CSSE2010 & CSSE7201

AVR Project

Semester One, 2024

Due: 4:00pm, Friday 24th May

Weighting: 20% (100 marks)

The University of Queensland

School of Electrical Engineering and Computer Science

Objective

As part of the assessment for this course, you are required to undertake an AVR project which will test you against some of the more practical learning objectives of the course, namely your C programming skills applied to the ATmega324A microcontroller.

You are required to modify a program to implement additional features. The program is a basic template of the game "Battleship" (described in detail on page 3). The AVR ATmega324A microcontroller runs the program and receives input from multiple sources and uses an LED matrix as a display for the game, with additional information being output to a serial terminal and – to be implemented as part of this AVR project – a seven-segment display and other devices.

The version of Battleship provided to you has very basic functionality – it will present a splash screen upon launch, respond to push button presses or a terminal input "s"/"S" to start the game, then display the boards for the game Battleship; one for the human player and one for the computer player. You can add features such as moving the cursor, game scoring, pausing, audio etc. The various features have different tiers of difficulty and will each contribute a certain number of marks. Note that marks are awarded based on demonstrated functionality only, regardless of how much effort or code has gone into attempting such functionality.

Don't Panic!

You have been provided with approximately 1500 lines of code to start with – many of which are comments. Whilst this code may seem confusing, you do not need to understand all of it. The code provided does a lot of the hard work for you, e.g., interacting with the serial port and the LED matrix display. To start with, you should read the header (`.h`) files provided along with `game.c` and `project.c`. You may need to look at the AVR C Library documentation to understand some of the functions used. Several intro/getting started videos are available on Blackboard, which contains a demonstration of some of the expected functionality to be implemented and walks through setting the project up with the provided base code, and how to submit. Note that the requirements in this document takes priority over anything shown in the feature demonstration videos.

Grading Note

As described in the course profile, if you do not score at least 10% on this AVR project (before any late penalty) then your course grade will be capped at a 3 (i.e., you will fail the course). If you do not obtain at least 50% on this AVR project (before any late penalty), then your course grade will be capped at a 5. Your AVR project mark (after any late penalty) will count 20% towards your final course grade.

In addition, tiers may have more marks available in the features than the total marks allocated for that tier. You cannot earn more marks than the total allocation, however, you will be given all marks you are awarded for each feature up to this limit. This means that you can still get full marks for a tier, even if you are not awarded full marks for each feature, if you complete features with marks that would total above the allocated marks for that tier.

Your total marks for Tier B cannot exceed your total marks for Tier A. Your total marks for Tier C cannot exceed your total marks for Tier B.

Code Style

No marks are awarded nor deducted in the marking criteria for code style. However, poorly styled code will be detrimental to debugging and expanding the code, and so may lead to a loss of marks that way. If your code is sufficiently poorly styled, then asking for assistance from course staff may be refused until your style is improved. This includes, but is not limited to, if your code:

- Utilises `goto` or digraphs/trigraphs;
- Fails to bitshift where doing so would be expected;
- Declares variables without meaningful names;
- Uses integer variables without exact width declarations (e.g. `int` instead of `int8_t`);
- Has inconsistent or contradictory indentation and/or brace positioning.

Academic Merit, Plagiarism, Collusion and Other Misconduct

You should read and understand the statement on academic merit, plagiarism, collusion and other misconduct contained within the course profile and the document referenced in that course profile. You must not show your code to or share your code with any other student under any circumstances. You must not post your code to public discussion forums or save your code in publicly accessible repositories. You must not look at or copy code from any other student. All submitted files will be subject to electronic plagiarism detection and misconduct proceedings will be instituted against students where plagiarism or collusion is suspected. The electronic plagiarism detection can detect similarities in code structure even if comments, variable names, formatting etc. are modified. If you copy code, you will be caught.

Program Description

The program you will be provided with has several C files which contain groups of related functions. The files provided are described below. The corresponding `.h` files (except for `project.c`) list the functions that are intended to be accessible from other files. You may modify any of the provided files. You must submit all files used to build your project, even if you have not modified some provided files. Many files make assumptions about which AVR ports are used to connect to various IO devices. You are encouraged not to change these.

- `project.c` – this is the main file that contains the game event loop and examples of how time-based events are implemented. You should read and understand this file. A majority of the modifications and additions to the project code will be in this file and `game.c`.
- `game.c/.h` – this file contains the implementation of the game components and is used to store the state of the game. You should read this file and understand what representation is used for the game state and the board representations. A majority of the modifications and additions to the project code will be in this file and `project.c`.

- `display.c/.h` – this file contains the implementation for displaying the current state of the board. This file contains useful functions for displaying the board to the LED matrix.
- `buttons.c/.h` – this contains the code which deals with the push buttons. It sets up pin change interrupts on those pins and records rising edges (buttons being pushed).
- `ledmatrix.c/.h` – this contains functions which give easier access to the services provided by the LED matrix. It makes use of the SPI routines implemented in `spi.c`.
- `pixel_colour.h` – this file contains definitions of some useful colours for use with the LED matrix.
- `serialio.c/.h` – this file is responsible for handling serial input and output using interrupts. It also maps the C standard IO routines (e.g., `printf()` and `fgetc()`) to use the serial interface so you are able to use `printf()` etc. for debugging purposes if you wish. You should not need to look in this file, but you may be interested in how it works, and the buffer sizes used for input and output (and what happens when the buffers fill up).
- `terminalio.c/.h` – this encapsulates the sending of various escape sequences which enable some control over terminal appearance and text placement – you can call these functions (declared in `terminalio.h`) instead of remembering various escape sequences. Additional information about terminal IO will be provided on the course Blackboard site.
- `spi.c./h` – this file encapsulates all SPI communication. Note that by default, all SPI communication uses busy waiting (i.e., polling) – the "send" routine returns only when the data is sent. If you need the CPU cycles for other activities, you may wish to consider converting this to interrupt based IO, similar to the way that serial IO is handled.
- `timer0.c/.h` – sets up a timer that is used to generate an interrupt every millisecond and update a global time value that is used to time various game events (such as the curser flashing). This can be useful for implementing any features that require precise timing.
- `timer1.c/.h` & `timer2.c/.h` – largely empty files, that can be used for features that require timers/counters one and two.

NB: When you create a new project in Microchip Studio, a `main.c` file will automatically be created, containing an empty `main()` function. `project.c` also contains a `main()` function, but Microchip Studio will preferentially look the `main()` function in the `main.c` file, if it exists. Please ensure that you delete the `main.c` file so that Microchip Studio will look for the `main()` function in `project.c`.

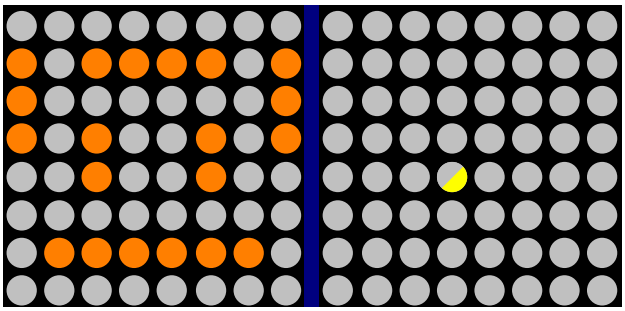
Battleship Description

This AVR project involves creating a replica of the classic game "Battleship". Battleship is a two-player game that coarsely simulates a naval battle. Each player has a grid board, on which they place narrow pieces representing ships. They then take turns in firing shots at each other into announced grid squares, after which their opponent tells them if they've hit one of their ships, or missed. In this implementation, there will be one human player and one computer player. The human player inputs their move on their turn with the IO board and/or terminal, while the computer player takes their turn automatically.

Each player has the following ships in their fleet, of the given length:

- Carrier (6);
- Cruiser (4);
- Destroyer (3);
- Frigate (3);
- Corvette (2); and
- Submarine (2).

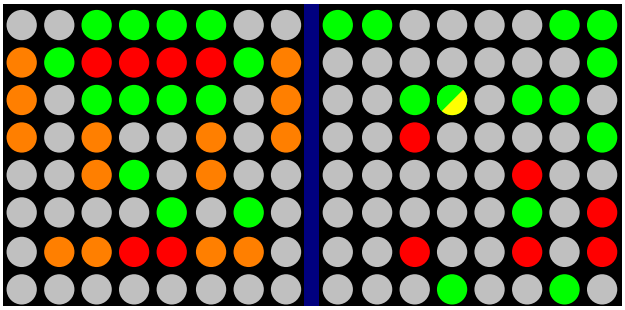
A diagram of the LED matrix, as it appears when the game starts, is shown in Figure 1, and a diagram of the LED matrix as it may appear during the game is shown in Figure 2.



Human Board Computer Board

Figure 1: LED Matrix Diagram for initial layout of Battleship

The left half of the LED matrix is the human board; it contains the human player's ships, and the computer player will be firing shots at grid spaces on that board to try to hit those ships. The human player's ships are visible to them, and shown in orange. The right half of the LED matrix is the computer board; it contains the computer player's ships, and the human player will be firing shots at grid spaces on that board to try and hit those ships. The computer player has the same set of ships as the human player, but they are not visible to the human player, and are likely in different locations. The human player can select specific grid squares on the computer board by moving the flashing yellow cursor. The flashing is indicated with the half-yellow pixel in Figure 1 and Figure 2. Between the two boards is a gap in blue; this does not appear on the LED matrix, but is instead shown to separate the two boards, though the LED matrix does have a seam at this position.



Human Board Computer Board

Figure 2: Potential LED Matrix Diagram for mid-game Battleship

As each player takes shots, the LED matrix fills in to show the result of the shot. If the shot hits a ship, that pixel is coloured red. If the shot misses, then that pixel is instead coloured green. This allows the human player to find the computer player's ships.

When each pixel of a ship has been hit, that ship is sunk. When all of a player's ships have been sunk, that player loses, and their opponent wins.

Initial Operation

The provided program has very limited functionality. It will display a splash screen which detects the rising edge on the push buttons B0, B1, B2 and B3, as well as the input terminal character "s"/"S". Pressing any of these will start a game of Battleship.

Once started, the provided program detects a rising edge on the push button B0, but no action is taken on this input (this will need to be implemented as part of the **Move Cursor Push Buttons** feature).

Wiring Advice

When completing this AVR project, you will need to make additional connections to the ATmega324A microcontroller to implement particular features. To do this, you will need to choose which pins to make these connections to. There are multiple ways to do this, so the exact wiring configuration will be

left up to you, and you should communicate this using your submitted feature summary form (included at the end of this document. A standalone version is also available on Blackboard). Hint: Before implementing any features, read through them all, and consider what peripherals each feature requires and any pin limitations this imposes. If you do not do this, you may find yourself in a situation where the pins that must be used for a peripheral for a later feature are already connected to another peripheral, requiring you to rewire your breadboard and update your code before you can use the new peripheral. Some connections are defined for you in the provided base code and are shown in grey in the table on the next page.

Wiring Table

Port	Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	Pin 1	Pin 0
A								
B	SPI connection to LED matrix				Button B3	Button B2	Button B1	Button B0
C								
D							Serial RX	Serial TX
							Baud rate: 19200	

Program Features

Marks will be awarded for features as described below. Part marks will be awarded if part of the specified functionality is demonstrated. Marks are awarded only on demonstrated functionality in the final submission – no marks are awarded for attempting to implement the functionality, no matter how much effort has gone into it, if there is no evidence of functionality when the program is run. You may implement higher-tier features without implementing all lower-tier features if you like (subject to prerequisite requirements). The number of marks is not an indication of difficulty. It is much easier to earn the first 50% of marks than the second 50%, and marks may be deducted if features interact negatively with one another, and the number of potential iterations grows quadratically with the number of features implemented.

You may modify any of the code provided (unless indicated otherwise) and use any of the code from learning lab sessions and/or posted on the course Blackboard site. For some of the easier features, the description below may tell you which code to modify or there may be comments in the supplied code to guide you.

Note: The course has a pass hurdle of 10% for this AVR project, which can be achieved by completing the first two features (**Splash Screen** and **Move Cursor with Push Buttons**), and being awarded full marks. See Grading Note for further details.

Minimum Performance — Tier X: Pass/Fail

Your program must have at least the features present in the code supplied to you, i.e., it must build and run, show the splash screen, and display the initial game when a push button or "s"/"S" is pressed. No marks can be earned for other features unless this requirement is met, i.e., your AVR project mark will be zero.

Splash Screen — Tier A: 4 marks

Modify the program so that when it starts (i.e., the AVR microcontroller is reset) it outputs your name and student number to the serial terminal, in the indicated location (remove the placeholder text, including the chevrons <>). Do this by modifying the function `start_screen()` in file `project.c`.

The Terminal Summary section shows all features that require printing text to the terminal. You may find it useful to read through this and plan your terminal layout to avoid running out of room or having messages collide with each other.

Move Cursor with Push Buttons — Tier A: 6 marks

The provided program does not allow moving the cursor. Modify the program so when push button B0 (connected to pin B0) is pressed, the human player's cursor on the computer board moves right (away from the human board). Similarly, pressing push button B1 (connected to pin B1) should move the cursor down, push button B2 (connected to pin B2) should move the cursor up, and push button B3 (connected to pin B3) should move the cursor left.

If the cursor would be moved off the edge of the computer board, it should wrap around to the other side of the board. For example, if the cursor is on the far right side of the LED matrix, then pressing B0 should move the cursor to be adjacent to the seam between the two board (in the same row). The cursor should never end up on the player board. Whenever the cursor is moved, the flashing cycle should be reset, such that the cursor instantly lights up yellow, and remains yellow for 200ms. The cursor should move when the button is pressed; no behaviour is expected for when the button is released, nor if the button is held down.

Hints: In the `play_game()` function in the file `project.c`, when push button B0 is pressed, the function `move_cursor(1,0)` in the file `game.c` is called. This function is currently empty; start by filling in the `move_cursor()` function (there are some hints to get you started). Note that the cursor variables are unsigned integers in the base code. This makes some methods of edge wrapping easier, but makes others harder. If you wish to use one of the latter methods, you should change the variables to signed integers.

Move Cursor with Terminal Input — Tier A: 6 marks

The provided program does not register any terminal inputs once the game has started. Modify the program such that pressing "w"/"W" moves the cursor up, and "a"/"A", "s"/"S" and "d"/"D" move the cursor left, down and right respectively, in a similar manner to the previous task. Note that both the lower case and upper case of each letter should execute these movements as described. Also note that the inbuilt serial functionality handles keyboard inputs that are held down for you.

Just like in the previous task, the cursor should wrap around the edges, and the flashing should be reset whenever the cursor moves. Holding down a key will usually send multiple instances of that key to the terminal. Unlike the push buttons, this means holding down a key will result in the cursor repeatedly moving.

On the splash screen, the game can be started by pressing "s"/"S"; looking at the function `start_screen()` should give you an idea of how to read serial input from the terminal. Do not make other keys start the game from the splash screen.

Human Turn — Tier A: 8 marks

*Requires **Move Cursor with Push Buttons** and/or **Move Cursor with Terminal Input** features to be implemented.*

When the user presses the "f"/"F" key, they should fire at the location indicated by the cursor. If the location contains a section of one of the computer player's ships, that location should be coloured red; otherwise, it should be coloured green. It should remain that colour for the remainder of the game. Until the cursor is moved off that location, and when the cursor moves over that location in the future, it should flash between yellow and red/green. When the location is fired at, it should instantly change

colour if the cursor is in the "off" state, but should have no effect if the cursor is in the "on" state, until it transitions. Firing a shot does not effect the timing cycle of the cursor.

Hint: Each location uses an eight bit number to determine what is at that location. The three least significant bits determine what is in that location, with 000_2 being no ship, and 001_2 through 110_2 enumerating the ship types. The next bit is 1 if the location contains the end of a ship. Then the next bit is 1 if the ship is horizontal, or 0 if the ship is vertical. This leaves three unused bits. These can be used to store the damage state of the ship, and later in the **Sinking Ships** task, the sunken state of the ship.

After the **Computer Turn – Basic** feature (below) is implemented, the turns should alternate between the human and computer players. If the **Computer Turn – Basic** feature is not implemented, the human player should instead take multiple turns in a row.

You may wish to create your own function in `game.c` to implement the human player's turn, and call this from `project.c` when "f"/"F" is pressed. If you do, ensure that you declare it correctly in `game.h`.

Computer Turn – Basic Moves — Tier A: 6 marks

*Requires **Human Turn** feature to be implemented.*

While the program knows the location of all of the human player's ships, the computer player should not. As such, it needs to find the human player's ships by playing the game.

After the human player has taken a turn, the computer should automatically take a turn. For the basic strategy, the computer player will fire at the top left location of the human board on their first turn. On the following turns, they will fire one location to the right of the previous location. Once they have fired on the entire top row of the human board, they will fire at the leftmost location of the second row, and then move right again. When the computer player has fired on a location, that location should be coloured red or green, depending on if that location does or does not contain a piece of one of the human player's ships.

Invalid Move — Tier A: 4 marks

*Requires **Human Turn** and **Computer Turn – Basic Moves** features to be implemented.*

The human player should not be able to fire on a location they have previously fired upon. If they attempt to do so, nothing should happen, and it should remain their turn. In addition, you should print a message indicating that the human player attempted to make an invalid move to the terminal. If they attempt to make multiple invalid moves in a row, you should print different message, of increasing intensity; you should have at least three unique messages. Each of these message should appear in the same location, replacing the previous. Once a valid move has been made, clear this message. When the cursor is on a location that cannot be fired at, it should flash dark yellow instead of bright yellow when the cursor is in the "on" state. You will need to define a colour for this. You may use any value that produces a colour that a) would be described as [dark] yellow and b) is notably darker than the yellow in the base code is. Additionally, if the cursor is in the "on" state when a shot is fired, it should immediately turn dark yellow.

Sinking Ships — Tier A: 4 marks

*Requires **Human Turn** and **Computer Turn – Basic Moves** features to be implemented.*

When each segment of a ship has been hit, that ship is sunk. When this happens, a message should be printed to the terminal. You should designate an area of the terminal of at least six lines high, and of appropriate width, where these messages will be printed. If one of the human player's ships has been sunk, then "I Sunk Your <Ship>" should be printed to the left side of this area, left aligned, replacing

"<Ship>" (including the chevrons) with the name of the sunken ship. If one of the computer player's ships has been sunk, then "You sunk my <Ship>" should be printed to the right side of this area, right aligned, again replacing "<Ship>" with the ship name. Each message for each player should print on a new line, and the messages already printed should remain in place. For example, towards the end of the game, a section of the terminal could look something like:

```
I Sunk Your Cruiser                    You Sunk My Destroyer
I Sunk Your Destroyer                   You Sunk My Corvette
I Sunk Your Frigate                      You Sunk My Cruiser
I Sunk Your Corvette                     You Sunk My Frigate
                                         You Sunk My Carrier
                                       You Sunk My Submarine
```

For this case, the human player had their cruiser, destroyer, frigate, and corvette sunk in that order, and the computer player had their destroyer, corvette, cruiser, frigate carrier and submarine sunk in that order. The order of the ships between the two players is not derivable from these messages.

You may instead use "The Computer Player Sunk The Human Player's <Ship>" and "The Human Player Sunk The Computer Player's <Ship>" for the messages, with the same formatting as above.

When a ship is sunk, all of its segments should immediately change to dark red on the LED matrix, and remain so for the rest of the game. You will need to define a new colour for this. You may use any value that produces a colour that a) would be described as [dark] red and b) is notably darker than the red in the base code is. The final location fired at should never appear [standard] red; if the cursor is in the "off" state when the ship is sunk, it should immediately turn dark red.

Towards the end of this document, there is a Terminal Summary, which shows all the information that needs to be displayed on the terminal for the various features throughout the game. It also contains a note about not spamming the terminal.

Game Over — Tier A: 6 marks

*Requires **Sinking Ships** feature to be implemented.*

When all of one player's ships have been sunk, that player loses, and the other player wins. When this happens, print a message to the terminal stating whether the human or computer player has won. Do not remove other messages from the terminal (unless the **High Score** feature below has been implemented). In addition, colour every location that had not been fired at on both boards of the LED matrix either dark orange if it contains a ship (including the human player's [standard] orange ships), or dark green if it does not. This will result in no pixel on the LED matrix being unlit. You will need to define new colours for these. Furthermore, turn off the cursor, and stop its flashing cycle.

When "s"/"S" or any IO board push button is pressed, the game should return to the splash screen (for both the LED matrix and terminal). At this point, nothing from the previous game should remain visible. Any settings from later features (e.g. **Seven-Segment Timer** below) should be retained, with the setting shown on the splash screen. These settings should be changeable as per the respective feature descriptions. A new game can then be started by pressing "s"/"S" or any IO board push button again.

The existing infinite loop of the `main` function of the base code calls `new_game`, `play_game` and `handle_game_over` forever; you should not add additional calls to these functions, nor call the `main` function. In addition, if you have created your own global variables in `game.c`, you are encouraged to assign their initial value in `initialise_game`, and not in the global scope aether outside of any function, so that their value will be reset at the start of each game loop, and not only when the device is powered on.

Ship Setup – Human — Tier A: 8 marks

Before the game proper starts, the human player should be able to lay out their ships in positions of their choice. The board should initially contain no ships. Then, for each ship, going largest (carrier) to smallest (submarine), the human player should be able to move about a representation of the ship with the IO board buttons and/or WASD keys, as they would for the cursor in the game. If the ship is at the edge of the human board, attempting to move it towards that edge should do nothing. Pressing "r"/"R" should rotate the ship by 90 degrees. The positions of the ship before and after rotation should overlap by one pixel; you may use any implementation that follows this, and keeps the ship fully within the human board when rotated. Pressing "f"/"F" should place the ship, turning it orange, and move on to the next ship. While a ship is being moved, each segment should be coloured green if it does not overlap with a previously placed ship (the carrier will always be green, as no ships will be placed before it), or coloured red if it does overlap. You may choose if the entire ship being placed is coloured red, or only the overlapping segment(s); please note your choice on your feature summary. If the current ship does overlap, it should not be able to be placed. During ship setup, the terminal should state the name of the ship that is being placed.

If the user presses the "a"/"A" key from the splash screen (instead of "s"/"S" or a push button), then the ships should be placed in the default locations, and the game should start from the first turn, skipping the ship placement.

Cheating — Tier A: 4 marks

When the "c"/"C" key is pressed, the positions of the computer player's ships should show on the computer board for one second; this does not end the human player's turn. The undamaged segments should show as orange. The damaged segments should remain red or dark red, the missed shots should remain green, and the cursor should remain yellow. Sea locations that have not yet been fired at should remain black. After one second, the orange segments should turn black, while the other segments should remain as they are. While the ships are revealed, the game should continue running as normal without any disruption or lag; the player should still be able to move the cursor and fire shots, the cursor should continue to blink (and should alternate between yellow and orange while on a revealed ship), and so on. You may choose how to handle "c"/"C" being pressed while the ships are revealed (so long as the choice is reasonable e.g. crashing the game is not a valid choice). Note the choice you made for this on your feature summary.

Seven-Segment Timer — Tier A: 6 marks

*Requires **Human Turn** and **Computer Turn – Basic Moves** features to be implemented.*

On the human player's turn, they have a limited amount of time to make their shot. The length of time is selectable by pressing the number keys "1" through "5".

- Mode "1" – No time limit;
- Mode "2" – 60 second limit;
- Mode "3" – 30 second limit;
- Mode "4" – 15 second limit;
- Mode "5" – 5 second limit.

If the human player runs out of time, then their turn ends without them taking a shot (giving the computer player an extra shot). The mode should be selectable both from the splash screen and during the game. You may choose what happens if the user attempts to start a game without choosing a mode (e.g. use a default mode, or prevent the game from starting if a mode is not chosen). If the mode is changed mid turn, then the timer should reflect how much time the player has taken this turn. For example, if the player has spent ten second so far on their turn in mode "2", and they change to mode

"3", then the seven-segment display should change from showing "50" to showing "20". If they instead changed to mode "5", this should instead instantly end their turn. Changing from mode "1" will always give the player the full time allocation, even if they had previously been in another mode on the same turn.

Use the seven-segment display to show how much time the human player has remaining on their turn. For mode "1", turn off the display. For the other modes, show the time with both digits where there is at least ten seconds remaining, and show the time with one decimal place when there is less than ten seconds remaining; the value "9.9" should appear on the seven-segment display at the appropriate time.

You should ensure that the seven-segment display correctly displays at all times. This includes when the game ends, as well as for later features that may otherwise interfere with the display, such as the pause feature. The timer display should truncate the time remaining i.e. show the time remaining rounded down. Consequently, during the "11" -> "10" -> "9.9" transition, you must ensure that "10" is displayed on the timer for a full second. You may choose to slightly fudge the initial time for aesthetic reasons; if you do, please note this on your feature summary.

Game Pause — Tier B: 6 marks

*Requires **Human Turn** feature to be implemented.*

When the "p"/"P" key is pressed, the game should pause, and remain so until "p"/"P" is pressed again. While paused, all inputs, other than "p"/"P", should be ignored. In addition, the cursor should stop blinking while paused, but when unpaused, its progress through its flash cycle should continue. For example, if the game is paused 120ms after the cursor flashed on, then it should flash off 80ms after the game is unpaused. It should neither flash off immediately after unpausing, nor reset its cycle so that it flashes off 200ms after unpausing. Holding down the "p"/"P" key should result in the cursor apparently flashing at half its normal rate. If the **Seven-Segment Timer** feature above is implemented, then the timer must stop counting down when the game is paused, but must continue to display both digits of the remaining time. When the game is paused, a message indicating this should be displayed on the terminal. When the game is unpaused, remove this message. If the **Sound Effects** feature below is implemented, it must pause, as described in its section. All other aspects of the game should similarly pause. While the game proper is running (i.e. the part of the game where shots may be fired), the game must be able to be paused. You may choose to allow other parts of the game (splash screen, game over screen, ship placement if the **Ship Setup – Human** feature has been implemented, etc.) to be paused. If you do, please note this on your feature summary.

Computer Turn – Seach & Destroy — Tier B: 8 marks

*Requires **Computer Turn – Basic Moves** features to be implemented.*

This feature adds a second algorithm for the computer to use on their turn. On the splash screen, the player should be able to toggle between the two modes, by pressing the "y"/"Y" key. The terminal should display what algorithm the computer is set to at all times, but should only be able to be changed on the splash screen.

For this algorithm, the computer player has two modes; "search" and "destroy". If the computer player has hit any of the human player's ship segments, and not completely surrounded that segment (orthogonally) with additional hits, it will be in "destroy" mode. It will choose any segment that it is yet to fire at that is adjacent to one of the known human player's ship segments, and fire at that segment. If there are multiple such segments, it should choose one at random.

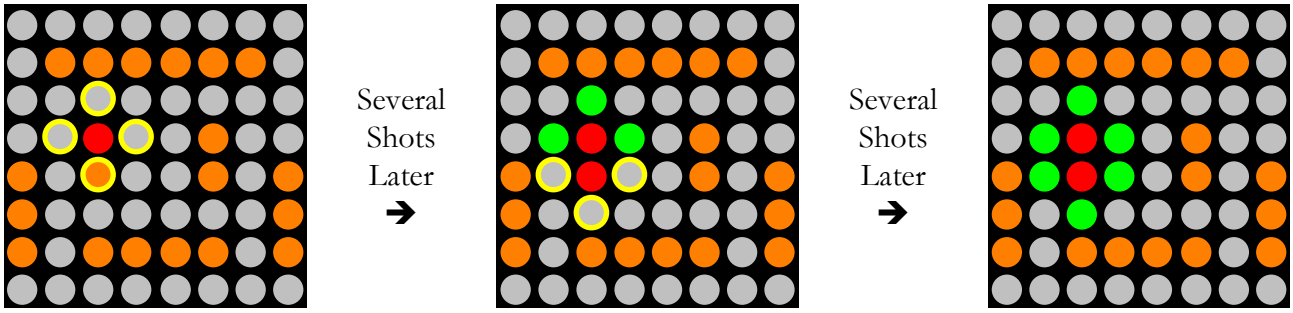


Figure 3: Example of Computer Turns in "destroy" mode.

Figure 3 shows an example of "destroy" mode, showing the human board of the LED matrix at three sequential points in time. In the left subfigure, the computer player has successfully hit a section of one of the human player's ships, as indicated by the red circle. The circles with the yellow borders show the surrounding locations that the computer player will fire at next. The middle subfigure shows the outcome of these shots. Because of the hit below the original shot, there are now more locations that the computer player will fire at, again indicated by the circles with the yellow borders. The right subfigure shows the outcome of these shots. As every location containing a section of one of the human player's ships has been surrounded, the computer would leave "destroy" mode at this point. Note that this is just one sequence of possible shots; if the computer player had first fired at the location below the original shot, then there would be six possible adjacent locations, and the computer could fire at these in any order.

If the conditions are not met for "destroy" mode, the computer player should instead enter into "search" mode; the first turn will always be in "search" mode. While in search mode, the computer player should fire at random locations it has not yet fired at. The order of locations should be different for each game.

A Note on Randomness: Computers are designed to be deterministic, and so cannot produce truly random numbers (without specialised hardware). As such, they instead produce pseudo-random numbers. This can be thought of as a large but finite ordered list of large numbers, with adjacent elements having a non-obvious relationship with one another. Whenever a random number is needed, the next element is returned (returning to the top when the last number is returned). The range of numbers needed is usually much smaller than the range of numbers in the list, and so the given number is reduced down to the required range. This results in many elements from the list matting to the same usable number, which obfuscates what exact element was returned, and thus makes it difficult to predict the next number, without getting a large number of samples. However, the list must start from a different element each time the program starts, otherwise it will return the same numbers in the same order each time. Ideally, the list would start from a random element, but if a truly random element could be chosen, then the list would not be needed in the first place. As such, something external must be used; this is called the seed. For inconsequential applications, it is common to use data from user inputs with enough precision that a human could not get a seed of their choice e.g. the time taken between powering up and the first user input, at millisecond precision. Once a seed has been chosen, any number of random numbers can be returned from the list without needing to reseed. Reseeding will not be beneficial nor detrimental to the quality of the random numbers, but will be unnecessary additional instructions. The function to seed a random number in C is `srand`, and it is common on many systems to use `time(NULL)` from `<time.h>` as the seed argument, however, this will not work on the Atmega324A. You will need to use another method to determine the current time. `rand` can be called to return a random number; this number will be between 0 and `RAND_MAX`, which will be at least 32767.

Ship Setup – Computer — Tier B: 6 marks

*Requires **Cheating** feature to be implemented.*

When the game starts, the computer player should place their ships in random positions and rotations. It is suggested that you have the ships placed in order from largest to smallest. See the note above about random numbers.

One method that can be used to implement this is, for each ship, enumerate each viable positions (i.e. assign a number to each position where the ship is fully on the board, and not overlapping another ship, for both horizontal and vertical rotations). Then, generate a random number up to the number of viable positions. Place the ship on the board in the position corresponding to that number.

If the user presses the "a"/"A" key from the splash screen (instead of "s"/"S" or a push button), then the ships should be placed in the default locations, and the game should start from the first turn, skipping the ship placement. If **Ship Setup – Human** is implemented, this means that both players will have either the original static setup, or a custom setup (placed for the human player, random for the computer player). Indicate which method of ship placement was used on the terminal during the game and on the game over screen.

Cheatin' 2 – Electric Boogaloo — Tier B: 4 marks

*Requires **Sinking Ships** feature to be implemented.*

If the human player presses "b"/"B", "n"/"N" or "m"/"M", then multiple shots should be fired in one move. Pressing "b"/"B" should fire on the location of the cursor, as well as the eight pixels surrounding it. If the cursor is located at an edge or corner, then only six or four pixels should be fired at, respectively. Pressing "n"/"N" should fire on the location of the cursor, as well as every pixel in the same row. Pressing "m"/"M" should fire on the location of the cursor, as well as every pixel in the same column. Using a cheat should constitute the human player's turn, passing the turn to the computer player. The human player should be able to use each of these cheats no more than once per game. If you have implemented the **Invalid Move** feature, you should treat an attempt to use the same cheat a second time as invalid, and print the "invalid move" message to the screen; this should not end the human player's turn. The human player may use these cheats even if they have previously fired at every location that would be targeted by the cheat, if so, they will complete their turn without having an effect on the game board, and be unable to use that cheat for the remainder of the game.

Salvo Mode — Tier B: 6 marks

*Requires **Sinking Ship** feature to be implemented.*

In salvo mode, each player fires a number of shots on their turn equal to the number of surviving ships in their fleet. All shots are fired before the player received information about whether their shots hit or missed a ship segment. If you have implemented **Computer Turn – Seach and Destroy**, you should have the computer assume that each shot missed to determine where the next shot in the same turn should be aimed. To prevent a massive first turn advantage, for the first six turns (three from each player), the active player ramps up the number of shots fired; one on the first turn from the human player, two on the second turn from the computer player, three on the third turn from the human player, and so on, until the sixth turn, where the computer player fires six shots (unless a player loses one or more ships in the first six turns, where they will be capped at the number of their remaining ships). LEDs L_0 through L_5 should show how many shots the active player has remaining during this turn, with one LED lit per shot. The lit LEDs should be adjacent, and L_0 should be the last LED to turn off. You should provide some indication on the LED matrix of which locations have already been fired upon this turn (but not yet resolved); you may choose any method that effectively communicates

this information. Note this method in your feature summery. A player should not be able to fire upon the same location twice in one turn.

While the splash screen is being displayed, the user can press "z"/"Z" to change between regular mode (one shot per turn) and salvo mode. The mode should be displayed at all times, but should only be able to be changed on the splash screen.

If you have implemented the **Seven-Segment Timer**, if a player runs out of time, their turn ends, but they still fire any shots they have already made. In addition, after making a selection, the player gets additional time, depending on the mode they have selected:

- Mode "1" (no time limit) – N/A;
- Mode "2" (60 second limit) – +30 seconds per shot;
- Mode "3" (30 second limit) – +15 seconds per shot;
- Mode "4" (15 second limit) – +5 seconds per shot;
- Mode "5" (5 second limit) – +1 seconds per shot.

The additional time cannot bring the player's total remaining time above what they had at the start of their turn. E.g. in mode "2", if the player makes their first shot after 15 seconds, without this requirement, their remaining time would increase 75 seconds. However, since the turn time in mode "2" is 60 seconds, they must instead be given 60 seconds after this shot.

If you have implemented the **Cheatin' 2 – Electric Boogaloo** feature, using a cheat should count as one of the human player's shots for their turn. They cannot use the same cheat again that turn (nor for the rest of the game), but they may use a different cheat.

Sound Effects — Tier B: 6 marks

*Requires **Game Over** feature to be implemented.*

Implement sound effects using the buzzer for each of the following events:

a) Human player hits computer player ship segment;
b) Human player sinks computer player ship;
c) Human player wins;
d) Computer player hits human player ship segment;
e) Computer player sinks human player ship;
f) Computer player wins.

When event b) or e) occurs, you should not play the sound effects for a) or d). When event c) or f) occurs, you should not play the sound effects for a), b), d), or e). The sound effects for b), c), e) and f) should be a series of tones, not just a single note. The sound effects for a) and d) may be either a single note or a series of tones. You should make the sound effects thematically related, such that the sound effect for b) is a "fancier" (or equivalent) version of the sound effect for a), and similarly c) for b), e) for d), and f) for e). In addition, the sound effect for d) should be a "corrupted" (or equivalent) version of the sound effect for a), and similarly e) for b) and f) for c). The sound effects should be proper tones, with a noticeable duration, and not a single click for each note.

While the sound effects are playing, the game should remain playable; it should not freeze until the sound effects have completed. If the "q"/"Q" key is pressed, the sound effects should be silenced, until the key is pressed again. Starting a new game should not reenable the sound effects. When the game is paused, the sound effect should also pause, and resume from where it left off when unpaused.

On your feature summary form, indicate how the sound effects are thematically related.

Firing Animation — Tier C: 6 marks

*Requires **Human Turn** and **Computer Turn – Basic Moves** features to be implemented.*

When the computer player fires a shot, play a short animation that conveys what location they are targeting. If you have implemented Salvo Mode, each shot should occur sequentially. Ensure that the animations are not long enough that waiting for every shot gets frustrating. You may overlap the animations by a small amount if you wish. Describe your animation in your feature summary. During the animation, the user should be able to move the cursor, but should not be able to fire a shot until the animation is complete. For other inputs, you should allow ones that do not progress the game state while the animation is running, but disallow and ignore any inputs that would. If the **Seven-Segment Timer** feature is implemented, the timer should not decrement while the animation is running.

Joystick — Tier C: 6 marks

*Requires **Move Cursor with Push Buttons** feature to be implemented.*

Use the joystick to move the cursor. You may choose an arbitrary orientation for the joystick, but the directions must be rotationally consistent. Positive x must go in the opposite direction of negative x, and positive y must go in the opposite direct of negative y. In addition, the direction of positive x must be $90°$ clockwise of the direction of positive y. You must implement a delay between each movement of the cursor. This delay should be shorter (i.e. the cursor should move faster) the further the joystick is tilted. However, even at the shortest delay, the cursor must be reasonably controllable, and the user should be able to move the cursor to any given location with reasonable accuracy. Because of variations in joysticks, different joysticks will read different values when in the upright position. You must account for this, and ensure that when the joystick is upright, it does not move the cursor, even if different joysticks are used. If the joystick is tilted diagonally, the cursor should move diagonally, and in one step; the cursor should not alternate horizontal and vertical movement when doing so. The cursor should be able to wrap around the edges as it does for the buttons and keys, including for diagonal movement.

Computer Turn – Smart Targeting — Tier C: 6 marks

*Requires **Computer Turn – Search & Destroy** feature to be implemented.*

This is another method that the computer can use to determine where they fire at on their turn. Pressing "y"/"Y" on the splash screen should rotate between the three methods. This method is similar to **Search & Destroy**, but with some extra logic in each mode to improve the computer player's performance.

If the conditions are met for "destroy" mode, then the computer should act similarly to how it would for the **Search & Destroy** method. However, when determining which location of an unsurrounded ship segment to fire at, it should first choose one with the greatest number of combined consecutive hit ship segments in a straight line from the potential target. If there are multiple preferred targets, the one that is closest to the centre of the grid should be chosen; you may choose what metric to use for this (suggested either Euclidean L^2, Manhattan L^1 or Chebyshev L^∞ distances) – please note on your feature summary which metric you use. If there are still multiple preferred targets, you may decide how the computer player determines which one is chosen.

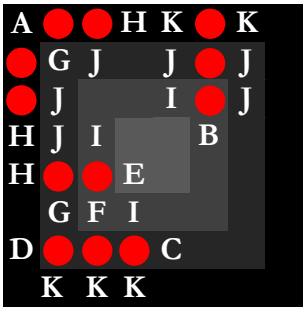
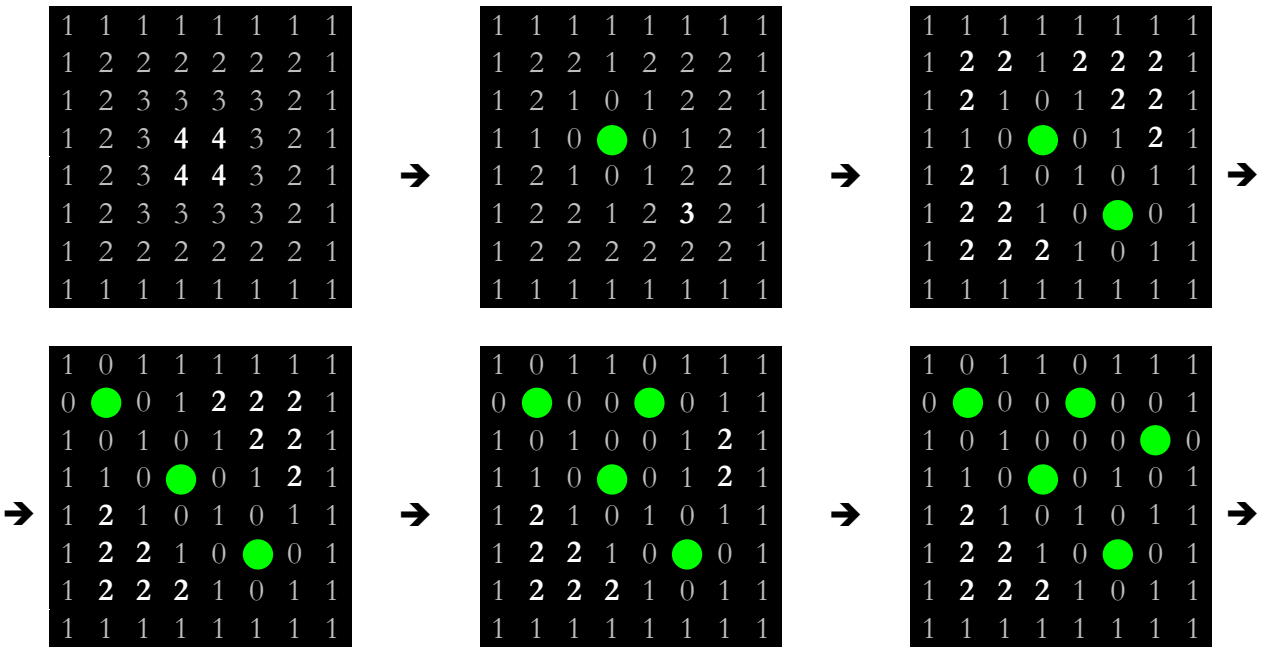


Figure 4: Example of Smart Targeting targets in "Destroy" mode.

Figure 4 shows an example of which locations would be targeted while in "Destroy" mode. Chebyshev distances are used for this example, and the grid is shaded for demonstration purposes so that the central locations are lightened. The computer player would prefer them in order `A` through `K`. If location `A` were unhit, it would be targeted. If it were, then `B` would be targeted instead if it were unhit. If it were (and was a miss), then `C` would be targeted instead, and so on. Location `A` is adjacent to four hit locations in a straight line, even though they are split between two ships. As such, it is the most preferred target. Locations `B`, `C` and `D` are all adjacent to three hit locations, and so are next, and ordered from the location closest to the centre outwards. Locations `E`, `F`, `G` and `H` are all adjacent to two hit locations, and so are next, and also ordered from the centre outwards. Note that there are multiple locations `G` and `H`; you may have the computer player target these in any order you wish (so long as each `G` is targeted before any `H`). Locations `I`, `J` and `K` are all adjacent to only one hit location, and so are last, again from centre outwards; there are multiples of each of these.

If the conditions are <u>not</u> met for "destroy" mode, the computer player should instead enter into "search" mode. The computer player should calculate how far each location they are yet to fire at is from each location adjacent to one they have fired at, and/or the edge, using the Manhattan distance (i.e. number of orthogonal moves). Then, it should fire at the location that with the greatest distance; you may decide how tied maximum distances are handled. However, once all locations have a Manhattan distance of zero, the first tiebreaker must be the number of adjacent unfired locations. If this location contains a section of one of the human player's ships, then the computer player will (likely) enter into "destroy" mode next turn. Otherwise, it will remain in "search" mode, but perform a new calculation for the distances.



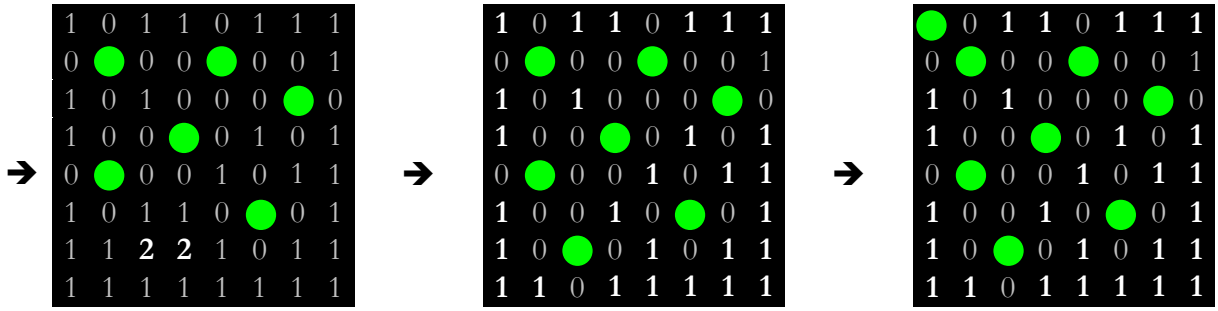
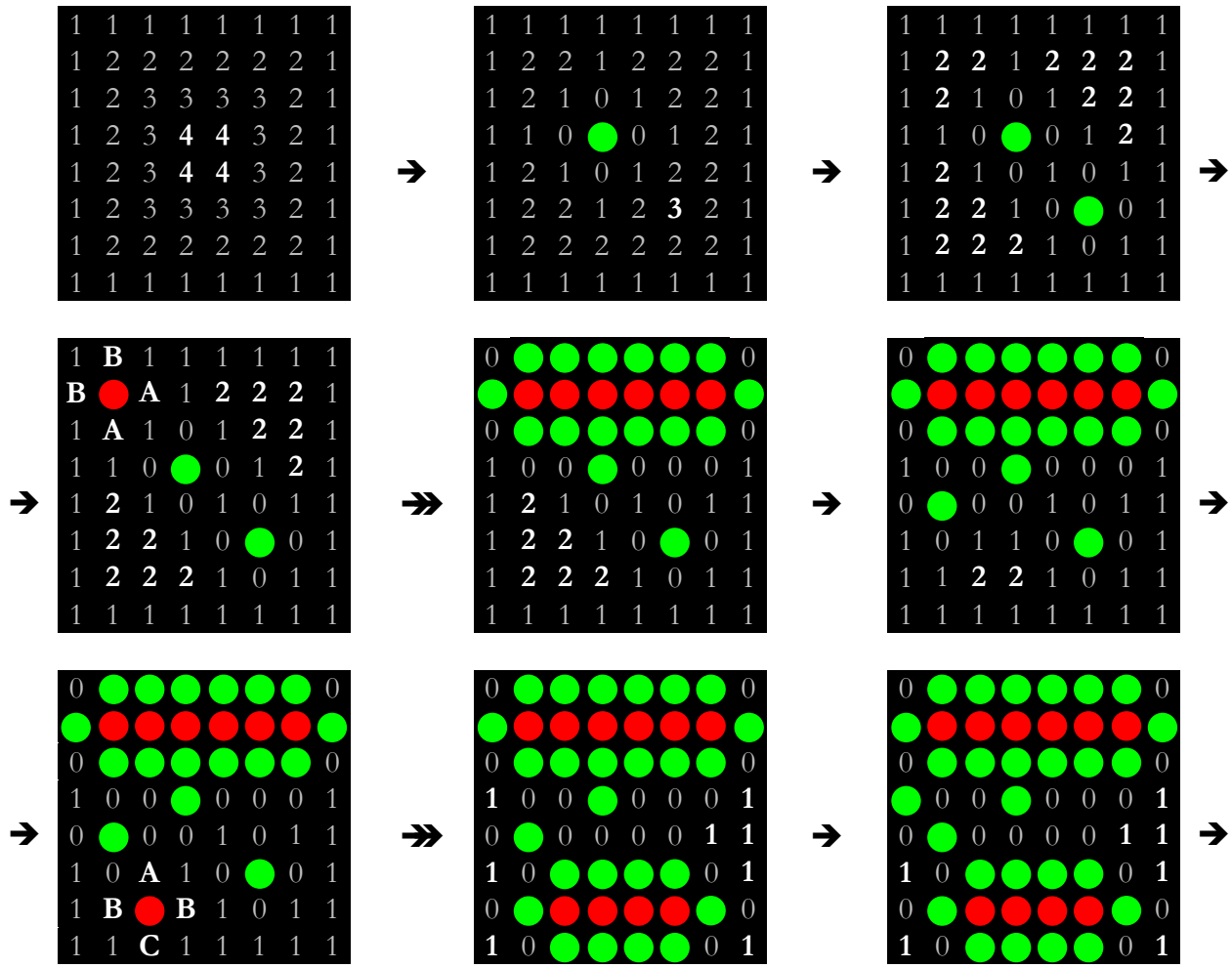


Figure 5: Example of Smart Targeting in "search" mode.

Figure 5 shows an example of the computer in "search" mode over nine turns. The numbers show the Manhattan distances from locations adjacent to previous shots and/or the board edge, with the maximums bolded. In the first subfigure, any of the fours could have been chosen; the tie breaking for this example is topmost, then leftmost. You may choose a different tiebreaker. After this first shot is taken, this updates all the Manhattan distances; notably, everything orthogonally adjacent to this shot now has a Manhattan distance of one, and the maximum is now three. After another shot, the maximum is two, and after a few more, it is one. Note that if any shot hit a segment of one of the human player's ships, then the computer would have gone into "destroy" mode, which could have radically changed the Manhattan distances once the computer player returned to "search" mode.

NB: The two "modes" are named such for descriptive reasons; you do <u>not</u> need to remember the modes between turns, and may instead determine which mode to use at the start of each turn.



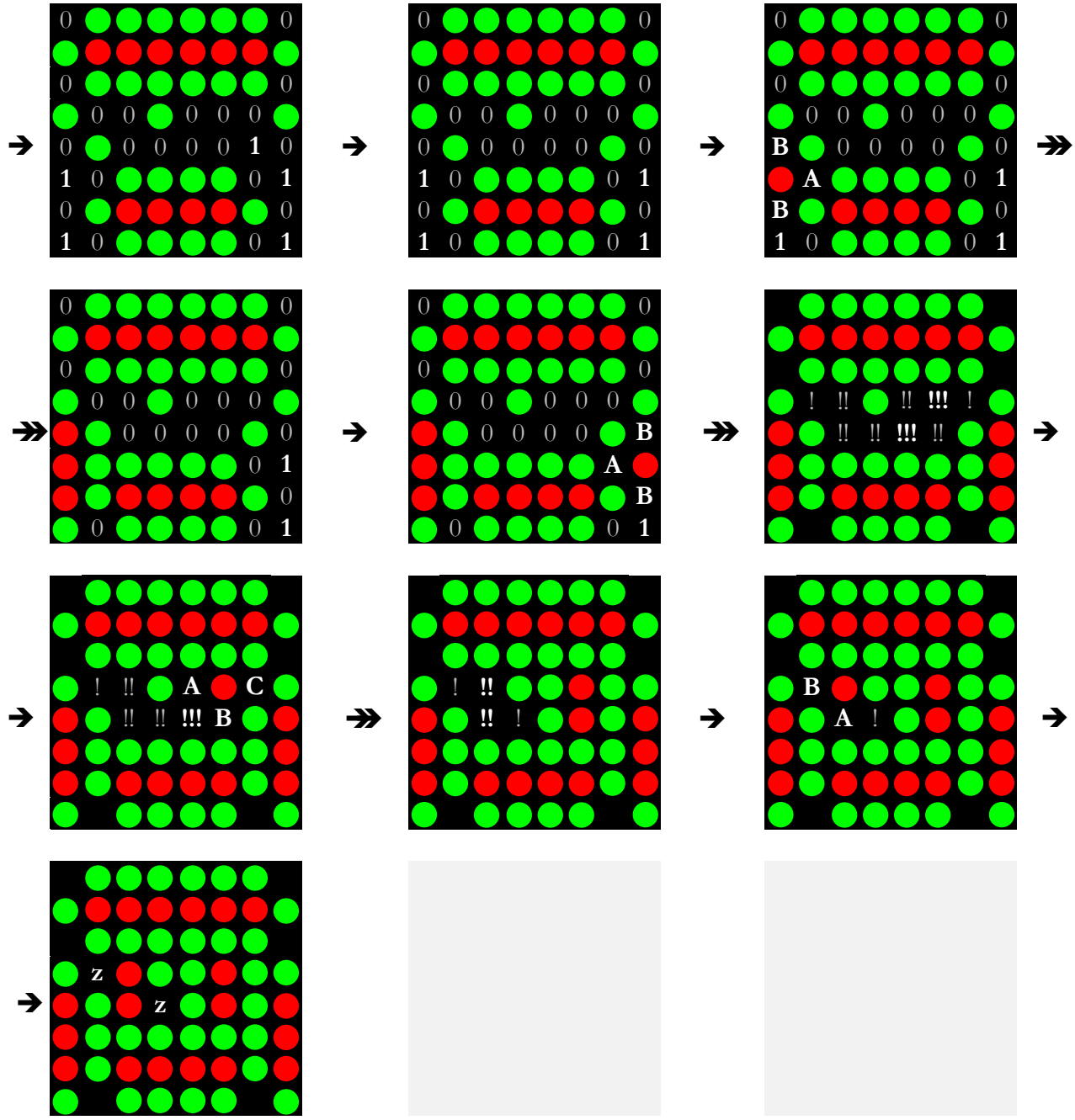


Figure 6: Example of Smart Targeting complete game.

Figure 6 shows a mixture of "search" and "destroy modes. Once one segment of a ship has been found, then for compactness, several turns are skipped, and the resulting grid is shown at the point the computer player returns to "search" mode. Once all the Manhattan distances are zero, the values are replaced with a number of exclamation marks showing how many adjacent locations are unfired upon (with blanks being zero locations; since the minimum length of a ship is two, there cannot be a ship segment in these locations). If the **Game Over** feature (above) has been implemented, the game will end after the last subfigure; the locations marked with a z will <u>not</u> be fired upon before the game ends.

High Score — Tier C: 8 marks

*Requires **Game Over**, **Ship Setup – Computer**, **Salvo Mode** and **Computer Turn – Smart Targeting** features must be implemented for full marks.*

NB: This feature goes beyond the content that is taught in the labs. You will be expected to gain the necessary knowledge to complete this feature yourself by consulting the datasheet and relevant references online. Staff will only provide limited assistance.

When a game is complete, if the human player won, calculate a final score. For each of the human ships, square the number of undamaged locations on that ship; sum these values together to calculate the ship score. E.g. if the carrier is undamaged, it will contribute a value of $6^2 = 36$; if half the segments are damaged (regardless of which they are), it would contribute a value of $3^2 = 9$, the same as an undamaged destroyer/frigate. If any ship is sunk, it would contribute a value of 0. The maximum ship score is 78 for a completely undamaged fleet. Then, find the number of locations on the computer grid that the human player did not fire at; this is the accuracy score. Since the human player has won, none of these locations can be ship locations. Since the grid is $8 \times 8 = 64$, and there are a total of $6 + 4 + 3 + 3 + 2 + 2 = 20$ ship locations, the maximum accuracy score is $64 - 20 = 44$. To produce the final score, multiple the ship score and the accuracy score together; the maximum is $78 \times 44 = 3432$. You may also wish to display this score on the terminal during the game.

Use the EEPROM to store these high scores, along with a name of the player that achieved that score. Prepopulate the leaderboard with some made-up scores from some made-up players (these scores should be beatable, and cover a large range of scores).

At the end of every game, display the high score table. If a player beats a score on the scoreboard, display the scoreboard with their score in the appropriate position, and a blank for the name. The player can then type in their name, pressing enter to submit, and the new name and score should be displayed in a different colour. If a player submits a blank name, do not store their score. Otherwise, the lowest score should be removed, and at the end of subsequent games, the high score table should be displayed with the new entry. The new high score should be remembered even if the AVR becomes unpowered. Pressing "s"/"S" while the high score table is displayed (and not asking for player name entry) should return to the splash screen.

Ten scores and names should be stored for each combination of computer player firing method (3), salvo mode on/off (2), and static/random computer ship starting positions (2), for a total of twelve independent leaderboards. Each player name should be up to seven characters, and consist of any printable ASCII characters (you do not need to consider tab characters).

If the player enters the sequence "r"➔"R"➔"r" while the high score table is being displayed (and not asking for player name entry), the EEPROM should be reset to the pre-populated names and scores. You, and the marker, will generally have to do this initially after programming your code on a new device.

While on the splash screen, if "h"/"H" is pressed, the high score table for the chosen game settings should be displayed, as though a game just ended.

Terminal Summary

If all features are implemented, then during the splash screen, the terminal should display:

- The "Battleship" ASCII art title – **Splash Screen**;
- Your name and student number – **Splash Screen**;
- The selected game speed – **Seven-Segment Timer**;
- The computer turn method – **Computer Turn – Seach & Destroy** and/or **Computer Turn – Smart Targeting**;
- The computer ship placement method – **Ship Setup – Computer**; and
- Normal or Salvo mode – **Salvo Mode**.

While the human player is placing their ships, the terminal should display:

- The name of the ship being placed – **Ship Setup – Human**;

- The selected game speed – **Seven-Segment Timer**;
- The computer turn method – **Computer Turn – Seach & Destroy** and/or **Computer Turn – Smart Targeting**;
- The computer ship placement method – **Ship Setup – Computer**; and
- Normal or Salvo mode – **Salvo Mode**.

During the game, the terminal should display:

- A message when an invalid move is attempted – **Invalid Move**;
- Two lists of sunken ships – **Sinking Ships**;
- The selected game speed – **Seven-Segment Timer**;
- An indicator of if the game is paused – **Game Pause**;
- The computer turn method – **Computer Turn – Seach & Destroy** and/or **Computer Turn – Smart Targeting**;
- The computer ship placement method – **Ship Setup – Computer**; and
- Normal or Salvo mode – **Salvo Mode**.

During the end screen, the terminal should display:

- Which player won – **Game Over**;
- The selected game speed – **Seven-Segment Timer**;
- The computer turn method – **Computer Turn – Seach & Destroy** and/or **Computer Turn – Smart Targeting**;
- The computer ship placement method – **Ship Setup – Computer**; and
- Normal or Salvo mode – **Salvo Mode**;

You may find this summary useful when planning out your terminal layout. You may assume that the terminal will be at least as wide as the ASCII art title that displays on the splash screen in the provided code.

All information displayed on the terminal should be identifiable as to what it represents. For example, the game speed should have text to identify it, and not just be raw numbers, which could cause them to be confused with other information.

You must not spam the terminal with serial communication; you should only send a communication when information changes, and you should only update the relevant information, instead of clearing the entire screen and rewriting all the information. When the IO board is used for serial communication, the RX and TX lights will light up when data is being transferred. If the RX light is constantly lit, then you are constantly sending information, and you will not be awarded full marks. If the RX light turns on for a significant amount of time when you send data, then you may be sending more data that necessary, and if so, you will not be awarded full marks. Either of these issues can also cause the terminal screen to stutter.

Assessment of Feature Implementation

The program improvements will be worth the number of marks shown above. You will be awarded marks for each feature up to the maximum mark for that feature. Part marks will be awarded for a feature if only some part of the feature has been implemented or if there are bugs/problems with your implementation (which may include issues such as incorrect data direction registers). Your additions to the game must not negatively impact the playability or visual appearance of the game. Note also that the features you implement must appropriately work together, for example, if you implement game pausing then audio should pause.

Features are shown grouped in their tiers of approximate difficulty (Tier A, Tier B, and Tier C).

Submission Details

The due date for the AVR project is 4:00pm Friday 24h May. The AVR project must be submitted via Blackboard. You must electronically submit a single `.zip` file containing only the following:

- All of the C source files (`.c` and `.h`) necessary to build the project (including any that were provided to you – even if you have not changed them);
- Your final `.hex` file (suitable for downloading to the ATmega324A AVR microcontroller program memory, see the getting started video to locate the `.hex` file); and
- A `.pdf` feature summary form (see below).

Please name your `.hex` and `.pdf` in the form `proj_4nnnnnnn.hex/.pdf`, where "`4nnnnnnn`" is your eight-digit student number.

Do not submit `.rar` or other archive formats – the single file you submit must be a `.zip` format file. All files must be at the top level within the `.zip` file – do not use folders/directories or other `.zip`/`.rar` files inside the `.zip` file. If only the `.hex` file is submitted with no source files, then your submission will not be marked. The `.hex` file will be used if the marker is unable to compile/build the submitted source files.

If you make more than one submission, each submission must be complete – the single `.zip` file must contain the feature summary form, the `.hex` file and all source files needed to build your work. We will only mark your last submission and we will consider your submission time (for late penalty purposes) to be the time of submission of your last submission.

It is the responsibility of the student to ensure that their submission is correctly uploaded to the Blackboard submission portal with all of the files they intend to submit. This can be ensured by downloading your `.zip` file after submission is made, un-zipping the submission to check all files are correctly included and checking whether you are able to compile the project successfully. It is suggested that you use the Axon lab computers for this check.

The feature summary forms are on the last pages of this document. A separate electronically fillable `.pdf` form will be provided on Blackboard. Do not submit this entire document. This form can be used to specify which features you have implemented and how to connect the ATmega324A to other devices so that your work can be marked. If you have not specified that you have implemented a particular feature, we will not test for it. Failure to submit the feature summary with your files may mean some of your features are missed during marking (and we will not remark your submission). You can electronically complete this form, or you can print, complete and scan the form. Whichever method you choose, you must submit the `.pdf` file with your other files. If you have any assumptions or comments to convey to the marker, include these on the feature summary form.

Incomplete or Invalid Code

If your submission is missing files (e.g., will not compile and/or link due to missing files) then we will substitute the original files as provided to you. No penalty will apply for this, but no changes you made to the missing files will be considered in marking.

If your submission does not compile and/or link in Microchip Studio version 7 for other reasons, then the marker will make reasonable attempts to get your code to compile and link by fixing a small number of simple syntax errors and/or commenting out code which does not compile. A penalty of between 10% and 50% of your mark will apply depending on the number of corrections required. This will apply based off of compiling and linking with Microchip Studio on the Axon lab computers, regardless

of the results of compiling or linking with other programs. If it is not possible for the marker to get your submission to compile and/or link by these methods, then you will receive 0 for the AVR project (and will have to resubmit if you wish to have a chance of passing the course). A minimum 10% penalty will apply, even if only one character needs to be fixed.

Compilation Warnings

If there are compilation warnings when building your code, then a mark deduction will apply – 1 mark penalty per warning up to a maximum of 10 marks. The warning list generated by Microchip Studio on the Axon lab computers from a clean build will be the canonical list for these penalties. To check for warnings, rebuild all your source code (choose "Rebuild Solution" from the "Build" menu in Microchip Studio) and check for warnings in the "Error List" tab. If you are using other software for your project, you are encouraged to perform a final build on Microchip Studio on the lab computers.

General Deductions

If there are problems in your submitted AVR project that do not fit into any of the above feature categories, then some marks may be deducted for these problems. This could include problems such as general lag, errors introduced to the original program etc. The number of marks deducted will depend on the severity of the issues.

Late Submissions

As per the course profile, where an assessment item is submitted after the deadline (i.e., 4:00pm Friday 24th May), without an approved extension, a late penalty will apply. Assessment submissions received after the due time (or any approved extended deadline) will be subject to a 10% late penalty per day (or part thereof), up to a maximum of seven days. After seven days, a 100% late penalty will be applied.

Requests for extensions should be made via the process described in the course profile (before the due date) and be accompanied by documentary evidence of extenuating circumstances (e.g., medical certificate). The application of any late penalty will be based on your latest submission time.

Notification of Results

Students will be notified of their results via the My Grades section of Blackboard.

CSSE7201

The AVR programming described in this document constitutes part two of the CSSE7201 assessment, which contains the 10% pass hurdle for the course. Please see Blackboard for part one, and/or the course profile for more information regarding the assessment details.

Version

This is version thirteen of the project document. Changes between the current version and the previous version are shown in green. Changes between the previous version and version one are shown in blue.

The University of Queensland – School of Electrical Engineering and Computer Science
Semester One, 2024 – CSSE2010/CSSE7201 Project – Feature Summary

An electronic version of this form will be provided. You must complete the form and include it (as a `.pdf`) in your submission. You must specify which IO devices you have used and how they are connected to your ATmega324A. Failure to include this form with your submission will result in no marks being awarded for the project. Failure to specify connections and/or attempted features will result in no marks being awarded for the relevant features.

Student Number								Family Name	Given Names
4									

Port	Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	Pin 1	Pin 0
A								
B	SPI connection to LED matrix				Button B3	Button B2	Button B1	Button B0
C								
D							Serial RX	Serial TX
							Baud rate: 19200	

Feature	✓ if attempted	Comment (Anything you want the marker to consider or know?)	Mark	
Splash Screen			/4	
Move Cursor with Push Buttons			/6	
Move Cursor with Terminal Input			/6	
Human Turn			/8	
Computer Turn			/6	
Invalid Move			/4	
Sinking Ships			/4	
Game Over			/6	
Ship Setup – Human			/8	
Cheating			/4	
Seven-Segment Timer			/6	/50
Game Pause			/6	
Computer Turn – Search & Destroy			/8	
Ship Setup – Computer			/6	
Cheatin' 2 – Electric Boogaloo			/4	
Salvo Mode			/6	
Sound Effects			/6	/30
Firing Animation			/6	
Joystick			/6	
Computer Turn – Smart Targeting			/6	
High Score			/8	/20
Total: (out of 100)				
General deductions: (errors in the program that do <u>not</u> fall into any above category, e.g., general lag in gameplay)				
Penalties: (code compilation, incorrect submission files, etc. does <u>not</u> include late penalty)				
Final Mark: (excluding any late penalty which will be calculated separately)				