Real-Time and Embedded Systems

Project 03: Bank Simulator

Author: Liam Sullivan
Email: lds7381@rit.edu
Submission Date: 10/24/2022

Analysis and Design

This exercise used FreeRTOS on a STM32 microcontroller to simulate a full day of banking transactions. The bank consisted of three tellers and customers that would come into the bank at random times and need to have a transaction with a teller. If all of the tellers were full the customers would have to wait till the next teller is available to be serviced. The transactions between the tellers and the customer take a random amount of time and once they are serviced they will exit the bank. A diagram showcasing how the bank will flow through the day with customers and tellers was created and can be seen below in Figure 1.

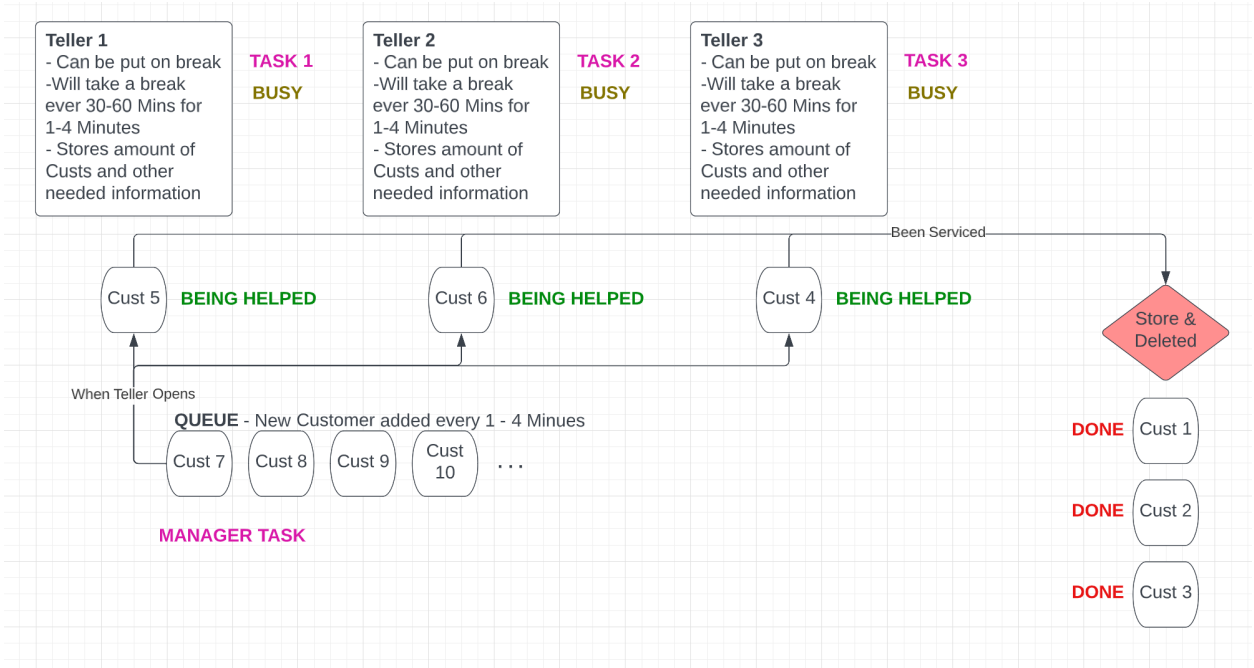

Figure 1: Bank Teller and Customer Transaction Diagram

From the above figure we can see all of the tasks that will need to be performed by the FreeRTOS on the microcontroller (seen in pink in Figure 1). Three separate tasks will be needed for each one of the tellers in the bank as well as a manager task to monitor the day, tellers, and customers in the queue. It can be seen that a queue will be needed to hold the customers in while all of the tellers are full and customers cannot be serviced. These tasks and the queue were created using FreeRTOS api as well as the RNG, UART, GPIO, and TIMER peripherals. All of the tasks were set to the same priority as to make sure one does not have higher priority than the other.

Each one of the teller tasks used their own teller struct. This teller struct, ‘teller_t’, can be seen below in Figure 2 to showcase all of the different data it contains to run the teller task.

```
typedef struct teller_t {
    /* teller information */
    uint8_t  teller_id;
    customer_t current_customer;
    osMessageQueueId_t *queue_handle;
    osSemaphoreId_t *sem;
    teller_status_t status;
    uint32_t button_pin;
    GPIO_TypeDef *GPIO_port;
    uint32_t cust_start_time;
    uint8_t end_of_shift;
    uint8_t break_flag;
    uint8_t force_break_flag;
    uint32_t next_break_time;
    uint32_t break_time;
    uint32_t end_break_time;
    /* Stored information */
    uint32_t customers_helped;
    uint32_t total_help_time;
    uint32_t avg_customer_help_time;
    uint32_t avg_teller_wait_for_customer;
    uint32_t max_wait_for_customer;
    uint32_t start_wait_for_customer;
    uint32_t stop_wait_for_customer;
    uint32_t total_wait_time;
    uint32_t total_customer_queue_wait;
    uint32_t max_customer_help_time;
    uint32_t breaks_cnt;
    uint32_t avg_break_time;
    uint32_t longest_break_time;
    uint32_t shortest_break_time;
    uint32_t total_break_time;
    uint32_t max_cust_queue_wait;
    uint32_t start_force_break_time;
    uint32_t end_force_break_time;
} teller_t;
```

Figure 2: teller_t Struct used for the teller task

The teller task would run based on this information stored in the teller’s struct. The teller mostly performed based upon its current status. This allowed the teller to do its base tasks of getting a new customer, transacting with a customer, and being able to go on break randomly or by button press. When a new customer is gotten it is taken from the created FreeRTOS queue and given a random amount of time to transact with the teller (it's set in the customer’s struct). The created FreeRTOS queue was a queue of customer structs which contained information that can be seen in Figure 3.

```
typedef struct customer_t {
    /* customer information */
    uint32_t customer_id;
    uint32_t help_time;
    customer_status_t status;
    /* Stored Information */
    uint32_t time_spent_in_queue;
    uint32_t queue_start_time;
    uint32_t queue_end_time;
}   customer_t;
```

Figure 3: Customer Struct

This struct allowed the teller to know all information needed to transact with the customer and store their required information. When transacting with the customer the task will go idle until it is time to end the transaction. When going on break the teller will also go idle until the randomly generated break time is done or until the corresponding teller button is unpressed. Using the teller struct also allowed each of the tellers to save their own information throughout the day instead of attempting to share statistics with other tellers. Then after the day was done each teller would compare their save statistics to get the final results of the day. The teller tasks will run until the day is over, the queue is empty and they have no more customers to service. A semaphore is then released to signify the task is completed and the task is then deleted.

The singular manager task was used to manage the bank. This task would open and close the bank, monitor the time of the day, monitor the statuses of tellers, Monitor the size of the queue, put tellers on break when it is time, create customers to be put into the queue, and get the final statistics of the day. To open and close the bank this task would get the current time from the timer TIM2 which was set to increment every 100 microseconds. It would then add the amount of ticks equal to “seven hours”, 420000, if 100 milliseconds is equal to 1 minute. This time is the time the bank then closes. When the bank opens customers can start being added into the queue and the bank being open is printed to the terminal using UART. When the time is greater than the close time the bank closes and no more customers are added into the queue. However, the bank will not fully close until all of the tellers are done servicing their customers and no more customers are left in the queue. This is done by acquiring semaphores from each of the teller tasks, this makes it so the bank cannot close until the teller tasks have completed transacting with their current customer and customers still in the queue. This task would monitor the time of day, the size of the queue, and the statuses of the tellers. It did this by printing this information out to UART. The time was gotten by converting the timer time to the timer of 7AM - 2PM, the queue size was gotten from the FreeRTOS api, and the statuses of the tellers were gotten using the pointers to the tellers. The printed UART monitor of the day can be seen below in Figure 4.

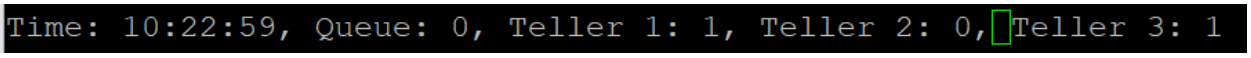

Figure 4: UART Day Monitor

All of the information needed for the monitor was sent into the manager task as a part of the parameter that took in a manager information struct. Along with the UART monitor, the queue size was also shown using a seven segment display attached to the microcontroller. This manager task also came up with the random intervals that the managers would go onto break. This was done by setting the break flag of the teller using the teller's pointer to its teller struct when it was the generated time for the teller to go onto break. The same procedure was performed when the corresponding break button was pressed as well, just with a forced break flag instead. Each type of break's length was saved as well as counted for the overall break count. For this task to add customers into the FreeROTS queue the task would generate a new customer struct at random intervals equal to a simulation time of every 30 seconds to 8 minutes. When these customers were added to the queue they also saved the time that they entered the queue as to save their time they spent in the queue. Finally, once the day was completed the final statistics of the day are printed out. This is done by taking all of the information that is stored in each of the teller's struct's and comparing them with one another. This allowed the task to get and compare all of the information of the day for all of the customers and tellers. This information is then printed out to UART, this can be seen below in Figure 5 which showcases a full day simulated.

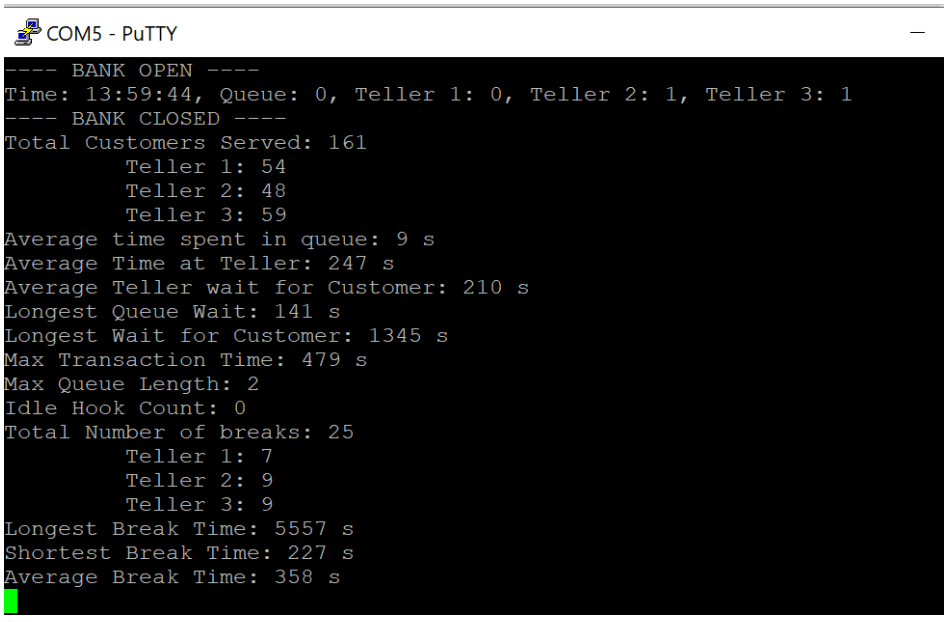

Figure 5: Full Simulation Day UART Output

In Figure 5, a successful full “seven hour” day simulation can be seen. This simulation in real time took a total time of around 42 seconds which is what the expected time of this simulation was calculated to be. From this we can see the bank opening and closing, then printing out all of the statistics that were collected throughout the day. All of the statistics printed above in Figure 5 are the necessary statistics that were needed for this simulation. All of these values are within the ranges they were expected to be in and showed correctness in the reporting (One force break was taken, large longest break time, and idle hook count could not be implemented correctly in time).

To create random values for the customer creation, customer transaction times, and break times a RNG peripheral on the STM32 was used. This peripheral was enabled and allowed the program to get a random bitstring of 32 bits from the RNG’s direct register. This random 32 bit was then used in a function that took in a max and a min range and scaled the random 32 bit unsigned int into the range. This allowed the program to be able to get any random number within a range whenever needed. This made it possible to have variation in how each one of the days is simulated and allows different statistics to be gotten on each run.

Test Plan

One of the biggest tests that had to be done was that all teller tasks would be able to pull customer structs from the created queue. This test had to be performed to ensure that all of the tellers in the bank would be able to pull from the queue to be able to perform each customer’s transaction. For this test to be successful all of the tellers would need to be able to pull a customer from the queue, wait a random amount of time then get another customer all while the other teller tasks and manager tasks were running. This is the base of how the bank runs and allows the tellers to be able to smoothly pull from the queue and transact without issue. When this test was first being performed onto my code, it was not performing as expected. The queue would fill to its maximum capacity prior to any of the tellers would pull from the queue. This test showed that the manager task was set too high of a priority and would be the only task running until the queue was full and the task went idle. This test made me realize how the priorities for the tasks in FreeRTOS were set up incorrectly and that all of the tasks in the program needed to be running on the same priority or one task would be performing solely until idle.

Another test that had to be performed during the creation of this project was testing the buttons of the shield on top of the STM32 microcontroller. When the forced breaks were implemented into the program that would force a teller to take a break if their corresponding button was pressed only one of the buttons was recognized as being pressed. To test the button

presses a function that checks the teller's corresponding button's GPIO input was created. From this function it was seen only the teller 2's button was functioning. Using the debug of the STMCUBE IDE, it was seen that the wrong ports were being sent to teller 1 and 3. After this was fixed the buttons were performing as expected. However, testing that these buttons were working as expected instead of trying to just implement them saved a lot of time debugging and attempting to find where in the code the button press was going wrong.

A final test that was always performed after a simulation was performed was just a common sense test on the printed statistics. When the statistics were first being created the way I was displaying the time of the statistics was incorrect. This led to statistics that would have been impossible to get. Such as the longest queue time being shorter than the average or the longest time at the teller being longer than 8 minutes. Being able to know what some values should be or having an expected range allowed the statistics to be tested after every single run to make sure all parts of the program were running as expected.

Project Results

Using the design and tests that were talked about above the resulting simulation was able to simulate a full day of transactions and successfully save all of the statistics. When the simulation is run it can be seen to average around 150 customers being served between all of the tellers for a full day. This is within the expected range of customers that will be served in the full day. The day also takes a real time of around 42 seconds to fully run the simulation which is the calculated time for the full simulation of 420 minutes to occur. This ensures that the timing of the program is on time and values used in designing the program are correct. This also ensured that the statistics that were printed out at the end of the day have correct simulation time values. This shows that the simulation can successfully measure timing statistics accurately across the program.

The use of FreeRTOS in this simulation was key to the success of how this simulation is run. Using FreeRTOS allowed the program to schedule the tasks to perform at the times they needed to. This allowed all three tellers to successfully be able to run at the same time and continually add new customers into the queue for transactions to occur. This allowed the bank simulation to be able to have its three tellers as well as a manager. FreeRTOS also allowed for the implementation of a queue that was safe to be accessed by all of the tasks. This made it so all tasks would be able to pull from and add to the queue worry free and easily. The use of semaphores with FreeRTOS also ensured that these tasks were completed correctly and that the bank did not miss any customers that are still being performed and remain in the queue. All of these FreeRTOS features were able to successfully be implemented and have the bank run a successful simulation with all of the features required.

Overall, the simulation is able to run with all of the required undergraduate and graduate features of the project. The design implemented overall was able to successfully complete a full bank day simulation over many tweaks and changes. However, these changes helped create the final design talked about above and led to a successful project that can simulate a full bank day and report statistics of the day.

Lessons Learned

One of the largest lessons learned in this project was the overall use of FreeRTOS. I have seen FreeRTOS prior to this project in my co-op, however, I never worked much with it and it seemed like a scary concept to learn. From this project I learned kinda just how FreeRTOS operates and why it is useful in embedded systems. One of the biggest lessons learned with RTOS was priority management, when I started this project I thought that each task would need a different priority and didn't really understand how priorities worked. After many debug sessions and reading about priorities I was able to realize that all my priorities needed to be the same level and learned a lot about how priorities need to be when performing with multiple tasks.

The use of semaphores was also something that I learned how to use during this project. Prior to this project I thought semaphores were used as almost a notification system and not as a way that was used to protect data. I wanted to force myself to implement at least one semaphore in some way in this project so that I could feel more comfortable about using them. I ended up using it as a way to protect the manager task from ending the day too early. Using it in this way made me see how semaphores are used and made me more comfortable to use them in future projects where they will be a great advantage. Another data protection method learned but not used was mutexes. I had a couple of mutexes in the program but realized I didn't need them with the way I was doing my statistics collection with the teller struct. However, I was able to learn and understand why these would be needed in freeRTOS and how they can be useful in future projects as well.