Timing Analysis Report

To evaluate the reliability of our system, it was necessary to conduct data analysis and statistical tests to observe and analyse the real-time performance of the system. To this end, our group conducted a series of measurements to determine the time required to complete specific tasks and performed critical instant analysis to verify whether each task could be completed before the respective deadlines. To study the overall behaviour of the system, we collected statistics from the FreeRTOS() operating system to show the time the real-world performance of the operating system.

Methodology

To determine the timing requirements for tasks and interrupts, we computed the average execution time for each task across all 32 intervals. This was then validated with the FreeRTOS stats buffer timer which appeared to show similar estimated time values when timed for worst case scenarios. The timing results obtained from this analysis are presented below.

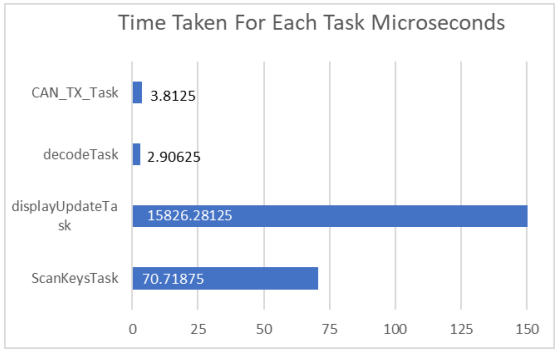


Figure 1: Timing Analysis for Tasks in The System

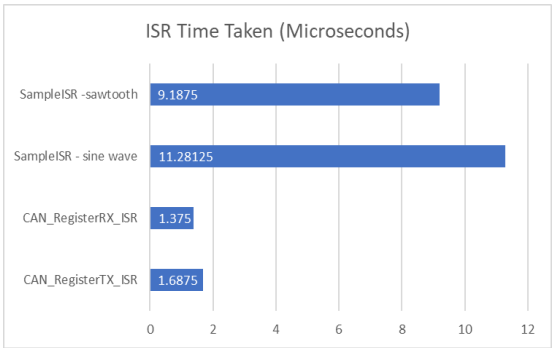


Figure 2: Timing Analysis for Each ISR

The aggregated timing statistics indicate consistent pattern wherein the displayUpdate() tasks exhibit significantly longer execution times compared to the other tasks. This can be attributed to the task's requirement to draw the relevant output on the display screen, which significantly increases the time required for this task. Additionally, the scanKeysTask() task exhibits relatively longer execution times, which is justifiable due to the need to scan through the keys, knobs, and joysticks, making this an overall time-consuming task.

Critical Instant Analysis

To determine the critical instant analysis of each task we need to determine both the deadlines and interrupt times of tasks within the system. We can determine the overall execution time of each task by taking the sum of our measured time and the relevant interrupt for the task (in figure 1).

For instance, the scanKeysTask() only has a single interrupt and hence the time taken for this task can be determined as follows:

$$T_{scanKeysTask()} = T_{scankeysmeasured()} + T_{sampleISR} = 70.7 + 11.3 = 82.0\,\mu s$$

Determining Task Deadlines

The deadlines for the displayUpdateTask() and scanKeysTask() are definable based on the port max frequency from STM32 and primarily use the vTaskDelayUntil() parameter. In our implementation, we extended the scheduling interval for displayUpdateTask() to 150ms for as it's acceptable to wait for a period of 0.15s for the task to occur and it's a task of lower priority.

Calculating Deadlines CAN Transmission

The CAN_TX_TASK has an explicit deadline that is contingent on the execution of the scanKeyTask(). This is because the CAN task is responsible for broadcasting the sounds to all the keyboards in the system.

Primarily the deadline for this task is defined as follows

$$CAN_{TASK(deadline)} = \frac{scan_{key_{deadline}}}{num_{keys}} = \frac{50ms}{12} = 4.167\,ms$$

This refers to the maximum allowable time for each CAN transmission.

Considering that the queue has a total length of 36, we can therefore assume the maximum time allowable for this task to be aggregated across 36 executions.

$$Aggregated - deadline = Queue\_size * CAN_{Task(Deadline)} = 36 * (50)/12 = 150ms$$

Likewise, for the decode task, the queue length is the same size and hence we will have a maximal aggregated interval of 150 ms.

Timing Analysis

After determining the interruptions and measured time per task, the following table was produced. Here, CAN_TX_TASK() and decodeTask() consider the aggregated time taken across 36 intervals.

Task Name and Time Including Interrupts	**Initiation Time – Tau (ms)**	**Execution Time – Ti (μs)**	**Priority Assigned**
scanKeys()	50	82.0	2
displayUpdate()	150	15826	1
CAN_TX()	150	198.0	1
decode()	150	154.1	1

When considering the priority of the following tasks the analysis is the same for all 3 of the displayTask() for 36 intervals, CAN_TX_TASK() for 36 intervals, and decodeTask() as they have equal initiation times and are all lowest priority tasks. The decision to do this was because it is important to ensure that display can be adequately handled to provide an essential user-interface, and this is equally important as CAN_TX_TASK() and decodeTask(). CAN and decodeTask() are important too as we need to ensure that sound can be broadcasted to different speakers.

Scan Key Task Timing

We can conduct a timing analysis for the Scan Key Task as follows.

$$Tn = \frac{T_{displayUpdate()}}{T_{scanKeys()}} = \frac{150}{50} = 3$$

Hence, we know 3 occurrences of scanKeysTask() will occur within the displayTask() deadline)

Latency Calculation

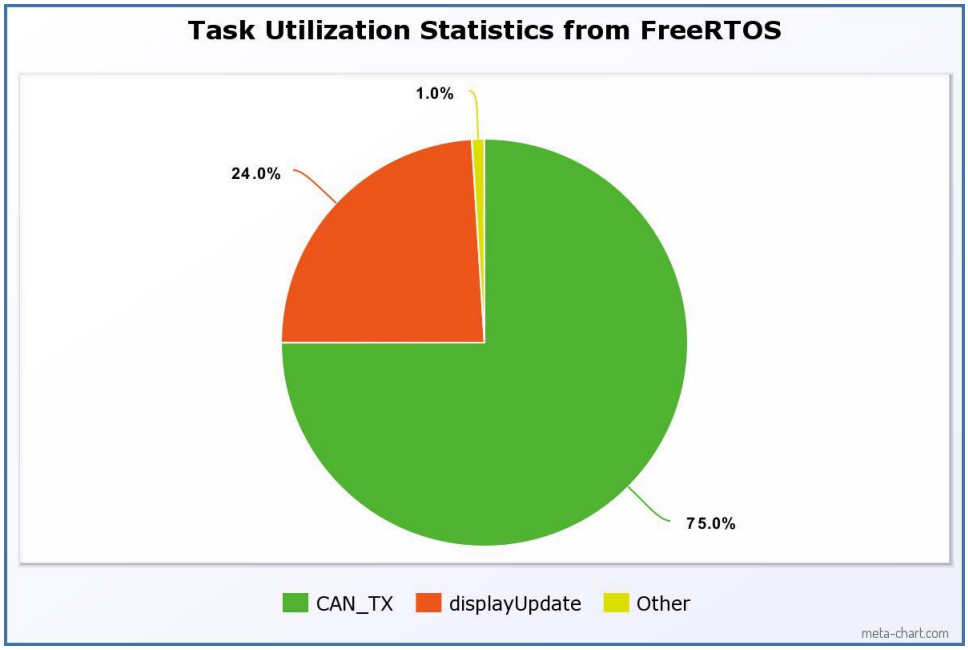
$$Ln = T_{CAN-TX()} + T_{decode()} + T_{displayUpdate()} + 3 * T_{scankeys()}$$

$$= \ 198 + 154.1 + 15826 + 3 * (82.0) = \ 16424.1 \ \mu s$$

According to the latency calculation of the critical instant, we can observe how $Ln < T_n = 150ms$ which results in the code being relatively safe according to time constraint requirements and critical timing path analysis.

Real World Timing Statistics

When the real-time operating system is running, we decided to consider a realistic operation of the CAN_TX_TASK and CAN_RX_TASK rather than disabling mailboxes for simplifications. It is also realistic to have all tasks running simultaneously. We hence conducted an analysis by enabling CAN mailboxes and then simply timing the rest of the tasks in the scheduler. After carrying out the FreeRTOS task utilization analysis the following ratios were determined for the percentage of time that the tasks ran.



It appears how CAN_TX and RX_TASKS can have a significant effect on the actual timing when it is running within real-time and disabling the mailboxes slightly oversimplifies the analysis step. Additionally, we see that displayUpdate() still gets carried out for an adequate and significant 24% of the duration which is significantly more than the time spent on scankeys() which is expected due to it taking a large duration to display pixels on screen. Overall, this analysis does confirm that all tasks are running smoothly with all tasks being carried out by the operating system. It does indicate however, that some tasks execution times cannot be oversimplified.