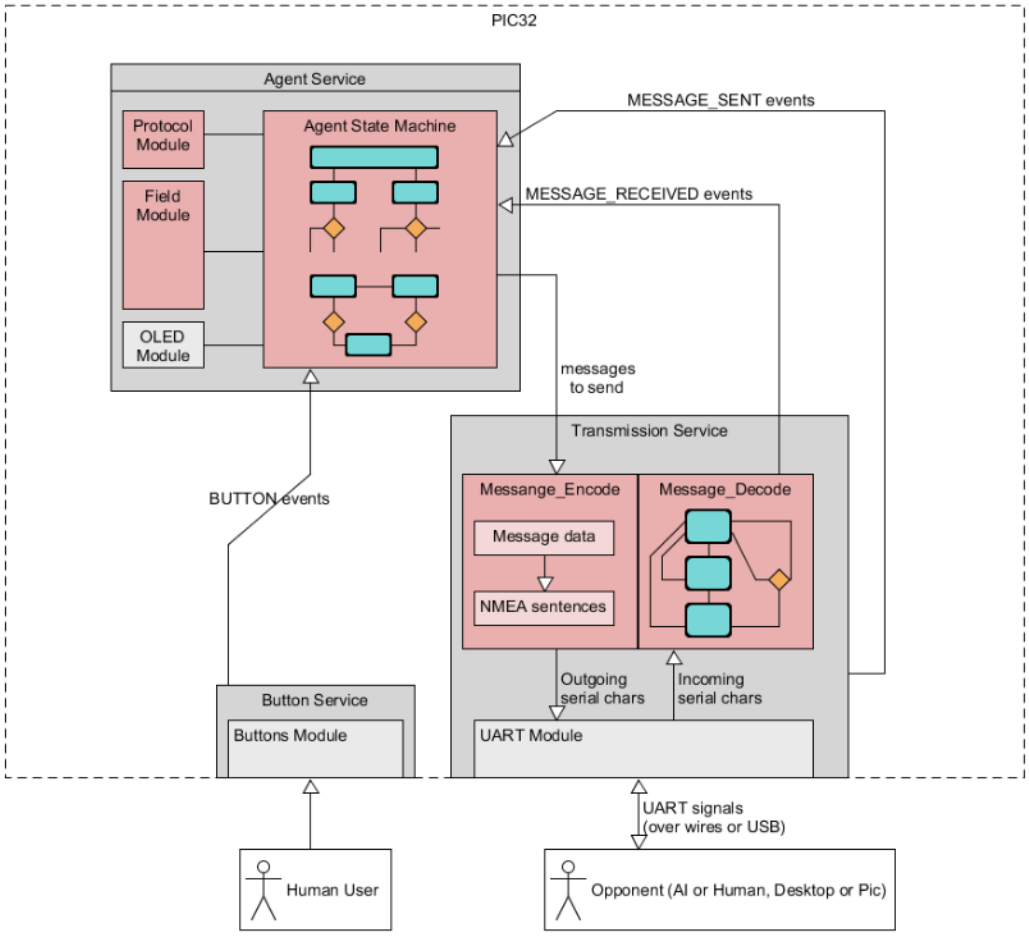
Introduction

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. You will play this game using two u32 boards (or a u32 and any other player that can communicate via UART).

We have provided a main file for the top-level operation of your embedded players. It uses a standard event-based framework, in which a constellation of reactive services exchange information in the form of events.



The top level framework is already complete, and you can find it in your zip folder. There are three modules:

1. A transmission service handles wired communication between the two agents. This module detects incoming characters, translating incoming electrical signals into data that other modules can easily work with.

2. A button service detects button events. This should be unsurprising.
3. An agent service manages the high-level game playing activity and updates the user display.

Each service has already been written for you, in full. However, these services depend on submodules which you are responsible for implementing. Four modules above have been highlighted:

1. The "**Message**" module is an interface between the raw sequence of arriving UART characters, and the rest of the system. Communication between agents is handled using a specialized communication protocol. This module has two responsibilities: To translate incoming messages into convenient data structures, and to translate outgoing data into messages.
2. The "**Agent**" module handles the steps in the communication protocol between the two agents, and manages the user interface. It is implemented as one large state machine, which calls functions from two submodules:
    3. The "**Field**" submodule is responsible for managing BattleBoats moves. It is the "AI", insofar as anything in this system can be considered intelligent. It processes two data structures called "fields", which keep track of each "board". One field represents the agent's estimate of the opponent's field, and the other contains a record of its own field, tracking the opponent's progress.
    4. The "**Negotiation**" submodule is responsible for handling the cryptographic scheme that the two agents use to fairly decide which agent goes first.

As partners, you are required to split up the work of these modules. One partner will select two module to implement, and the other partner will take the other two. HOWEVER, you are required to write test code for your partner's modules.

So, for example, Alice and Bob decide that Alice will write Agent.c and Negotiation.c, while Bob will write Message.c and Field.c. That means that Alice must write MessageTest.c and FieldTest.c, and Bob must write AgentTest.c and NegotiationTest.c. You must explicitly state your choices at the beginning of your READMEs (see below).

This means that both students must communicate effectively to make sure both partners understand the behavior of each library. This comes with several advantages, which you will read more about below.

Submitting this lab:

Both students are considered to have submitted identical files for every required file except the README.txt. That is, we only grade each project once, and you may choose as a team which repo we use to pull the final version.

The README is an exception: Each student is required to write their own README.txt as an individual.

Both students are required to maintain their work in their own personal repo as they work on the lab. However, (except for the README), only one student is required to push a final version. Both students should state EXPLICITLY AT THE TOP OF THEIR README which repo contains the final version.

FOLLOWING THESE DIRECTIONS PREVENTS GRADING ERRORS!

For example:

CruzID: alice@ucsc.edu MY REPO CONTAINS THE FINAL VERSION OF THE PROJECT. I WROTE: -AGENT.C -NEGOTIATION.C. -MESSAGETEST.C -FIELDTEST.C Please note we implemented the Field AI for extra credit, and it beats the staff AI 95% of the time. Introduction: In this lab, we…	CruzID: bob@ucsc.edu ALICE'S REPO CONTAINS THE FINAL VERSION OF THE PROJECT. I WROTE: -AGENTTEST.C -NEGOTIATIONTEST.C. -MESSAGE.C -FIELD.C Our AI destroys yours, give me extra points for that. Introduction: Lab09 was absolutely wild. I …

We will not share your README with your partner.

If you are taking the Solo option, your README should still convey all information that the graders need clearly at the top of the file:

> CruzID: carol@ucsc.edu
>
> I WROTE:
> -AGENT.C
> -NEGOTIATION.C.
> -AGENTTEST.C
> -NEGOTIATIONTEST.C
> -FIELDTEST.C
>
> I implemented a human agent for extra credit.
>
> Introduction:
> This lab was so unbelievably ….

Required Files:

- `Agent.c`
- `AgentTest.c`
- Field.c
- FieldTest.c
- `Message.c`
- `MessageTest.c`
- Negotiation.c
- NegotiationTest.c
- Lab09_main_ec.c (optional, if you want to implement a human agent)

Concepts

- Hashing and Encryption
- Checksums and message protocols
- Finite state machines
- Random Number Generation
- Teamwork

Reading

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

Lab Files:

- **DO NOT EDIT THESE FILES:**
  - **Agent.h / Message.h / Negotiation.h / Field.h** – Header files for the modules you will implement. Each is described in more detail below
  - **BOARD.c/h, Buttons.h, Oled.c/h, OledDriver.c/h, Lab9supportLib.a** – Standard CE13 libraries, which should be familiar to you by now.
  - **FieldOled.c/.h –** A wrapper for the field module that draws the BattleBoats field display.
  - **UART1.c/.h, CircularBuffer.c/.h** – Until now, we've been using a blocking version of the UART interface (namely, printf). While this is a useful simplification for new students, it doesn't work so well in real embedded systems. These files provide a non-blocking alternative.
  - **Message_correct.o / Negotiation_correct.o / Field_correct.o / HumanAgent.o** – Compiled and assembled working versions of each module you are required to implement. These can be included in your project in the same manner as a .c file, allowing you to test modules in tandem with working versions. If all 4 are included, you can see what a working version of the project is like.
  - **BattleBoats.h** – Defines the top-level system events
  - **Lab09_main.c** – This is the top-level main file that handles the event distribution system. You will read more about this system below. When we grade this work, we will use this file exactly as included in Lab9.zip. If you want to make any changes to this file (perhaps to assist with debugging), we strongly recommend that you make a copy and modify that instead.
    - This file should compile, as long as you have included complete .o or .c files for Agent, Negotiation, and Message.
- **CREATE THESE FILES:**
  - **AgentTest.c**
  - **MessageTest.c**
  - **NegotiationTest.c**
  - **FieldTest.c**
  - **Agent.c**
  - **Message.c**
  - **Negotiation.c**
  - **Field.c**

In this lab, we also provide .o files for each .c file we require. You can use these to write your own test harnesses, or to make your board work even if you haven't finished part (or any!) of the project.

Assignment requirements

When compiled, your project should be able to play a game of BattleBoats against any other correctly-implemented agent. This includes an agent programmed using the given .o files, or an agent running in a different language on your laptop or desktop communicating via serial, or a human typing carefully on coolTerm.

Very few teams produce a fully functional project. However, there is plenty of partial credit available. Each module has its own requirements. Each module's functions must follow the specification which are explicitly stated in the .h files for each module.

- **Field.c, Message.c, Agent.c, Negotiation.c:** Each function in each of these libraries should be defined, except where noted in the .h comments. Use the .h files for specifications.
- **FieldTest.c, MessageTest.c, AgentTest.c, NegotiationTest.c**: By now, you should have a good idea of how to write a good test harness. Be thorough and creative as you test.
    - Your tests should mimic the behavior of other modules, interacting with the module you are testing. For example, you should be able to pass events into AgentSM() as though they were being sent from the Transmission service.
    - You may also wish to include interactive portions in your test harnesses. For example, it is very useful to include an interactive play session in FieldTest.c.
- **Full Project:** Implement an artificial intelligence agent that uses the four modules you create, and runs when compiled with the given Lab09_main.c.
  The agent must:
    - In normal operation (ie, not an extra-credit addition), all behavior should be at least somewhat random. There should be no fixed pattern or hard-coded layout.
    - Your agent should be able to play several full games with any other correctly coded agent.
    - Your agent should be able to detect and report when its opponent fails to uphold the protocol, or when transmissions fail for some reason. It is not required that your agent be able to recover from such errors, though you may find it helpful.
    - Your agent should know when it has won or lost the game, and report victory or defeat.
    - Your agent should follow the state machine diagram faithfully.

    - Your agent should correctly record the history of the game, both on its own board and the opponent's board.
    - Your agent should be able to correctly handle all four negotiation outcomes (challenger heads, challenger tails, accepting heads, accepting tails).

- **Extra Credit opportunities:** There are several ways to obtain extra credit in this lab. If you do so, you MUST state what you did near the top of your README, and you should use #define switches to allow graders to switch between the basic version and the extra credit version.
    - Write your own HumanAgent.c that allows a human to place boats and make guesses using the buttons and switches.
    - Create an agent with a smart enough AI to defeat the staff AI more than %55 of the time.
    - Create an agent that cheats in the first-turn negotiation to gain an unfair advantage.
    - Create an agent that cheats during gameplay (however, its cheating must be undetectable by the opponent!)
- **General Requirements:**
    - Follow the standard style formatting procedures for syntax, variable names, and comments.
    - Your code should be well structured and readable.
    - Include useful comments. This is especially important in this lab.

- **Readme:** Each partner should create **their own** readme file named README.txt containing the following:
    - Your partner's name
    - The required information described above: Which submission would you like us to grade? Who wrote what? What extra credit did you do (if any) and how can we observe it in action?
    - Did you collaborate with anyone? What was the nature of the collaboration?
    - A lab report, containing:
        - An overview of the BattleBoats system. Describe how it works, as you understand it.
        - What worked well? What didn't? Describe at least one testing strategy you found effective.
        - What did you learn from this lab?
        - What did you like about this lab? What would you change about this lab?

Grading

The same code base will be graded for both students, although their READMEs are graded separately.

If you choose the solo option, you will implement two of the four student modules (along with the test harnesses of those two modules). One module must be Agent, and the other must be either Message or Field. Your grade will be doubled for those two modules (but not their test harnesses)[2].

This assignment again consists of 27 points. There are also 3 points of extra credit available.

- 14 points -- Modules:
  - 2 Negotiation.c
  - 4 Message.c
  - 4 Field.c
  - 5 Agent.c
- 7 points -- Test harnesses:
  - 1 NegotiationTest.c
  - 2 MessageTest.c
  - 2 FieldTest.c
  - 2 AgentTest.c
- 2 points -- Integration (how well everything works when put together)
- 2 points -- Code Style:
- 2 points -- Writeup:
- 3 Extra Credit:
  - 1 point for each extra credit item, with a limit of 3
- Possible Deductions:
  - Any code that does not compile receives 0 credit.
  - The usual deductions apply: No gotos, no compiler warnings, etc.
  - Up to -3 If your code does not follow the indented structure of the project (for example, using global variables to pass information between modules, or implementing the functionality of one module in the functions of a different module).
  - Up to -3 for hard-coded behavior
  - Other deductions at grader discretion

[2] Sharp observers will note that this yields a maximum of 22 points for the modules and test harnesses, which is one point more than is available to a member of a partnership. The extra point counts as extra credit, and serves as compensation for the greater challenge that solo students face. In practice, this last point is very challenging to obtain, and is less fun than the other extra credit opportunities.

BattleBoats rules:

The rules of BattleBoats are nearly identical to the classic game Battleship:

1. A coin flip determines which player goes first
2. Both players place all of their boats on their own field.
    a. Traditionally, there are four boats of various sizes. Each boat takes up more than one square, and no two boats can overlap.
3. Starting with the winner of the coin flip, players take turns. Each turn:
    a. The attacking player makes a guess, stating a row and a column to fire a shot.
    b. The defending player describes the result of that shot. The shot can:
        i. MISS, i.e. land in open water.
        ii. HIT a ship, but not sink it. The portion of the ship that lies in that square is damaged.
        iii. SINK a ship, if it has hit the last undamaged square of a ship.
    c. Both players record the results of the shot, and then the roles are reversed.
4. The game is over when one player is out of ships.

Battleboats diverges from these rules in three small ways. First, it uses different grid dimensions than the Hasbro game, as well as different ship sizes. Second, it uses a more complicated method for determining the first player. And finally, communication between players is handled using a very specific protocol. All are described below.

Lab09_main.c:

You are expected to read, and to some extent, comprehend this file.

It follows a structure that should be quite familiar to you by now. Event checkers run in the ISRs. They can set module-level variables that signal events. In the main loop, these variables are polled, and when they are set, they are passed into other services.

In this case, the Buttons service and the Transmission services detect events (either BUTTON events or *_MESSAGE_RECEIVED events) and pass them into the Agent service. The Agent service controls the OLED and also can deliver data to the Transmission service, where it can be passed to the opponent.

Though the system's topology is fairly standard, it will take many readings through the lab documentation to understand it. Keep going, and talk frequently with your partner.

Field Module:

The Field module is responsible for doing the job of the plastic grids in the classic game Battleship:

An agent is responsible for keeping track of two grids: One grid for their own ships, whose locations and statuses are known from the beginning of the game, and a second grid for their knowledge of the enemy's ships. The Field module defines a data structure (called a "Field" for holding the information in a grid, and several functions to read and modify these structures).

As noted, BattleBoats uses specific dimensions for the grids, and for each ship. You can find these values in Field.h. Note that boats are described as having "sizes" and "lives." Both describe the number of squares a ship occupies, but a ship's "lives" dwindle as the opponent scores hits.

The Field module also contains two functions, which are optimistically referred to as "AI" elements: One is responsible for placing the ships on the agent's own field during initialization. The other is responsible for generating guesses about the enemy's board.

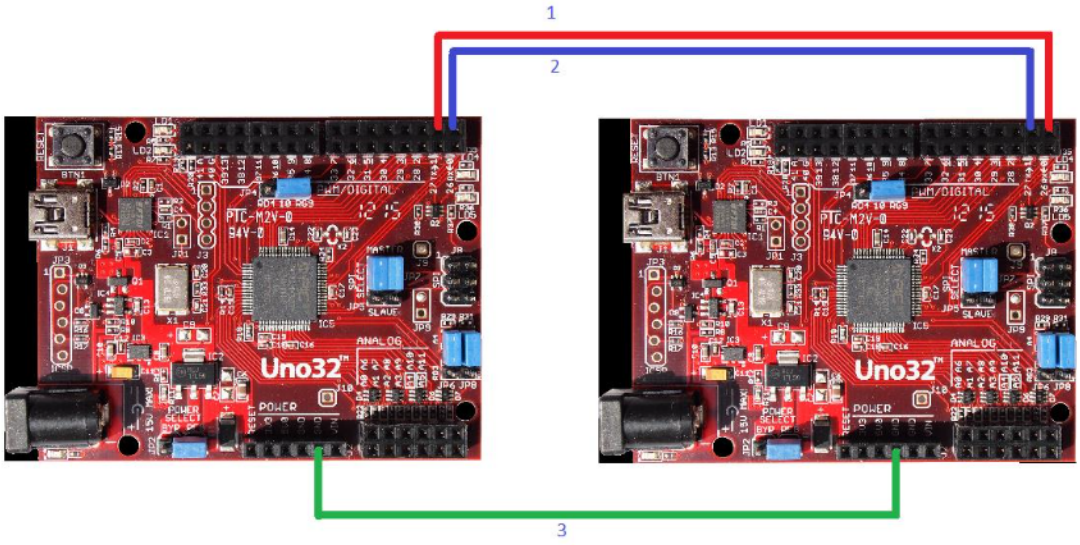
To satisfy the requirements of the lab, neither of these functions need to be particularly intelligent. They should not make illegal moves (for example, putting a ship half-off the grid), nor should they repeat guesses. Simply guessing randomly from all unknown squares is sufficient.

However, there is plenty of room for a more intelligent agent! The staff agent has a modicum of intelligence, and so will usually defeat a random AI, but it can also be easily beaten by an agent with two modicums of intelligence. Extra credit is available for reliably defeating our AI (and it is not too hard to conduct thousands of games in a few seconds).

UART communication:

One of the most common forms of embedded communication is the UART protocol. You've been using this protocol all quarter to communicate over USB cables between the u32 board and your Desktop.

UART can also be used to communicate directly between two devices, without using a USB intermediary. In this lab, you will use the jumper wires included in your lab to connect their UART pins, as in the following diagram:



In the above diagram, wire 1 connects the TX pin of the left board to the RX pin of the right board. Wire 2 connects the RX pin of the left board to the TX pin of the right board. Wire 3 ensures that both boards have a common ground.

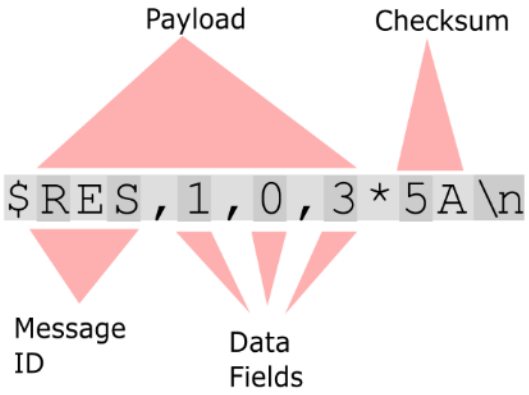
Note that these pins are the same ones that the USB connection uses! This means you cannot "eavesdrop" on the communication between the two board using CoolTerm! But, that also means that you can conduct your own game with a single board using CoolTerm, provided that you type carefully and keep a checksum calculator handy.

Message Module:

This library implements a low-level parser based on the NMEA0183 protocol, with a different set of messages[3], a smaller MESSAGE_ID field[4], and a slightly different end delimiter[5]. Our NMEA messages consist consist of the following elements:

'$'	The start-of-message identifier, always a dollar-sign
MESSAGE_ID	A 3-character string identifying the type of message.
‘,’	A comma separates the MESSAGE_ID from the subsequent data
DATA1,DATA2,DATA3,...	A comma-separated list of data, all encoded as ASCII decimal digits[6]
*XX	A message ends with an asterisk and then a checksum byte encoded as two separate ASCII hexadecimal characters (like '0A'). This checksum is calculated from ALL bytes between the ‘$’ and the ‘*’.
‘\n’	A newline character signifies the end of the string.

Here is an example:



The “payload” is the useful information in the message, the rest is for framing and error detection.

---

[3] To give you an idea of what NMEA protocol is *for*, these are some real NMEA message IDs: GGA = GPS Time, position, and fix data, HDT = Heading from True North, GBS = GNSS satellite fault detection.

[4] Real NMEA messages have 5 character IDs, and the first two characters identify the “talker,” or the device that is sending the message. Since there are only two devices in BattleBoats, we left these out.

[5] Real NMEA messages end in `"\r\n"`

[6] Real NMEA messages can use other ASCII characters in these fields as well.

Checksums

When serial messages are sent electronically, occasionally a single bit will become flipped, or lost. This is a small event in physical terms (it only takes one stray cosmic ray), but it can be a huge event in terms of the data that bit represents. 0b010000000 is very different from 0b00000000!

One strategy to counteract this is for the sender to include a redundant piece of information in each transmission that summarizes every other bit in the transmission. That way, if a bit flips, the receiver can detect the error by computing their own checksum, and realizing that their own calculations do not match the sender's calculation. The receiver will not know which bit flipped, but they will know that *some* bit flipped[7].

This strategy is called a *checksum*, even though many checksumming strategies do not actually use sums. The NMEA protocol uses an XOR checksum. Each ASCII char in the payload is XORed together, and the final 8-bit result is given as two ASCII chars, each representing one nibble of the checksum in hexadecimal[8] format. So, for example, if you wanted to send the payload message "CAT", you would do the following:

Letter	ASCII value (hex)	Binary value
C	0x43	0b 0100 0011
A	0x41	0b 0100 0001
T	0x54	0b 0101 0100
	XOR'd result:	0b 0101 0110
	XOR'd result in hex:	0x 56

Half of your Message module will be responsible for translating C structs into strings that the Transmission service can send. This will require building the payload string first. Once the payload string is constructed, your module will need to calculate the checksum of the payload, and compose the full NMEA message.

Decoding

Half of your message module will be responsible for receiving input characters, one at a time, and interpreting messages once one has arrived completely. These messages will then be passed along to the Agent state machine via the Transmission service. This is quite similar to the

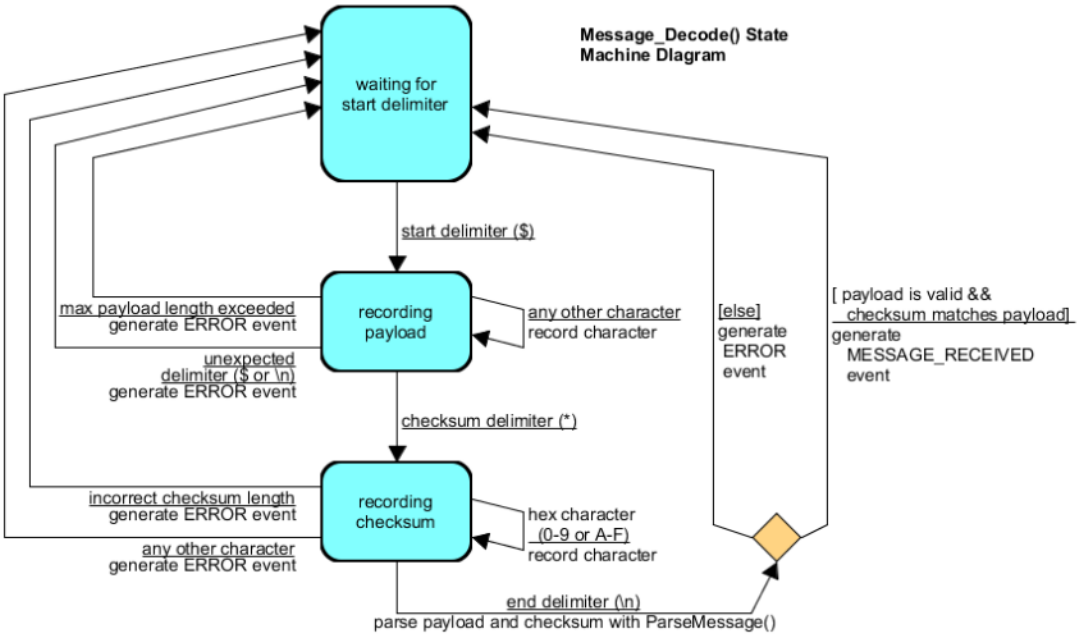
---

[7] You might be wondering: What if two bit flips occur, and they cancel each other out? It's true, it can happen! But, if one bit flip is very unlikely, then two bit flips are very very unlikely. If one message in a million experiences a bit flip, then one message in a *billion* experiences two bit flips, and only one message in *eight billion* experience bit flips that cancel each other out. That said, the NMEA protocol is quite common, so this must happen all the time.

[8] This strategy goes through a lot of representations! A data field is sent as a decimal, encoded in ASCII, XORed in binary, translated into two nibbles, and then each nibble is represented by a hexadecimal digit, in ASCII. Whew!

MorseDecode module from Lab9, which took in one Morse symbol at a time, detected when full sequenced had arrived, and interpreted those sequences.

One nice property of NMEA messages is that they can be parsed by a fairly simple state machine:



Your Message module should implement this state machine.

Negotiation Module:

The purpose of the negotiation module is to provide helper functions for a "commitment scheme," a cryptographic algorithm to determine turn order fairly.

Among humans, we can determine turn order by flipping a coin. Alice might flip a coin and ask Bob to call it. But, this relies on a trusted third party – the coin – to prevent Alice from telling Bob he guessed wrong every time.

PIC32s do not have access to a coin (at least, not over UART), so we use a "commitment scheme." In this scheme, Alice uses a one-way function (a function that is easy to calculate, but very hard to invert) to give Bob proof that she has committed to a secret number. The basic coin flip commitment scheme goes like this:

1. Alice generates a secret (random) number, A.
2. Alice uses her secret number to generate a commitment number, #a, which she sends to Bob.
3. Bob generates a (random) number, B. He sends it to Alice.
4. Alice sends A to Bob.
5. At this point, both Alice and Bob have A and B, so both Alice and Bob can both use A and B to determine whether the outcome was heads or tails by finding the bit-parity of (A XOR B)[9].
6. Finally, Bob can compare #a and A to ensure Alice kept her commitment.

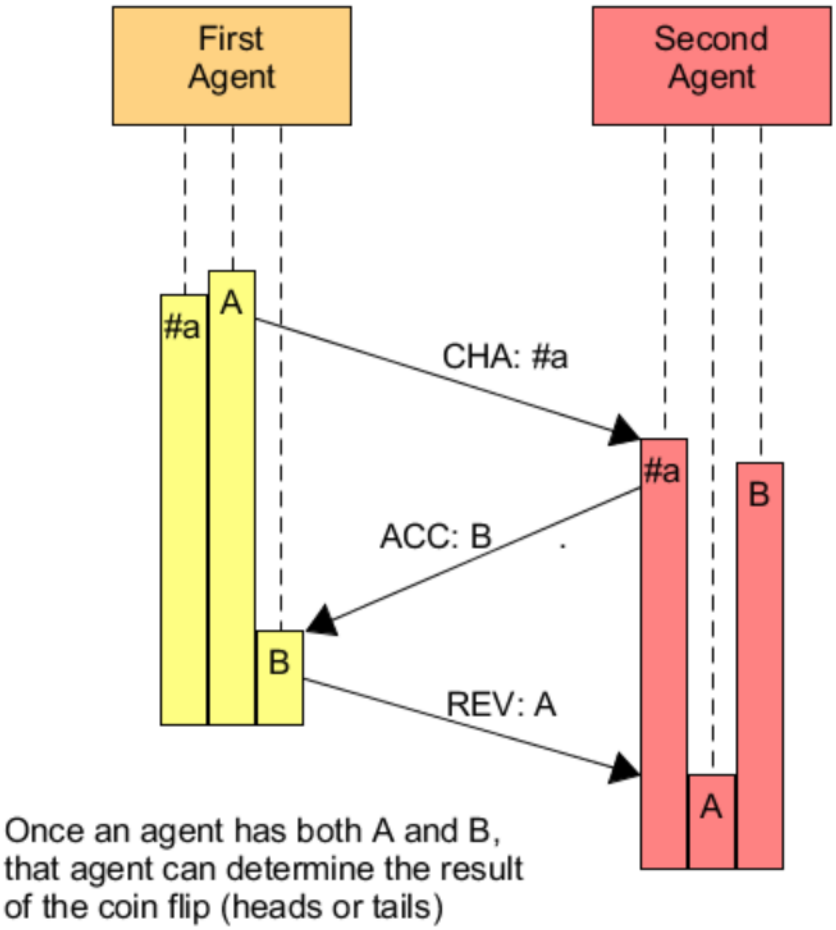
This scheme relies on a secure hash (that is, a hash that is prohibitively difficult to invert, and whose output appears random). We don't use such a hash, to give you the opportunity to cheat, and because we prefer conceptual clarity to cryptographic security in this course. We'll use the "beef hash":

$$\text{Hash}(A) = A^2 \bmod 0\text{xBEEF}$$

Simple strategies to exploit this weak hash exist for both Alice and Bob.

Here is the scheme in the form of a sequence diagram:

[9] Parity is just the number of ones in a binary string, modulo 2. So, the decimal number "42" has bit-parity 1, because decimal 42 = 0b00101010, which contains 3 ones. 3 is odd, so the parity is 1.

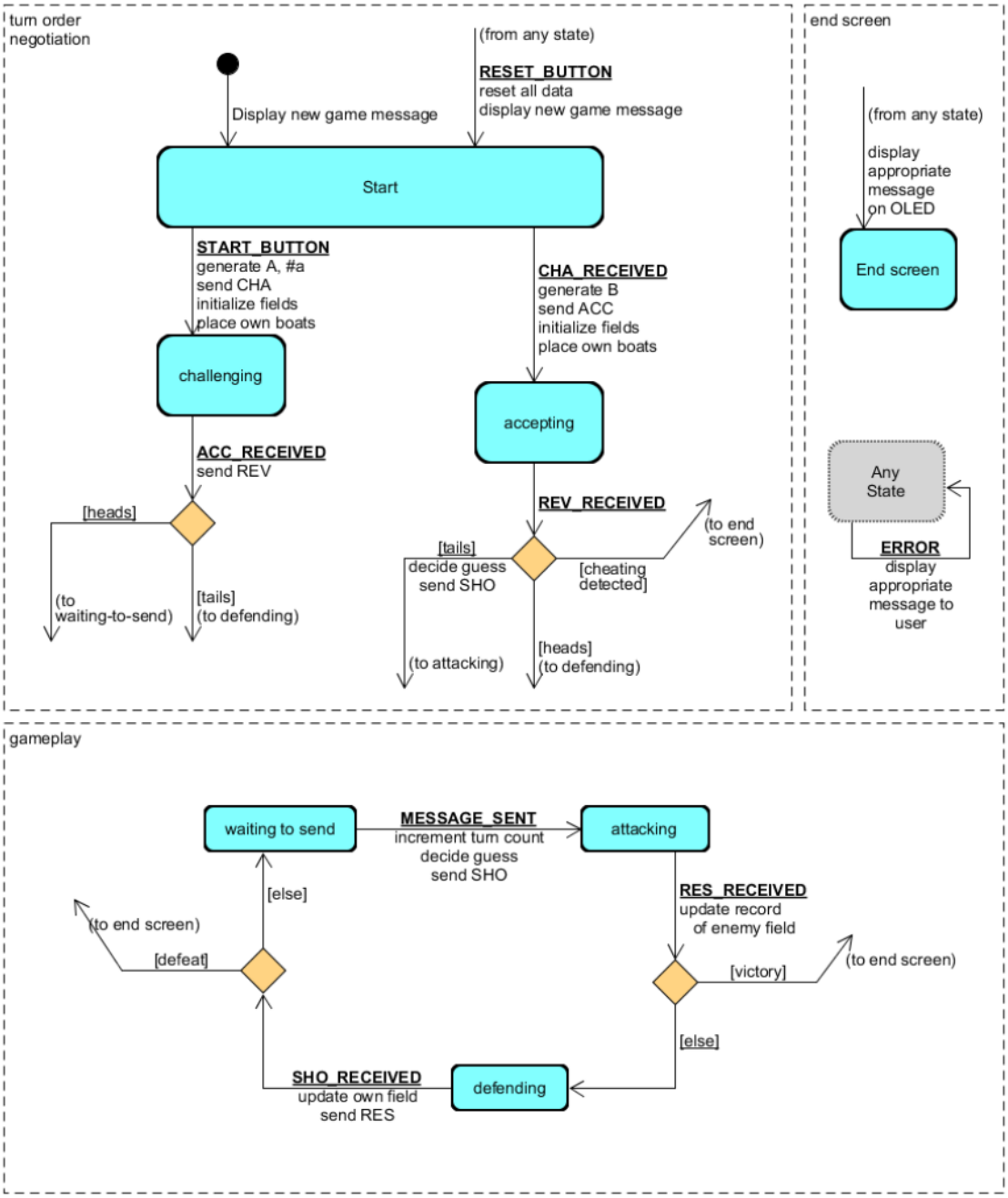


Note that either of our agents can take either role, depending on which party initiates the negotiation.

Agent State Machine:

At the hub of the modules you implement is the Agent state machine. This module is responsible for coordinating each of the other modules. Though it is not the top-level code in this project, it is the one that is making the interesting decisions.

The agent will use the following State Machine:



This diagram is broken into several blocks for conceptual clarity, but it is still a flat state machine, like each other one you have made in this lab.

Randomness and rand():

For this lab you will be using the `rand()` function provided by the C standard library

`rand()` only provides you with a random 32-bit number, but for most uses, like in the case of this lab, you will want that number to be within a smaller range (for example, if you want to select a row on the grid at random). 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.

One interesting property of rand() on the PIC32[10] is that it is entirely deterministic! This might seem as though it is not random at all, and if you're thinking that, you're right: rand() is not actually random! It works by keeping a much larger static variable (usually 32 or 64 bits), and wandering through the 2^32 possible values in random-seeming way[11] on each call. Though it's not *really* random, it is good enough for most purposes.

This has one huge drawback: If you reset your system, that variable goes back to its original static value. This means that, unless you take certain steps to add real randomness, your agent will play out the same sequence of moves each time it powers up![12]

To solve this, we "seed" the random number generator with unpredictable information supplied by some other source. Lab09_main.c takes advantage of every button press to seed that static variable with the time. Humans are quite unpredictable, so this strategy works well.

Some other common strategies include reading noisy ADC pins, reading the noise from photos taken by digital cameras, reading uninitialized memory, comparing the drift between two different timers, and using variables set by the compiler at compile time.

Testing your Agent:

[10] Some microprocessors have special hardware to generate random(er) numbers.

[11] You could say rand is hashing itself on each call.

[12] This "Groundhog Day" effect can actually be quite useful in some situations. For example, you can use this to repeat a sequence with a hard-to-catch bug in it.

To test your agents, it's going to be easiest to test them one step at a time. This means testing Protocol, Negotiation, 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.

Some tips on testing:

- It is *extremely* useful to use #define/#ifdef/#endif flags to turn various kinds of tests on and off. For example, #define IGNORE_CHECKSUM can be used to prevent your message parser from generating errors when bad checksums are encountered. This makes it possible (even fun) for a human to engage with an agent.

- Be creative! Try simulating a match in your FieldTest.h. See if you can automate a full game's worth of messages in MessageTest.h.

- This lab specifies a *minimum* amount of error catching sophistication, but it is a good strategy to be more sophisticated. For example, you can take advantage of the Event params to encode all sorts of information in your ERROR events. You can use this to report the char that caused your decoder to choke, for example.

- Poke around! There are several debugging tools built into the code we've given you, and your efforts to explore our code will be rewarded.

- Use MPLAB X configurations (in Project Properties->Manage Configurations) to switch quickly between test harnesses. You are required to turn in 4 test harnesses, but it may be useful to have many more.

- This is really a lab about debugging strategy.

Errata:

Sample Transmission sequences:

Challenger going first:

$CHA,43182*5A	
	$ACC,57203*5E
$REV,12345*5C	
	$SHO,4,8*58
$RES,4,8,1*55	
$SHO,2,2*54	
	$RES,2,2,0*58
	$SHO,3,3*54
$RES,3,3,1*59	
$SHO,0,0*54	
	$RES,0,0,1*59
	$SHO,0,2*56
$RES,0,2,0*5A	
$SHO,1,0*55	
	$RES,1,0,1*58
	$SHO,3,9*5E
$RES,3,9,0*52	
$SHO,3,0*57	
	$RES,3,0,1*5A
	$SHO,4,7*57
$RES,4,7,1*5A	
$SHO,2,0*56	
	$RES,2,0,3*59
...	
...	
...	

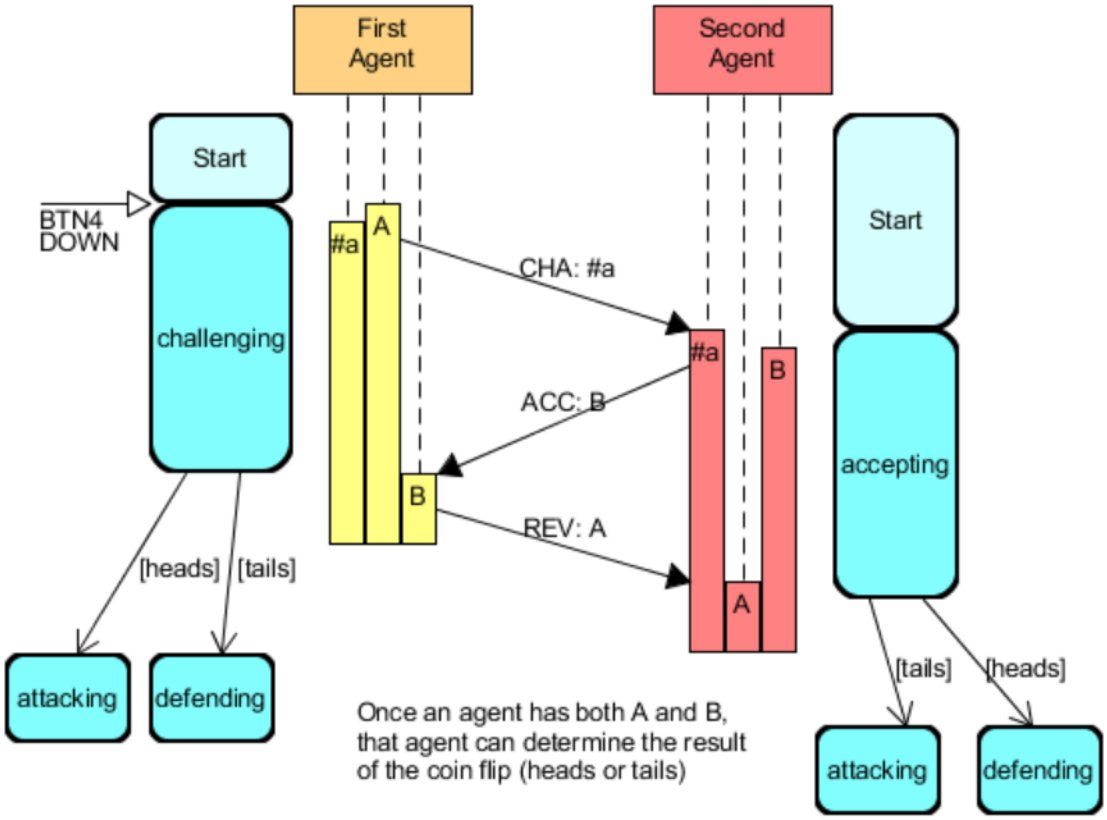
Acceptor going first:

	$CHA,44082*5C
$ACC,41331*59	
	$REV,57204*59
	$SHO,4,8*58
$RES,4,8,0*54	
$SHO,5,9*58	
	$RES,5,9,0*54
	$SHO,3,3*54
…	
…	
…	

.

State-Annotated Sequence Diagrams:

Protocol sequence diagram for turn order negotiation phase:



Protocol sequence diagram for gameplay phase: