Analysis Of FreeRTOS On ESP32

Submitted by:
Ali Anwer

What is ESP32 ?

- Low-Power SoC Micro-Controllers
- Developed by Espressif Systems
- Various Variants (Generations?) of ESP32
- Built-in WIFI and Bluetooth capabilities
- Offers “Light Sleep” and “Deep Sleep” modes for Power Saving
- Mainly 2 types of ISA (Instruction Set Architecture) : EXTENSA and RISC-V

ESP32 SoC families

- ESP8266 : L106 32-bit MicroProcessor
- ESP32: XTENSA LX6 32 bit MicroProcessor (single or dual core)
- ESP32-S2: XTENSA LX6 32 bit MicroProcessor (single core)
- ESP32-S3: XTENSA LX7 32 bit MicroProcessor (dual core)
- ESP32-C3: RISC-V 32 bit MicroProcessor (single core)

What is FreeRTOS?

- Real Time Operating System for MicroControllers.
- Developed by Richard Barry in 2003.
- Open Source and MIT License.
- Previously maintained by Real Time Engineers Ltd, now maintained by AWS.
- Can be used on 35+ MicroControllers (can be ported to new MCUs as well)
- Active development!
- Can be used with Rust Language

ESP-IDF?

- ESP-IDF is a development framework created by Espressif Systems.
- Contains Toolchain, Component libraries,bootloader,data partition tables,HAL and some examples codes that can be used to create applications.
- ESP-IDF can be used to build binary file, flash binary file to respected data partition and monitor Serial Port using terminal.
- Compatible with Windows, Linux , MacOS.
- Provides menuconfig to configure “SDK” for application. This can be used to enable peripherals as well as configure security parameters for ESP32 (depends on Variant of ESP32 for some options) and other system related configuration.

ESP-IDF and FreeRTOS

- ESP-IDF provides FreeRTOS as part of SDK and can be used without any configuration and porting.
- FreeRTOS from ESP-IDF contains some differences such as creating task affinity to specific core.

Analysis of FreeRTOS on ESP32

2 approaches taken in observing FreeRTOS behavior

- Simulation of ESP32 on Web Based Emulator(Wokwi) to perform Behavior Analysis
- Hardware based Analysis on ESP32-S3 to perform Timing Analysis

Behavior Analysis of FreeRTOS on ESP32 Using Simulations

Toggling GPIO to Analyze time required to perform action such as

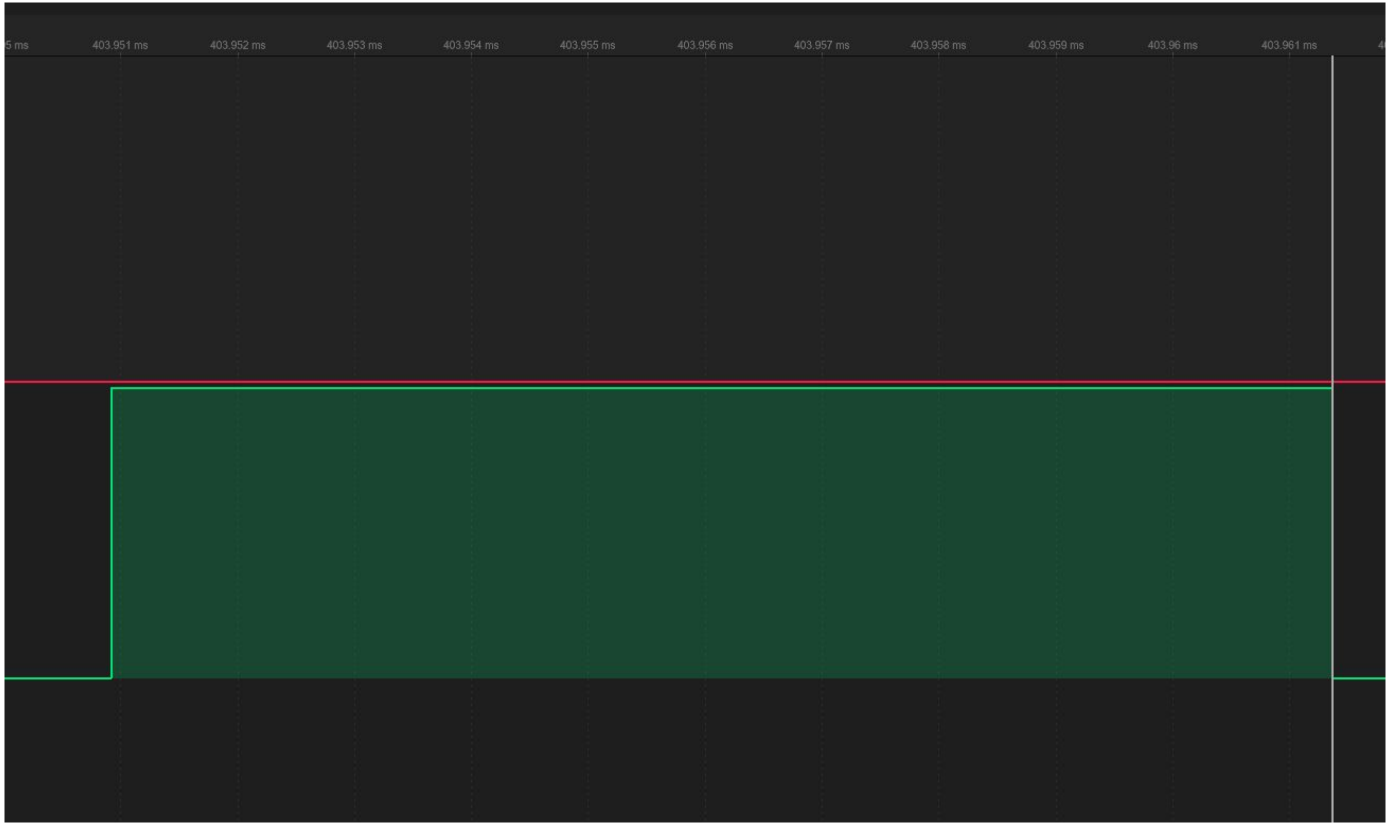
- Insert values in Queue
- Retrieving values from Queue
- Software timers
- Behavior of FreeRTOS on Interrupt

Queue

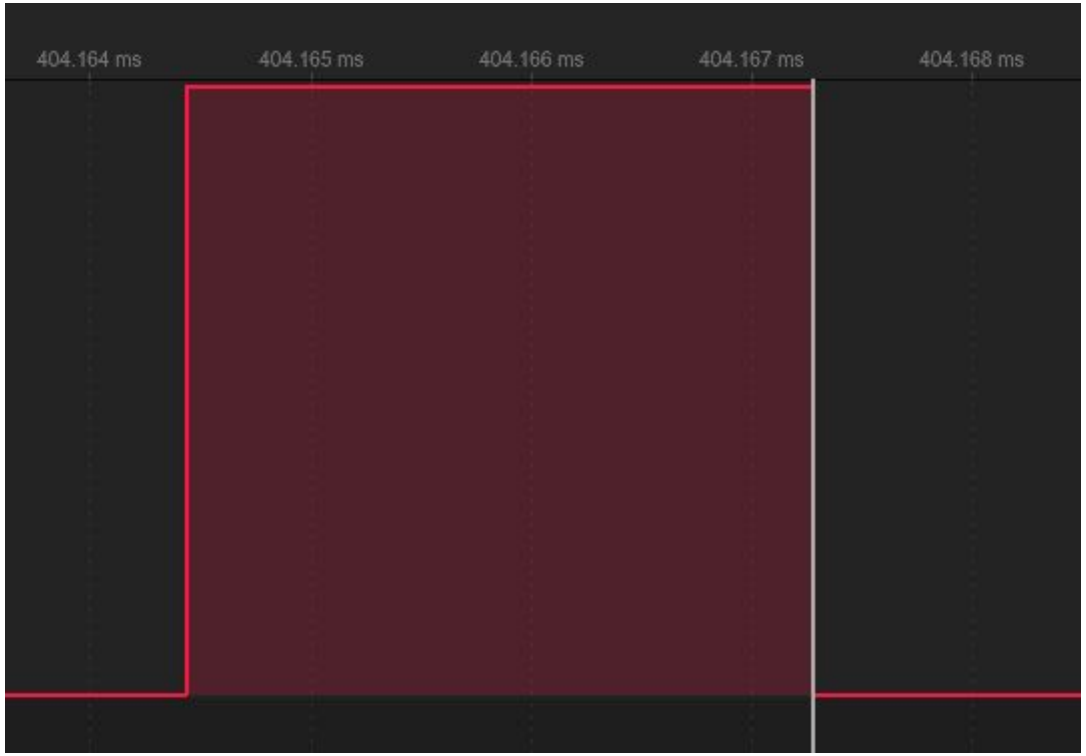
Test:Queue of size 1, where Sender and Receiver having same priority. Sender Task is initiated first which means first value is placed in Queue and than Receiver Task is executed. This ensures that there is always one value since scheduler is round robin

- Behavior of Sender Task is observed using Green Signal
- Behavior of Receiver Task is observed with Red Signal

Starting and ending of “Sender Task” execution is illustrated by toggling GPIO. At start of execution, GPIO is put to High State and at end of execution, GPIO is put back to Low State.



Similarly ,starting and ending of “Receiver Task” execution is illustrated by toggling GPIO. At start of execution, GPIO is put to High State and at end of execution, GPIO is put back to Low State.



From previous analysis, it can be concluded that:

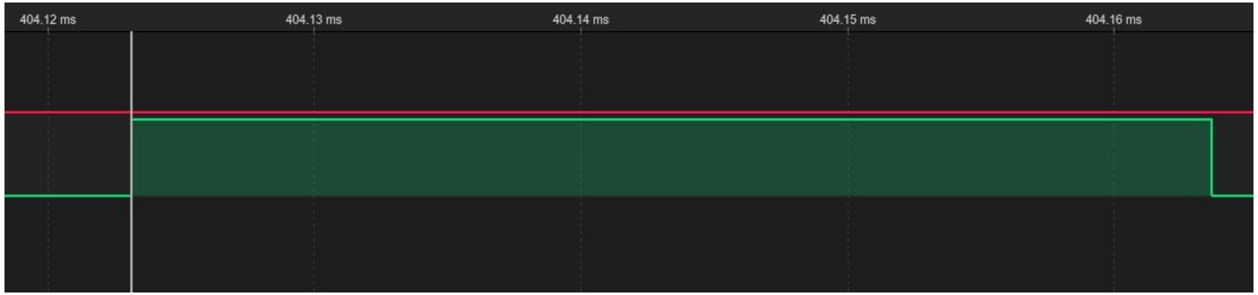
- Since Queues work in FIFO manner,putting value in Queue takes more time as compared to extracting value. Reason for taking more time to put values in Queue is that before value(s) in Queue, FreeRTOS needs to check if there is any space available in Queue or not but in-case of extracting values, it extracts value in-front of Queue regardless of number of elements unless specified which element to extract.
- Signal toggle time contains queue operation time as well as GPIO Toggle time.

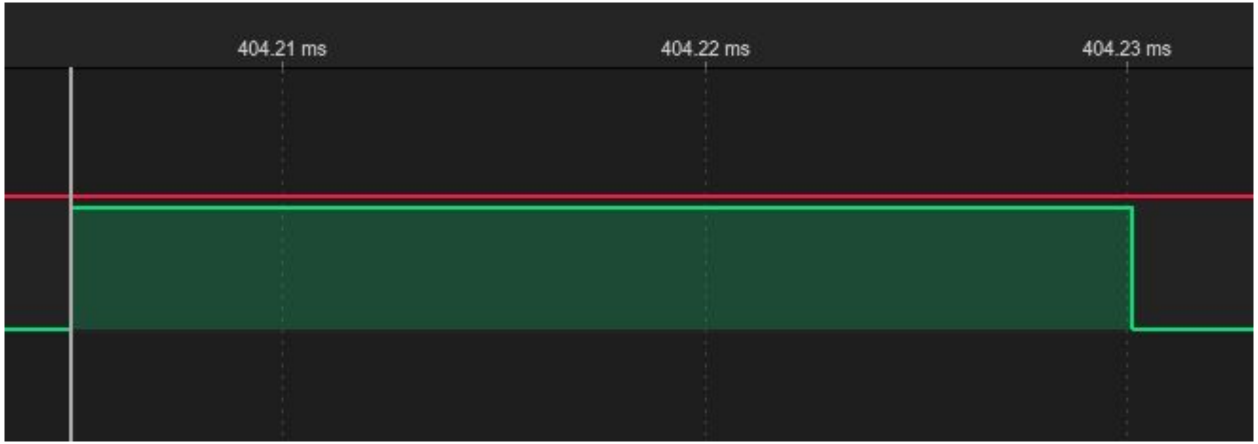
Queue

Test : Sender Task has higher priority and Receiver Task has lower priority with no delay in task execution.

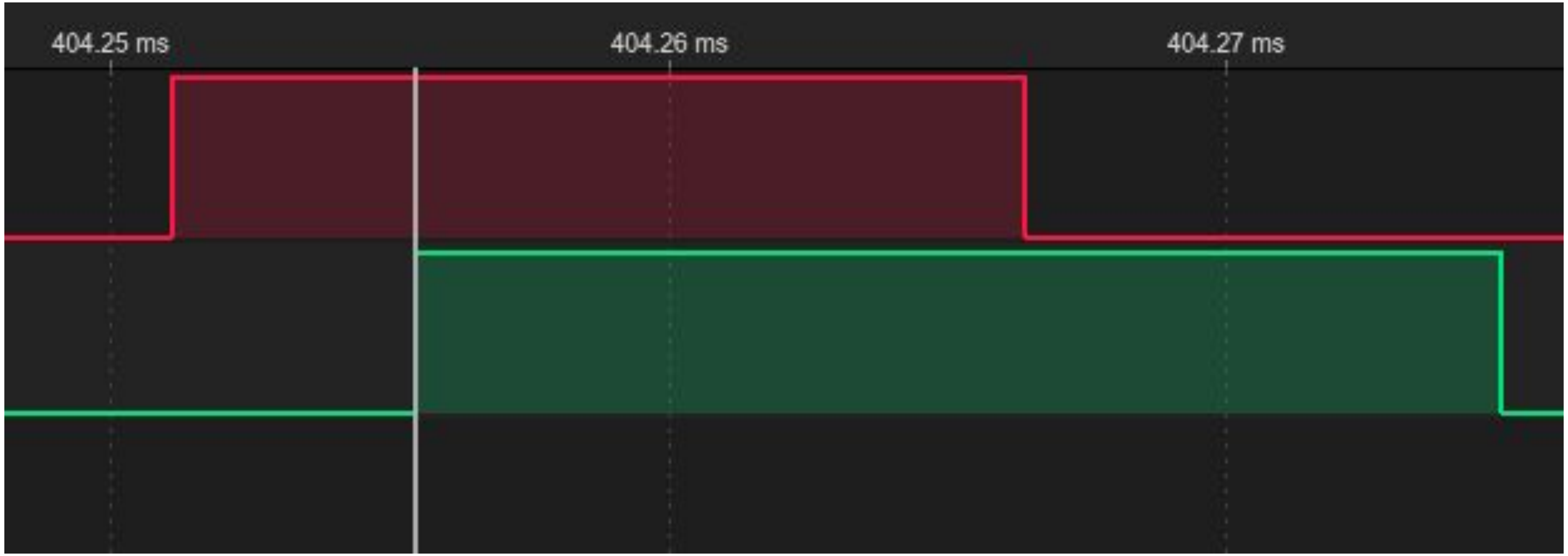
- Queue is set to contain 10 elements
- Sender Task is programmed to put 5 elements in queue at a time.
- Receiver task is programmed to retrieve 5 values at a time.

Here , a Queue is initialized with size of 10. First Sender will keep putting values in Queues since it has higher priority. Implementation of this function is limited to put 5 values at one time . there this function will execute 2 times before Receiver Task is executed.

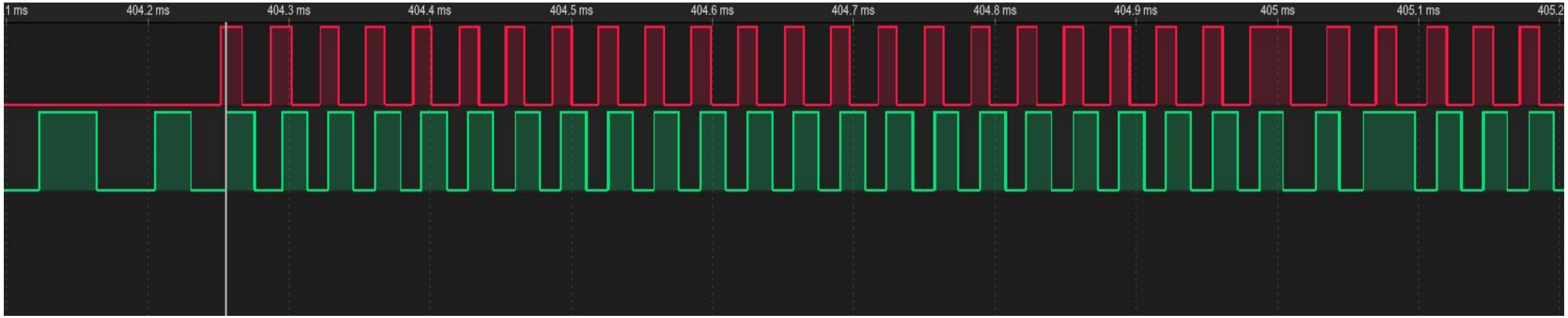




After 10 values are put in Queue, Sender Task is put in a blocked state and Receiver Task starts pulling out values from Queue but as soon as first value is pulled , Sender Task will preempt Receiver task since Sender has higher priority than Receiver Task. This is shown in following image below



Receiver task is implemented to get only 5 values in one complete iteration , therefore, it has low execution time as it takes less time to get value from Queue as shown in previous test. In this test , as soon as value is removed from Queue, a place is empty therefore, Sender Task Preempt Receiver Task and Pre-emption behavior looks like this



Observation:

- It can be observed that Green Signal (Task putting values in Queue) is executed 2 times before Red Signal is high (Task getting values from queues). Since Green Task has higher priority, therefore , it should preempt Red Signal and this behavior is confirmed by observing pattern.
- Time taken for 1st execution of Sender Task : 404.164 - 404.126 = 0.038ms
- Time for Sender Task in second iteration : 404.230 - 404.205 = 0.030ms .
- From previous test, it was estimated that time taken for putting one value in Queue and GPIO toggling operation took 0.009ms , therefore , putting 5 values took approx 0.035ms

Software timers

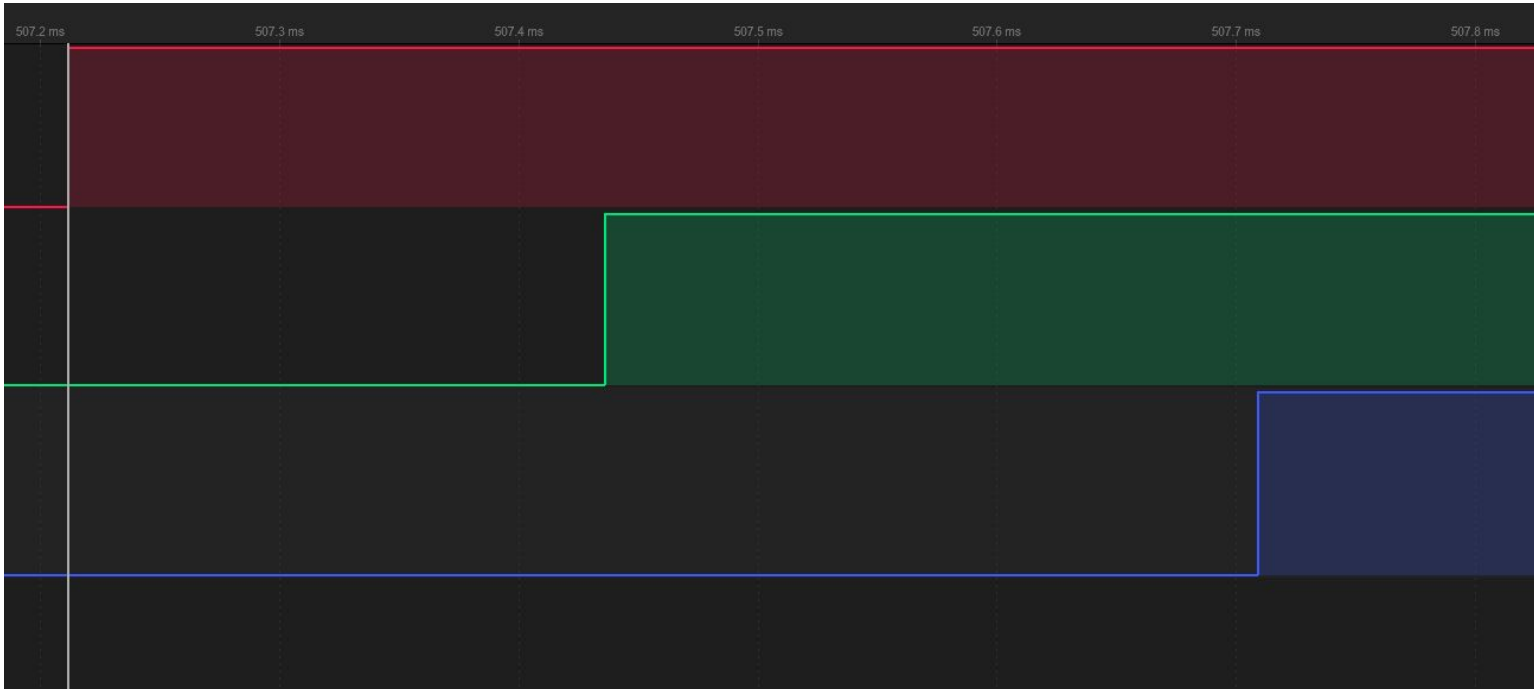
TEST**:**

All timer callbacks have same time to expire : 100 ms
Experimental setup: 3 timer based tasks. All have same callback function and using conditional to Toggle GPIO based on timer ID.

- Task1: Red Signal toggle based on expiring timer
- Task2: Green Signal toggle based on expiring timer
- Task3: Blue Signal toggle based on expiring timer

1 cycle

Rising Edge (LED turn on) (First call to timer expire callback)



Observation:

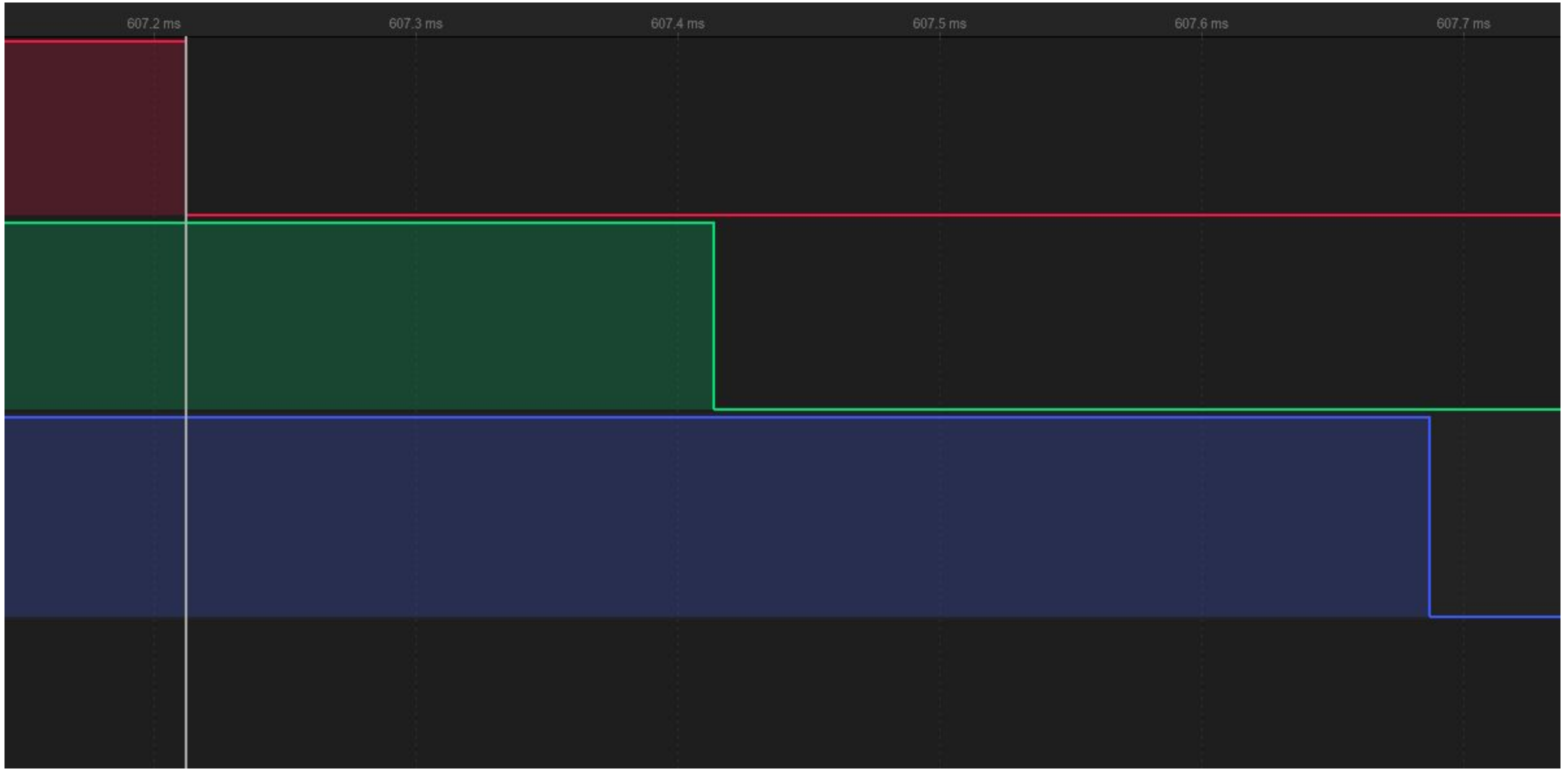
- Time to start task1 : 507.212 ms
- Time to start task2 : 507.436 ms
- Time to start task3 : 507.709

Difference between turning GPIO on (High State)

- Task2 - task1 = 0.224 ms
- Task 3 - task 2 = 0.273 ms

One important information to consider is, all 3 timers have same callback function and timer specific operation is happening on comparing Timer ID using conditionals, because of this reason, we have higher execution time.

Falling Edge (LED turn off) (second call to timer expire callback)



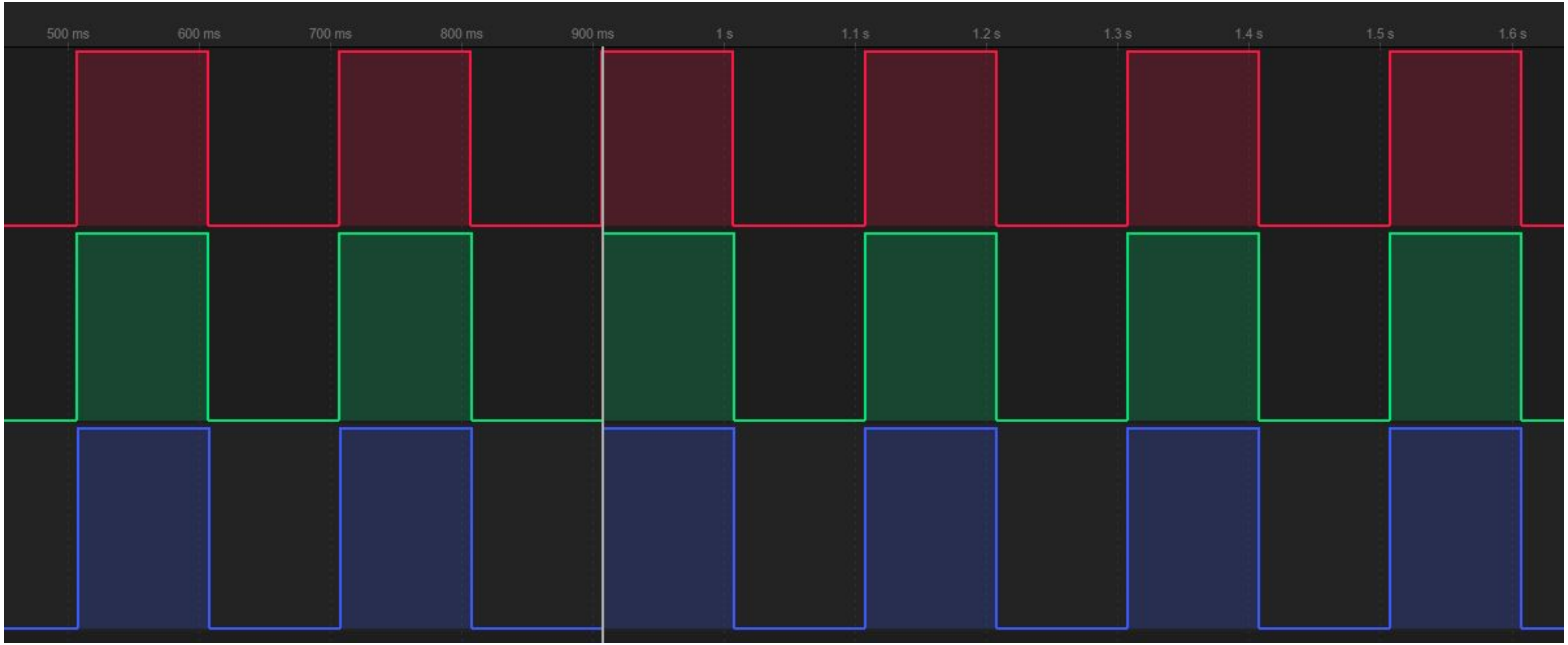
Observations:

- Time to start task1 : 607.212 ms
- Time to start task2 : 607.414 ms
- Time to start task3 : 607.687 ms
-

Difference between turning GPIO OFF (Low State)

- Task2 - task1 = 0.202 ms
- Task 3 - task 2 = 0.273 ms

All cycles overview



SOFTWARE TIMERS

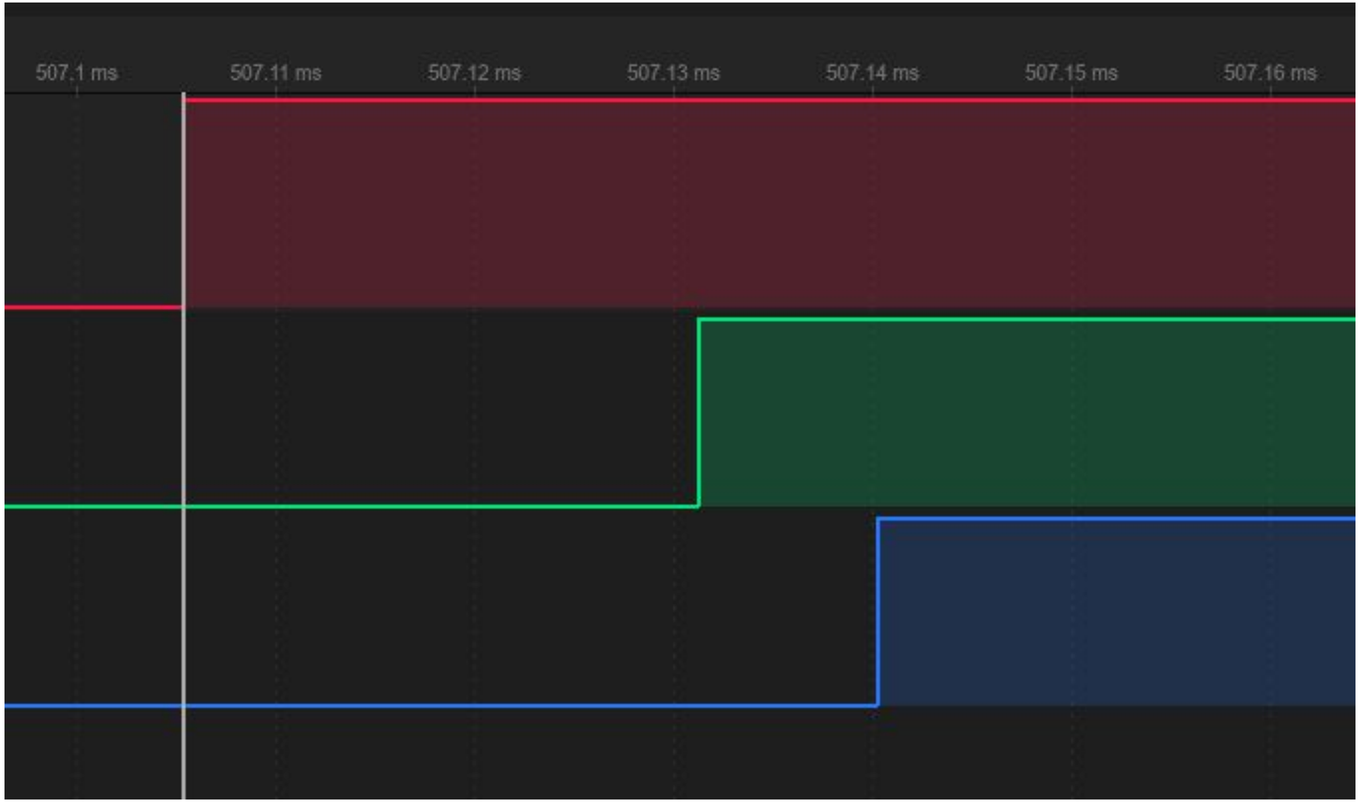
TEST**:**

All timer callbacks have same time to expire : 100 ms
Experimental setup: 3 timer based tasks. All have separate callback function to Toggle GPIO.

- Task1: Red Signal toggle based on expiring timer
- task2: Green Signal toggle based on expiring timer
- task3: Blue Signal toggle based on expiring timer

1 cycle

Rising Edge (GPIO turn on) (First call to timer expire callback)



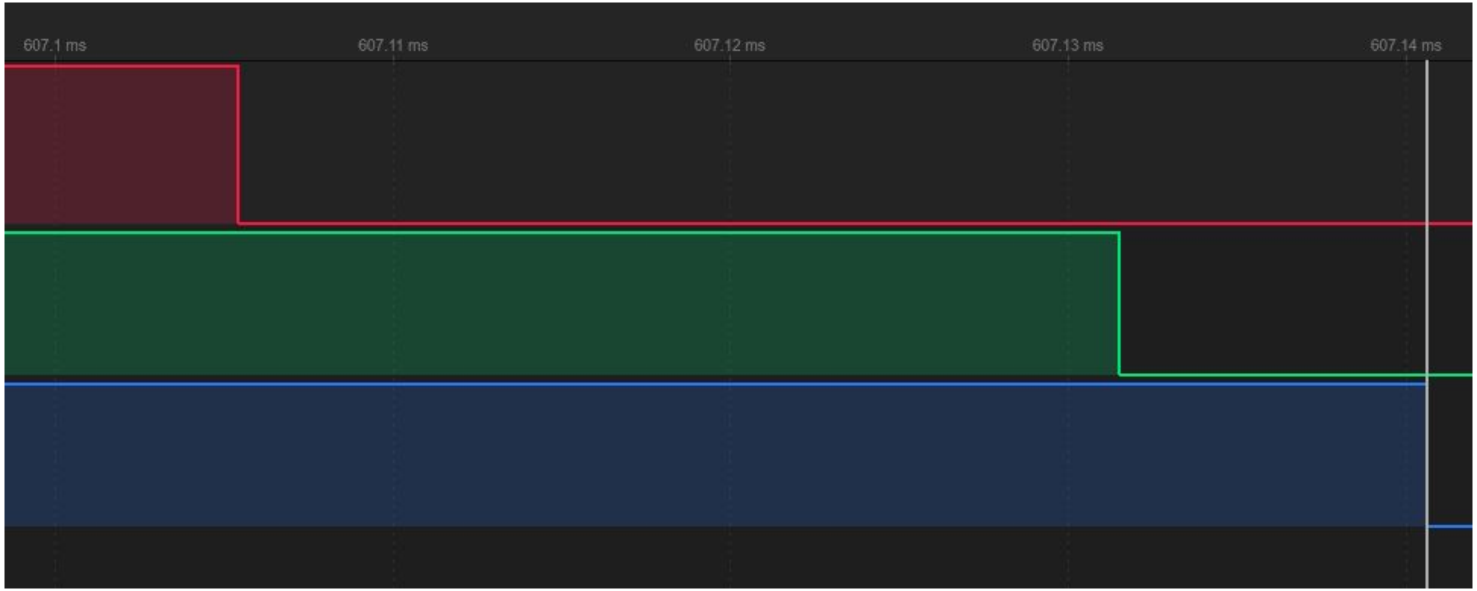
Observations

- Time to start task1 : 507.105 ms
- Time to start task2 : 507.131 ms
- Time to start task3 : 507.140 ms

Difference between turning GPIO ON (High State)

- Task2 - task1 = 0.026 ms
- Task 3 - task 2 = 0.009 ms

Falling Edge (LED turn off) (second call to timer expire callback)



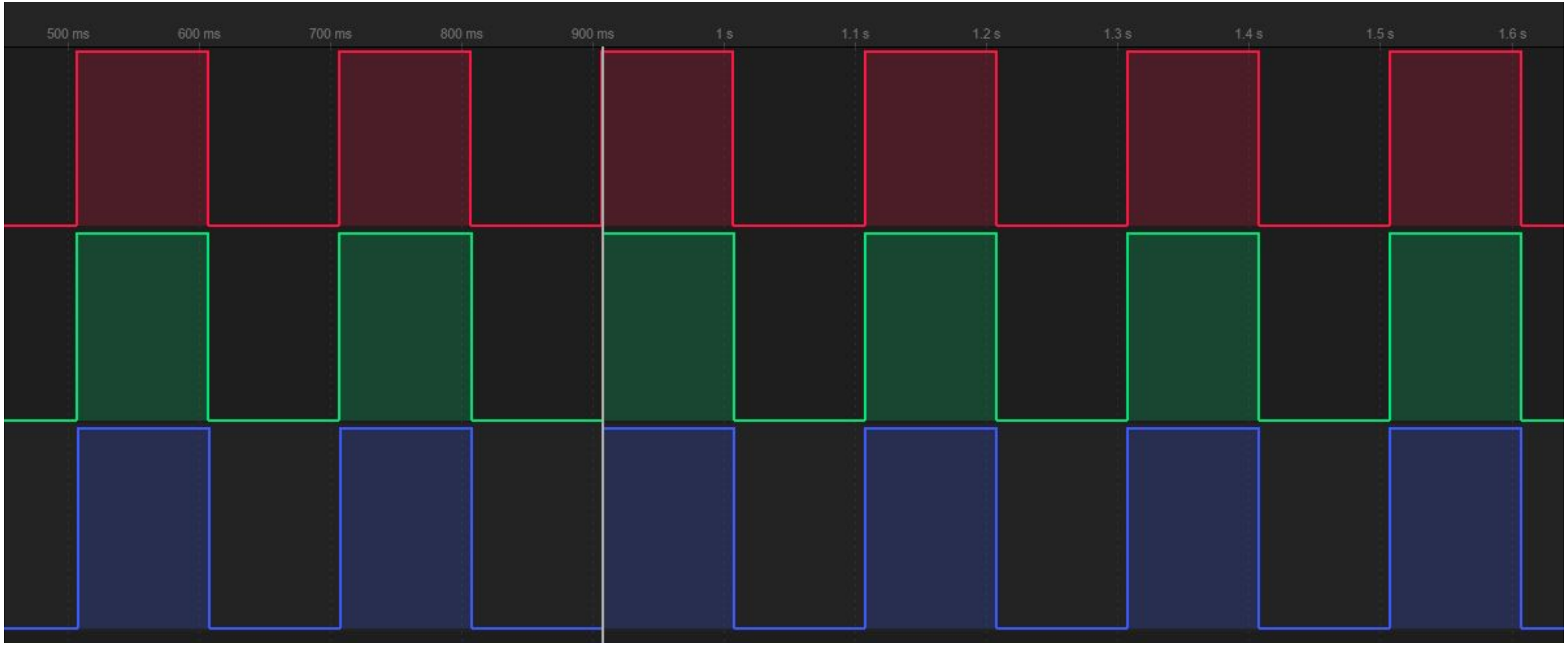
Observations

- Time to start task1 : 607.105 ms
- Time to start task2 : 607.131 ms
- Time to start task3 : 607.141 ms

Difference between turning GPIO OFF (Low State)

- Task2 - task1 = 0.026 ms
- Task 3 - task 2 = 0.009 ms

All cycles overview



Memory Consumption Analysis(Software Timer based 3 Tasks execution)

With same Callback function

Heap size at different steps of program execution

- From main 1 360576 // first statement in C “main” function
- From main 2 360476 // after setting GPIO Direction (Input and Output)
- From main 3 360308 // after calling/initializing timers
- From RED 365292 //same function but comparison based on timer ID
- From GREEN 365292 //same function but comparison based on timer ID
- From BLUE 365292 //same function but comparison based on timer ID

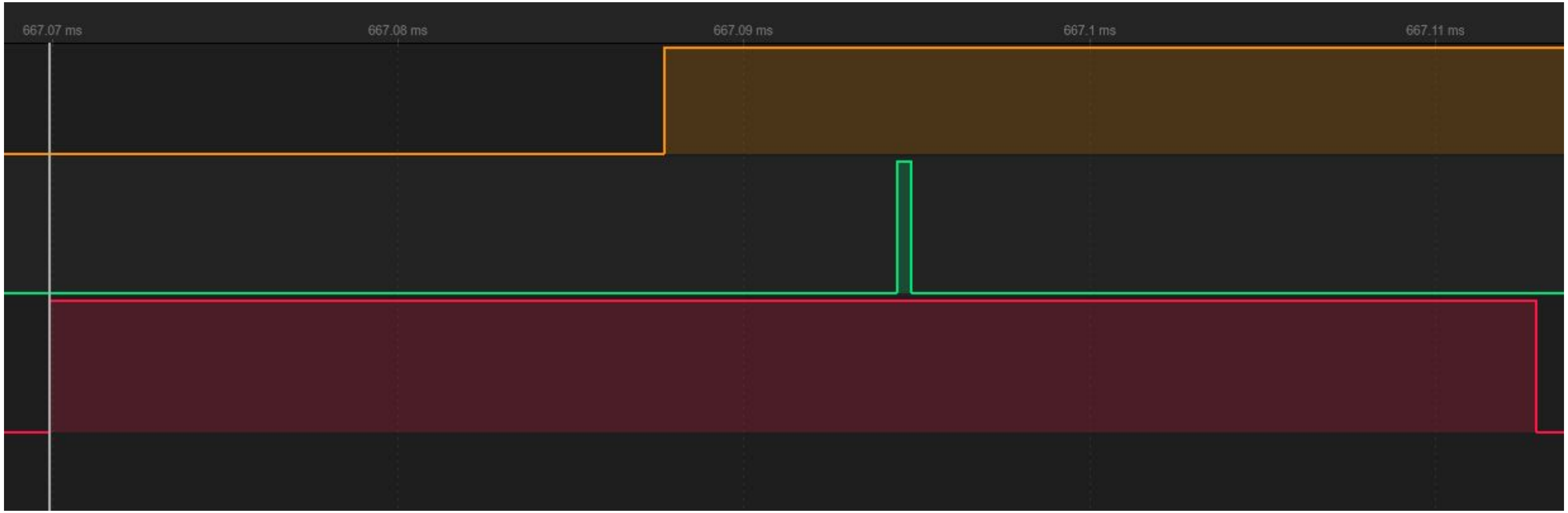
With different callback function

Heap size at different steps of program execution

- From main 1 360624 // first statement in C “main function”
- From main 2 360524 // after setting GPIO direction (Output)
- From main 3 360356 // after timer tasks are initialized
- From RED 365340 // inside callback with Red Callback
- From GREEN 365340 // inside callback with Green Callback
- From BLUE 365340 // inside callback with Blue Callback

Hardware Interrupts

Test: Simple Hardware Interrupt behavior



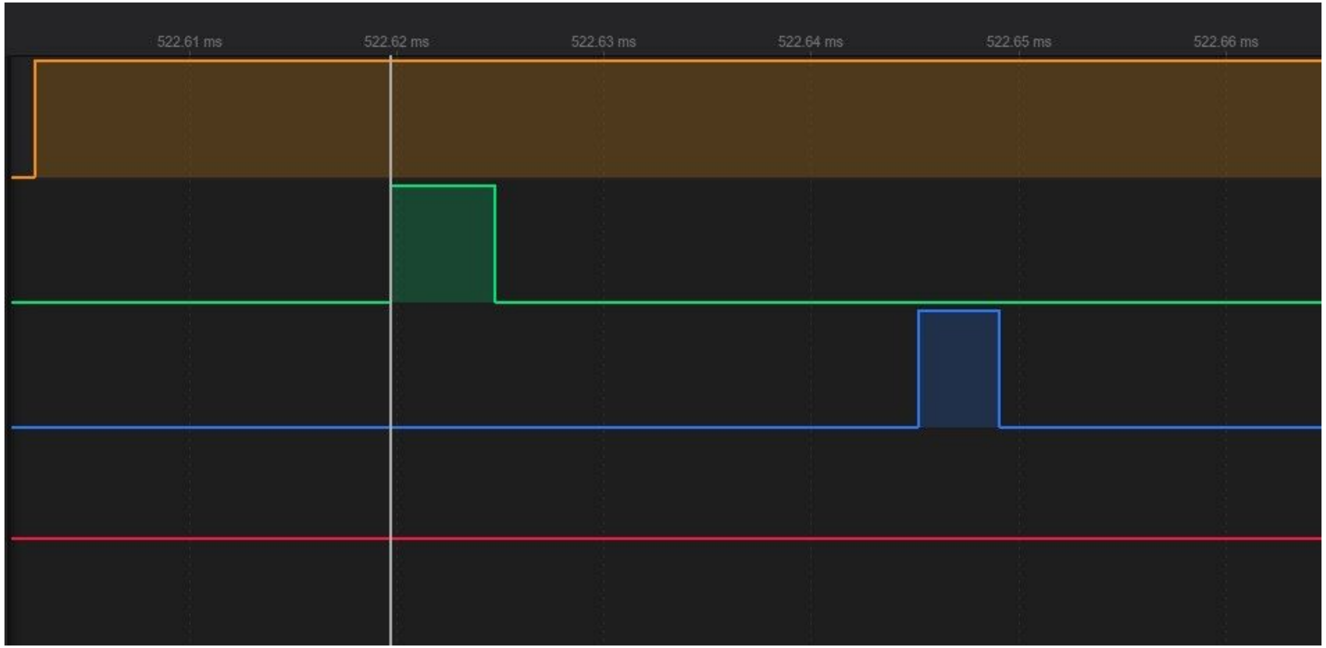
Hardware Interrupts

- Here, it can be observed that as soon as interrupt it generated at GPIO using push button, ISR(Interrupt Service Routine) is called and execution represented by Green Signal here. Since ISR has higher priority to regular tasks, therefore, ISR is completely executed and after that, execution of normal task(s) is continued.

- ISR returns the context back to task being executed , therefore, it is necessary to specify at end of ISR to perform context switch so that if Tick Period is expired and new task is scheduled by Scheduler, new task is put in execution rather old task (The task interrupted due to Hardware Interrupt).

Hardware Interrupt

Test: Interrupt Task Deferment



Hardware Interrupt

- Interrupt Task Deferment is a technique used to make ISR as short as possible.
- In this method, a binary semaphore is used to block “normal” task containing instructions to execute when hardware interrupt occurs.
- In ISR, Semaphore is “given” (released) from ISR and “normal” task that contains instructions when interrupt occurs is put in execution by Scheduler. This particular task is executed because of higher priority and being called (yielded) from ISR.
- As seen from previous Signal Behavior, First Green Signal (ISR) is toggled, than Blue Signal (task called from ISR) is toggled and after that, execution of other task is continued.
- ISR took more time in this test because of Semaphore Operation(s).

Hardware Analysis

Hardware Details

MCU : ESP32-S3 W-ROOM-1 CAM

Processor: LX7 Dual core capable of running at upto 240MHz

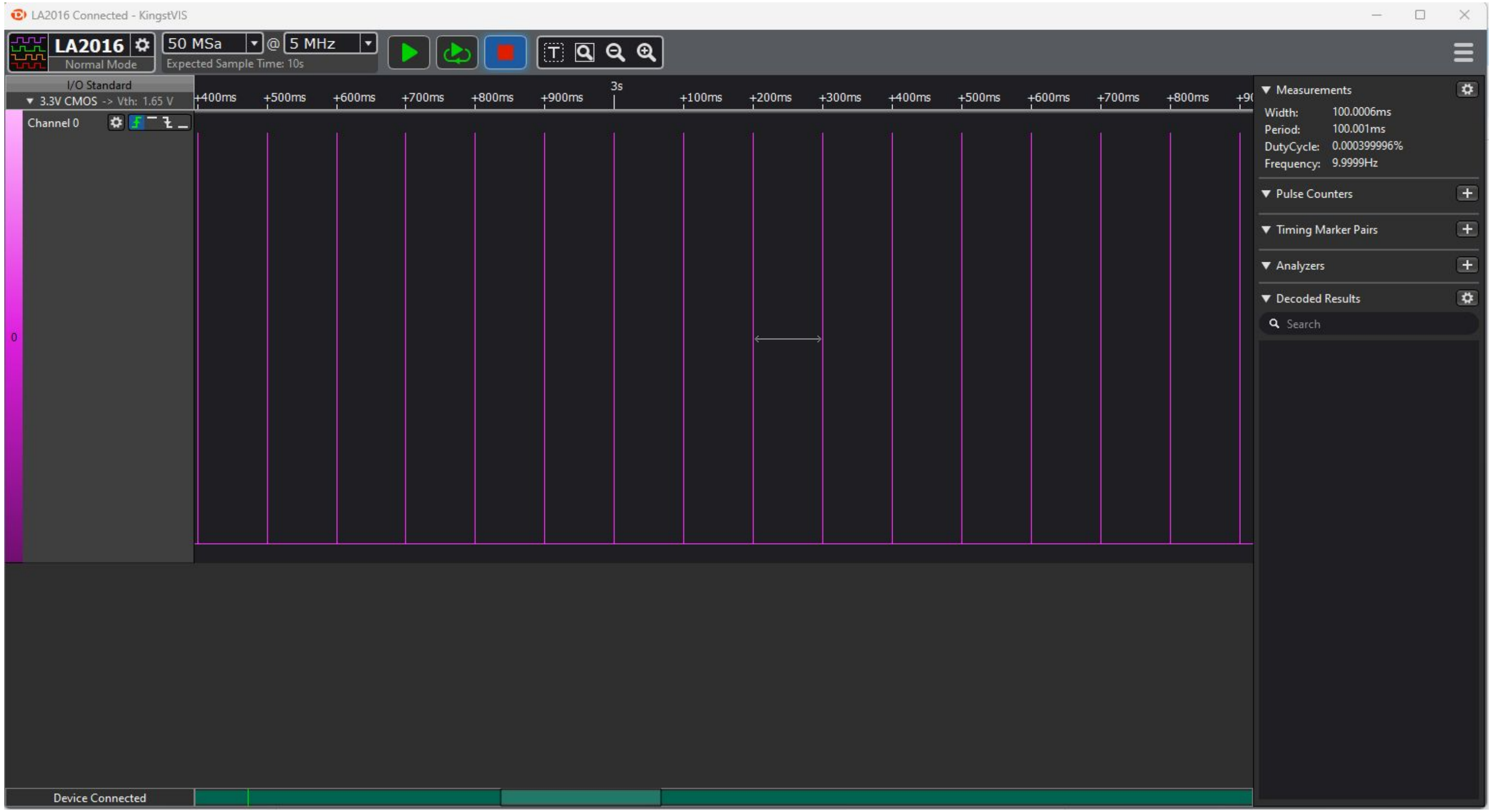
Crystal frequency : 40MHz

PSRAM : 8MB

WIFI , BLUETOOTH, USB OTG

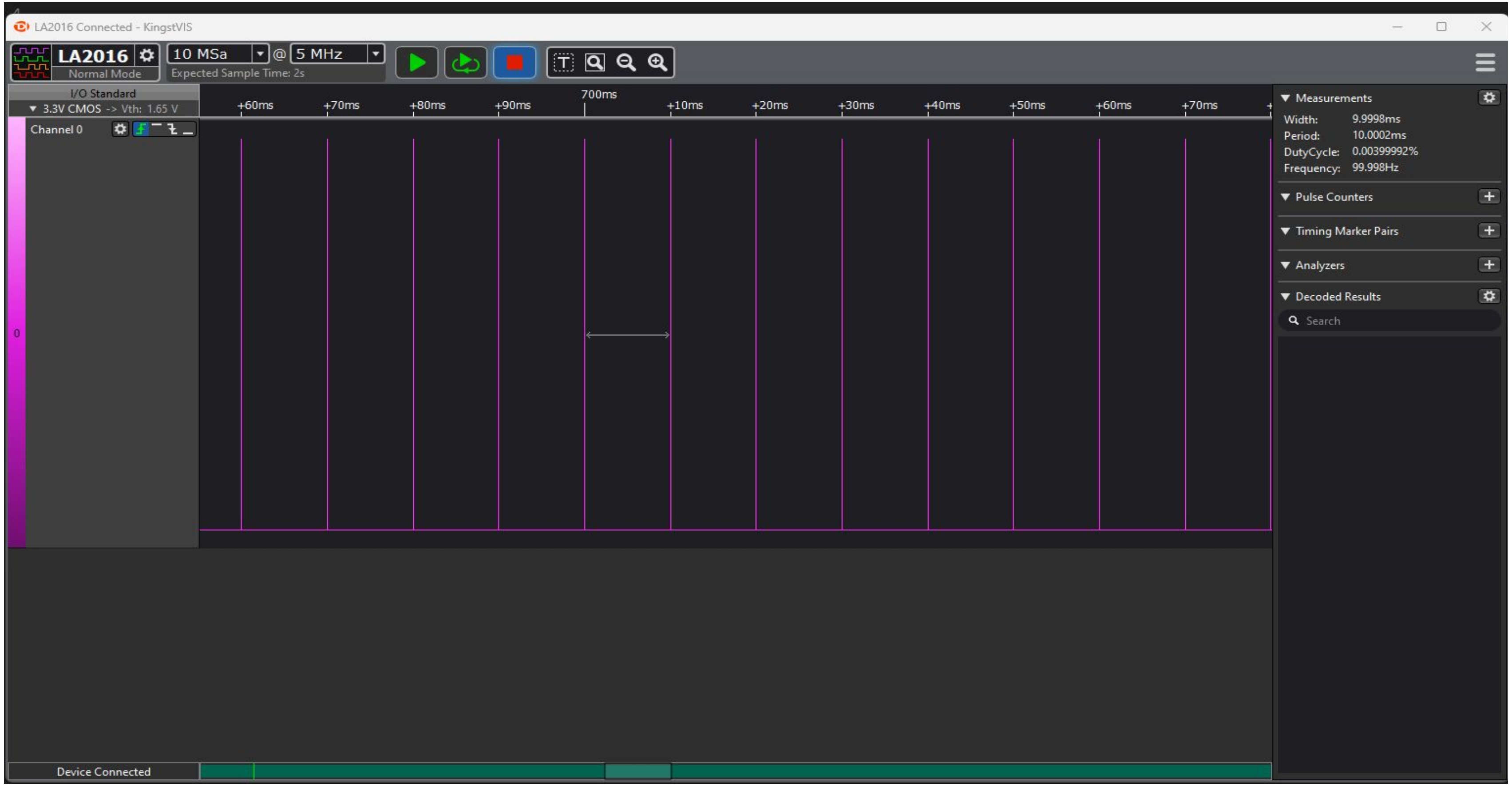
Capability of FAT File System

Test 1: 100 ms time delay with TICK_RATE_HZ equals 100 Hz (10 ms time period)



Width: 100.0006ms
Period: 100.001ms
DutyCycle: 0.000399996%
Frequency: 9.9999Hz

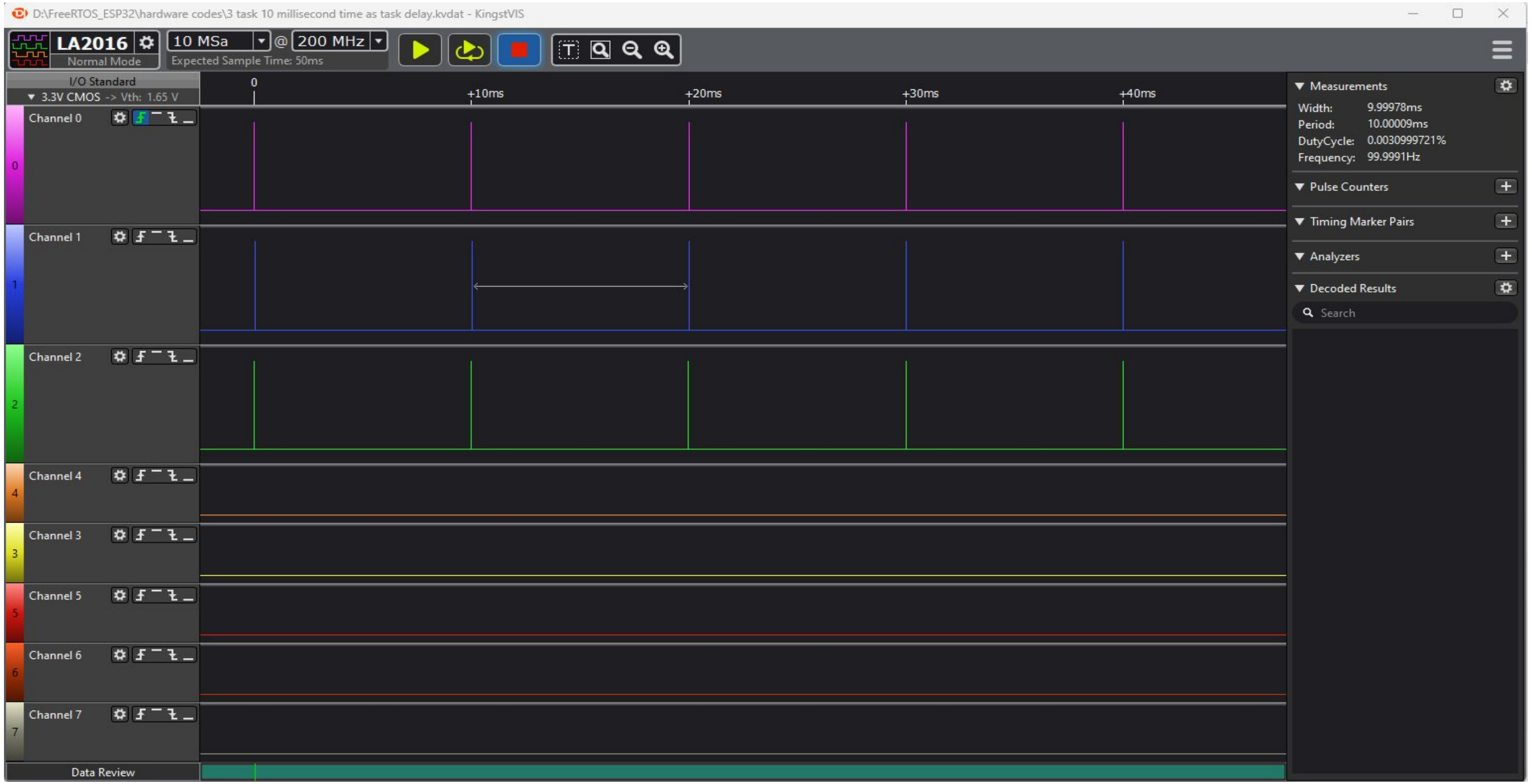
Test 2: 10 ms as time delay with TICK_RATE_HZ equals 100 Hz (10 ms time period)



Measurements

Width: 9.9998ms
Period: 10.0002ms
DutyCycle: 0.00399992%
Frequency: 99.998Hz

Test 3: 3 tasks with same priority , 10 ms delay and TICK_RATE_HZ equals 100 (default)


D:\FreeRTOS_ESP32\hardware codes\3 task 10 millisecond time as task delay.kvdat - KingstVIS

LA2016 Normal Mode
10 MSa @ 200 MHz
Expected Sample Time: 50ms

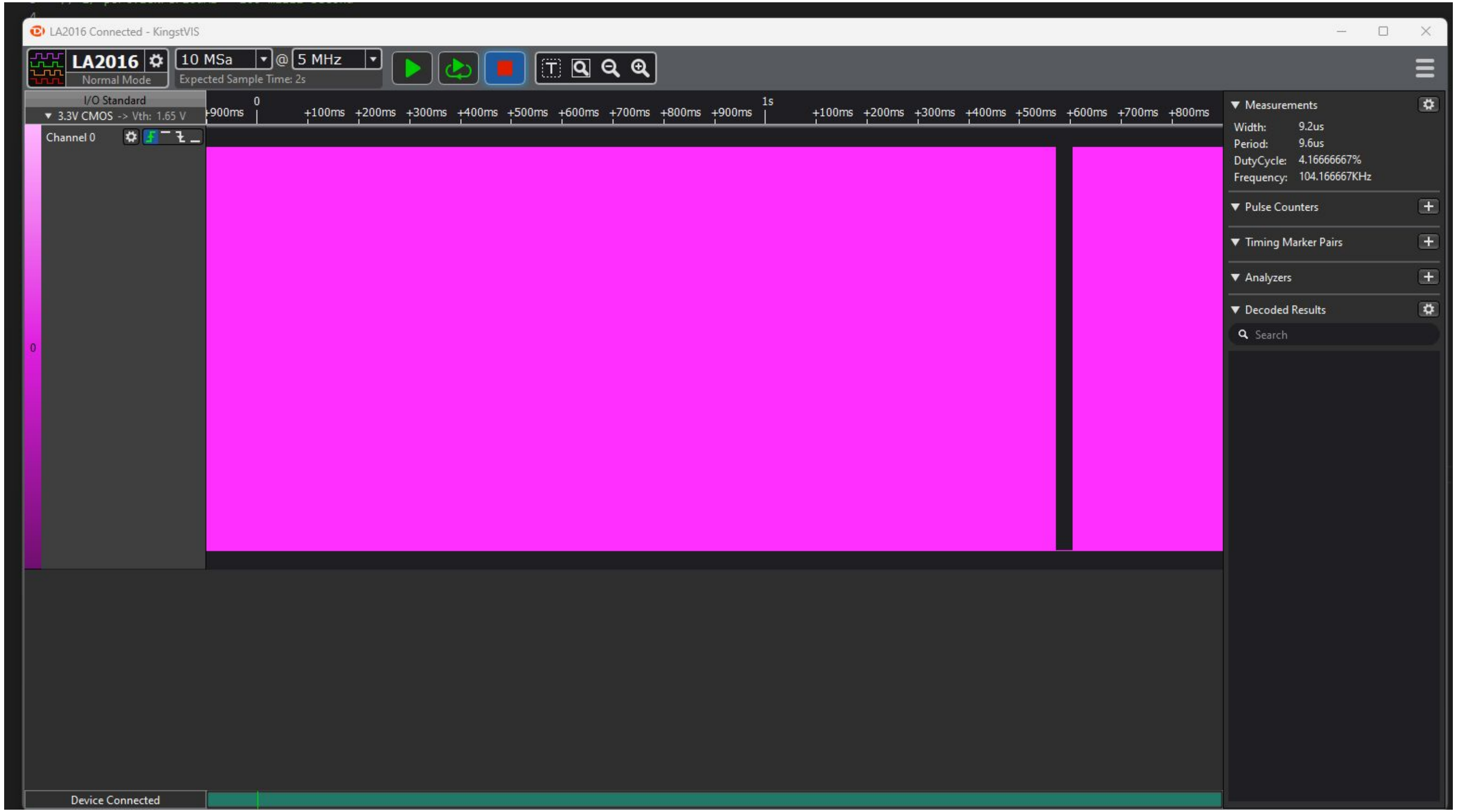
I/O Standard
3.3V CMOS -> Vth: 1.65 V

Measurements
Width: 9.99978ms
Period: 10.00009ms
DutyCycle: 0.0030999721%
Frequency: 99.9991Hz

Observations for test 1, test 2, test 3:

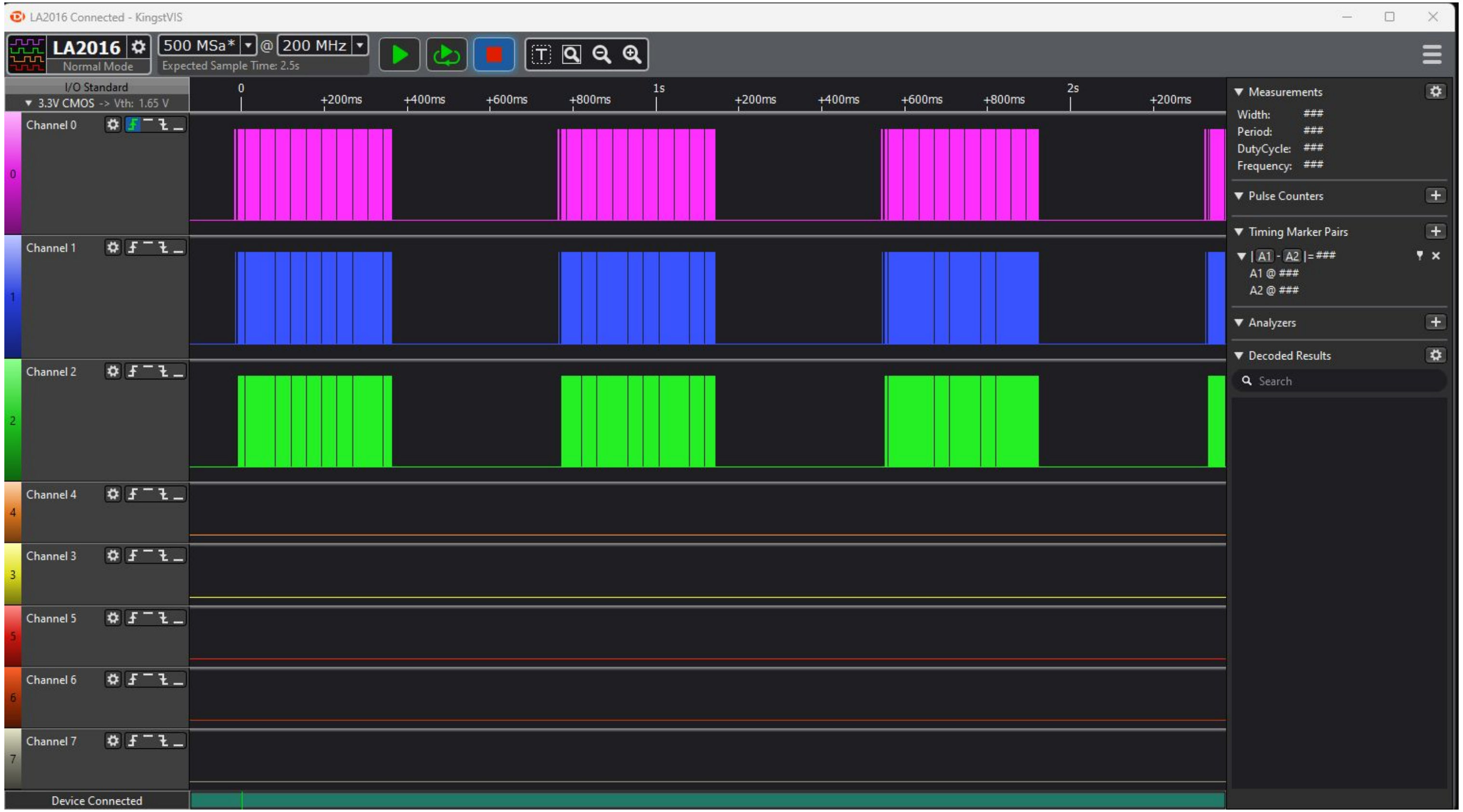
- It can be observed that execution time for task is not consistent.
- From data gathered during execution, tasks are sometimes executed with small delay and even sometimes little early than specified period.

Test 4: 1 ms Task delay with TICK_RATE_HZ equals 100 (default; 10 ms time period)



Width: 9.2us
Period: 9.6us
DutyCycle: 4.16666667%
Frequency: 104.166667KHz

Test 5: 3 tasks with same priority,1 ms Task delay with TICK_RATE_HZ equals 100 (default; 10 ms time period)



Observation for test 4 and test 5:

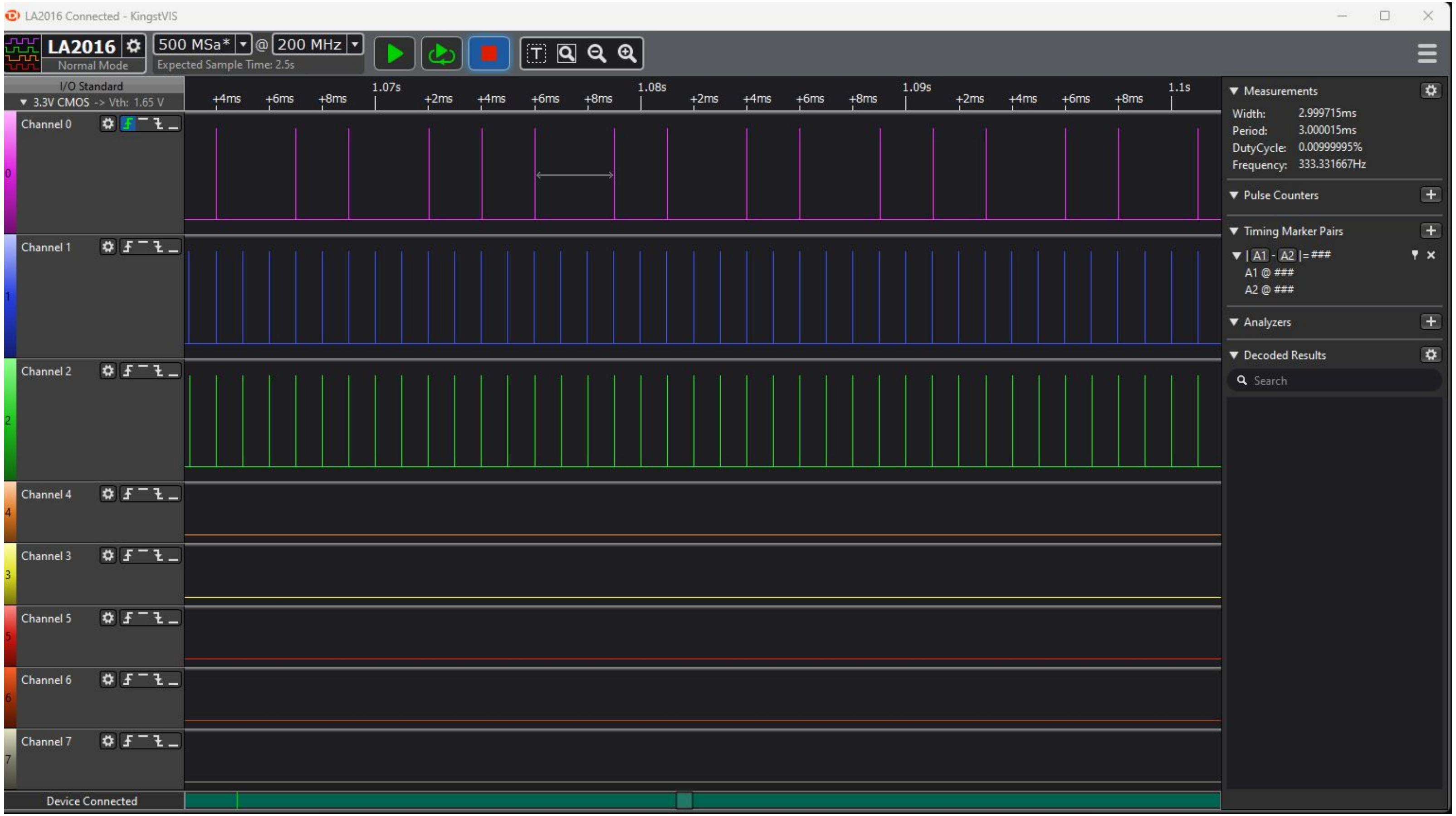
- When one task was executed with 1 ms time delay (task should execute after 1 ms) in test 4, it is observed that there delay between each time task is executed was 9.6 micro seconds instead of 1 ms. This is caused because TICK_PERIOD_HZ is set to 100 Hz which makes 10 ms time period. For this reason, there was no theoretically no time delay each time task is executed. However, in practical, there was time delay of 9.6 micro-seconds because this is the time required by Scheduler for switching between tasks.

- Similarly, in test 5, time delay was specified at 1 ms with Tick Period to be 10 ms, this caused unusual and unexpected behavior as observed in test 5 Signal patterns.

Changing TICK_RATE_HZ for smaller Tick Interrupt(s)

- TICK_RATE_HZ can be changed from default 100 MHz. Range includes 0 MHz to 1000 MHz.
- Below 1000 MHz is not supported by ESP-IDF
- Defining TICK_RATE_HZ at 1000 MHz can cause some undesirable results, therefore, caution should be taken to when designing/writing tasks’ “task”.
- Following Examples show how undesirable behavior can occur

Test 6 :3 task, same priority , 1 ms delay time


LA2016 Connected - KingstVIS

LA2016 Normal Mode

500 MSa* @ 200 MHz

Expected Sample Time: 2.5s

I/O Standard

3.3V CMOS -> Vth: 1.65 V

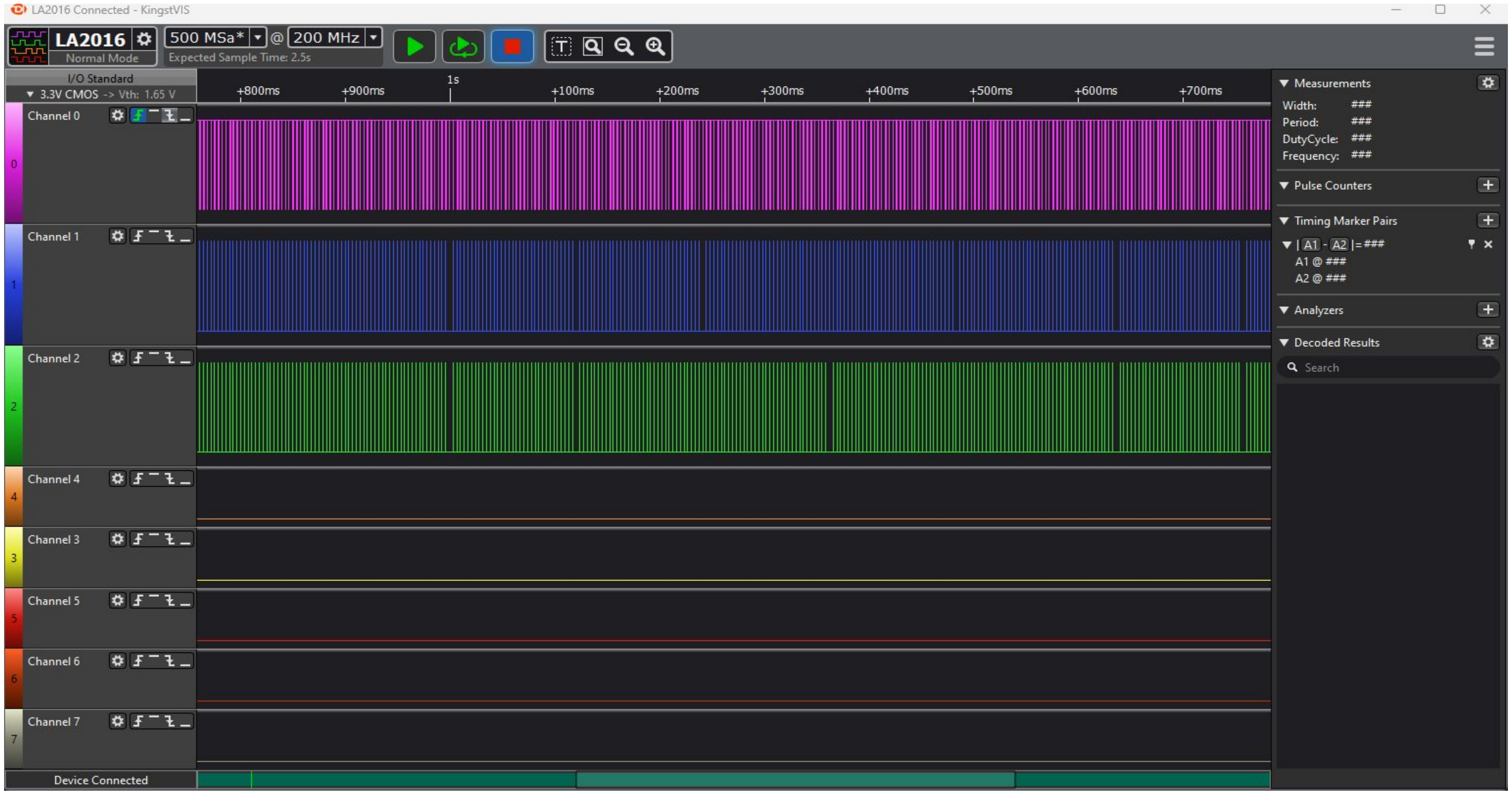
Measurements

```
Width:      2.999715ms
Period:     3.000015ms
DutyCycle:  0.00999995%
Frequency:  333.331667Hz
```

Timing Marker Pairs

```
| A1 - A2 | = ###
A1 @ ###
A2 @ ###
```

Test 7 :1 millisecond delay, 3 tasks, 2 tasks same priority, 1 task higher priority



LA2016 Connected - KingstVIS

LA2016 Normal Mode

500 MSa* @ 200 MHz

Expected Sample Time: 2.5s

I/O Standard

3.3V CMOS -> Vth: 1.65 V

Measurements

Width: ###

Period: ###

DutyCycle: ###

Frequency: ###

Pulse Counters

Timing Marker Pairs

| A1 - A2 | = ###

A1 @ ###

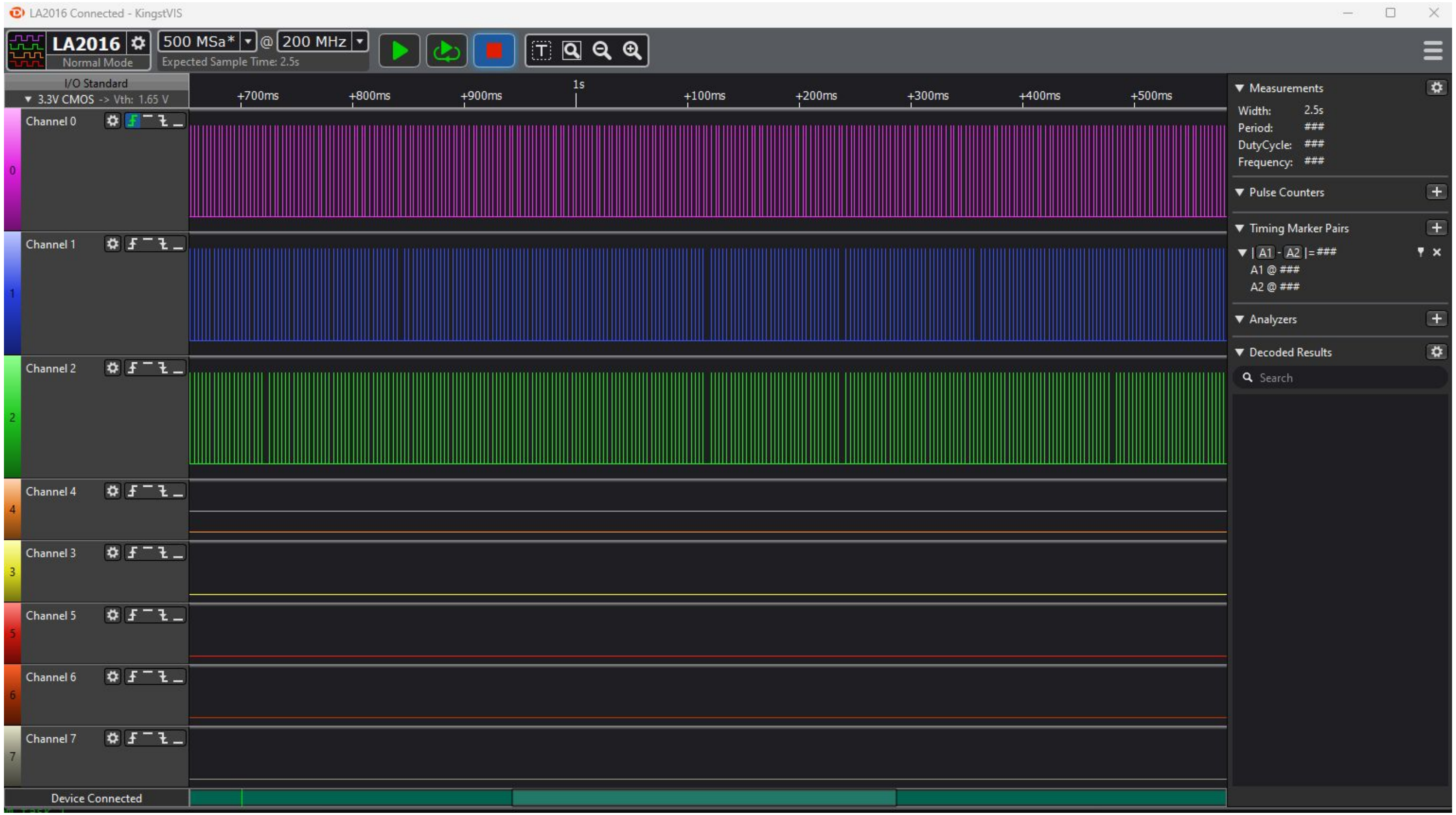
A2 @ ###

Analyzers

Decoded Results

Search

Test 8: 1 millisecond delay, 3 tasks, 2 tasks same priority, 1 task higher priority



Observations:

- Test 6: Here , it can be observed that task 1 (channel 0) has a different behavior as compared to other 2 tasks. This shows that with this small delay time , a small change such as logging info from a task can affect the behavior of task execution to a great extent.

- Test 7: 2 tasks have same task, one task is logging info on serial port (high priority).Task 2 and 3 are toggling GPIO, task 1 is toggling GPIO as well but logging on console between toggling state.

- Test 8: 2 tasks have same task, one task is logging info on serial port (high priority).Task 2 and 3 are toggling GPIO, task 1 is toggling GPIO as well but logging on console after toggling state

Findings

- Performance in regard to Behavior, FreeRTOS is following expected behavior
- Performance in regards to Timing, It can be concluded that FreeRTOS is performing good when Tick Interrupt is set to higher time period, whereas, decreasing time period can cause unexpected behavior due to overheads
- Time taken by Scheduler to perform context switching around 9.2 microseconds. This time depends highly on the Stack Size of Task and number of operations defined in Task.
- Overall, for applications that does not involve extremely precise and Hard Real-Time Deadlines, FreeRTOS can be used for achieving Multi-Tasking on MCUs with limited computation and memory.

Resources and Contact Information

Source codes, Notes as well as VCD files can be found at following git repository

http://gogs.es-lab.de/ali_anwer/Analysis-Of-FreeRTOS-On-ESP32.git

Contact information as well as more information can be found at:

Contact/Imprint - Embedded-Systems Lab (es-lab.de)

Disclaimer

Intention of this work to study Analysis of FreeRTOS on ESP32 SoC for “Educational” and “Research” Purposes. It does not represent in any way Espressif Systems and FreeRTOS.