CG2271 Mini Project Report

Done by: Sean Lim Hao Xiang, Kishore R and Glen Wong Shu Ze

This report explains the Real-Time Operating System (RTOS) architecture and the usage of Global Variables in our mini project.

GLOBAL VARIABLES: The global variables used in this project are shown below.

`rx_data`	Holds the serial data that is being sent from the BT06 module into the FRDM board.
`curr_motion_state`	Enumeration that can hold the value of either stationary or moving.
`has_completed`	Serves as a Boolean flag that only gets updated when the robotic car finishes the maze.

RTOS ARCHITECTURE: The architecture involves 5 tasks and 1 interrupt shown below.

Tasks	**Functionalities**
`tBrain`	Decodes the data from the Serial Port and perform the necessary actions.
`tMotorControl`	Controls the action of the motors.
`tGreenLED`	Controls the green LEDs.
`tRedLED`	Controls the red LEDs.
`tAudio`	Produce appropriate audio tunes based on the location of the robotic car.

Interrupt	**Functionalities**
`UART2_IRQHandler`	Serial data coming in from the BT06 module is captured using this interrupt.

There are 8 unique rx_data that can be sent from the BT06 module into the FRDM board shown below.

`rx_data`	**Functionalities**	**Value**
`DATA_STATIONARY`	Signal that is automatically sent when the user is not pressing any button on the app.	0
`DATA_BLUETOOTH_SUCCESS`	BT06 receives and sends this data when a successful Bluetooth connection has been established.	1
`DATA_BLUETOOTH_DISCONNECT`	-	2
`DATA_UP`	Signals that the user wants the robot to move up.	3
`DATA_DOWN`	Signals that the user wants the robot to move down.	4
`DATA_LEFT`	Signals that the user wants the robot to move left.	5
`DATA_RIGHT`	Signals that the user wants the robot to move right.	6
`DATA_END`	Signals that the user wants the robot to play ending tune.	7

RTOS ARCHITECTURE DIAGRAM

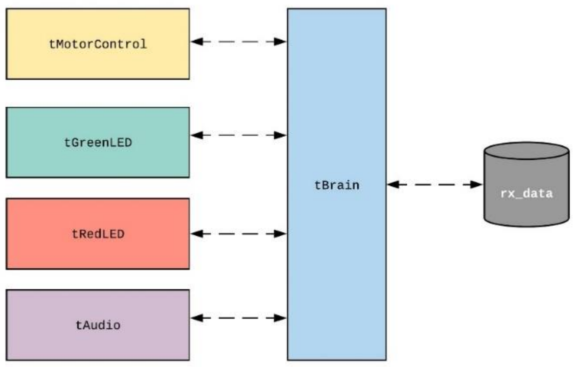

Figure 1: RTOS Architecture diagram of the system

RTOS WORKFLOW

Establishing Bluetooth connection

The diagram is a sequence diagram when user successfully connects to the BT06 bluetooth module.

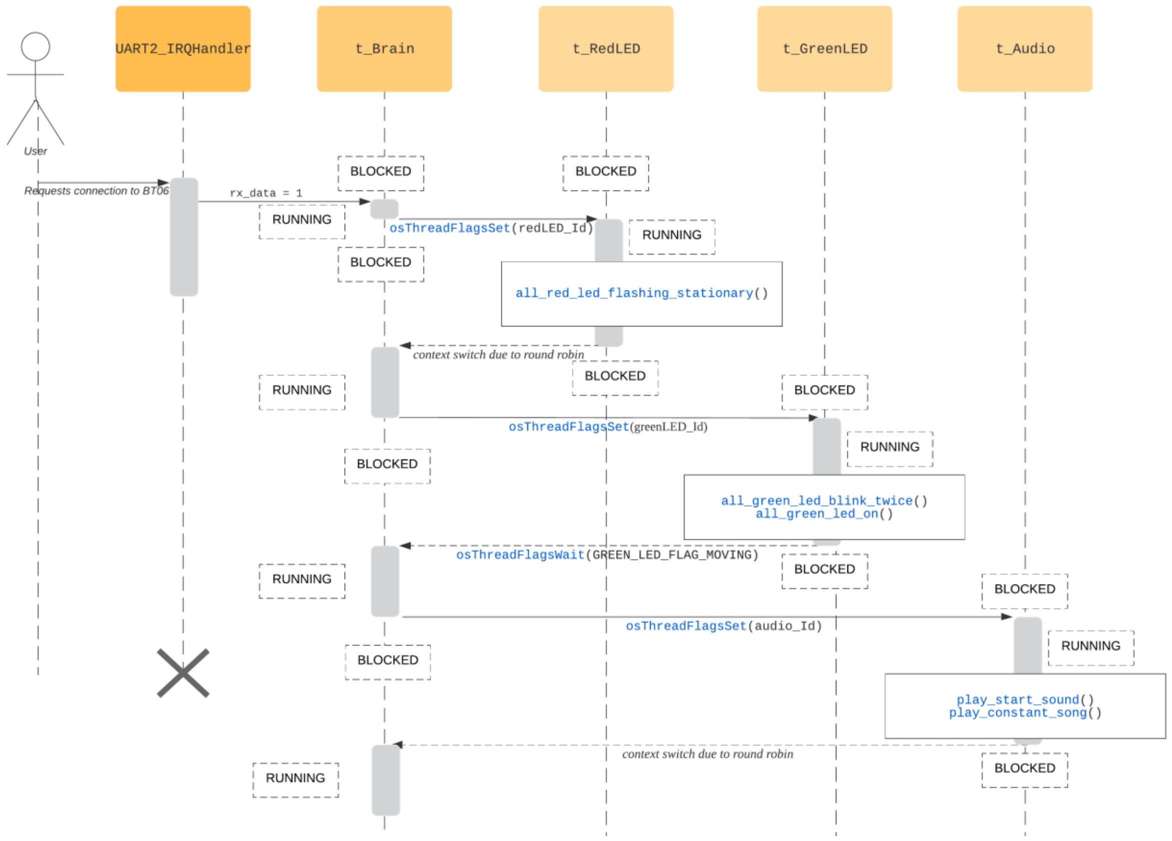
User
UART2_IRQHandler
t_Brain
t_RedLED
t_GreenLED
t_Audio
Requests connection to BT06
rx_data = 1
BLOCKED
RUNNING
osThreadFlagsSet(redLED_Id)
all_red_led_flashing_stationary()
context switch due to round robin
osThreadFlagsSet(greenLED_Id)
all_green_led_blink_twice()
all_green_led_on()
osThreadFlagsWait(GREEN_LED_FLAG_MOVING)
osThreadFlagsSet(audio_Id)
play_start_sound()
play_constant_song()
context switch due to round robin

Figure 2: Sequence diagram when a successful Bluetooth connection is established

Explanation on the sequence diagram

When user connects to the BT06 module via Bluetooth using his Android application, the application sends a UART data of the value 1 into the module. The module then transmits this data into the FRDM board which the board assigns it to `rx_data`. When the OS context switched to `t_Brain`, the updated `rx_data` gets detected and sets the `RedLED` thread flag. This results in a context-switch into the `RedLED` task which will enable the red LED to blink at a rate of 1Hz (or 500ms on, 500ms off). After blinking for 5ms, `RedLED` task context-switches back to `t_Brain` due to round robin scheduling. `t_Brain` will then set the `GreenLED` thread flag which results in a context-switch into the `GreenLED` task which enable the green LED to blink twice followed by turning all the green LEDs on. This task then goes to blocked state since it is waiting for the green LED thread flag for moving and context-switched to `t_Brain`. `t_Brain` then sets the audio thread flag which enable it to play start sound followed by the constant song that will play continuously in the background. Similarly, this process will last for 5ms and gets context-switched due to round robin scheduling.

Movement: UP, DOWN, LEFT, RIGHT

The diagram is a sequence diagram when user presses up, down, left or right on the Android application.

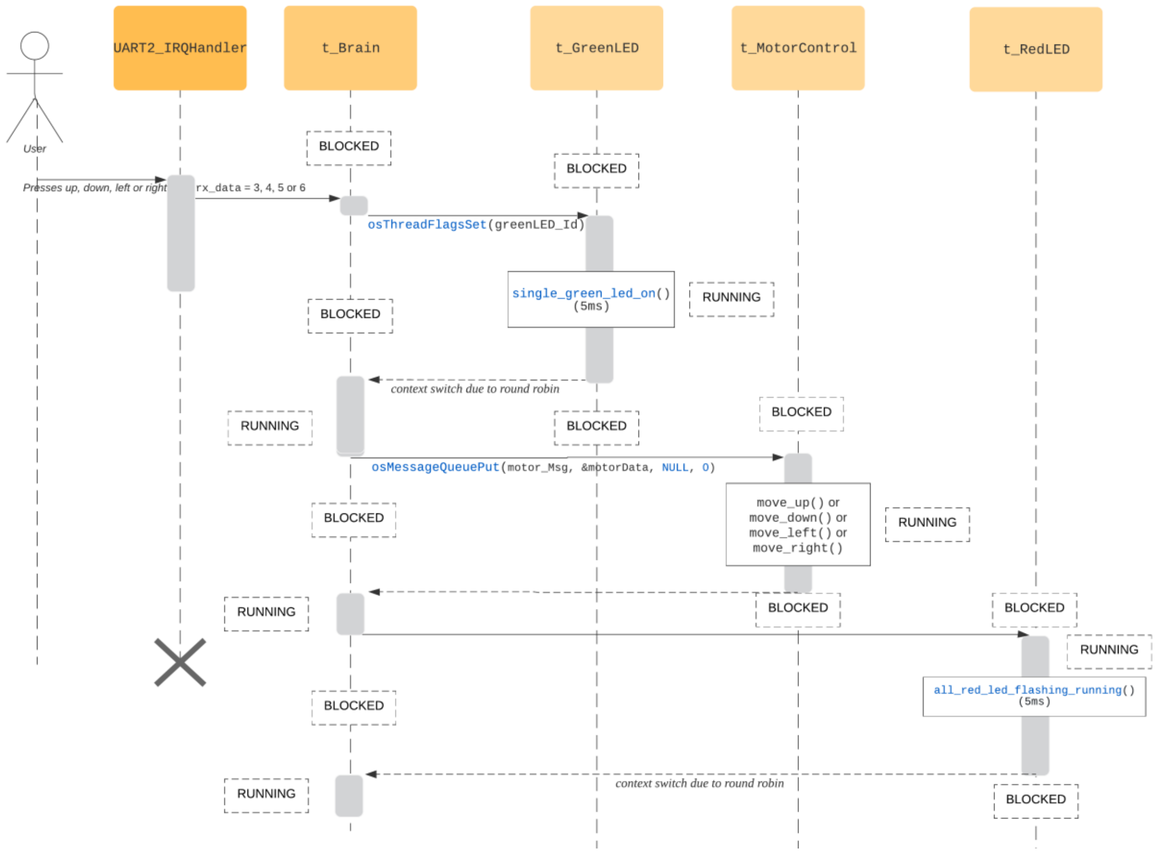
UART2_IRQHandler
t_Brain
t_GreenLED
t_MotorControl
t_RedLED
User
Presses up, down, left or right
rx_data = 3, 4, 5 or 6
BLOCKED
osThreadFlagsSet(greenLED_Id)
single_green_led_on() (5ms)
RUNNING
context switch due to round robin
osMessageQueuePut(motor_Msg, &motorData, NULL, 0)
move_up() or move_down() or move_left() or move_right()
all_red_led_flashing_running() (5ms)

Figure 3: Sequence diagram for movement control

Explanation on the sequence diagram

When the user presses and hold the up, down, left or right button on the Android application, this will send the UART data of 3, 4, 5 or 6 into the module. The module then transmits this data into the FRDM board which the board assigns it to `rx_data`. When the OS context switched to `t_Brain`, the updated `rx_data` gets detected and set the `GREEN_LED_FLAG_MOVING`. This enable the green led to be in running mode for 5ms before getting context-switched back to `t_Brain`. `t_Brain` then sets the global variable `curr_motion_state` to be moving, sets the `motorData` to be the direction that user wants the car to travel and then place this message packet into the message queue. This will result in a context-switch to the t_MotorControl task that will move the robotic car accordingly and context-switched back to `t_Brain`. Due to round robin scheduling, `t_RedLED` task will be scheduled and since the `curr_motion_state` has been updated to be moving, the task will call `all_red_led_flashing_running` which enables the red LEDs to blink at 2 Hz (or 250ms on, 250ms off). It will then context-switched back to `t_Brain` after 5ms due to round robin scheduling.

**Note that the audio thread will get scheduled as well and will be playing the constant song. (Not shown)

Ending tune

The diagram is a sequence diagram when user requests for the ending tune.

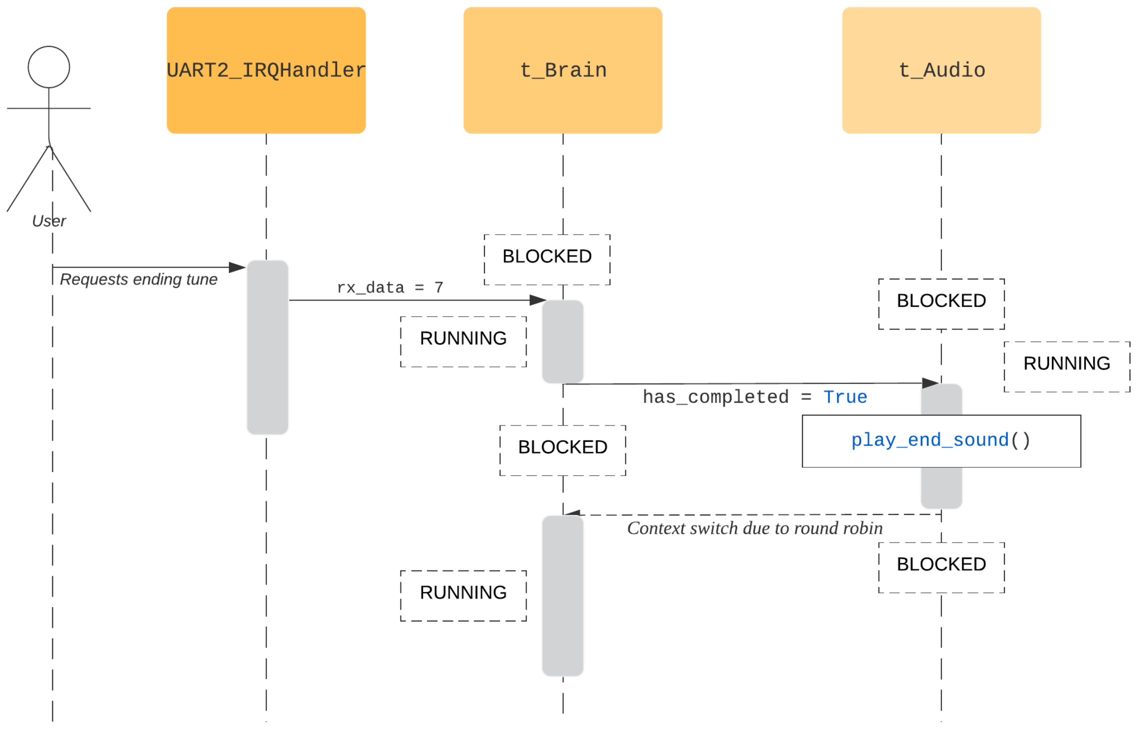

Figure 4: Sequence diagram for ending tune

Explanation on the sequence diagram

When the user presses on the end button to signal the end of the maze, the Android application will send the UART data of 7 into the BT06 module. The module then transmits this data into the FRDM board which the board assigns it to `rx_data`. When `t_Brain` is being scheduled, the updated `rx_data` will get updated and sets the global Boolean variable `has_completed` to true. When `t_Audio` gets scheduled, the updated `has_completed` variable gets detected and the task will call the `play_end_sound()` function which will play the end tune. Similarly, it will get context switched out of `t_Audio` after 5ms.

CONCLUSION

Overall, the use of RTOS helps in improving the predictability and determinism. It allows us to be able to know that the time taken for each task is at most 5ms due to round robin scheduling. However, determining which RTOS components to use was not trivial since all of them are seemingly useful and similar.

We eventually ended up using only thread flags and message queues. Thread flags are especially useful for green and red LEDs and audio since many of the requirements allow needed the task to perform a single task once i.e. playing the start sound once and blinking the green LED twice for one time at the start of challenge. We used message queues to synchronize the different movement of the wheels.

On the other hand, global variables are also useful and cannot be ignored since they helped in the completion of requirements. Using of global variables allow us to complete the requirements without having to create too many threads. For instance, the use of global Boolean has_complete allows us to know when to play the end song without having to create another thread solely for the end song.