Introduction

The basic idea for the project is quite simple: a less complex version of the game Pop-It. The game presents the user with basic arithmetic problems based on the set difficulty. Every 4 rounds, the user must either: clap, press one of the joystick buttons, or shake the device. Doing any of the previously mentioned actions counts as a wrong answer. User is rewarded with a 10 second time increase (game time set by user at start of game) and a pleasant sound when answered correctly. If the user answers incorrectly, an unpleasant sound plays. Numerical answers and problems are provided and displayed through the computer terminal. A beeping sound indicates when time is about to runout by increasing in both frequency and duration.

The most interesting part for this project is the methodology used to develop the game. Starting from the bottom down, specific hardware handlers are used to take input from Joystick, terminal (UART), and onboard gyroscope, and sound recorded. For output, there are handlers to display characters on the LCD screen, terminal, and play sounds. The handlers are essentially just wrapper functions/files which utilize either the BSP drivers or HAL drivers provided by STM but abstract most of the setup complexity or simplify the input to the drivers. On top of these are the specific game mode threads (one for each targeted input, so either shake, clap, press joystick, or input math problem answer). These threads communicate through a single message queue to determine which input was provided by user first. An additional thread is used to handle sounds to be played during the game. Lastly, a thread is used to coordinate all threads, starting them all when an input is required, and stopping all when an input was received.

Development Strategy

The basic premise taken here was to implement one functionality at a time and test it out to prevent it messing with future features. The following timeline shows the order in which I originally wished to develop the game:

Week	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Hardware Handlers	SAI	DFSMD	Gyro	LCD	Joystick	UART	Initializing
Communication and Timers	Thread Msg Queue	Sound Msg Queue	Game Timer	Sound Timer	Debugging		
Threads	Math Mode	Shake Mode	Joystick	Clap	Sound	Game Manger	Debugging

While this was the original plan, outside sources (interviews and outside jobs) got in the way and the actually development did not start until about the seconds week so more than one of these activities were done in a single day (leading to subpar performance). This design is not good because I ran into a lot of bugs while implementing the threads. These bugs included, threads not communicating correctly or terminating immaturely (no input was ever given), faulty UART communication, noisy sound output (probably related to clock configuration), just to include a few.

To debug these problems, I would run just one tread at a time and make sure it worked correctly. This included correctly utilizing its hardware handlers and taking in input. Usually the bug was at the handler level so it would take lots of digging in the reference manual or talking to the professor. Next, I would make sure the thread worked well while other threads ran which was where most of the problems arise. Incremental additions of functionality would help with finding the exact root of the cause. If given more time, I would have started much earlier to allow for inconveniences to arise as they always do.

Description of Thread UML Diagrams

Thread: Game_Manager_Manager

Hardware Handlers Used: (UART and LCD)

Procedures:

GetInputDifucultyandTime())

- Starts and joins thread Input_Mode_Manager. This thread terminates by itself after it puts into a message queue the desired difficulty and game time.

StartGameTimer()

- Takes user input time and starts Game_Timer with desired timer.

Looping Procedures (Until Game_Timer expires**)**

Generate_Random_GameMode()

- Generates random targeted game mode:
  (0 – MATH, 1 – CLAP, 2 – JOYSTICK, 3 - SHAKE)
- If Joystick is the target mode, generates random button to be pressed

Start_All_Threads()

- Starts all threads with their required parameters:

  (Math: Difficulty, Clap: Time Left, JoyStick: Target Button, Shake: NULL)

Display_Game_Mode():

- Show target game mode on LCD screen

WaitForMessage()

- Waits for any message to be put on Queue

Adjust_Time_PlaySound()

- If correct answer was given and thre target thread set the message:
  - Play correct answer sound (push sound to sound queue)
  - Increase time left by 10 s
- Else:
  - Play wrong answer sound

Thread: Input_Mode_Manager

Hardware Handlers Used: (Joystick and LCD)

Procedures:

GetInputDifficulty():

- Display target difficulty on LCD screen and take input from Joystick to change difficulty
- Waits for center button to be pressed

GetInputTime():

- Display current time setting on LCD screen and take input from Joystick to change difficulty
- Waits for center button to be pressed

PushGameTimeAndDifficulty()

- Pushes the desired difficulty, time setting, and Thread Id (message mark) onto queue
- Thread Terminates right after this procedure

Thread: Math_Mode_Thread

Hardware Handlers Used: (UART)

Procedures:

RandDecimalDigit()

- Generantes a random decimal digit. This method is ran 2 times to gerate 2 1-digit numbers if difficulty is set to easy, 4 time to generate 2 2-digit numbers if set to medium, or 6 times to generate 2 3-digit numbers if set to hard.

Transmit_Problem():

- Show problem on terminal using UART Handler

WaitForAnswer():

- Calls to clear UART receive buffer
- Waits for just one byte to be received. It loops one time if difficulty set to easy, 2 if medium, and 3 if hard.

PushAnswerToQueue()

- Check if answer matches the correct one and sets corresponding message on queue (actual playing of sound is handled by game manager that reads message after thread exits)
- Thread terminates after this

Thread: Clap_Mode_Manager

Hardware Handlers Used: (DFSDM)

Procedures:

ClearRecordings()

- Uses DFSDM handler to clear the recording buffer

WaitForSound()

- Method in DFSDM contantly calculates the maximum sound heard and sets a flag when it crosses a threshold in DFSDM handler

PushAnswerToQueue():

- Once sound threshold is crosses, the thread pushes a correct answer message onto queue and attaches ClapMode Id to message.
- Thread Terminates after this.

Thread: Joystick_Mode_Manager

Hardware Handlers Used: (JOYSTICK)

Procedures:

ClearLastButtonPressed()

- Class method in Joystick handler to clear last button pressed flag

WaitForButtonPress()

- Waits for any button to be pressed and captures value

PushAnswerToQueue():

- Method checks if button pressed matches that which the initial caller (Game_Mode_Thread) requested and pushes corresponding message on queue
- Thread Terminates after this.

Thread: Shake_Mode_Manager

Hardware Handlers Used: (Gyroscope)

Procedures:

WaitForDevicetoMove()

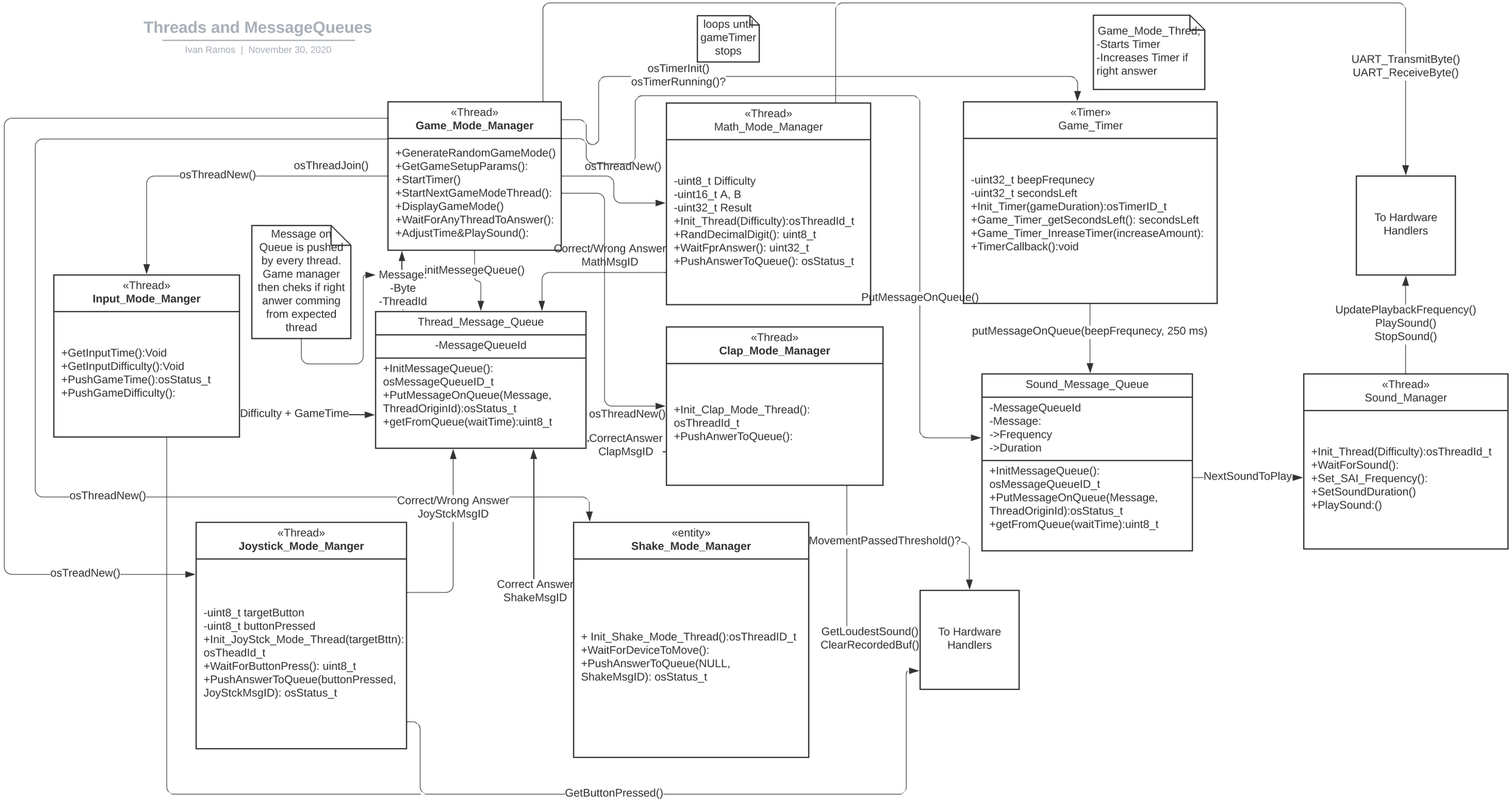
- Uses Gyro Handler to see when movement passes a certain threshold

PushAnswerToQueue():

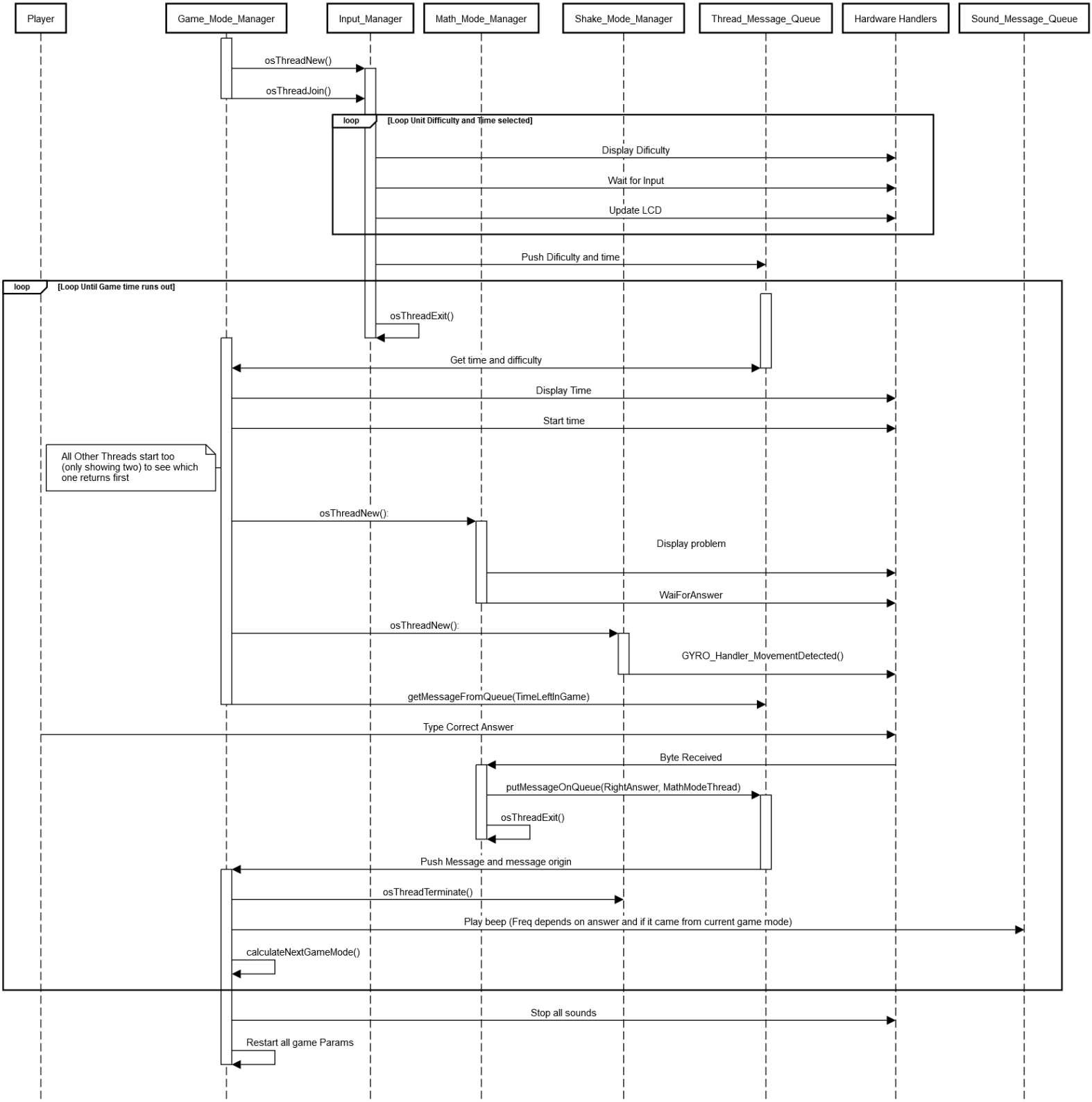
- Method pushes correct answer onto queue and attaches ShakeMode Id
- Thread Terminates after this.

Threads and MessageQueues

Ivan Ramos | November 30, 2020



Thread Intractions



MessageQueue: Thread_Message_Queue

Description: Used to see which thread terminated first and if the answer provided by user was the expected one. Game Manager justs waits on this queue after a certain game has been picked. A message for this queue includes a flag (correct or wrong answer), and an ID for the thread that pushed the message onto it.

MessageQueue: Sound_Message_Queue

Description: Used to handle the next sound to be played by Sound_Manager thread. Any thread can request a sound to be played. A message in here includes the requested frequency and duration of the sound.

Timers: GameTimer

Description: Used to control for how long the timer runs and to play a beep every second (in callback). The callback also increments the requested frequency after every second and loops back to a base frequency after it reaches it's highest value (N = 40 for PLL in SAI).

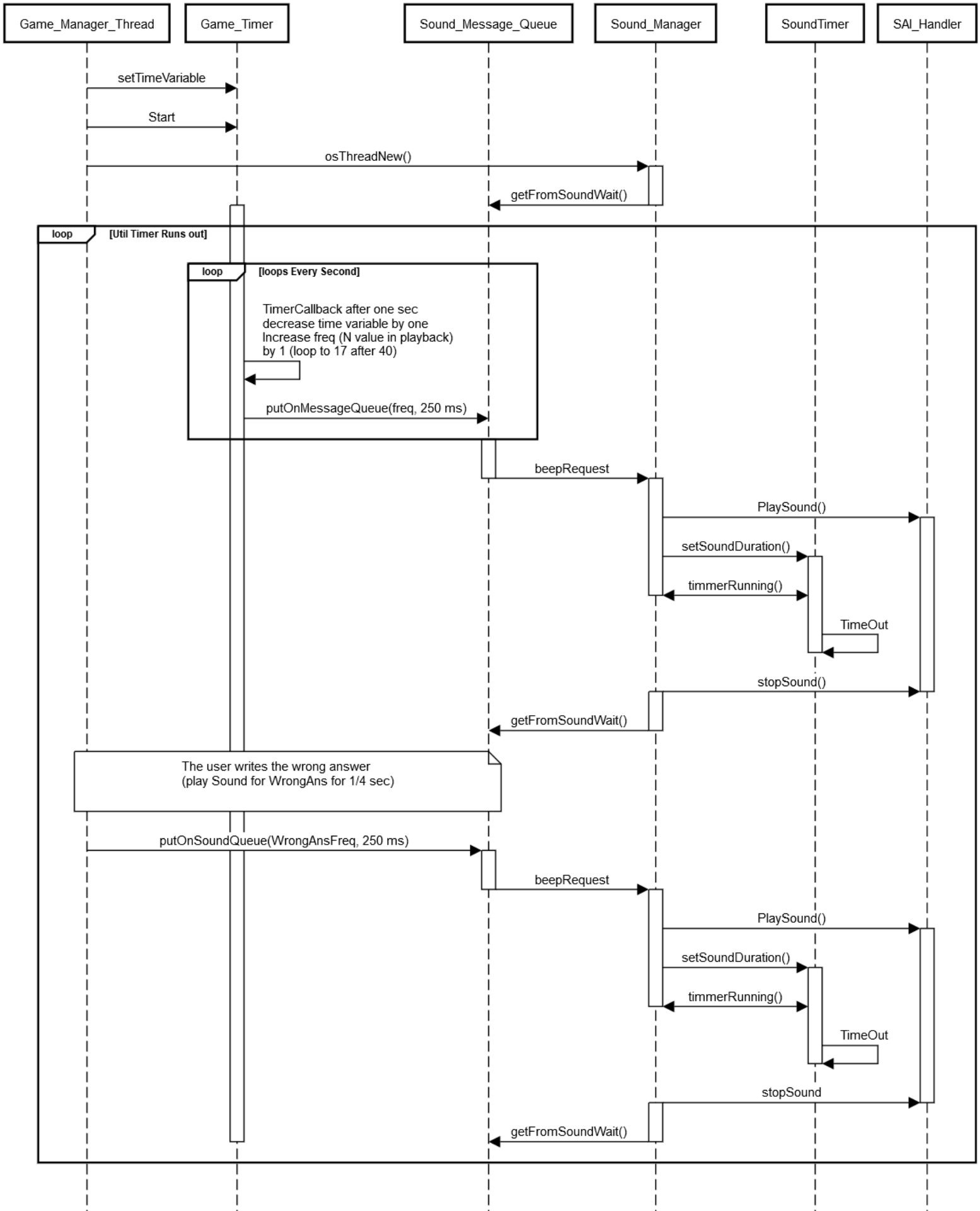
Timers: SoundTimer

Description: Used by SoundManager Thread to set a desired sound duration. Only the SoundManager can push messages onto this queue.

Challenges For Using Threads:

Threads are very hard to debug. It is hard sometimes to synchronize them and they can break other things in your program. If they do not run continuously sometimes, they do not work properly, or some bugs do not appear until other threads are also running. What helped me was using break point in multiple threads to see if maybe we were not terminating them correctly, sometimes logging was necessary to see behavior for continuous run. This is definitely a very powerful for of programming. I did not have to think much about timing or alternating between different input methods, but it does take some time to get used to this form of programming. I'll definitely try to finish the project on my own time over the break and make sure it works as intended.

Sound Manager and Game Timer



Peripheral Descriptions

LCD_Handler

- This is essential just a wrapper class for the BSP_LCD_GLASS file. It used the methods included in the BSP file to:
    - Display Time: Takes in an array of numbers and displays them in order
    - Display String: Takes in an array of chars, the size of the array, and displays all of them in order
- No interrupts are used, just a flag needsUpdate, to indicate when when we should update the time provided (array changes but display does not)

SAI_Handler

- Very similar to the one used in the SAI_Example provided by STM (All initialization is the same)
- A single buffer is used and we can change the frequency of the sound through a public method
- Methods:
    - SetFrequency(): Updates the playback update
    - SetVolume(): Updates volume in the playback update method
    - PlaySound: Starts DMA transmit of sound
    - StopSound: Stops DMA transmit of sound
- This Handler implements the same callback interrupt as the one in the demo to continuously play a sound

JOY_Handler

- Also a wrapper class build around the BSP_JOYSTICK provided by STM. From the file, this class uses the initialization methods.
- Uses interrupts to set a variable that indicates the last button pressed.
- Methods:
    - ClearLastPressed(): clears last button pressed variable
    - GetLastButtonPressed(): just a getter for the last button pressed variable

GYRO_Handler

- Wrapper built around the BSP_GYRO file. From this file, this class uses the initialization methods and a method that retrieves the x, y and z raw values of the gyroscope.
- GYROSCOPE_THRESHOLD used to prevent small movements being registered by handler
- Methods:
    - Movement_Detetected: calls BSP_GYRO function to retrieve x, y and z values, takes absolute value, and returns a 0xFF if it crosses the set threshold.

UART_Handler

- Utilizes the HAL drivers to initialize UART communication. It uses the setup provided by STM32Cube4 which provide me with the initialization, MSP and MX Init functions, and an easy to use interphase to set baud rate and other settings.
- The handler does not use any interrupts, instead, it uses methods to call the HAL_UASRT_Transmit and Receive drivers provided in the HAL drivers. This way I knew when to expect a response from the user and when to update the display.
- Methods:
    - ReceiveByte(): Waits until a single byte is received
    - Transmit(): Sends a single byte to the terminal

DFSDM_Handler

-Set up in much the same was as the DFSDM demo in the STM playback demo. The demo provided a way to easily initialize the device and the filter. It uses a callback interrupt to continuously write to the recorded value buffer (writes over the instead of stopping the recording)

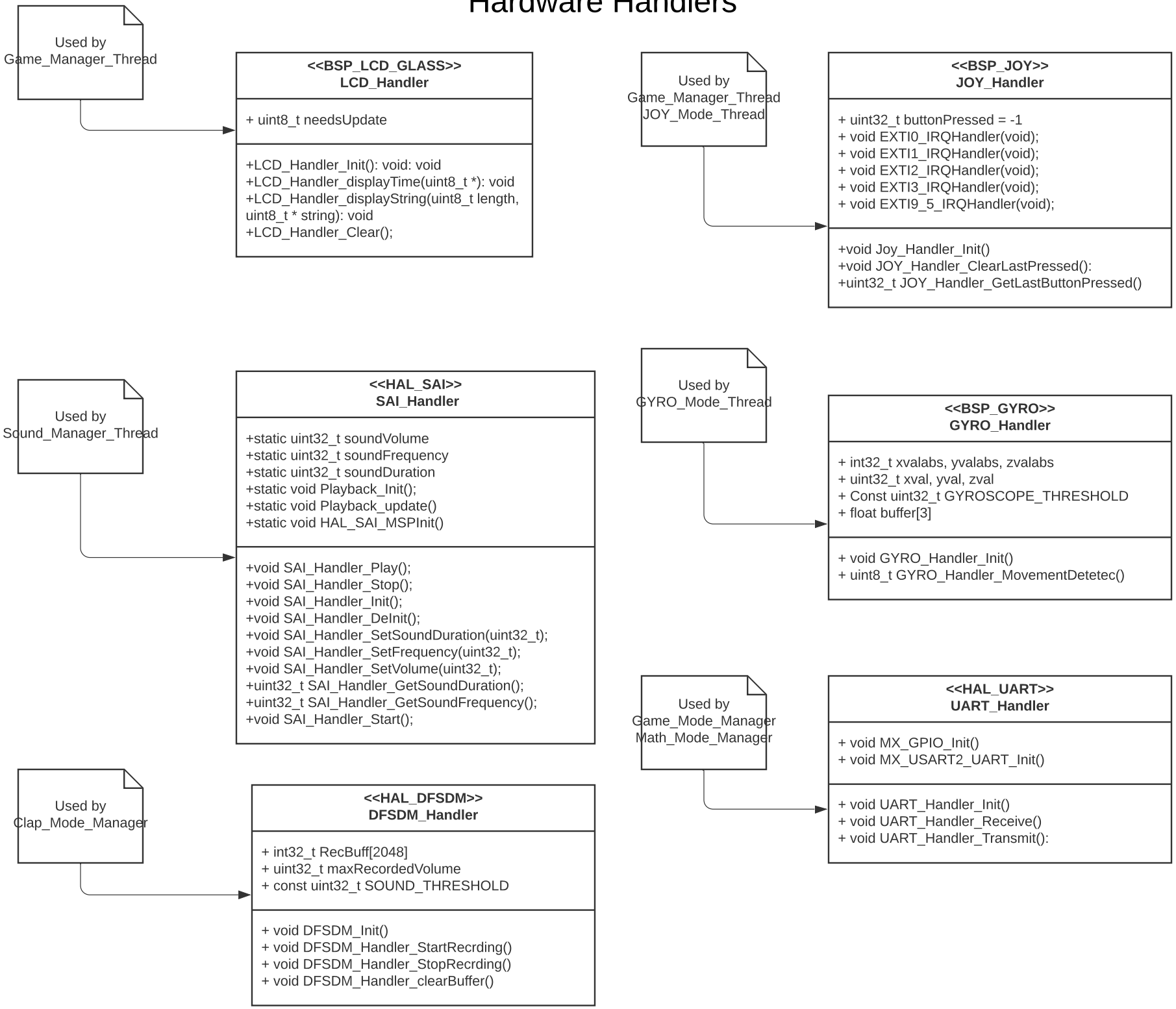
- Methods:
    - DFSDM_Handler_StartRecording()
        - Uses the HAL_FilterRegularStart_DMA driver to start recording into the recording buffer.
    - DFSDM_Handler_StopRecording()
        - Uses the HAL_FilterRegularStop driver to stop recording
    - DFSDM_Handler_GetLoudestSound():
        - Returns the largest value currently inside the recording buffer
    - DFSDM_Handler_SoundPassedThreshold()
        - Returns 0xFF if the loudest sound in the recording buffer crosses a threshold set inside of the handler
    - DFSDM_Handler_clearBuffer()
        - Clears the recording buffer

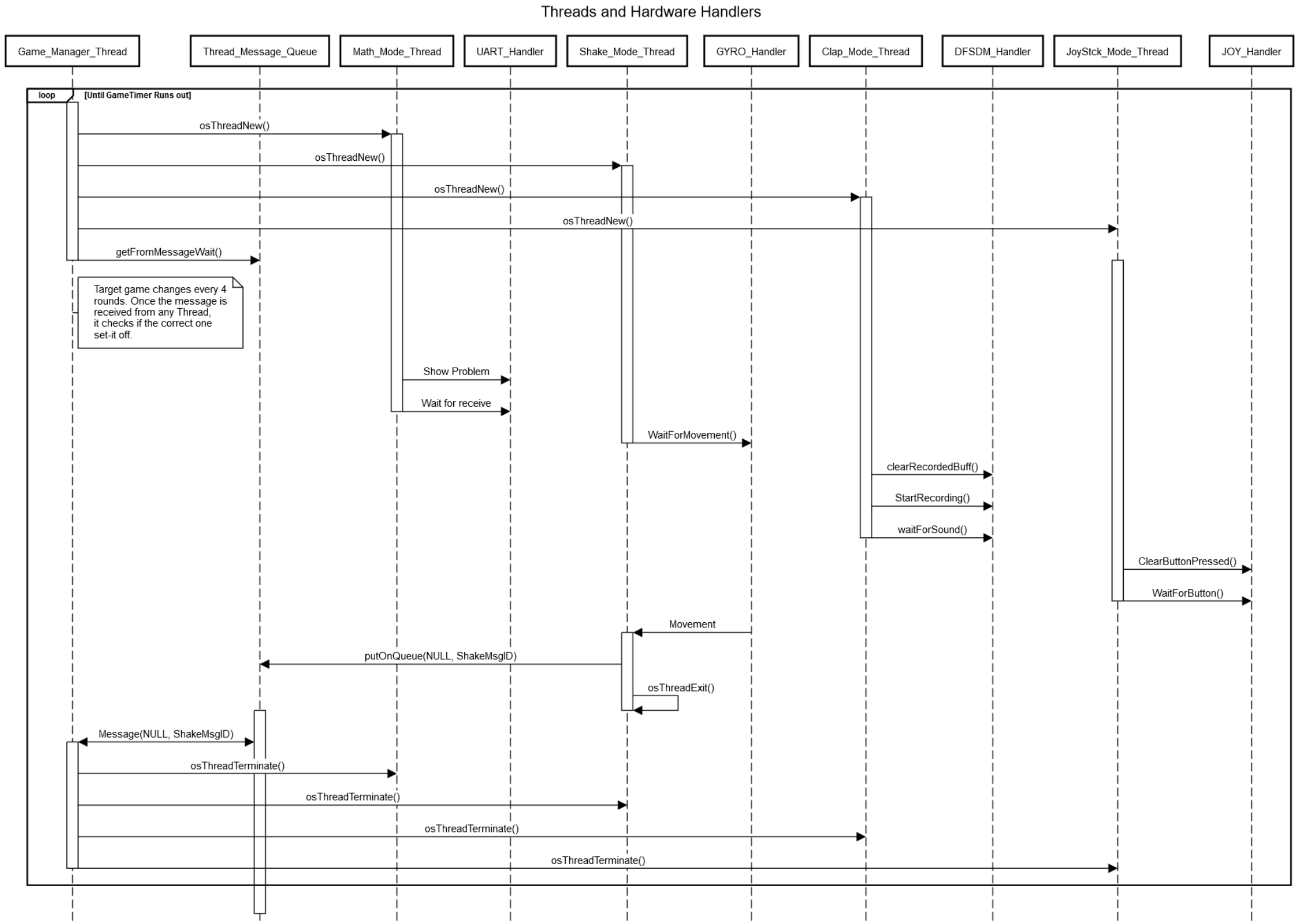
Challenges While Implementing Hardware Abstractions

Some devices are much, much easier to implement than others. The STM32Cube4 helps but it does not allow you to make small changes unless you understand the under-workings. The biggest challenge was the UART and the sound recording since we did not had a required homework for any of these two. Timing issues is also a big hurdle. The more peripherals you include, the harder it is to keep track of which one uses what and adding another peripheral can sometimes mess with the system clock timer. An interesting thing I found was that you can essentially replace the HAL_Delay function with osDelay and everything (seemingly) works just fine. The biggest hurdle and what took the most amount of time was getting the RTOS working. It took some digging around the STM32 packages to find the required source files since for some reason the package provider could not find all the required files.

Looking back to the beginning of the semester, I see that directly using my own drivers for interacting with the hardware is much easier for some devices (not the LCD or SAI or DFSMD)

Hardware Handlers




Threads and Hardware Handlers
Game_Manager_Thread
Thread_Message_Queue
Math_Mode_Thread
UART_Handler
Shake_Mode_Thread
GYRO_Handler
Clap_Mode_Thread
DFSDM_Handler
JoyStck_Mode_Thread
JOY_Handler
loop
[Until GameTimer Runs out]
osThreadNew()
osThreadNew()
osThreadNew()
osThreadNew()
getFromMessageWait()
Target game changes every 4 rounds. Once the message is received from any Thread, it checks if the correct one set-it off.
Show Problem
Wait for receive
WaitForMovement()
clearRecordedBuff()
StartRecording()
waitForSound()
ClearButtonPressed()
WaitForButton()
Movement
putOnQueue(NULL, ShakeMsgID)
osThreadExit()
Message(NULL, ShakeMsgID)
osThreadTerminate()
osThreadTerminate()
osThreadTerminate()
osThreadTerminate()

so what I'll try to do in my own time in the future is replace the UART drivers with my own (unless this proves to be too much trouble though.)

Current Bugs and Plan to Debug

As mentioned at the beginning of the report, the program currently has multiple bugs which keep it from operating correctly. In this section I provide some insight into how I would have planned to debug the problems:

1. Gyroscope keeps giving false positives.
    a. For some the gyroscope handler keeps returning before I move the device. This might be because I do not restart the gyroscope after every single run or maybe the buffer maintains it's previous values from when the thread was last ran. I would first try to run the Gyro thread by itself, and then see what the buffer holds after every run.
2. UART does not stop and wait for a receive value.
    a. We had a debugging session so I knew what the problem might be (incorrect MSP initialization). This should be an easy fix.
3. Random number is always just zero:
    a. Need to look into how the lfsr method we talked about during the debugging session works. Right now it is only return 0 but I believe all I have to do is right-shift the result by 8 bits.
4. Sound Manager does not correctly get the message from queue:
    a. This one is a treaky one because the other message queue (the used for threads) works just fine and the sound queue is essentially a copy of it (compiles fine so multiple definitions). This is probably just a problem with the initialization or the fact that I am calling it from the gameTimer callback function.
5. No rewarding sounds:
    a. This is just rooting from bug No. 4. Once I get the queue working I should be able to get the tones working.