NC STATE UNIVERSITY

Dept. of Electrical and Computer Engineering

ECE 560 : Fall 2019

Project#1 : I2C WITH NON-PREEMPTIVE TASKS

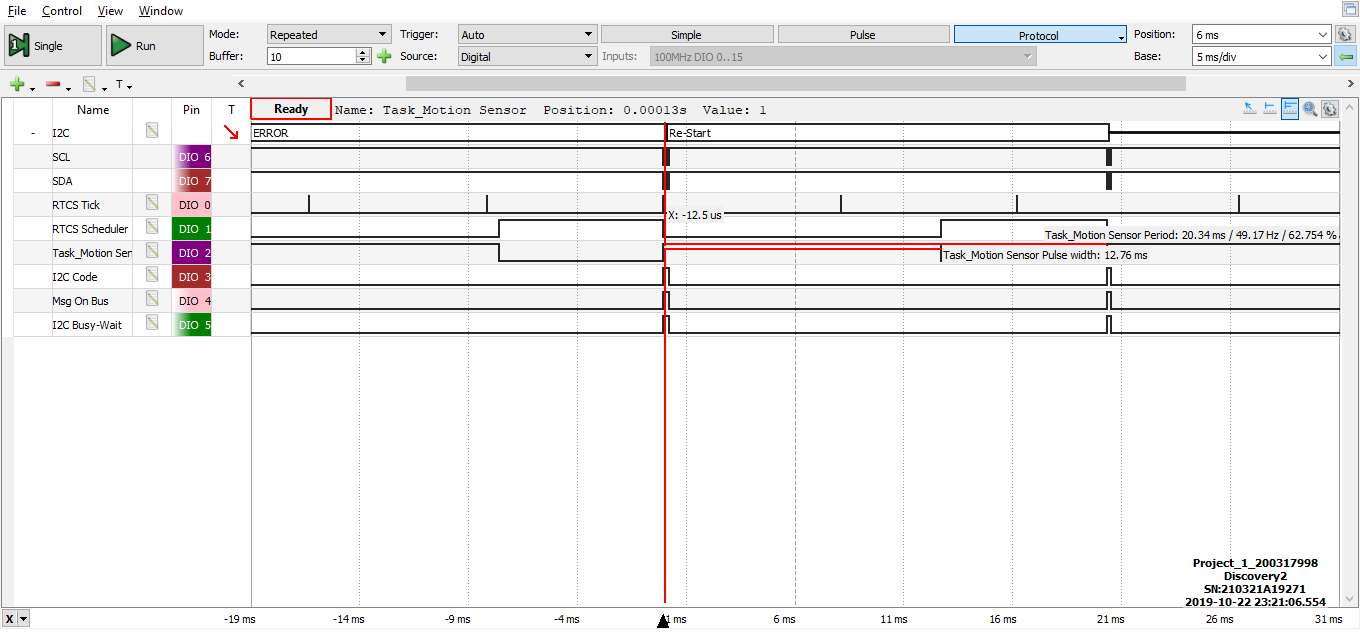
By

VISHNU SURESH MENON

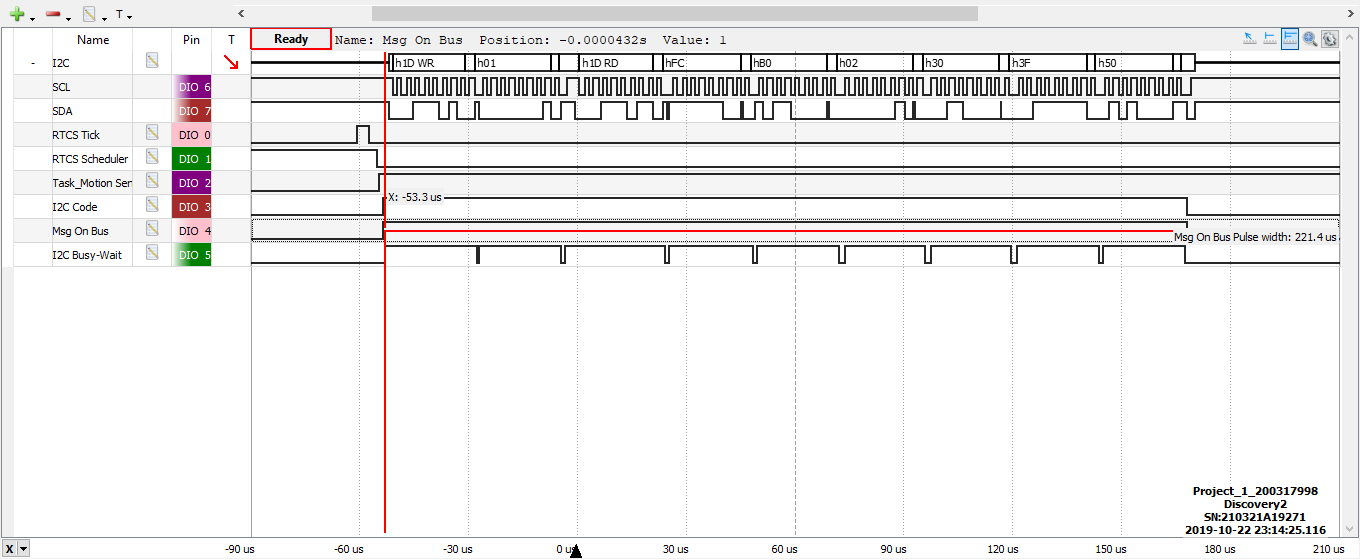
NCSU ID : 200317998

MODE 1 – BLOCKING

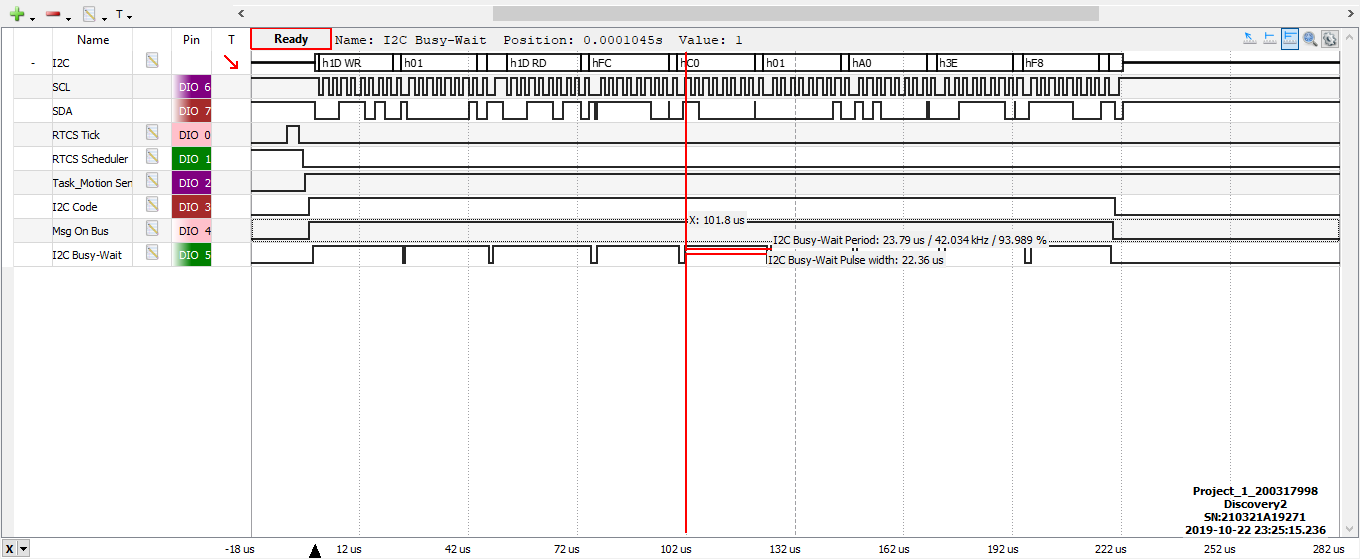
1. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C read operation.



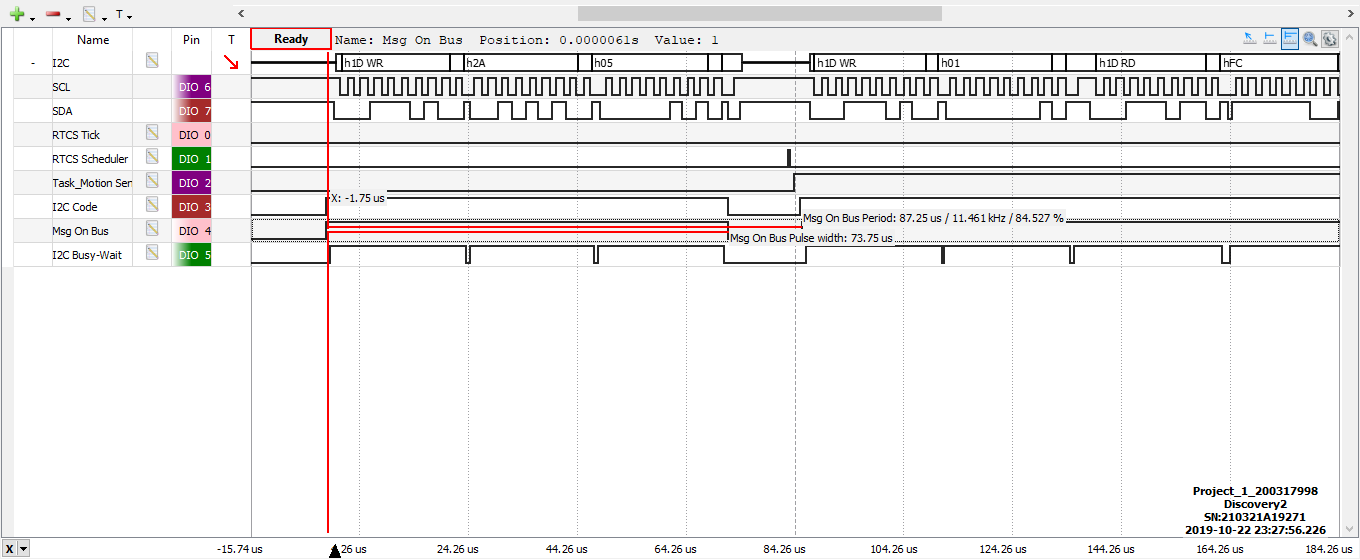
1. As per the above snapshot the task(Task Motion Sensor) is active for a time of 12.76ms.
2. The frequency at which Task Motion Sensor runs is about 49.17Hz



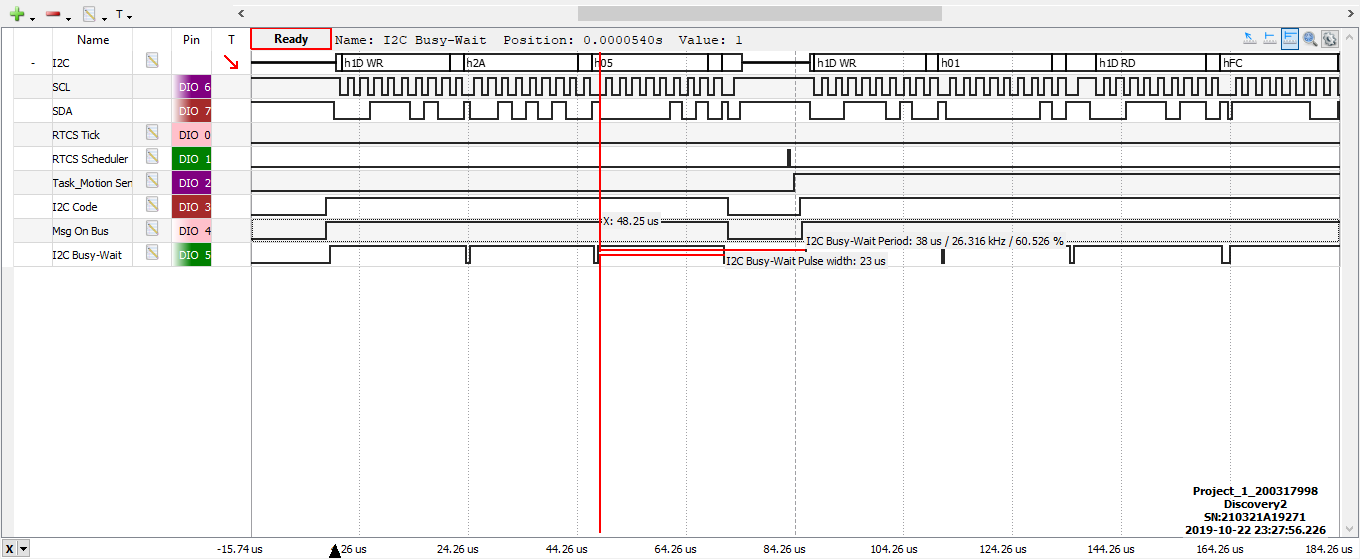
1. The above picture depicts that the Message On Bus signal is active for about 221.4us for one message which includes the writing of slave device address, register address, sending read command to the slave address and the reading of six bytes (data count = 6) of data along with the ACK signals.



1. The I2C Busy Wait is active for 22.36us as shown in the above screenshot ie; the active segment is the pulse width of this signal and from the above picture it is evident that there exist nine active segments. Therefore, the i2c busy wait is at logic high for about 201.24us(9\*22.36us) which almost nearby the time duration of message on bus signal.
2. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C write operation.



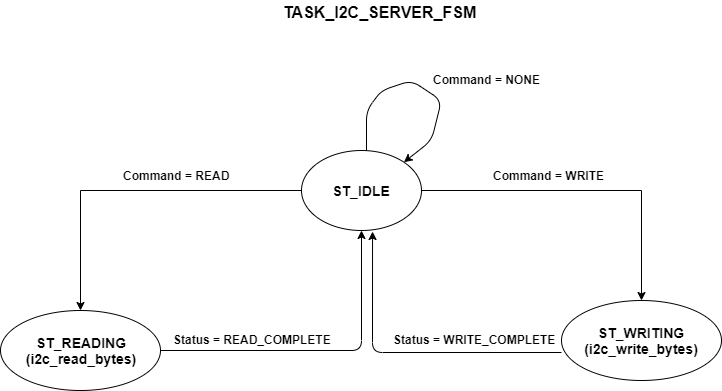
1. The above illustration shows that the Message on Bus signal is active for about 73.75us for one message which includes the writing of slave device address, sending the register address and writing one byte of data since the data count equals one.



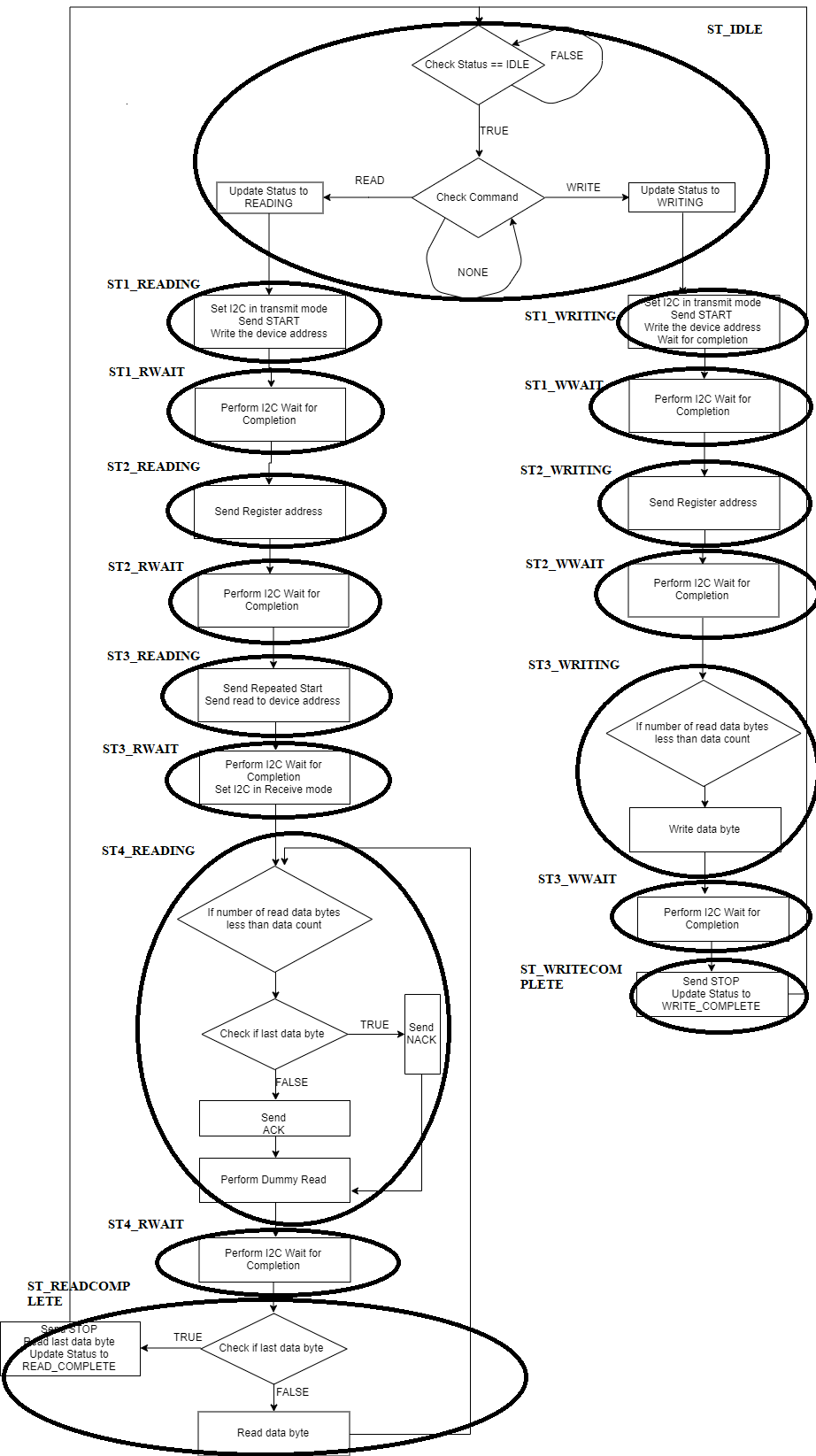
1. The I2C Busy Wait is active for 23us as shown in the above screenshot ie; the active segment is the pulse width of this signal and from the above picture it is evident that there exist three active segments. Therefore, the i2c busy wait is at logic high for about 69us(3\*23us) which almost nearby the time duration of message on bus signal.

MODE 2 - FINITE STATE MACHINE WITH RUN TO COMPLETION SCHEDULER

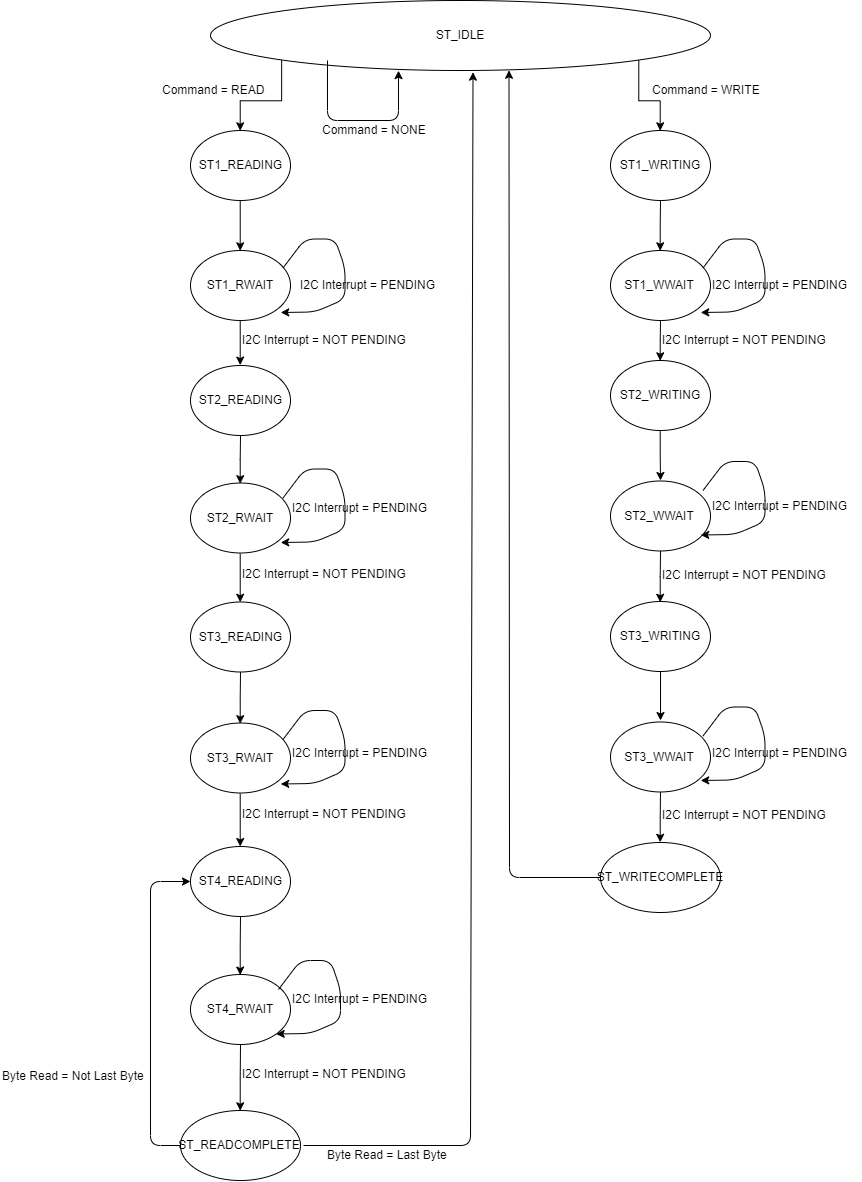
1. The below depicted is high level state diagram of the function Task\_I2C\_Server\_FSM function designed on the basis of handshaking details provided from the specification document.



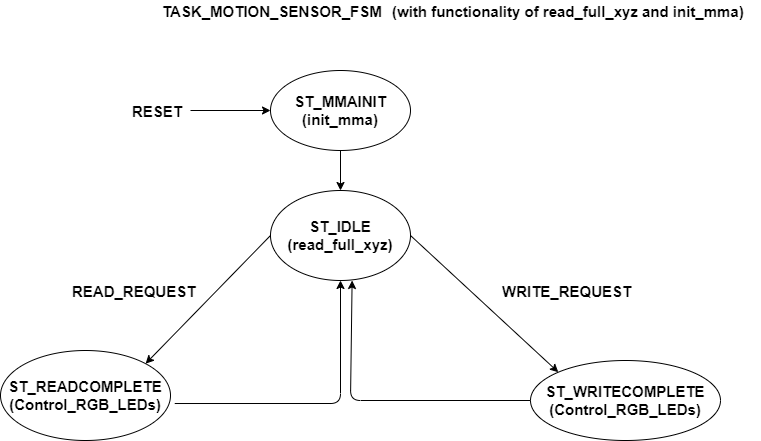
The below illustrated is the flowchart of the function Task\_I2C\_Server\_FSM function which represents the read and write functionality of I2C serial communication protocol. The control flow of the function is represented in the flowchart. Here, the entire function is broken into finite state machines by circling specific portion of code and identifying it as a state such that each state consumes a minimal amount of the CPU time. The function is divided into states under the assumptions that only delay calls take any time and all other code is zero time. Therefore, only single i2c wait functions are maintained in one state.



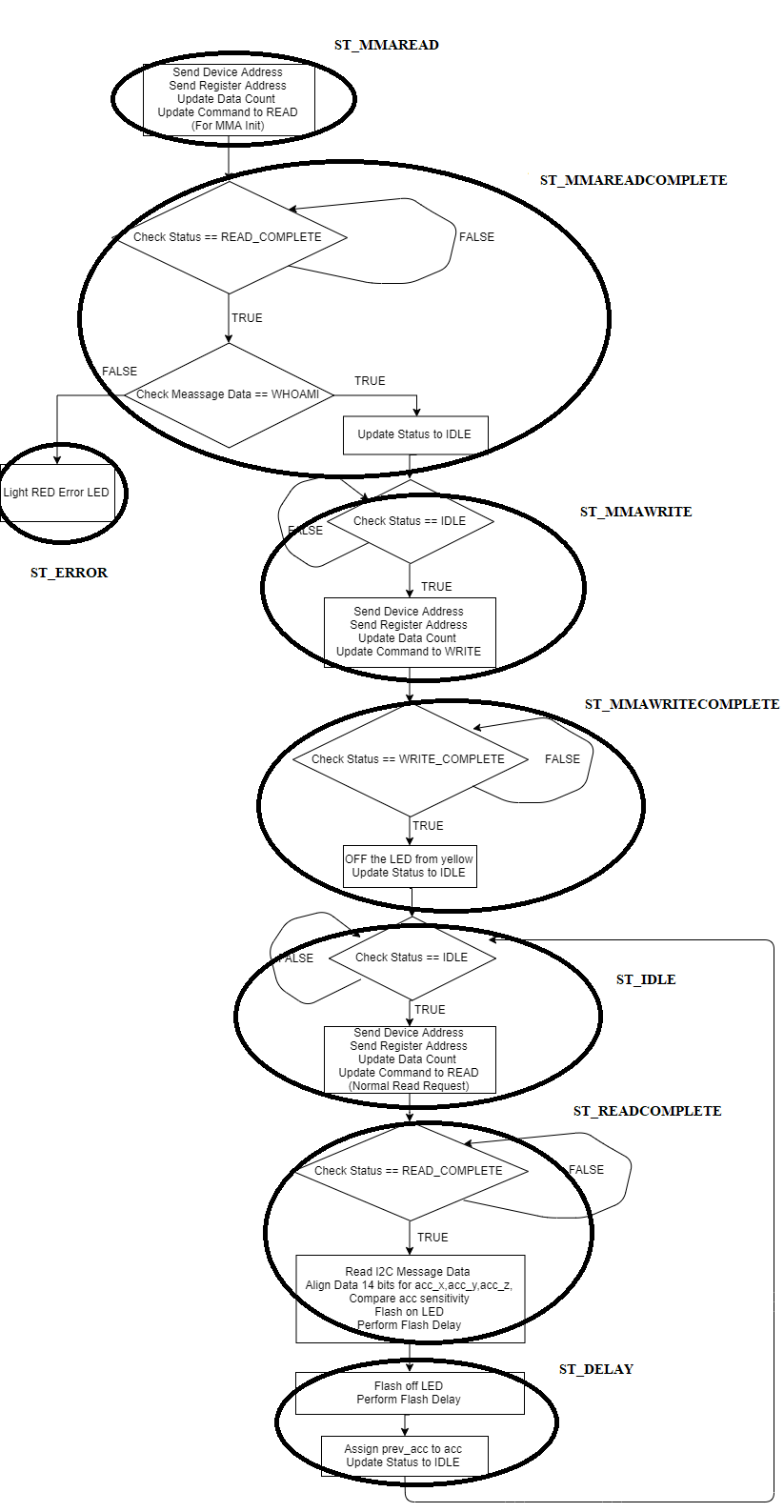
The detailed state diagram corresponding to the above-mentioned flowchart for the function Task\_I2C\_Server\_FSM is shown below. Here the identified states are connected together showing the transitions from one state to another.



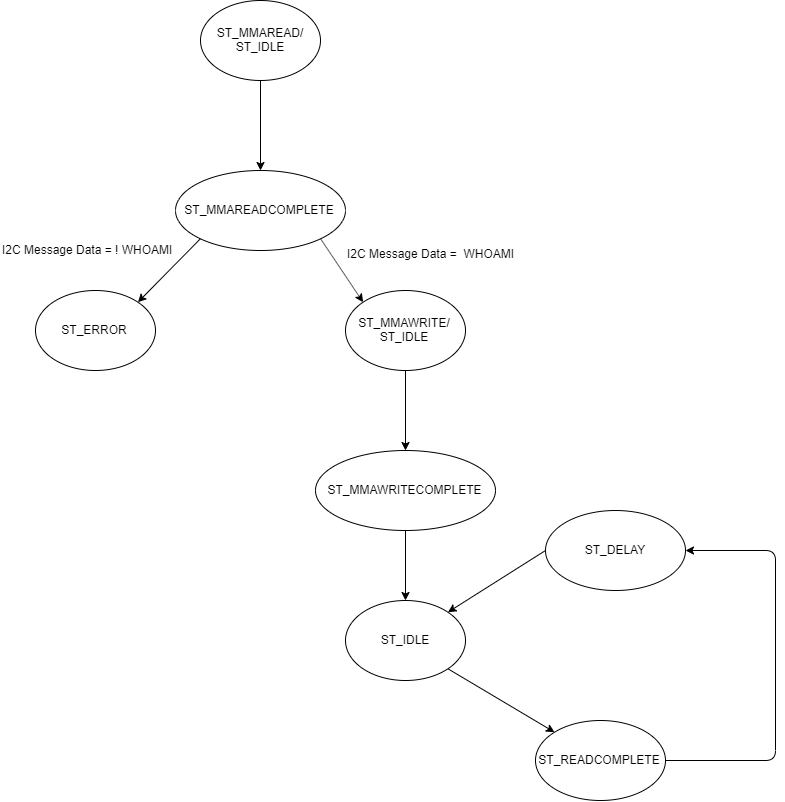
1. The below depicted is high level state diagram of the function Task\_Motion\_Sensor\_FSM function designed on the basis of handshaking details provided from the specification document.



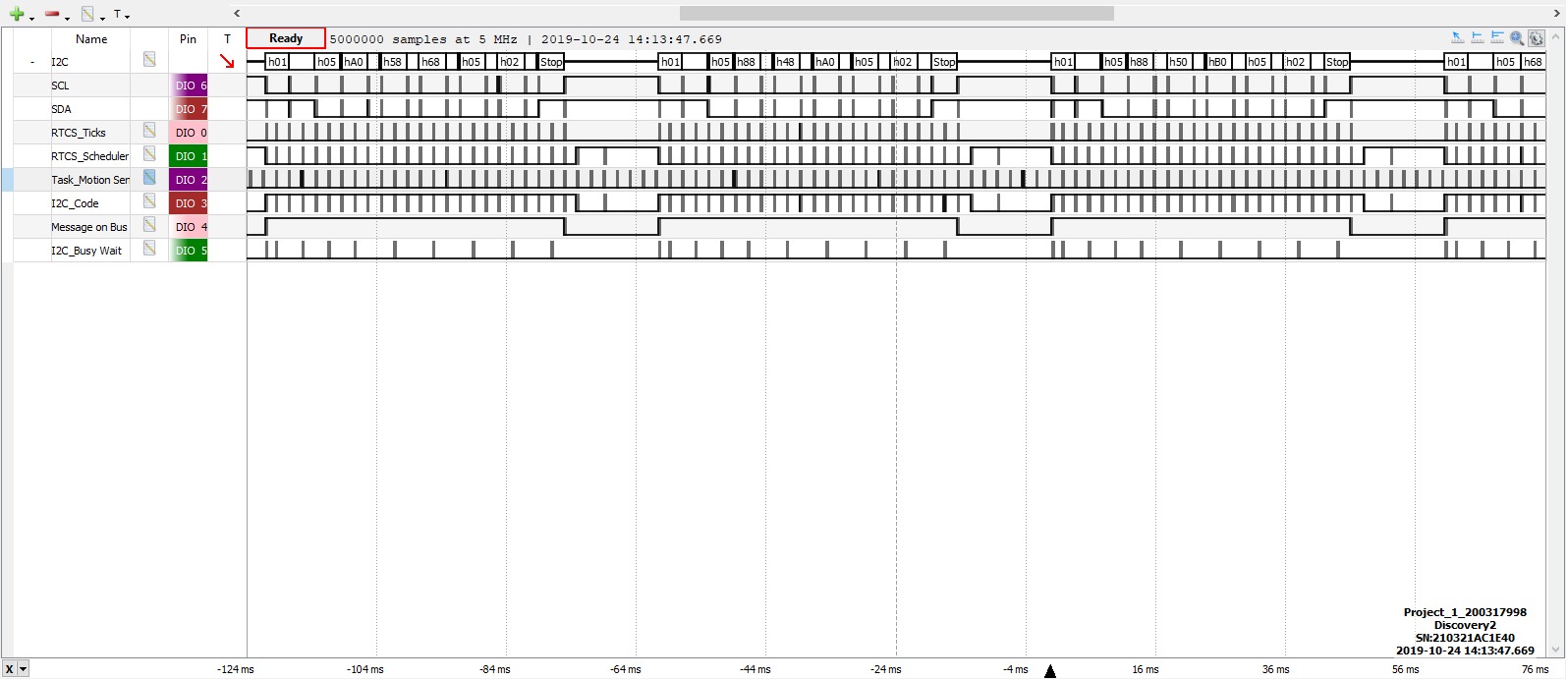
The below illustrated is the flowchart of the function Task\_Motion\_Sensor\_FSM function which represents the init\_mma, read\_full\_xyz and Control\_RGB\_Leds functionalities. The control flow of the function is represented in the flowchart and the entire function is broken into finite state machines by circling specific portion of code and identifying it as a state such that each state consumes a minimal amount of the CPU time. The function is divided into states under the assumptions that only delay calls take any time and all other code is zero time. Therefore, only single flash delay functions are maintained in one state.



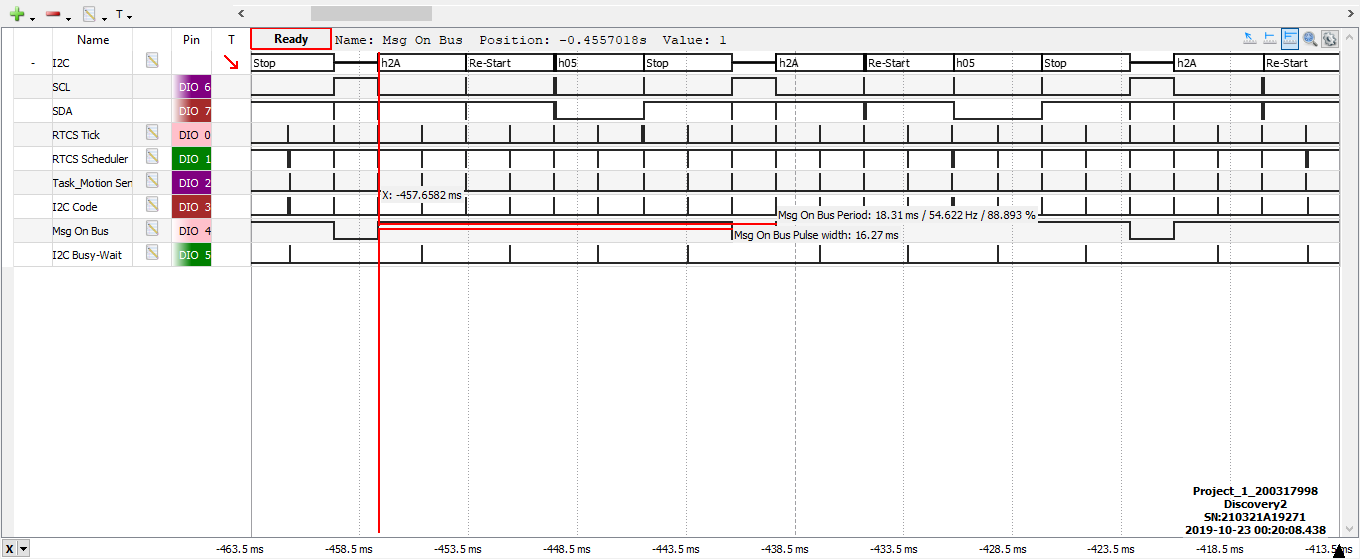
The detailed state diagram corresponding to the above-mentioned flowchart for the function Task\_Motion\_Sensor\_FSM is shown below. Here the identified states are connected together showing the transitions from one state to another.



1. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C read operation.

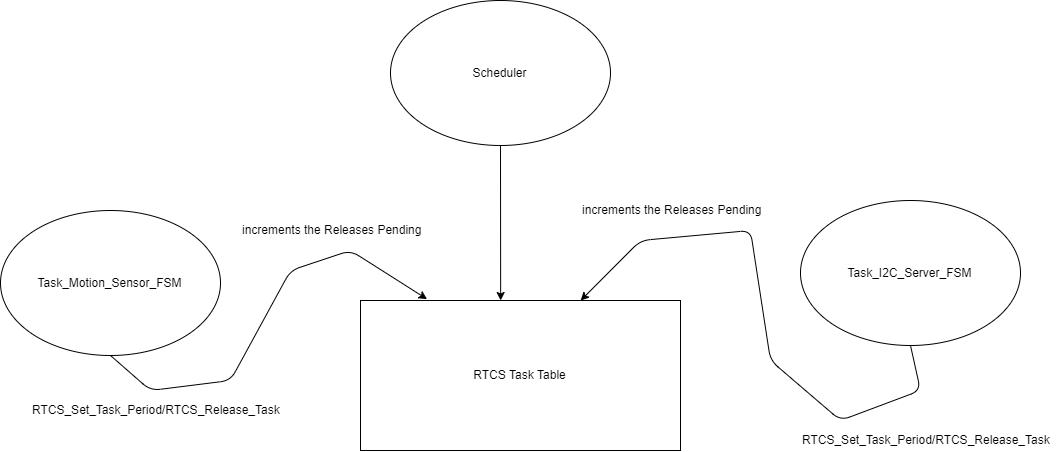


1. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C write operation.



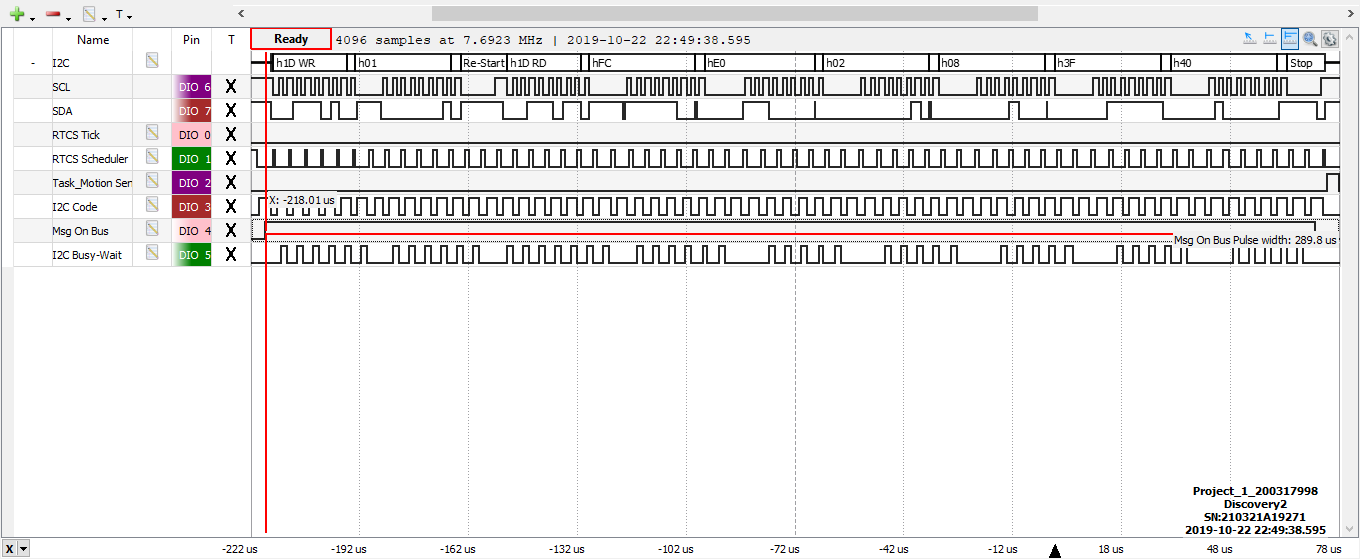
MODE 2 - FINITE STATE MACHINE WITH RUN TO COMPLETION SCHEDULER (EVENT TRIGGERED APPROACH)

1. In event triggering approach the state transitions are initiated by an external event rather than depending on a periodic timer-based triggering approach. In the previous case, each task gets executed periodically(irrespective of their priority) by the RTCS Scheduler. The Low power timer (LP\_Timer) used to generate the timer tick increments the Releases\_Pending of both the tasks in the RTCS\_Task\_Table based on the period of timer tick. In event triggering, we make use of the inbuilt RTCS functions like the “RTCS\_Release\_Task” and “RTCS\_Set\_Task\_Period” present in the original code. These functions are responsible enough to trigger the tasks by just incrementing their releases pending on the task table so that the scheduler can execute those tasks. Hence, upon invocation of these RTCS functions in each task, it behaves like an external event initiating a task to run. In this approach response time can be reduced and doesn’t let the low priority tasks run.

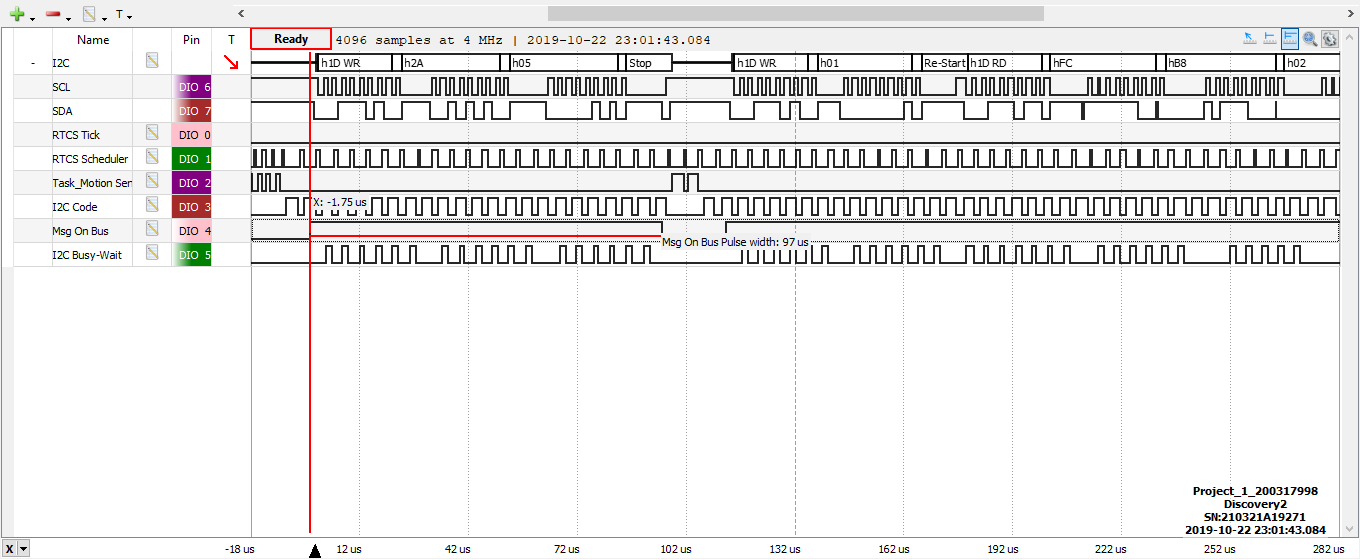


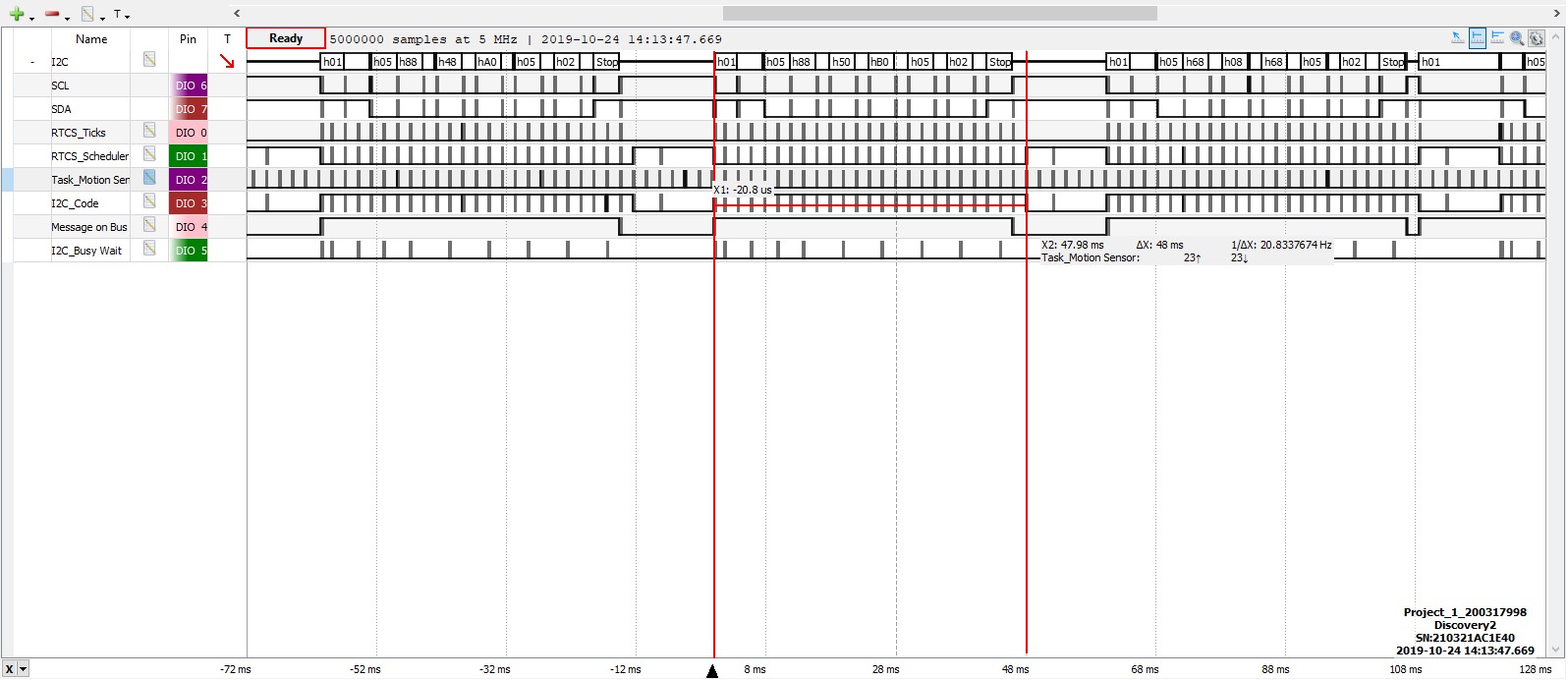
The above state diagram illustrates on how the invocation of RTCS functions from each state of both the tasks increments the releases pending of the task to be released on the RTCS table. Therefore, the scheduler can execute the required tasks to be executed based on the information fed on the RTCS table. This is how the flow of event triggering scheduling approach has been implemented in the project.

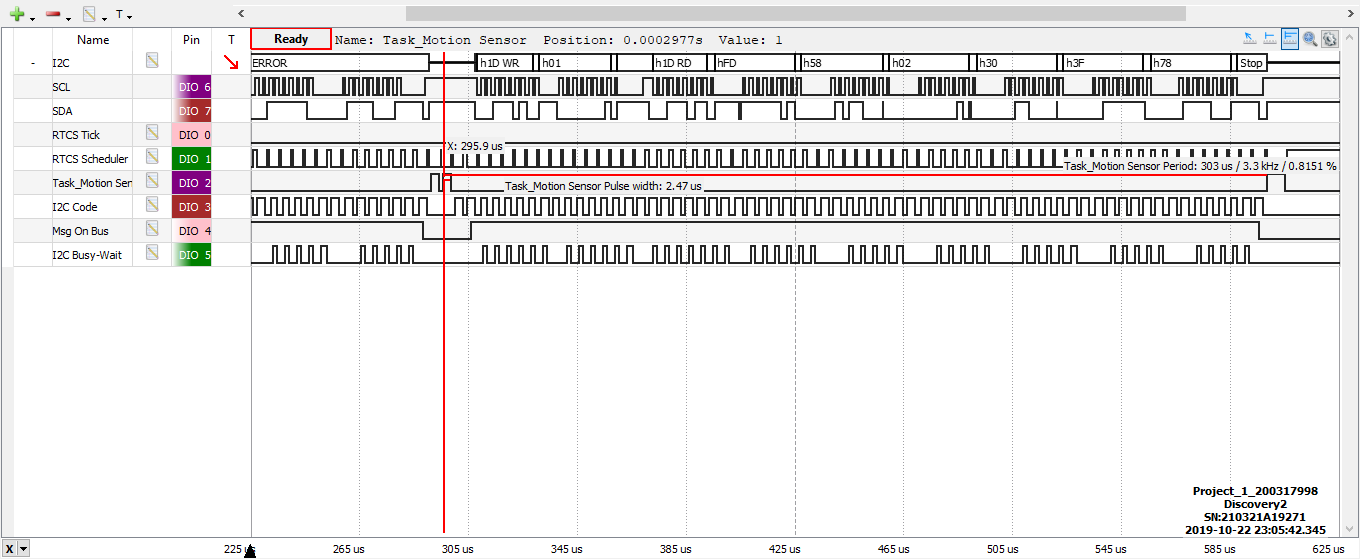
1. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C read operation.



1. The below screenshots from the logic analyzer illustrates the debug signals (RTCS Tick, RTCS Scheduler, Task Motion Sensor, message on bus, I2C busy wait) and the I2C bus signals (SCL, SDA) during the I2C write operation.







The time delay between Task\_Motion\_Sensor\_FSM sending a read request and starting to light the LED is the time difference between the pulses of the function Task\_Motion\_Sensor\_FSM. This is due to the fact that the read request is initiated from the Task Motion Sensor FSM function and the blinking of LED based on the resulting data is also executed in the same function. Therefore, the time delay between the pulses of this function ie; the duration where the Task\_Motion Sensor is at logic low and I2C message is on the bus is the approximate response time of this project implementation.

As per the first screenshot (FSM with RTCS)

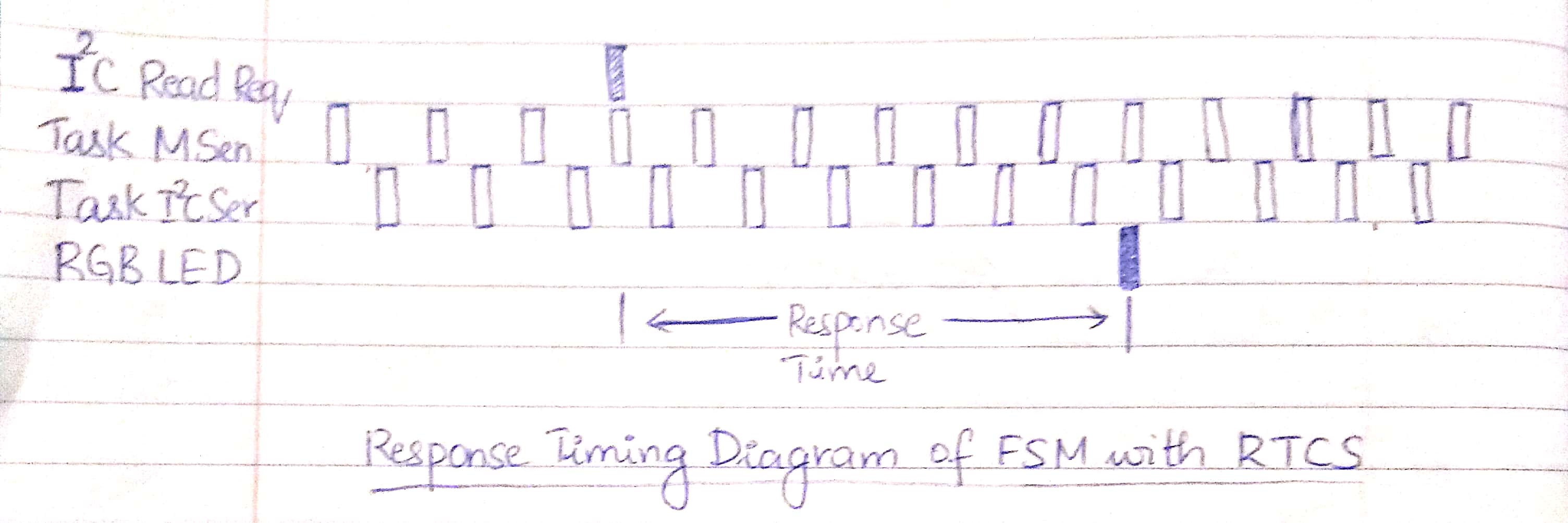
Response time = 48ms

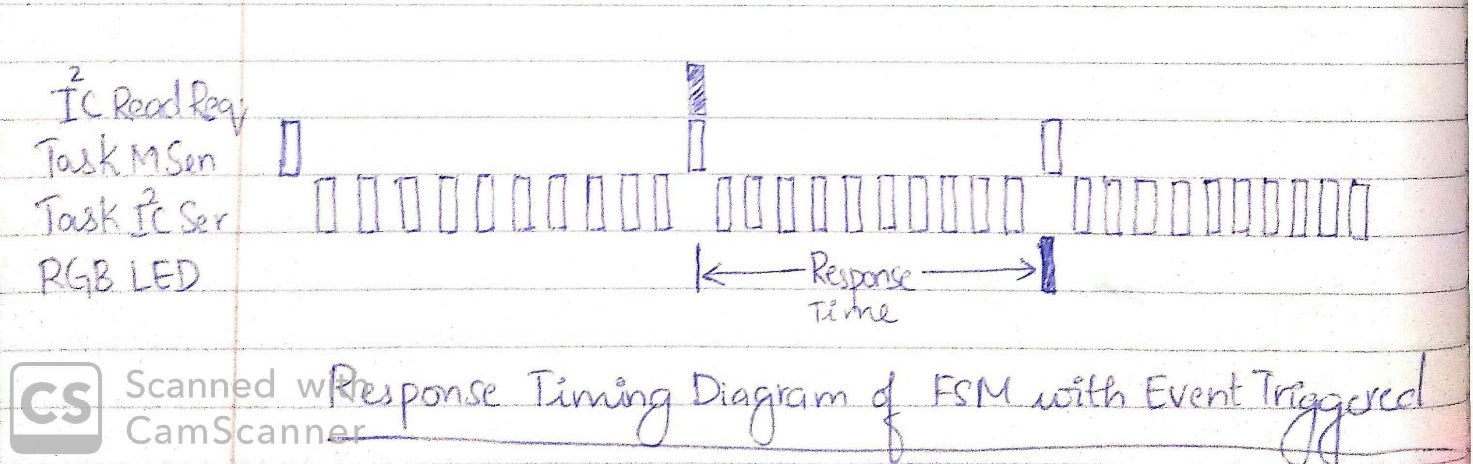
As per the second screenshot (FSM with RTCS with Event Triggered Approach)

Response time = 303us

From the above screenshots it is revealed that the response time is improved significantly in event triggering approach compared to the periodic time triggered based approach. This is due to the fact that in “FSM with RTCS” both the tasks are triggered by the periodic timer interrupt within a time period of one timer tick(2ms) which is more than the execution time of each broken states of the tasks. Therefore, tasks are executed alternatively on a periodic basis even if the tasks might be idle at that moment of time.

In event triggering approach the required tasks are executed on an event driven basis and doesn’t let the low priority tasks run. Moreover, the delay functions used in the project are consumed by using the hardware timer peripherals and the scheduler can be used to take care of other functionalities rather than polling the tasks and staying idle. Hence the duration of each tasks can get reduced in this approach resulting in an improved response time.





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*THE END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*