UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

**Lab 9 – Battleboats**
**Commit ID Form: https://goo.gl/forms/aoPKz2VXhSgVsYzN2**
**28 Points**

**Warning**

This lab is the hardest one in the class, and will take you much more time. This lab involves two state machines that are more complex than you have programmed before, and a new library to display graphics on the OLED. The difficulty is somewhat offset by having a partner; however you need to coordinate very well with them. You will need to start early and read this lab very carefully. **Failure to plan is planning to fail**.

## Introduction

For this lab you will be working with a partner to develop the libraries and agents necessary for a two-player game of BattleBoats, which resembles Hasbro's Battleship™ game. There are two supporting libraries along with two playing agents, one with artificial intelligence and one that is an interface for the human player. You and your partner may split up the work such that each does one of the two libraries and then you will both work together to implement the Agent. For extra credit you may implement a human agent.

## Concepts

- `const` variables
- Hashing
- Checksums
- Encryption
- Finite state machines
- `rand()`
- Teamwork

## Reading

- **CKO** – Chapter 5
- Wikipedia Article on BattleShip Game

**Provided files**

- **Agent.h** – This file describes the Agent interface for creating game-playing agents that interact with the BattleBoats game. Each agent must implement the three functions described within the Agent.h file. The provided HumanAgent.o implements the API described within, and so will your ArtificialAgent.c.
- **AgentTest.c** – provides a `main()` that is used for unit testing the Agent (human or Artificial) functions that you have coded. Just like in previous labs, you will need to code your own test harness, and exclude it from the project when done testing.
- **BOARD.c/h** - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values.
- **Protocol.h** – This file describes the communication protocol that will be implemented in the corresponding Protocol.c file. All communication with the other agent will utilize the functions within this library.
- **ProtocolTest.c** – provides a `main()` that is used for unit testing the protocol functions that you have coded. Just like in previous labs, you will need to code your own test harness, and exclude it from the project when done testing.
- **Field.h** – This file describes a library that implements many helper functions for managing a BattleBoats field. You or your partner will implement the described functions in a Field.c file. Also, the only way that the data within the `Field` structs should be accessed is via the functions within Field.c; the field member should never be accessed directly.
- **FieldTest.c** – provides a `main()` that is used for unit testing the Agent (human or Artificial) functions that you have coded. Just like in previous labs, you will need to code your own test harness, and exclude it from the project when done testing.
- **Tester.c** - This file is a bare file including `main()` that you can use to test your code with. It also calls `srand()` for you so that `rand()` is useable. You will need to press BTN4 before your code will run, as that's the last step in initializing the PRNG. This file is for debugging only and as such will NOT be graded for this assignment!
- **BattleBoats.c** – This is a source file that includes `main()` along with some other code. It also initializes the pseudo-random number generator using `srand()`. It should be included within an MPLAB X as normal. This function will not compile unless you have implemented all of the libraries and included an agent, either your HumanAgent.c, ArtificialAgent.c, or the provided HumanAgent.o file. **This file should not be modified at all!! If you need to test stuff use the provided tester.c**

- **HumanAgent.o** – This is a precompiled object file. This contains a human agent that can be used to play against your artificial agent.
- **Lab9SupportLib.a [Oled.h, OledDriver.h, Ascii.h, FieldOled.h, Buttons.h]** - Provides pre-compiled code for working with the OLED and the Buttons. FieldOled.h is a new file that is especially important as it provides a function that will draw the entire screen based on two field struct inputs. When adding this to your MPLAB X project, make sure it's the last file listed in the Libraries folder, otherwise you may get "undefined reference" errors when compiling with BattleBoats.c or HumanAgent.o.

**Assignment requirements**

- **Working with a partner:** Unless otherwise approved by the instructor, you will be working with a partner on implementing this lab. The libraries should be split between you and your partner, with one implementing Protocol and the other Field.[1] Once both libraries are completed and tested, then you should work together on HumanAgent.
- **Gameplay:** Gameplay will depend on whether the Uno32 board is compiled with the ArtificialAgent code or the HumanAgent code. See their respective sections for further details. Most of the gameplay mechanisms are already handled for you within the BattleBoats.c file. The only gameplay mechanics you need to handle are initializing the field, and generating guesses following the FSM specified for the Agents. Additionally, HumanAgent.c will require extra logic for handling user input.
- **Protocol:** This library implements a low-level parser based on the NMEA0183 protocol, with a different set of messages and a modified line ending. Communication is based around passing ASCII messages that look like:

| | |
|---|---|
| '$' | The start-of-message identifier, always a dollar-sign |
| MESSAGE_ID | A 3-character string identifying the type of message. |
| ‘,’ | A comma separates the MESSAGE_ID from the subsequent data |
| DATA1,DATA2,DATA3,... | A comma-separated list of data, all encoded as ASCII characters |
| *XX | A message ends with an asterisk and then a checksum byte encoded as two separate ASCII hexadecimal characters (like '0A'). This checksum is calculated from ALL bytes between the ‘$’ and the ‘*’. |
| ‘\n’ | A newline character actually ends the string. |

[1] You might want to consider each writing the test harness for the other's library. This will ensure that you both know how the code works, and have confidence that it works well.

The decoding stage of this library follows the state diagram provided later on in this manual. It also describes the turn-negotiation algorithm used.
    - An enum datatype must be declared that stores all of the states for the decoding FSM.
    - A struct datatype must be declared for storing all of the data for the ProtocolDecode(). This entails a character array big enough for the biggest protocol message, a variable to store the current index in that array when recording data, a variable of the datatype of the FSM state enum, and a uint8_t to store the checksum.
    - A static helper function for calculating the 1 byte checksum of a string.
    - A static helper function for converting an ASCII hex character to a uin8_t.
- **Field:** This library implements the field-related functionality in this lab. This includes initializing the fields, adding boats, and checking if a guess hits a boat.
- **ArtificialAgent:** Implement an artificial intelligence agent within an ArtificialAgent.c file that implements all the functions described in Agent.h. This file should be a collaborative effort by you and your partner.

  The artificial agent must:
    - Implement an `AgentInit()` function that randomly places the boats. There should be no pattern or hard-coded layout; it should be entirely random on a ship-by-ship basis. This also applies to their direction, so that all possible ship layouts **could** be generated by your code.
    - Implement an `AgentRun()` function that handles the communication protocol and the gameplay mechanics. Guesses must be influenced by some heuristic along with some randomness. If you are not using `rand()` as part of generating the next guess you will not receive full credit. This means that you shouldn't use a hard-coded guessing pattern or guess entirely randomly, as that won't get you full credit.
    - This `AgentRun()` function should follow the provided agent state machine, though there is no user input required as the agent generates its own guess. It should still check if it's in a termination state (WON, LOST, or INVALID) and return if it is before doing anything else (like event detection or running the FSM). In other words `AgentRun()` should do nothing but return 0 if the agent's FSM is within WON, LOST, or INVALID.
    - All of the states should be in a single enum datatype. The variable storing the current state of the Agent should be declared as that enum datatype.
    - Additionally the ArtificialAgent should have an approximately 1s delay during its guessing so that it appears to be thinking.[2]

[2] This delay should be set to (BOARD_GetPBClock() / 8), where BOARD_GetPBClock() is in BOARD.h. This delay ends up being about a second long.

    - Implement the `AgentGetStatus()` and `AgentGetEnemyStatus()` function according to the documentation within Agent.h.
    - Do not modify any members of the Field struct directly (though you may access them for passing them to other functions).
- **HumanAgent.c (Extra Credit):** Implement a human interface agent within a HumanAgent.c file by implementing the functions described in Agent.h, which are further expanded on below:
    - Implement an `AgentInit()` function. To prompt the user during boat placement, `AgentInit()` should have its own event loop where button events are continuously checked until placement is finished. When BTN4 is pressed, the boat direction is rotated 90 degrees clockwise (N->E->S->W). When BTN3 is pressed, the boat start position is moved right. When BTN2 is pressed, the boat start position is moved down. When BTN1 is pressed, the boat is placed at the shown location. During boat placement, all previously placed boats are shown along with the boat being placed if it is a valid location that doesn't collide with other boats. If the current boat placement is invalid, a CURSOR should be displayed at that field location instead.
    - Implement an `AgentRun()` function that handles the communication protocol and the gameplay mechanics by merely prompting the user for attack coordinates during their turn. This prompting is done by placing a CURSOR at 0,0 (upper-right corner of the field). The user then moves this cursor right when BTN3 is pressed and down when BTN2 is pressed. BTN1 selects the current position as the agent's guess. If this guess is on an UNKNOWN field location, then the agent should stop prompting the user (removing the CURSOR field position) and continue the game. If the user's currently selected square is already a HIT or EMPTY, then it should just be ignored. It should also track the state of both the user's game board and the enemy's and display both before the attack coordinates prompt. Note that this will require another event loop that checks for button events within ONLY the SEND_GUESS state.
    - This `AgentRun()` function should implement the provided state machine for its logic. It should also check if it's in a termination state (WON, LOST, or INVALID) and return if it is before doing anything else (like event detection or running the FSM). In other words `AgentRun()` should do nothing but return 0 if the agent's FSM is within WON, LOST, or INVALID.
    - Implement the `AgentGetStatus()` and `AgentGetEnemyStatus()` function according to the documentation within Agent.h.
    - Does not modify any members of the Field struct directly.
- **Code style:** Follow the standard style formatting procedures for syntax, variable names, and comments.

    - Be sure to add inline comments where appropriate. It'll help you remember what you've done when you come back to finished code, help us assist you in debugging, and you'll lose points if you don't.
- Add the following to the top of every file you submit as comments:
    - The name of the author(s) of this file.
- **Readme:** Each partner should create **their own** readme file named README.txt containing the following list of items. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.
    - First you should list your name, your partner's name, and the names of anyone else who you have collaborated with. Also list which libraries you did: either Protocol or Field.[3]
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? Did you work with anyone else in the class? How did you work with them and what did you find helpful/unhelpful? How was working with a partner? What worked/didn't in partnering with them?
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the point distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understanding of this lab or would more teaching on the concepts in this lab help?
- Submit all of your libraries and test harnesses (Field.c, FieldTest.c, Protocol.c, ProtocolTest.c), and your agents (ArtificialAgent.c, AgentTest.c and possibly HumanAgent.c), along with **your own** README.txt file.

**Grading**

This assignment again consists of 28 points:

- 6 points – Implementing Field

[3] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

- 3 points – Implementing FieldTest.c (and correctly testing Field)
- 6 points – Implementing Protocol
- 3 points – Implementing ProtocolTest.c (and correctly testing Protocol)
- 4 points – Implementing ArtificialAgent.c
- 3 points – Implementing AgentTest.c (and correctly testing Artificial Agent)
- 1 point – Create a README.txt file (each partner should submit their own)
- 1 point – GIT hygiene
- 1 point – Follow proper code style and formatting.

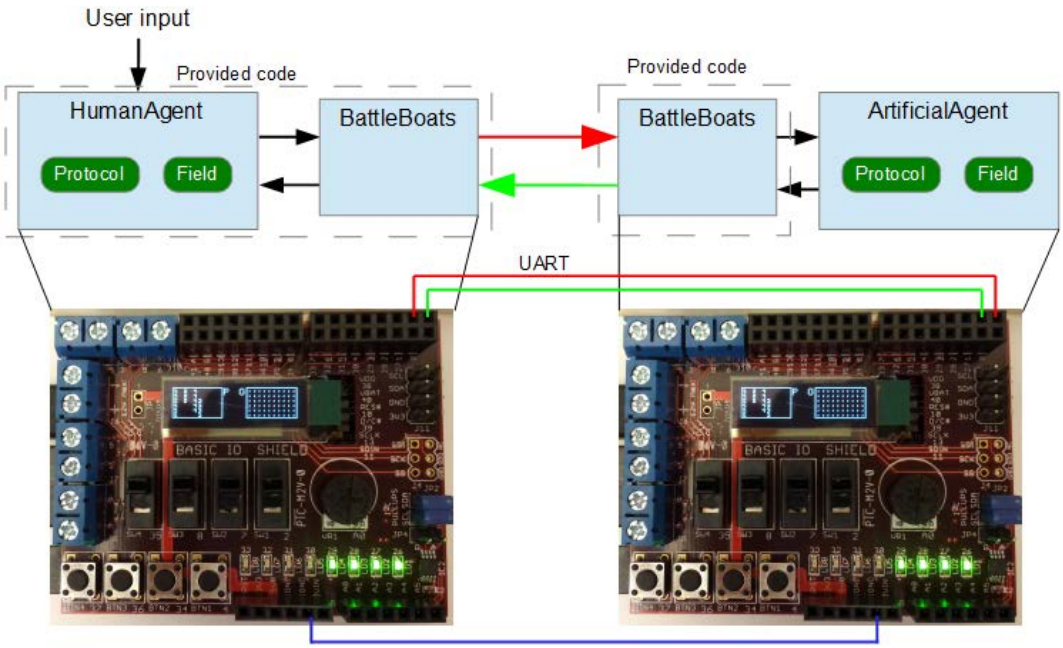
- Extra Credit (5 points) – Implement HumanAgent.c, an agent controllable by the user, according to the provided specifications.
    - 3 points – Implementing HumanAgent.c
    - 2 points – Implementing AgentTest.c (and correctly testing Human Agent)

You will lose points for the following:

- NO CREDIT for the lab for compilation errors (applies to both partners).
- -2 points: any compilation warnings (applies to both partners).
- -2 points: used gotos (applies to both partners).
- -1 point: Constants that were defined in header files were redeclared or unused with the value hard-coded instead (applies to both partners).

**Program Overview**

For this program you will be implementing all the supporting libraries for a BattleBoats game along with one of the agents that will play the game (really just an interface for a human to play it). This will require 2 separate Uno32s to run each agent to play against each other. Below is a diagram that outlines how the code is architected and how the Uno32s are connected. Each Uno32 has a UART1 TX and UART1 RX pin as their pins 0 and 1. These need to be connected to the opposite pin on the other Uno32, so UART1 TX on one is connected to UART1 RX on the other, as indicated by the Red and Green lines in the following diagram. Additionally, the Uno32's need a shared ground. If they are both powered from the same source, they will work properly. Otherwise, both Unos should be connected with another of the jumper cables that were provided to you. The Blue line shows where this jumper cable should go to connect the two grounds of the Uno32s.




The non-library code encompassed by the dashed line is code that has already been written for you and is included in HumanAgent.o, BattleBoats.c, and Lab9SupportLib.a. The UART is the serial port connection between the two BattleBoats players.[4] They can be any combination of ArtificialAgents or HumanAgents. You will use the jumper wires (only 3 are required per group) to connect pin 1 to pin 0 of the other for both Uno32 boards (which jumper cables are used doesn't matter). This means that you'll need to program each agent independently; this will easiest to do with your partner on neighboring computers with each person running MPLAB X and able to program one of the Uno32's.

**Encryption/Hashing**

Encryption is the encoding of information so that agents without a secret key cannot read it. It is commonly used when transferring sensitive information, such as credit card information and is an important part of the internet.

Hashing is an operation commonly used in conjunction with encryption. It is the calculation of a number that is unique based on the data fed in as input. It can be used to provide a way to identify large amounts of data with a much smaller identification number. It's commonly used for building complex data types, such as the hash table.

[4] UART stands for Universal Asynchronous Receiver/Transmitter, and is the common serial port used on most microcontrollers.

A trivial example of a hashing function is the XOR() operator. The data is XOR()d together using a specific chunk size depending on how unique the resultant hash should be given the size of the data input.

For this lab, you will XOR all of the bytes that comprise the guess and the encryption key, which will result in a single byte checksum.

Turn order must be negotiated in a way that neither agent can cheat in agreeing on which agent should guess first.[5] The algorithm is as follows:

1. Agent “A” generates a random 16-bit number that is its "guess" along with another 16-bit number that is used as the encryption key.
2. Agent “A” then transmits a checksum of both its guess and key (which is an 8-bit XOR of all of their bytes) along with an encrypted version of its guess (which is a 16-bit XOR of the guess with the encryptionKey).
3. During this time Agent “B” is doing the same thing.

Now both agents have the encrypted version of the guess and, since each knows the encryption and checksum algorithms, a way to verify that the final unencrypted version of the guess was in fact the same as the original guess transmitted.

4. Once Agent “A” has received Agent “B”s encrypted guess and checksum, it transmits the unencrypted guess and the encryption key (and Agent “B” does the same).
5. Agent “B” can now verify Agent “A”s information by verifying both the checksum and the encryption key (and Agent “A” does the same).
6. Now both can agree on who should go first by having either guessed higher or lower than the other agent depending on if the XOR of the LSB of their guesses is 1 or 0.[6]

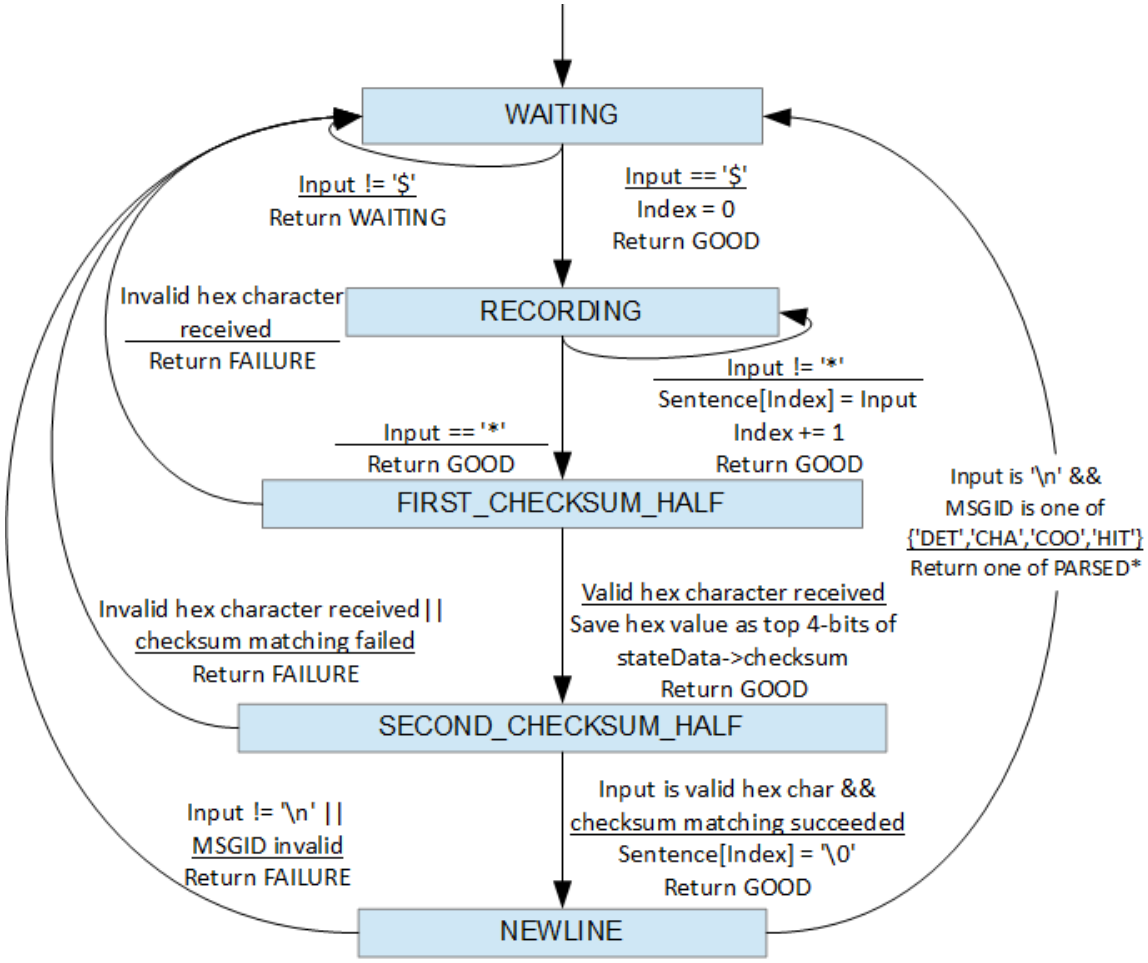
**Protocol Parsing State Machine**

The protocol parsing algorithm is shown in the finite state machine diagram below. You should be able to test this by sending in characters that make up a valid string and have the correct decoding. A typical message to decode might be:

```
$C00,3,5*45
```

[5] Going first gives that agent a large advantage in the game. Thus we need to determine a way to fairly assign who goes first.

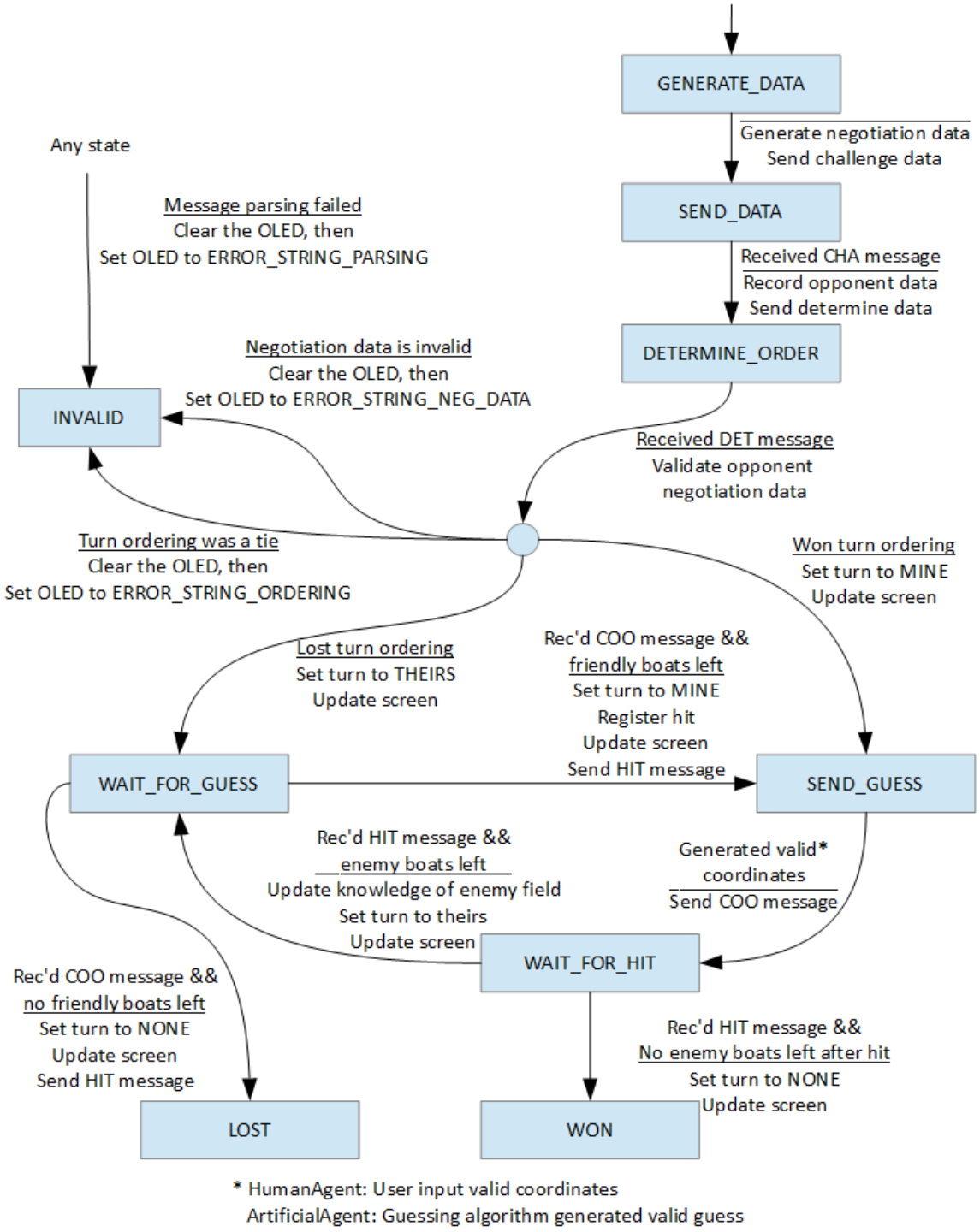
[6] That is, you need to check the XOR of the two guesses and check the LSB. If it is 1, then if guess A is larger than guess B, A goes first. If not then B goes first. If the LSB is 0 then if guess A is less than guess B, A goes first. If not, B goes first.




**Agent State Machine**

Each agent follows the state machine provided below. It is divided into two sections: one for initializing the game state and the other for processing turns.

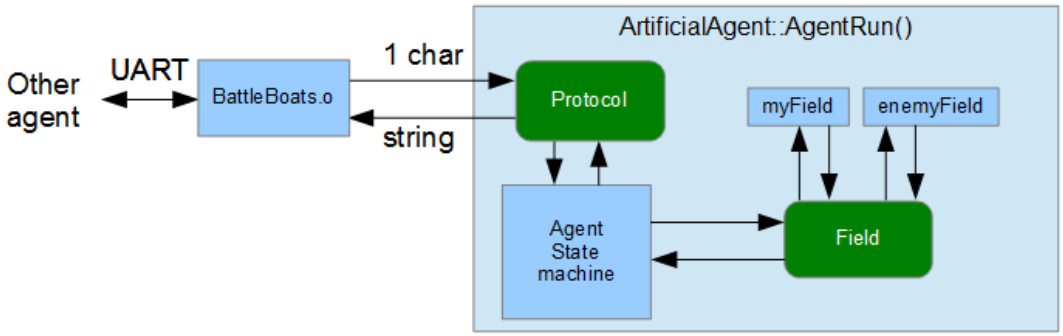
Note that while this machine refers to user input as it's directly applicable to the human agent, any agent will follow this same state machine with just a different way to generate that input. In the case of the ArtificialAgent, this input is actually generated by the artificial intelligence logic within the ArtificialAgent code.




**Artificial Agent**

The artifical agent should rely on the libraries that you will implement: Protocol and Field. This code will be organized into an ArtificialAgent.c file, which will also implement the agent state machine described above. The following diagram gives a rough overview

of how the libraries are used within the ArtificialAgent. Notice that the Field data is only modified through functions within the Field library.




The above diagram is a high-level overview of how the code is architected for this lab, and specifically, how the ArtificialAgent code will be organized. The green rounded-rectangles represent the libraries you create. The blue rectangles represent actual code in ArtificialAgent.c. Both `myField` and `enemyField` are `Field` structs that are stored in ArtificialAgent.c, but manipulated using the Field library within the state machine implemented within ArtificialAgent.c.

**`Rand()`**

For this lab you will be using the `rand()` function provided by the C standard library. The code that is implemented in `main()` (in both Tester.c and the provided BattleBoats.c file) handles seeding the random number sequence via `srand()`, so you won't need to handle that.

`rand()` only provides you with a random 16-bit number, but for most uses, like in the case of this lab, you will want that number to be within a different range. If that range is a multiple of 2, like 16, you can just use a bitwise-and to select the lowest n bits that make up that range. For example, to generate random numbers between 0 and 15 you could write `rand() & 0x0F`. For ranges that are not a power of two, use the integer modulo operator: `rand() % 15`. Note that in embedded systems the modulo operator is VERY computationally expensive compared to bitwise operations, so if you need a base-2 range, you should always use the bitwise-and method over the modulo.

**Testing HumanAgent/ArtificialAgent:**

To test your agents, it's going to be easiest to test them one step at a time. This means testing Protocol and Field well before moving on to the agents. Do this in your ProtocolTest.c and FieldTest.c files. Once you're sure that all of your code correctly works, you can start on the agent. These are fairly easy to test because you can

communicate directly with them by sending input from the terminal directly to the code via the USB cable.

To begin testing, disconnect your agents from each other. Now connect to it using your serial terminal emulator (Cutecom or RealTerm) at 115200 baud. Now we will emulate the messages the other agent will send. Below are two sets of valid turn negotiation data followed by a list of HIT and COO messages. Send these in the proper sequence and observe that your agent operates correctly. To test how your agent responds to invalid messages, just change any single byte of the data, which should cause the checksum to fail.

| | |
|---|---|
| **Negotiation Data Set 1** | $CHA,37348,117*46<br>$DET,9578,46222*66 |
| **Negotiation Data Set 2** | $CHA,54104,139*45<br>$DET,32990,21382*5e |
| **Negotiation Data Set 3** | $CHA,62132,70*79<br>$DET,52343,16067*50 |
| **Negotiation Data Set 4** | $CHA,36027,55*7a<br>$DET,7321,36898*6e |
| **HIT messages** | $HIT,3,8,1*43<br>$HIT,0,2,0*4b<br>$HIT,2,3,1*49<br>$HIT,5,6,4*4e<br>$HIT,0,3,0*4a<br>$HIT,1,7,1*4e<br>$HIT,4,8,0*45<br>$HIT,5,3,3*4c<br>$HIT,0,5,0*4c<br>$HIT,5,6,1*4b<br>$HIT,1,1,1*48<br>$HIT,1,0,0*48<br>$HIT,5,2,5*4b<br>$HIT,2,8,0*43<br>$HIT,0,6,0*4f<br>$HIT,5,9,0*45<br>$HIT,2,8,2*41 |
| **COO messages** | $COO,0,2*41<br>$COO,5,5*43<br>$COO,1,6*44<br>$COO,0,4*47<br>$COO,0,5*46<br>$COO,1,2*40<br>$COO,3,8*48<br>$COO,4,0*47<br>$COO,1,7*45<br>$COO,0,8*4b<br>$COO,2,2*43<br>$COO,4,1*46<br>$COO,0,0*43<br>$COO,2,9*48 |

**Frequently Asked Questions:**

***I get*** ../ArtificialAgent.o: In function `AgentInit':
ArtificialAgent.c:(.text+0x128): undefined reference to `FieldOledDrawScreen'
../ArtificialAgent.o: In function `AgentRun':
ArtificialAgent.c:(.text+0x454): undefined reference to `FieldOledDrawScreen'
ArtificialAgent.c:(.text+0x484): undefined reference to `FieldOledDrawScreen'
ArtificialAgent.c:(.text+0x578): undefined reference to `FieldOledDrawScreen'
ArtificialAgent.c:(.text+0x5a4): undefined reference to `FieldOledDrawScreen" ***when compiling.***

Remove Lab9SupportLib.a from the project and re-add it to your Libraries folder. Order matters when linking in MPLAB X, so re-adding Lab9SupportLib.puts it at the end fixing this issue.

***I get*** ../BattleBoats.c: In function `main':
/ BattleBoats.c:83: undefined reference to `Uart1Init'
/ BattleBoats.c:98: undefined reference to `ButtonsInit'
//BattleBoats.c:102: undefined reference to `OledInit'
/BattleBoats.c:105: undefined reference to `OledDrawString' ***when compiling.***

Remove Lab9SupportLib.a from the project and re-add it to your Libraries folder. Order matters when linking in MPLAB X, so re-adding Lab9SupportLib.puts it at the end fixing this issue.

***I get*** ../BattleBoats.c: In function `main':
//BattleBoats.c:62: multiple definition of `main'
build/default/production/_ext/271043436/Tester.o:Tester.c:(.text+0x0): first defined here ***when compiling.***

Only one file can define main() for a C program. In this case, main() is implemented both in Tester.c and in the BattleBoats.c file we provide you. You only need one, so remove the other from the project to compile.