Ense 352 – Fall 2022 – Term Project

Revision 1.1: 2022-12-04

Handed Out: 2022-11-03

Due: Mon 2022-12-05 by 23:55

Description

The ense352 term project this year is a "Battleships" game, to be written *in C or C++*. Your target is the nucleo64-f103rb board.

To get an idea of the steps involved in playing the game of Battleships, take a look at this video. It's brief; under 4 minutes. In addition the Wikipedia article is of general interest.

Your working project (software and hardware) will serve as a single **game unit** as described in the video. It's important at this point to emphasize that the Battleships game can be played by two people using only pencil and paper. No fancy molded plastic components and certainly no computers required.

Because our hardware is extremely limited, with respect to the input and output capabilities, our game unit will be quite primitive. It will, at least conceptually, have the following components:

- a **layout (bottom) grid** where the player places his or her ships
- a **targeting (top) grid** where the player records guesses, or "shots".
- During your turn you take a shot by calling out a grid coordinate. Your opponent responds with "hit" or "miss" and you record the location, and the hit/miss result in your **targeting grid**. If it was a hit, the opponent also records your shot on their **layout grid**.

In this implementation, some of these "components" will be physical I/O devices, some will be paper & pencil, and some software-based. The player must enter the information into the game unit via some input device. The game unit maintains a partial model of the game state, and presents information about the game state through some output device. What input/output devices, you ask?

Hardware

Of course, you're familiar by this time with the hardware platform. It has precisely

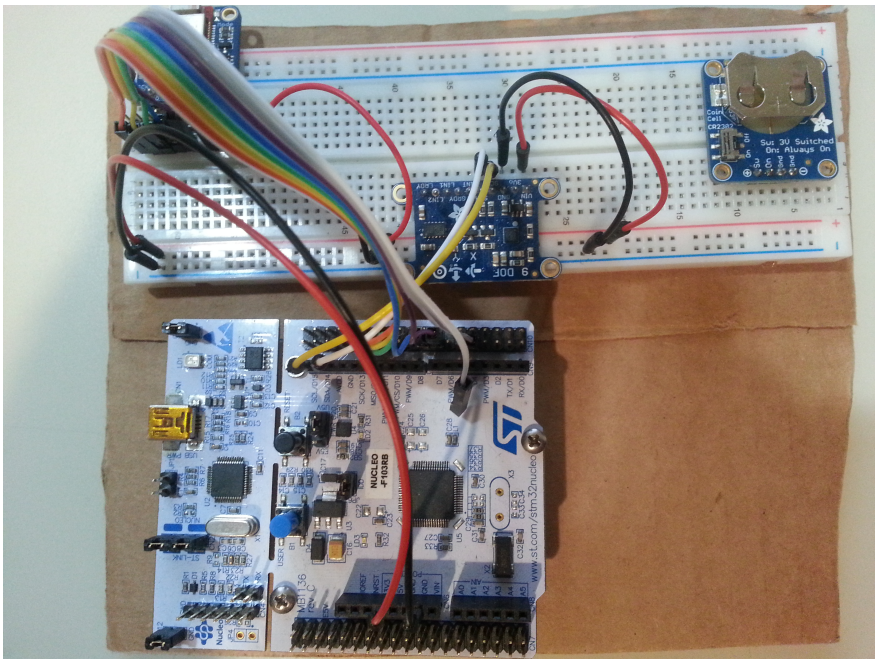
- one push-button switch
- one green LED

I can't really envision how to encode all game interactions with the above devices. We'll need to add some hardware, minimally some LEDs and switches. And that hardware is perhaps best just placed on a solderless breadboard.

Ideally, you would use a keyboard and monitor, but it's not really feasible to ask you to do something that complex. At this point it's hard enough just reading switches and blinking LEDs.

The parts kit you've obtained in enel384 should have enough parts to get you going.

I *highly* recommend you arrange your project as shown in the photo.



Here f103rb board is attached to a substrate (I used cardboard, and three metal standoffs to attach the board) as is the solderless breadboard (I used glue). With this setup you can construct your circuit on the breadboard and be reasonably sure of not dislodging your connections with every trivial bump. *Just ignore the specific devices on the breadboard. They are a BLE tranceiver, an IMU, and battery pack used in another class.* This setup has the virtue of keeping the f103rb and the breadboard together, which will save you *hours* of time debugging pulled connections. I have nylon standoffs you can use for mounting your f103rb. Each board requires three of these.

Interactions

To get started with the problem, think about the kinds of interactions we want to capture. Let's describe a scenario starring **you** and your **opponent**. A game consists of running through the following steps until one player's entire fleet is sunk:

1. you issue a grid coordinate (for a 10x10 grid, that's a pair of numbers between 1 and 10, first the row, then the column)

2. opponent responds with: "miss","hit", or "hit and sunk". How do they know this? They need to know where their ships are placed. They could perhaps get that information from their "layout grid" in which they simply record their own ship's positions on a piece of paper. Just like it's 1917 again.

   miss if it was a miss then the opponent has nothing else to do. You mark the shot as a miss on your "targeting grid". This is the equivalent of you placing a white peg in your shot's coordinate, on your targeting grid.

hit if it was a hit, both you and your opponent must do the equivalent of marking that coordinate with a "red peg". You put the red peg in your targeting grid, and your opponent places the red peg in their layout grid, on the part of the vessel that occupies that grid cell.

hit and sunk in this case both you and your opponent perform the same actions as in the "hit" case, with the additional step of recording that the vessel was sunk.

3. Now it's your opponent's turn: they do what you did in step 1.
4. You now respond with "miss", "hit", or "hit and sunk", just as your opponent did in step 2. And the play loops back to step 1.

You can elaborate all possible interactions in a series of use cases, refine them until they seem to cover all possible scenarios, then use that as the basis of your software design. This can be done *regardless* of the actual I/O mechanisms.

In fact, the entire game logic can be developed on your personal computer, then ported to the f103rb. Only the I/O software layer would differ. This idea of off-target development leads to the practice of using a layered architecture, probably the most ubiquitous software design pattern.

Hints

- Develop use cases that are hardware agnostic, to the extent possible. This can help with separating design from implementation details. You can get more information about use cases from wikipedia.
- We recommend the initial game configuration be controlled by compile-time parameters (as opposed to run-time inputs). This means you'd need to recompile the software before every game.

Ubquitous Language

Most software projects come up with a series of important terms (or, at least well-run, successful projects do this) early on, which allows all the interested parties to communicate accurately and succinctly. It's basically a glossary. Here's my attempt at our glossary:

game unit The hardware and software assembly that you, the developer, design and build, and use to play against other students.

attacking player The player whose turn it currently is. That is, the one who fires the shot. (also **attacker**)

responding player The player who's turn it is *not*. The one who is being shot at. (also **opponent** when the attacker is being discussed in a sentence)

player A general term for either an attacker or an opponent.

layout grid The grid on which each **player** places their ships and on which they record hits from their opponent.

targeting grid The grid on which each player records each shot, and the result of the shot (hit or miss).

Requirements

I'll present these in bullet form to avoid imposing an overly strict ordering of importance. In the following requirements, **shall** indicates a mandatory requirement, **should** indicates a desirable requirement.[1]

User-facing Requirements

- The software **shall** allow game play between two people, each having their own game unit, with interactions similar to those specified above in the "Interactions" section.
- The software **shall** use a 10×10 grid by default, to allow easy interfacing to other player's game units.
- The game unit **shall** emit distinct signals to indicate
  - waiting to receive, from the attacker, row and column input for a shot
    - verified, or failed to verify, to the attacker, the row and column input
  - the game unit **shall** prompt for all inputs from the player, and **shall** provide success or failure feedback for all inputs
  - the game unit **shall** indicate when a ship is sunk, and when the game is lost

Developer Requirements

- The game unit **shall** be responsible for
  - recording the player's ship positions, the grid size, and other game parameters (specified at compile time)
  - recording the opponents hits in the layout grid
  - announcing the result of each shot: miss, hit!, sunk, or all-sunk.
- Other game state is the responsibility of the humans:
  - recording the player's misses and hits in the targeting grid
  - announcing when the player has lost

Deliverables

1. A live demo of the working system. This would include playing a Battleships game with another person.
2. Working software submitted
3. Document the hardware, so that another person could implement what you've done.

[1] NOTE: We reserve the right to make changes to these requirements if, for example, there are grotesque errors discovered, or a better idea emerges. We will strive mightily so that these putative changes (1) happen early, and (2) make your job easier, rather than harder. We may, if needs be, place this document on github, convert it to markdown, and solicit pull-requests.

4. An ASCII readme file describing
   (a) what the game is
   (b) how to play
   (c) a table describing all the output signals, and what they mean
   (d) information about problems encountered, any basic features you failed to implement, extra features implemented
   (e) any configuration options

The final due date for all deliverables is Mon 5 Dec 2022 by 23:55. The demo portion will be done in the lab.

Evaluation

The base requirements, if fully realized, will result in a mark of 90%. The following optional extras, if fully realized, will add at most the indicated bonus, while keeping the mark to a maximum of 100%.

- **7%**: Demonstrate portable implementation: your SW compiles and runs on a desktop computer *or* the f103rb simply by auto-determining *at compile time* which target you're building for. The code base should have a *minimum* of platform-specific code. The rest of the code should be unaware of the change.
- **5%**: Use the 7 segment display from your enel384 parts kit in an interesting and useful way, and properly document it.