Final Project Report

W. Xie, J. Tompkins

12/19/2024

Introduction

The final project requires a further look into programming the STM32F769I-DISCO board than any of the previous labs. We decided to create a video game, and so to fit the further game requirements of a fairly complicated game that works well, we implemented a simplified version of the game Buckshot Roulette. This involved two DISCO boards, one for each player. Information is presented to the player primarily through the LCD screen built into the DISCO board with ASCII art/text information (rather than 3D models and sprites) as well as some information using the terminal. The two boards communicate using UART with one designated as host and the other as client. Other peripherals used include timers and the RNG module.

Buckshot Roulette - Gameplay Overview

a) General Description

Buckshot Roulette is a 1 vs 1 multiplayer game where the two players take turns using items and shooting a shotgun, either at themself or their opponent. The shotgun is filled with a mix of live and blank rounds, live rounds dealing damage to the player hit. Shooting the opponent with a blank round does nothing, however, shooting oneself with a blank round gives the player another turn. Therefore, there is a possible risk and reward for both shooting oneself and the opponent. To add to the gameplay complexity, each player has a selection of items they can use during their turn. These items have a wide range of possible effects that they impart on the players.

Specifically, for our implementation, we decided to have the shotgun house between 2 and 8 shells for each round, with at least one of each type (live and blank). At the start of each round, up to 4 inventory items are given to each player (depending on the space available in their inventory, with a maximum of 8 items in an inventory at one time). Then, the amount of total, live, and blank shells are generated and displayed to the players, so they know how many of each exist, but not the order of the shells. Once the shotgun is emptied, the round is over, and a new round begins (generating a new order and gifting inventory items).

Based on the items they receive and the amount of live and blank shells, the players must strategically choose when to use each item as well as which direction to shoot the shotgun. The game continues until one of the players runs out of health. Our implementation has players starting off with 5 health points and no limit on the maximum (as certain items regenerate health).

b) Items

Based on a variety of factors, including limitations of our implementation, deadlines, and item complexity, we decided to implement a subset of the items found in the official game. Multiple items are allowed to be used per turn, with the turn only ending once the gun is shot. As discussed in the future, there is also a time limit for each turn, and when this time elapses, a shot at the opponent is inferred as the desired move and is executed. Each item has a corresponding 4-bit Inventory Encoding.

Encoding	Item	Define String
0000	Empty Item	EMPTYITEM
0010	Beer	BEER
0011	Cigarette Pack	CIGPACK
0100	Handsaw	HANDSAW
0101	Handcuffs	HANDCUFFS
0110	Inverter	INVERTER
0111	Magnifying Glass	MAGGLASS

Table 1: Inventory Encoding

1. Beer

When the Beer item is used, the next shell is ejected from the gun, live or blank. An example strategy would be the player using this to remove a shell they know the status of and don't want to use or to eject a shell they don't know the status of in the hopes of narrowing down the possibility of future shells. The Beer item is encoded as 0010.

2. Cigarette Pack

When the Cigarette Pack item is used, the player gains one health point back. There is no limit on maximum health or the amount to be used per turn, so theoretically a player could use 8 of them to regenerate 8 health points in one turn. Strategy for this item is straightforward. The Cigarette Pack item is encoded as 0011.

3. Handsaw

When the Handsaw item is used, the barrel of the shotgun is sawed off, doubling the damage of the next shot. This acts on live and blank rounds, doubling a live round to dealing two health points of damage, and doubling a blank round to still deal no damage. Therefore, it is much more beneficial to use this on a live round, as it is essentially wasted on a blank round. This item does not stack, and so using it twice still results in either two or zero damage being dealt. The Handsaw item is encoded as 0100.

4. Handcuffs

When the Handcuffs item is used, it skips the opponent's next turn. This item does not stack, and so using it twice in one turn does not skip two of the opponent's. However, the player could use it once, shoot, get their turn back, and use it again to skip the opponent twice. This item does stack with firing a blank at yourself, however, as it never truly became the opponent's turn in that situation, and so using Handcuffs and then shooting a blank at yourself essentially gives you three turns in a row. The Handcuffs item is encoded as 0101.

5. Inverter

When the Inverter item is used, it switches the status of all shells in the gun: all live rounds become blanks and all blank rounds become live. This item could be stacked to switch the rounds over and back if the player desires to. The Inverter item is encoded as 0110.

6. Magnifying Glass

When the Magnifying Glass item is used, it displays the status of the next round (live or blank) to the players. This is done by displaying a message on the host's LCD screen as the players are intended to both be looking at both screens. The Magnifying Glass is powerful when used in tandem with other items, such as the Handsaw and the Inverter. The Magnifying Glass item is encoded as 0111.

c) Moves

The functionality of the game is executed through "moves." Each turn consists of multiple moves until a "final move" is selected. Selection of moves occurs through a keyboard input over UART 1 when it is the player's turn. This move is then directly executed (if the player is the host) or sent to the host to be executed (if the player is the client). This process will be covered in more detail in the next section of the document, Client vs Host. The keyboard input is interpreted as either shooting or selecting an inventory item to use. Entering a value of 1-8 corresponds to an inventory slot. The value is then updated depending on the item found in the slot based on the encoding.

Encoding	Move	Final Move?
0000	Shoot Self	Yes
0001	Shoot Opponent	Yes
0010	Use Beer	No
0011	Use Cigarette Pack	No
0100	Use Handsaw	No
0101	Use Handcuffs	No
0110	Use Inverter	No
0111	Use Magnifying Glass	No

Table 2: Move Encoding

Client vs Host

a) Introduction and High-Level Description

In order to properly facilitate the game flow with multiple players, a system was devised to have one DISCO board act as the authoritative host, driving the majority of the flow, and the other acting as a predictive client, which handles its portion of the logic and reports back to its player. Therefore, some functionality is present on both boards, and some is only on one or the other.

1. Joining

At the start of the game, each player must select to join as the host or client. In order for proper functionality, the host must join first. It then waits for a join request from the client. At this point, the client makes a Join Request. This consists of sending a UART message with the pattern 1XXXXXXX (the first bit is a one, the rest are "don't cares"). This is received and the game starts, going into the first round.

2. Rounds

Each round consists of multiple turns, alternating between the host and the client. At the start of the game, the host goes first. Then, for each following round, the player to go first is whoever did not close out the previous round (as to prevent a player from going twice in a row). At the start of each round, the host will use the RNG module to determine the inventory of each player and order of shells in the shotgun. Inventory information is sent to the client through a Game Status Update. The round goes until the shotgun runs out of bullets. Then, if the game is not yet over, a new round begins.

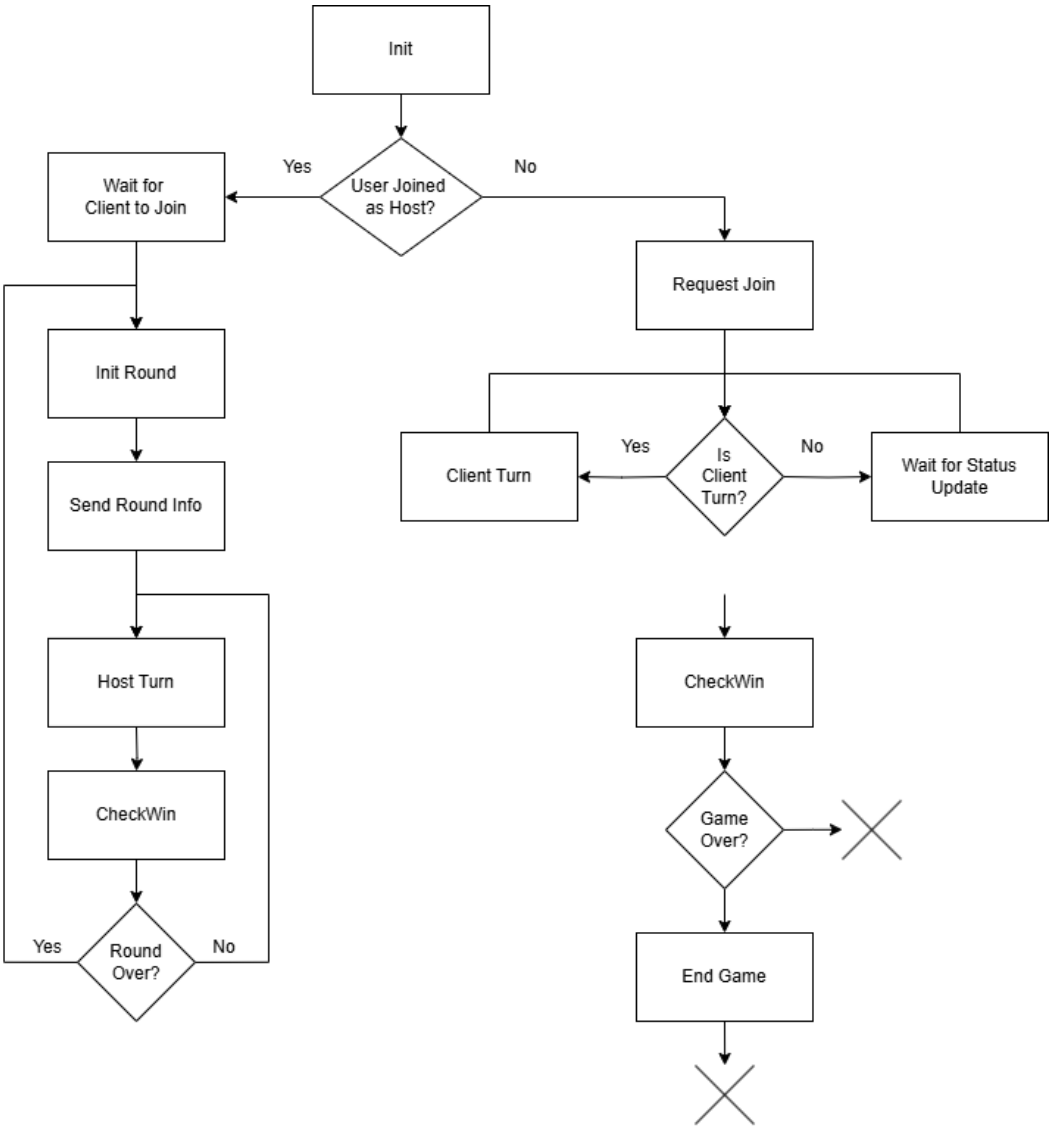

Figure 1: Overall Game Flow

3. Turns on the Host's End

Functionality for the client's turn and the host's turn on the host's end is very similar, with the only difference coming from which UART instance to poll information from. First, the turn timer is started to limit the amount of time a player has to make selections. Then, the host waits for moves from the player. When a move is received, the action is decoded. One possibility is an item has been selected to be used. In this case, the effect of the item is applied and the host waits for more requests. The other possibility is called a "final move." This is when the gun is shot, either by the player deciding to shoot at themselves or the opponent, or by the timer elapsing, in which case the gun is shot at the opponent by default. In this case, the shot is executed and the turn goes to the next player (which may be the same player in certain circumstances, such as when the Handcuffs item is used). After the shot, a Game Status Update is send to the client to inform them the turn is over.

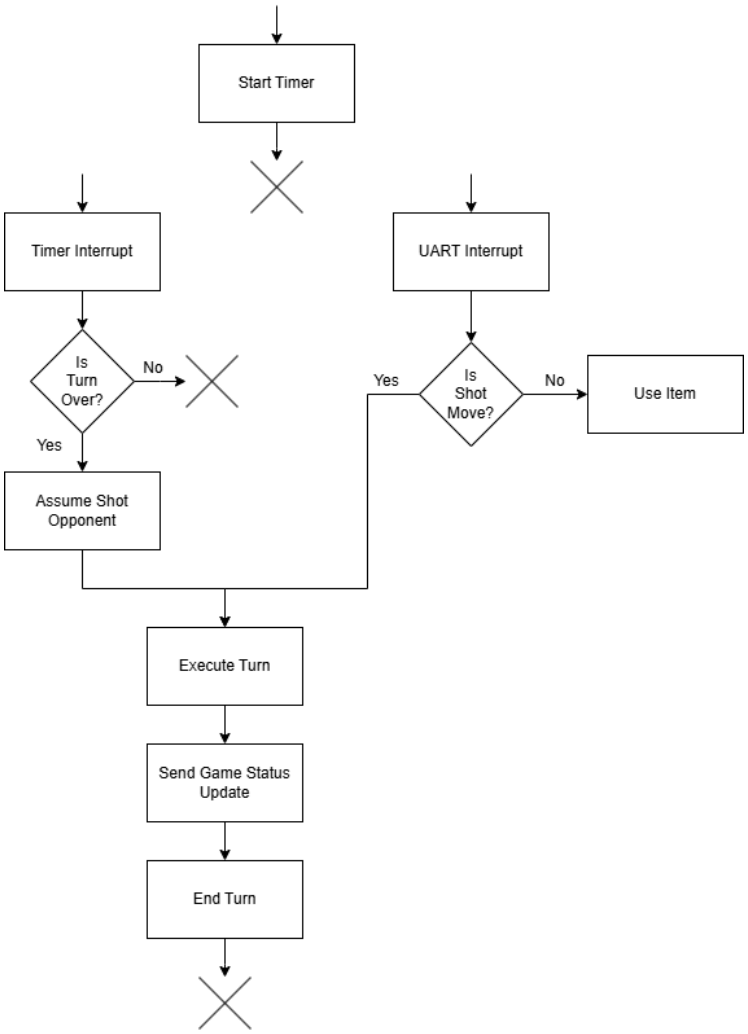

Figure 2: Host Turn

4. Turns on the Client's End

When it's the client's turn, the client will make a Move Request. It will then wait for a Game Status Update to know that its turn is over. If the client waits too long to make its Move Request, the turn timer may elapse, in which case the host will assume a shot at the opponent, execute the client's turn, and send a Game Status Update. When it's the host's turn, the client simply waits for the Game State Update.

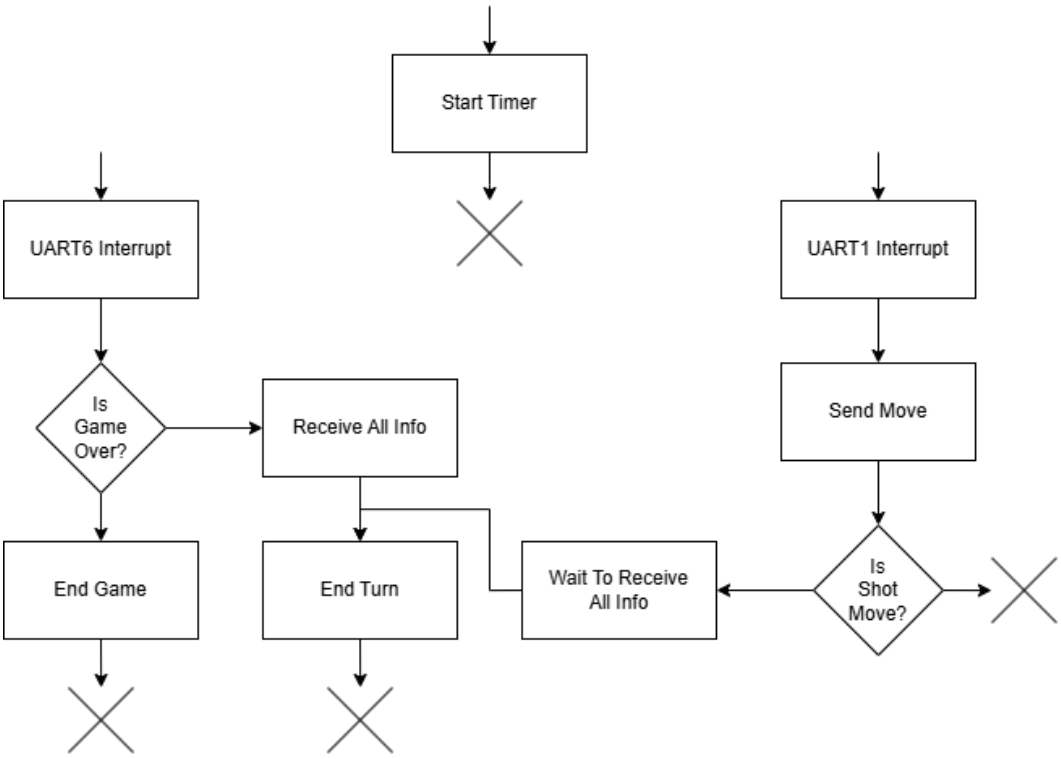

Figure 3: Client Turn

5. Game Over

After each turn, both players check each other's status. If a player has run out of health, the game is over, and a screen is displayed informing the players. This logic is seen in the Overall Game Flow figure above.

b) Low Level Description

1. Turn Flags & Logic

The main while(1) loop in the program contains the logic for one turn. As the game progresses, each iteration fo the loop corresponds to one turn. A few flags are then used to track the current game state to ensure both the host and client are properly tracking whose turn it is. One of these is "is_host", which tracks if this player is the host so that the proper

functionality and control flow is followed. Next, is_host_turn tracks if the current turn is the host's or the client's. This is used in conjunction with next_turn_is_host to determine the turn order, as certain items and actions lead to a player going multiple turns in a row. Within the turn, received_selection and received_final_selection are used to continue once a move has been selected. Then, for the client only, status_received is used to track if the host has sent a Game Status Update. Looking at the full control flow, first, both players start the timer at the start of each turn. Then, there is a branch on is_host, as the host and client follow a different structure. Finally, both players determine who's turn should be next and the timer is reset.

Host: The control flow for the host is very similar for both the client and the host's turn. The only major difference is if move selections come from UART1 (the host's turn, so the keyboard) or UART6 (the client's turn, so the other board). A branch on is_host_turn queues the proper UART to listen for inputs. Next, there is a wait to receive a final move, either through a shot move being selected or by the timer fully elapsing. Within this wait, the host waits for one move at a time. Each move is fed into ExecuteTurn(), which fulfills the action the user was hoping to make, either using an inventory item or shooting the gun. At this point, the status of the gun is checked to switch rounds if needed, reloading the gun and giving new inventory items to each player. After the final move is selection, a bit of housekeeping is done to update both LCD screens, check if the game is over, and send a Game Status Update to the client, and then the turn is ended.

Client: The client starts with a branch on is_host_turn. When it's the client's turn, the client listens for UART 1 and 6, as both keyboard characters and the other board can end the turn. It then waits in a while loop for a final selection to be received, either via a shot move being selected or from the host sending a Game Status Update. Within this wait, the client will wait for one move at a time, and when one is selected, it is sent to the host. Alternatively, during the host's turn, the client simply waits to hear a Game Status Update. After the move selection, there is another wait on status_received to ensure that the turn isn't ended unless a Game Status Update has been received.

2. Game Status Update

A Game Status Update can consist of multiple messages. First, a generic message is sent with part of the data field informing the client about the quantity of remaining messages.

Generic Message: For the generic message, the first two bits encode both win status as well as special cases for when it's the client's turn. As mentioned earlier, the client is predictive and will guess which turn is next, but the host is authoritative and so can tell the client "actually, it's your turn next." A value of 00 corresponds to neither player winning and the game continuing as normal. Note that this does not encode the host's turn, just whoever should be next. A value of 10 corresponds to the host winning the game and 01 corresponds to the client winning. Finally, a value of 11 means that neither player has won yet and the client is next. This is used in certain cases where the client has two turns in a row. Then, the next 3 bits encode the health of the client using straight binary enocoding. Note that this unoficially caps the health at 7, however, it can be higher in reality (which is tracked by the host). The final 3 bits encode if any further messages will be sent. This occurs at the start of each round to update the bullet count and inventory items. These

can take a value from 000 (which means no items will be send) to 100 (meaning 4 items will be sent). Note that 001, 010, and 011 are possible, telling the client a certain number of items will be sent. The client stores this value and then expects the corresponding number of messages (with 2 inventory items being sent per message as well as a bullet information message being sent after the inventory messages). Note that no bullet information message is sent unless at least one inventory message is sent.

Bits [7:6]	Bits [5:3]	Bits [2:0]
Win & Client Turn	Client Health	Expect Inventory Items
00 - Continue as normal 01 - Client Won 10 - Host Won 11 - Client Turn	[2:0] = 0-7 Capped at 5	000 - No items 001 - 1 item 010 - 2 items 011 - 3 items 100 - 4 items

Table 3: Generic Message Layout

Inventory Message: An inventory message consists of up to two inventory items being sent, each encoded in 4 bits using the Inventory Encoding. If only one item is being sent, then the Empty Item encoding is used, which is 0000. This tells the client that this slot is not a new item (if the client already has 7/8 slots full and can only accept one new item).

Bits [7:4]	Bits [3:0]
Item 1	Item 2

Table 4: Inventory Message Layout

Bullet Information Message: A bullet information message encodes the number of live rounds in the first 4 bits and the number of blanks in the last 4 bits. There will be between 1 and 7 of each, and so these are straight binary encoded with 0001 referring to 1 an 0111 referring to 7.

Bits [7:4]	Bits [3:0]
Number of Live Rounds	Number of Blank Rounds

Table 5: Bullet Information Layout

3. Move Request

A Move Request consists of sending a UART message with the pattern 0XXX for the first 4 bits (the first bit is zero and the next 3 are "don't cares"), and then a Move Encoding. As mentioned earlier, the player enters a character using their keyboard. A 1-8 selects an item from their inventory (in the respective slot, if there is an item in that slot). The Move Encoding then corresponds with the Inventory Encoding for that item (for example, Beer is 0010). A 0 refers to shooting the opponent, which takes a Move Encoding of 0001. A 9 referes to shooting oneself, which takes a Move Encoding of 0000. All together, the data

for a Move Request to use the Handsaw item would be 0XXX0100 (with the "don't cares" taking the form 000).

c) Results and Analysis

The functionality of the client and host logic is verified by analyzing the terminal output of each. We would walk through the game intending to test certain features, noting issues we found and implementing print statements when needed to walk through the status during certain operations. Seen below is an example of the terminal output from the host. Examples of some information given includes the data sent by the host, the turn logic (such as "Starting turn", who's turn it is, and where in the turn the program is, such as "Host received selection"), and other control and flag updates. Similar prints with slight differences in content are implemented on the client and the information is compared.

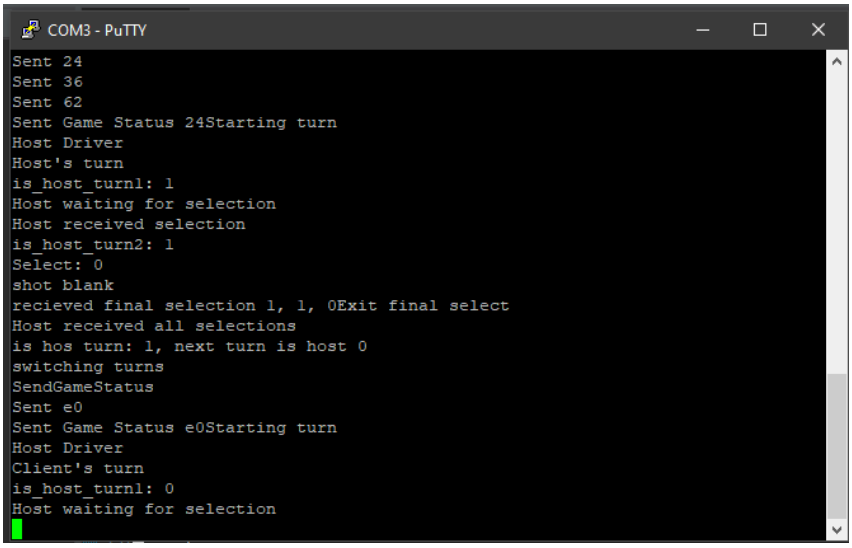

Figure 4: Example of Output to Host Terminal

UART

a) Introduction and High-Level Description

Due to the bidirectional, asynchronous nature of the communications between the client and host, we decided to use UART for these transmissions. These messages allow for the host to send information to the client (such as the client's health, inventory, and turn order), allows the client to make requests to the host (such as to join a game or use an item), and is used to get character inputs from the user as getchar() is blocking. There are two main functions used for sending UART between boards: readwriteUART() and the HAL_UART_RxCpltCallback(). The first function, readwriteUART(), handles sending and waiting to receive UART messages. All sends are done normally while all receives are done in interrupt mode. This allows the receive functionality to be put into the callback function.

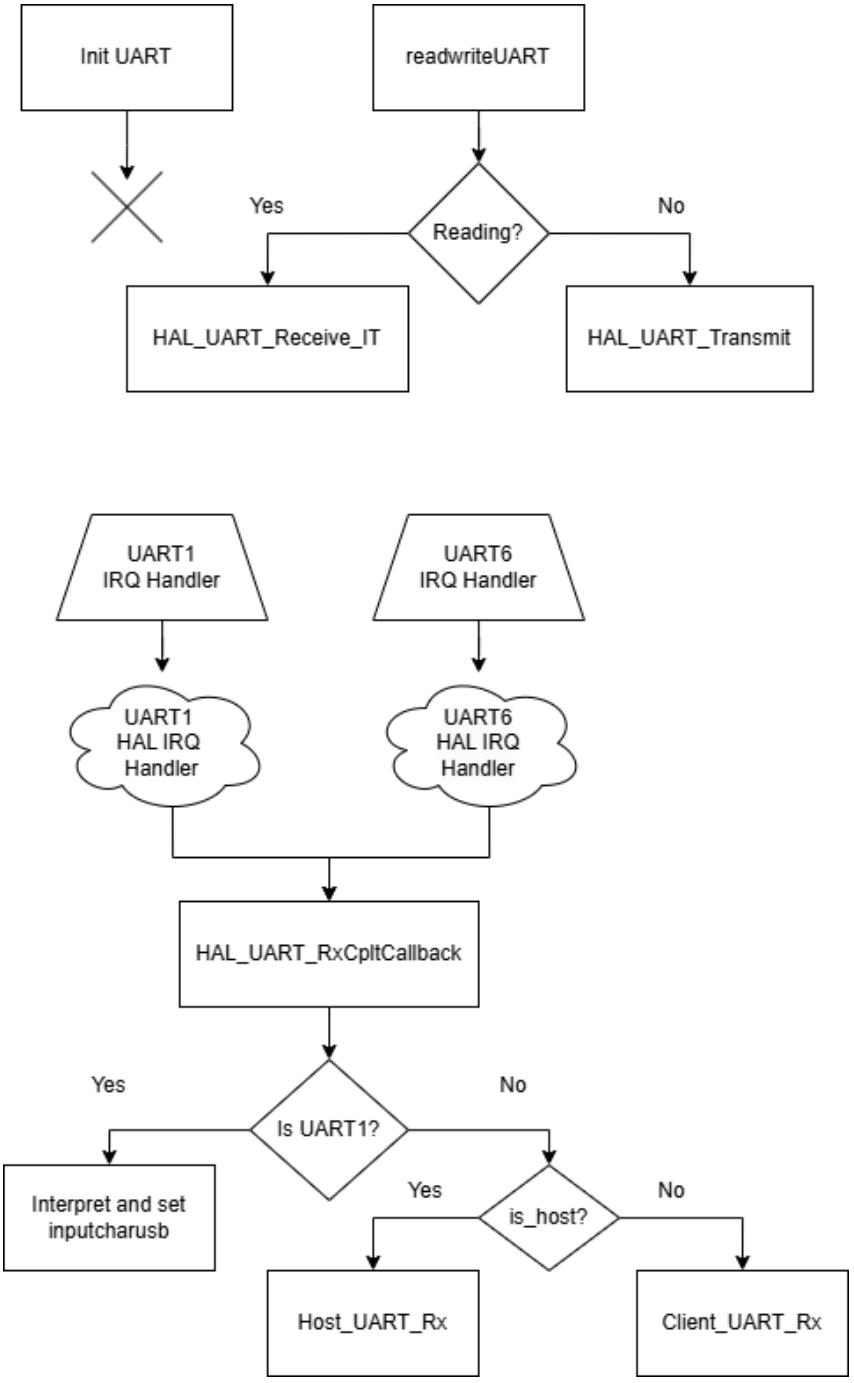

Figure 5: UART FlowChart

b) Low Level Description

First, the interrupts are enabled for both UART1 and UART6. UART1 is used for getting characters from the keyboard and UART6 is used for communicating between boards. THe interrupt handlers simply call the HAL handler. This will then automatically call the callback function.

1. HAL_UART_RxCpltCallback()

The callback needs to handle both instances.

UART1: UART1 refers to characters received from the keyboard, and so the callback must interpret the character and set flags to notify the main loop that a character has been received. First, the character is gathered from the transmission. This character can either refer to an inventory item or choosing to shoot the gun. For inventory items, the character will be a value from 1 to 8. This value is used to get the corresponding item from the inventory and that spot is cleared (so each slot is only used once). This also resets the value of timecounter, as they get the full turn duration for each item. Alternatively, if a shot was selected, the input character is set corresponding to which direction was intended (at self or opponent) and an additional flag, received_final_selection, is set, notifying that the turn should be over after this move. If the character doesn't match any of these values (0-9), it is set to 0xFF, which refers to an invalid selection. For both shots and inventory items, the flag received_selection is set, notifying the main loop that a selection has been made.

UART6: UART6 refers to transmissions received from the other board. This logic is further encapsulated based on if the player is the client or host, so the variable is_host is checked and the corresponding function is called (Host/Client_UART_Rx()).

2. Host_UART_Rx()

Two types of messages can be sent to the host: a join game request, which is sent once at the start, and a move request. These messages are distinguished by the first bit, with a 1 referring to a join request and a 0 referring to a move request. Thus, when a message is received, the first bit is checked. If a join request has been received, the flag in_game is set, which starts the game if it hasn't been already. If a move request has been received, the move is decoded and the inputcharusb variable is set to the corresponding move. This is then treated as if it was received from the keyboard.

3. Client_UART_Rx()

There are 3 types of Game Status Update messages that can be sent to the client. These are interpreted by control variables which are set based on the game state as well as bits within Game Status Update messages. The three messaged types are described further in the low level description of the Client vs Host section. The data of each message is passed into UnpackRecieve(), which takes data and a flag stating if the data should be interpreted as inventory items or not. It then looks at the data received and sets the values of various variables according to what was received.

4. readwriteUART()

This function, readwriteUART(), takes in data and a flag stating whether UART should be written to or read from. Writes are done via HAL_UART_Transmit() and reads are done via HAL_UART_Receive_IT().

c) Results and Analysis

Successful UART transmission can first be validated by capturing a waveform. This can be analyzed to ensure that the correct data is being sent. Then, most of the testing is done by sending the various types of transmissions and ensuring that the proper one is received and interpreting. The simplest to test is ensuring UART1 works by entering a keyboard character and comparing the received character with the one that was input. Then, for UART6, we tested the join request message. Once this worked, we knew that both UART instances were successfully transmitting, so any issues we ran into were more likely to be an incorrect encoding or decoding rather than with the physical transmission. As seen in the figure below, the host terminal prints various information when in debug mode. This information is compared to the actions taken. For example, about halfway through, "shot blank" can be seen, as well as flag status updates. As the player just selected to shoot, we know this is proper functionality.

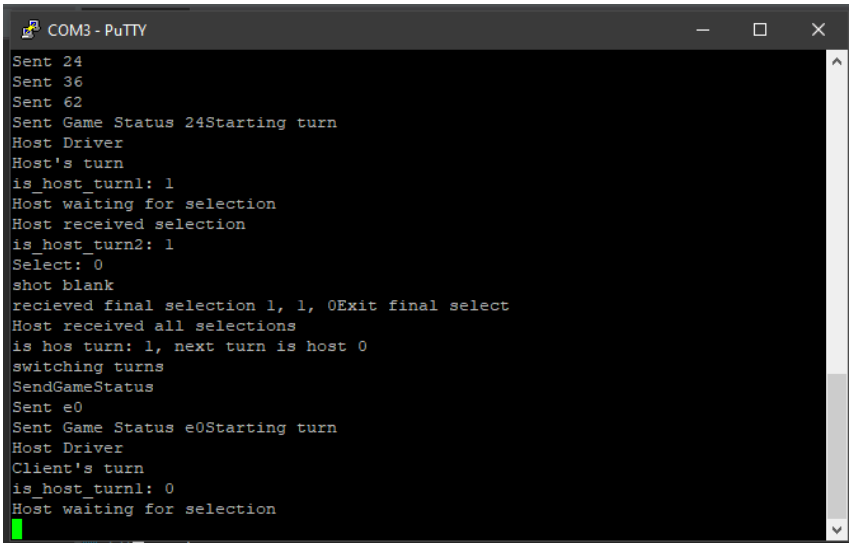

Figure 6: Example of Output to Host Terminal

LCD Screen

a) Introduction and High-Level Description

In order to display the game to the players, we have decided to use the LCD screen in order to display the game state. The HAL functionality for the board provided by STM offers a high level structure to manage LCD hardware, such as initialization functions, layer

management, and ASCII typing. For our use case, all that was used was initialization, drawing functionality, and color management.

Figure 7

b) Low Level Description

As this implementation is relatively high level from the STM HAL implementation, all that is required is to initialize the LCD, a layer and select it, color, font size, and turn on the display [1]. After that, all that is required is to clear the screen and display strings at certain lines.

1. LCD_STARTUP()

This function just calls initialization of the LCD, setting layer and selecting it, setting color and font size, turn on the display, and display the initial screen, prompting the players to choose being host or client, as seen below [1].

Figure 8: Initial Screen Example

2. Display_Lives()

This function shows the number of lives of yourself. This is done through a loop with the players lives and concatenating using strcat, a string function in the string library [2].

3. Display_Bullets()

This function is called every time the shotgun is emptied and the shotgun requries more bullets. This function calls the generation of new bullets and displays the amount of live and blank rounds.

4. Display_Next_Bullet()

This function is called every time the item magnifying glass or beer is used. This will either display the next bullet in the shotgun with the magnifying glass, or eject a bullet and show the type of bullet with the beer.

5. Display_Inventory()

This function is called in Update_Display(). This function will display your own inventory. As this requires the use of multiple lines, it uses loops to determine what needs to be written on each line.

6. Display_Turn_Cuffed()

This function displays whose turn it is, as well as if you or the enemy is handcuffed.

7. Update_Display()

This function calls Display_Lives(), Display_Bullets(), Display_Inventory(), Display_Turn_Cuffed(), and displaying the shotgun. This function is called every turn switch and after an item displays its effects after a timer is elapsed. A few examples can be seen below.

Figure 9: Example of Your Turn

Figure 10: Example of Enemy Turn

c) Results and Analysis

As this is a relatively high HAL implementation, there is not much that can be messed up, other than the interfacing of strings with private variables. A few issues had risen throughout using this. One of those was the use of the string module, as I was not familiar with the strcat function, thus I had some issues with that, as well as how it had interfaced with character arrays, as strings are much more defined in C++, rather than C [2]. Another issue was the use of inventory and having it interface with the LCD. For this abstraction, the use of defines made this much easier, as it made inventory to be made using names rather than numbers that I would have to track as the programmer. Lastly, an issue arose with Update_Display when calling it repeatedly, it seemed to be flashing. This can be quickly fixed by attaching it to user inputs so it doesn't seem to be flashing.

To test these functions, all that needs to be done is to call them with modifying their variables and displaying them. This can be easily shown in the figures above.

Timers

a) Introduction and High-Level Description

In order to limit the amount of time a player has for their move, timer 7 was used as a time limit. The time for each turn can be easily set by manipulating one comparison, which we initially set to 15 seconds before expanding to 30 seconds to provide more time to the player. The timer is started by both the client and host at the beginning of each turn and elapses after one-tenth of a second. Both players print the current time to the terminal to indicate how much time is left in the current turn. In addition, the host tracks the number of times the timer has elapsed, and when the turn time limit has been reached, the turn is ended.

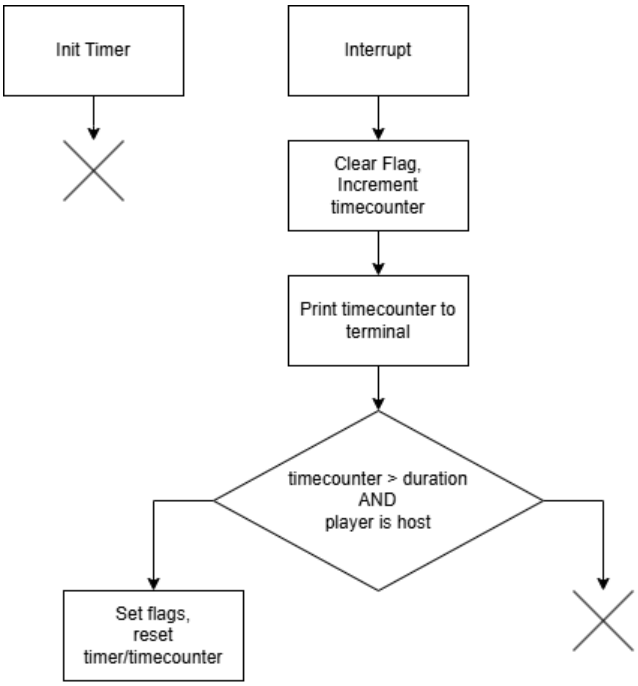

Figure 11: Timer FlowChart

b) Low Level Description

In order to use timer 7 in this way, it has the following configuration. Timer 7 is set to count up with a prescalar of 21599 and a period of 499. This in combination with the corresponding clock speeds leads to a full period of 100 ms. Next, the interrupt request is enabled to allow the timer to be run in interrupt mode. This occurs at the start of every turn, with each player calling HAL_TIM_Base_Start_IT(). Finally, the interrupt handler can be implemented. In this function, the flag is cleared, the tracker variable "timecounter" is incremented, and the current value is printed to the player. Then, if timecounter is greater than the turn limit and the player is the host, flags are set to end the player's turn with the desired move set to shoot the opponent, the timer is stopped, and timecounter is reset to 0.

c) Results and Analysis

The functionality of the timer is tested in two ways. First, the values printed to the terminal are compared with a clock to ensure they are accurate. An example of these values can be seen in the figure below, with the "300" (30.0 seconds) referring to the timer of the previous turn fully expiring (meaning the player ran out of time and their turn ended) and the "49" referring to the current turn being 4.9 seconds in. Second, a player opts to not select a final move for the duration of the turn and the game is observed to see if the end of turn is triggered with the player shooting their opponent.

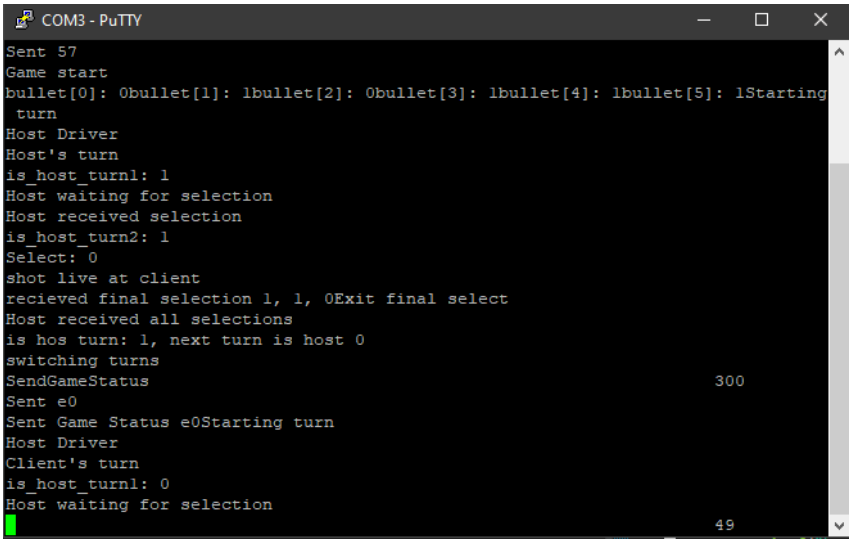

Figure 12: Timer Terminal Output

Random Number Generator (RNG)

a) Introduction and High-Level Description

In order to implement randomness into the game, we decided to use the RNG module. This provides a hardware-based random number generation, using a noise source to create random numbers. The HAL for RNG simplifies the process of initializing and using this peripheral. In order to use this module, the RNG HAL provides the initialization, de-initialization, and random number retrieval [1], [3].

b) Low Level Description

As our game will work fast enough, it didn't seem necessary to use this implementation with DMA or interrupts, thus polling was used to generate new numbers as it is the easiest to use [4].

For our game implement, the RNG is used for item generation and bullet generation.

For item generation, as we only have 6 items to use, an item can be generated using modulo on a generated number. For each item, each is found with the modulo 6, and adding 2 to each to align the item to be easily used with transfer with the client and host.

For bullet generation, an RNG request is first generated to determine the amount of bullets to be used. This is done by again using module 7, and adding 2 to find a number between 2 and 8. After this, another RNG request is made, and a function will count the first chose amount of bullets, using the binary of the number, and if it finds at least 1 blank and 1 live round, which corresponds to 1s and 0s, it will return with that order. Otherwise, it will shift over by one digit and check again. In the off-chance that it doesn't find a single combination, RNG will generate another number and start over again.

c) Results and Analysis

As for any issues with RNG, no problems had arised from it, as RNG was very easy to work with when polling. To implement all other functionality that we had wanted to implement, all this takes is a knowledge on basic C coding, which can be easily done without any bugs using unit testing.

As RNG doesn't have much moving parts, all that needs to be verified to work is the bullet generation and item generation.

For verification of the bullet generation functionality, this can only be tested by letting it run and checking all cases and if it violates any rules. Through testing, it has been found that it does not violate any rules, which include at least 1 live, at least 1 blank, and range between a total of 2 to 8 bullets. An example can be found in figure 12.

For verification of the item generation, this can be tested similarly, though it doesn't have to be nearly as rigorous, as it only can generate 6 possible items. An example of the generation of random items can be found in figures 9 and 10.

Conclusion

The implementation of Buckshot Roulette on the STM32F7 micro-controller utilizing a client-host architecture demonstrates the versatility and power of the STM32F7 series in handling embedded applications. This includes the utilization of UART, LCD Screens, Timers and RNG. This project showcases the ability of the micro-controller to manage real-time communication, data processing, and player-program abstraction. The client-host architecture allows for the host to act as the central controller, orchestrating game logic, random number generation, and and rule enforcement. Clients interface with the host to receive game states and send user inputs, ensuring a responsive and interactive experience. Using UART allows for UART to facilitate this client-host communication. STM32F7's ability to interface with LCDs allows the implementation of an intuitive user interfaces. The current implementation allows for future iterations of this project to include additional items as well as tuning and balancing game mechanics easily. By combining robust hardware capabilities with efficient software design, this project demonstrates the micro-controller's potential.

References

[1] "Rm0410 reference manual," 2018. [Online]. Available: https://www.st.com/resource/en/reference_manual/dm00224583-stm32f76xxx-and-stm32f77xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[2] "Strcat() in c," [Online]. Available: https://www.geeksforgeeks.org/strcat-in-c/.

[3] "Description of stm32f7 hal and low-layer drivers," [Online]. Available: https://www.st.com/en/embedded-software/stm32cubef7.html#documentation.

[4] *Mastering STM 32*. 2018. [Online]. Available: https://www.embedic.com/uploads/files/20201008/Mastering%20STM32.pdf.