Holds the ships location on a map. In deployment phase this screen is shown in screen |
| MINEFIELD | 0x20002000-  0x200021F7 | Holds the mine location on a map. In attack phase this screen is shown in screen |
| SHIP\_EMPTY  SHIP\_CIVIL  SHIP\_BATTLE | 0x20000800-  0x20000817 | Holds bitmaps for empty ship, battleship and civilian ship |
| CURSOR  MINE | 0x20000818-  0x2000081D | Holds bitmaps for mine cursor and mine |
| OLD\_SHIP\_LOC\_X  OLD\_SHIP\_LOC\_Y | 0x20001000  0x20001001 | Saves the last location of the ship cursor. Have important role in cursor movement. I will be further investigated in Ship cursor section. |
| OLD\_CURSOR\_LOC\_X  OLD\_CURSOR\_LOC\_X | 0x20001002  0x20001003 | Saves the last location of the mine cursor. Have important role in cursor movement. I will be further investigated in Ship cursor section. |
| SHIP\_MEMO | 0x20001100  0x20001149 | Holds locations of placed ships. First word works as a pointer. It holds the address of incoming ship memory |
| MINE\_MEMO | 0x20001150  0x20001199 | Holds locations of placed mines. First word works as a pointer. It holds the address of incoming mine memory |
| GP\_MEMORY | 0x20006000-  0x200060FF | Reserved for general purpose memory usage. |

In this memory block one of the most important memory addresses are “SHIP\_MEMO” and “MINE\_MEMO” blocks. These blocks are used for saving the Y.X location of the ships and mines. The data saved in these blocks have crucial importance when we determine the winner and loser. To obtain best performance from this memory space we use pointer approach. The first word of each memory space holds the pointer address of next used memory space. This approach gives us flexibility in ship and mine number. With this approach we can place up to 10 ships and mines. On the other hand, due to 20 second limitation in attack phase, code blocks internally limit ship and mine number.

Other big blocks of memory are 3 field maps. Empty field, playfield and minefield holds invaluable visual information. Despite they do not have any impact on calculations, these blocks have important role in the visualization of the game. In Figure 6 and Figure 7 on screen shots of these maps are shown with ship and mine symbols.

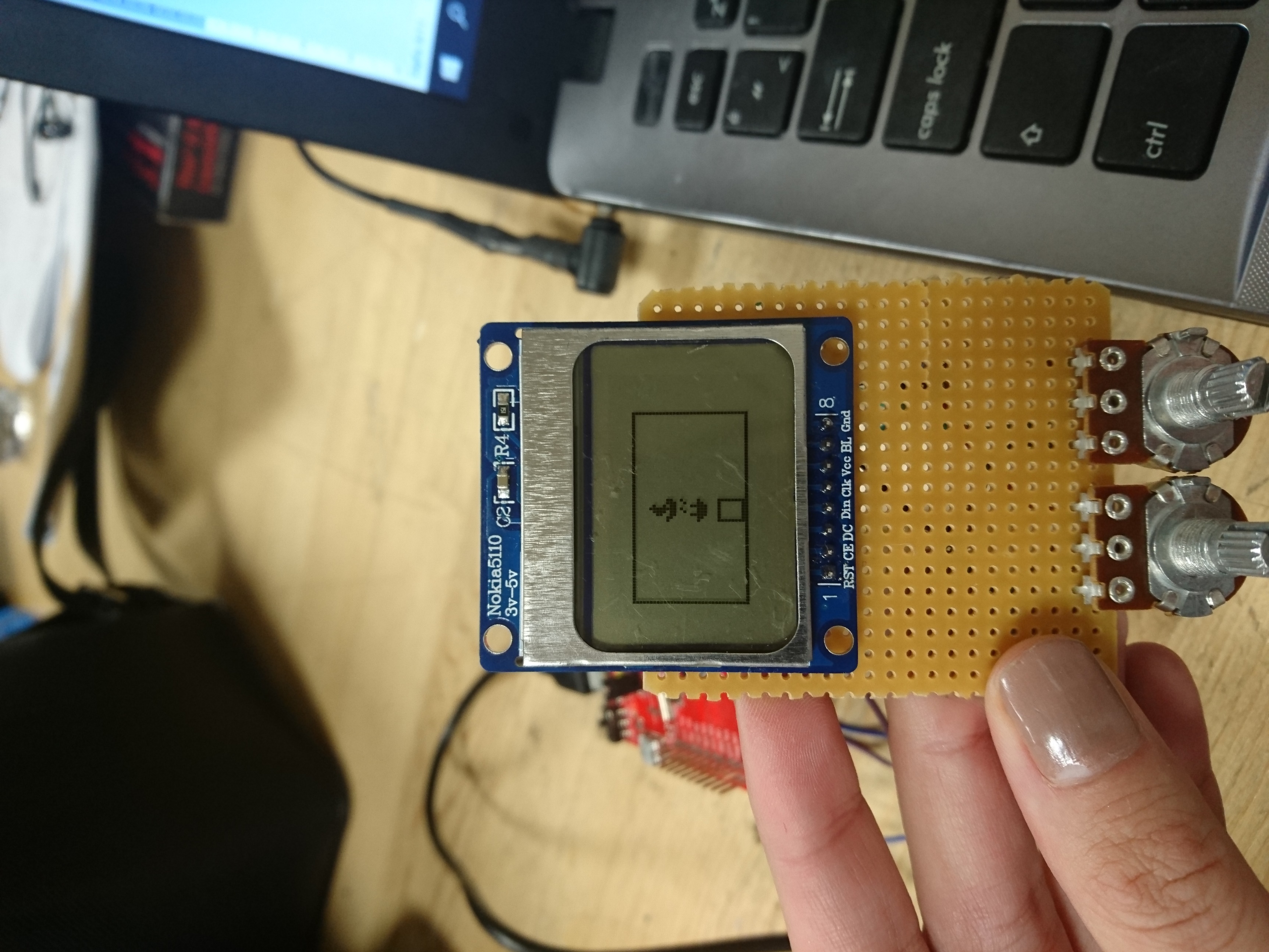


Figure 6: On screen visualization of PLAYFIELD with civilian ship(top), battleship(middle) and ship cursor(bottom).

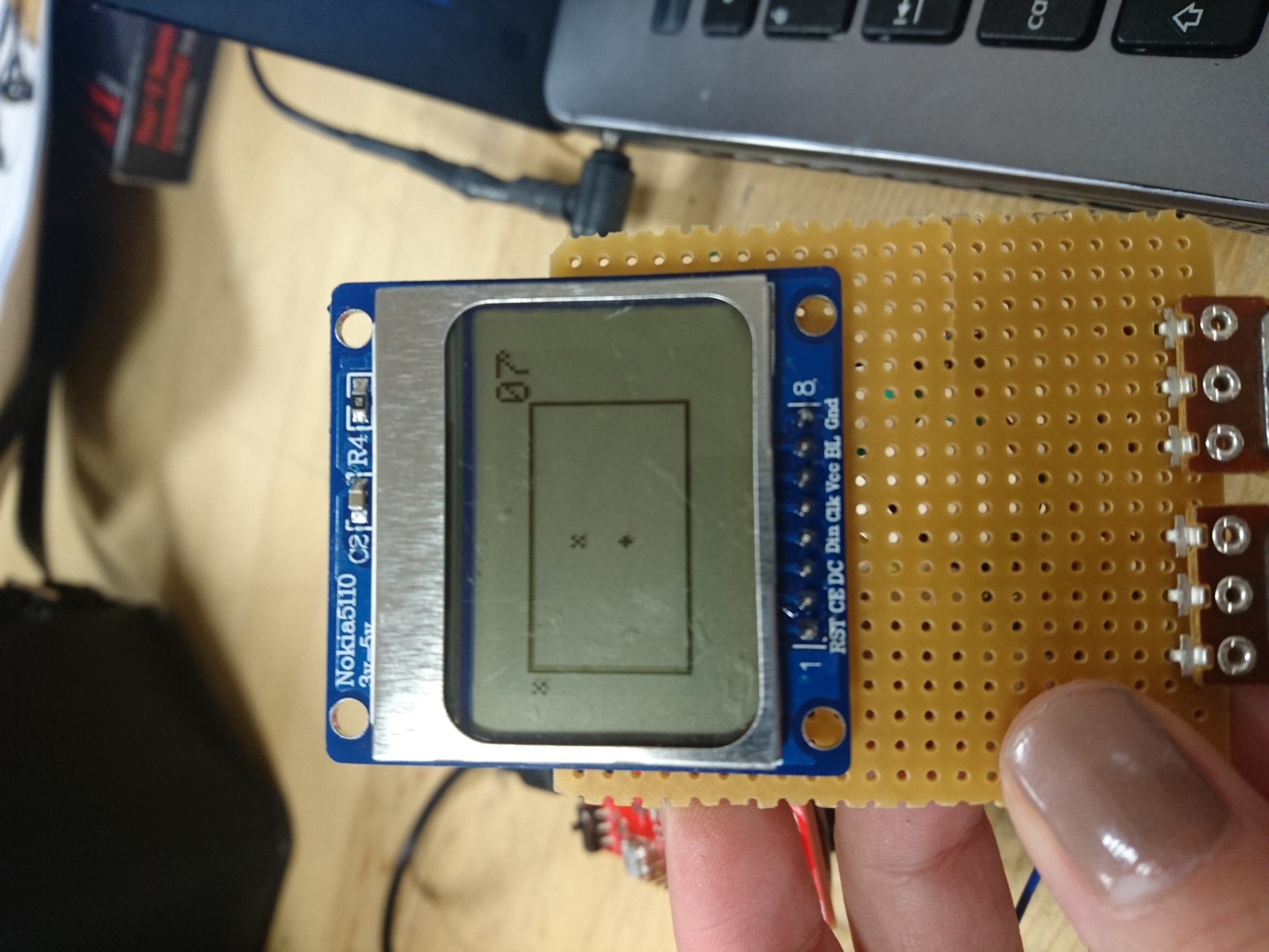


Figure 7: On screen visualization of MINEFIELD with mine(top) and mine cursor(bottom).

CURSORS

In the visual design of this game console, cursor location subroutines hold very important role. Without proper operation of cursor, it is impossible for player to place any ship or mine. To fit with different needs of the game we have two different cursor subroutines, “SHIP\_CURSOR” and “MINE\_CURSOR”. Despite their different names and areas of uses, there is only minor differences. Hence in this part we will explain the operation of the “SHIP\_CURSOR” subroutine and after that state the differences between “MINE\_CURSOR”.

Cursor units take their input very similar to “ADDRESS\_CHANGE” which shown in Figure 4. On the other hand, in this block we also have a ship type input which is located in most significant byte. If the type is 0x11 then it is cursor. For the input is 0x01, it means civilian ship and for 0x10 input it means battleship for us. The bitmap is shown in Figure 8.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 24:23 | | 16:15 | | 8:7 | | 0 |
| MMMM.MMMM | | 0000.0000 | | YYYY.YYYY | | XXXX.XXXX | |

Figure 8: Bitmap configuration of R4 data input

The operation of cursor can be divided into three parts as “clear old cursor”, “change field memory” and “cursor write”. At the start of the operation the main aim is clearing the old cursor, so we will continue with clear glitch free screen. For that we benefit from the “OLD\_SHIP\_LOC” memory. The previous location of cursor is kept in this memory address, and we first need to clear this location. To clear it, we set the memory address of screen to old cursor location. Then we send playfield visual data which we keep at “PLAYFIELD” memory array.

After clearing the old cursor, we come to a decision point. If no button is pressed, it skips memory write part and directly moves to “cursor write” part. If ship input is present, then moves to “change field memory “part. In change memory part, the corresponding ship bitmap is written onto the “PLAYFIELD” location by ORR operation with old memory. This ORR with past values preserves the old writings. The ship shapes and cursor shape can be observed in Figure 6.

At the last stage we need to write cursor value on screen. But before starting we write the current cursor location to old cursor location memory. With this step we will be able to clear this cursor in the next step. After saving the cursor location we adjust the screen address to cursor location. Then we ORR the cursor bitmap with corresponding byte of “PLAYFIELD” memory and write onto the screen. As a result, old cursor location is changed with the new cursor location in this block.

To briefly state the main differences between “MINE\_CURSOR” and “SHIP\_CURSOR”, the ship bitmap is 8x8 bit. On the other hand, mine bitmaps are only 3x3 bit. This difference can be solved only by changing numerical values in the code. Also, in the mine mode there is only one place mine input which shown by both 0x01 and 0x10 inputs. The mine shape and cursor shape can be observed in Figure 6.