Performance And Analysis of FreeRTOS on ESP32-S3

Submitted to:
Prof. Dr.-Ing. Ingo Chmielewski

Submitted By:
Ali Anwer

TABLE OF CONTENT

ESP32 Overview

ESP32 is successor of ESP8266 microcontroller series. It is developed by Espressif systems and manufactured by Taiwan Semiconductor Manufacturing Company (TSMC). ESP32 is claimed to be a low power, low cost System on Chip (SoC) microcontroller with WIFI and bluetooth integrated. Other features such as power management, low-noise receiver amplifiers are part of ESP32 MCU. ESP32 are available in 4 series based on type of processor used and have minor variations with-in series.

Based on the series , ESP32 can have a different MicroProcessor. Espressif Systems uses mainly 2 types of processors, namely Xtensa LX6 , Xtensa LX7 and RISC V. These are ESP32-S series , ESP32-C series, ESP32-H series and ESP32 series. Major difference between these series is the type of processor used and their processing capabilities. ESP32-S series uses single (Xtensa LX6 processor) and dual core (Xtensa LX7) 32 bit processors provided by Tensilica, A company based in silicon valley, now part of Cadence Design System. ESP32-C and ESP32-H series uses RISC-V single core 32 bit processor but with variation in number of pipelines in processor. ESP32 series use single as well as dual core 32 bit processors and offer largest variations in features.

ESP32 comes with WIFI and bluetooth integrated as part of MCU and does not require any additional module for WIFI or Bluetooth connectivity. ESP32 comes in different packaging variations such as ESP32-WROOM32 using Surface Mount Technology (SMT). Another variation found in different ESP32 MCU is availability of Pseudo-Static RAM (PSRAM or PSDRAM).

Xtensa LX Processors:

Xtensa LX Processor developed by Tensilica is a 32 bit architecture with 16-24 Instruction Set processor. Base Instruction Set Architecture (ISA) consists of approx 80 base Instructions and contains the capability of adding Instructions which can change the ABI and binary code produced by the compiler.

General Features of LX7 Processors

- Efficient real-time 32-bit base Xtensa processor architecture
- Configurable instruction and data caches and local memories
- Choose from pre-verified application-specific DSP ISAs
- IEEE 754-compliant single- and double-precision floating-point options
- Choice of low-power features extensibility with application-specific instructions,execution units, register files, and I/Os
- Multiple bus interface options including AXI4, AXI3, ACE-Lite, PIF, AHB-Lite, and iDMA

- Industry-standard debug features like JTAG and multi-core debug support
- Compatible with ARM® CoreSight™ debug and trace technology
- Processor-specific software, tools, and models generated automatically
- Mature C/C++ compiler with proven auto-vectorizing Capabilities

Efficient Base Architecture

- The Xtensa LX7 processor's 32-bit architecture features a compact instruction set optimized for embedded designs. The base architecture has a 32-bit ALU, up to 64 general-purpose physical registers, 6 special-purpose registers, and 80 base instructions, including 16- and 24-bit (rather than 32-bit) RISC instruction encoding
- Modelessly intermixed standard 16- and 24-bit instructions, as well as designer-defined FLexible Instruction length eXtension (FLIX) instructions of any size from 4 to 16 bytes, resulting in highly efficient code that is optimal for both memory size and performance
- Selectable 5-or-7-stage core ISA pipeline to accommodate different memory speeds

Extended DSP execution pipelines up to 31 stages
• Designer-defined instruction pipeline depths up to 31 stages

ESP32-S Series uses 2 types of Xtensa processors, namely Xtensa LX6 and Xtensa LX7.

ESP32-C3 ,RISC-V based ESP32 processor:

RISC-V is an Open Source ISA based on RISC ISA. RISC-V comes with a very small ISA as part of base ISA and have capability to add instruction as extension to base architecture. RISC-V is a load-store architecture The instruction set is designed for a wide range of uses. The base instruction set has a fixed length of 32-bit naturally aligned instructions, and the ISA supports variable length extensions where each instruction can be any number of 16-bit parcels in length. Subsets support small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale 19 inch rack-mounted parallel computers.

The instruction set specification defines 32-bit and 64-bit address space variants. The specification includes a description of a 128-bit flat address space variant, as an extrapolation of 32 and 64 bit variants, but the 128-bit ISA remains "not frozen" intentionally, because there is yet so little practical experience with such large memory systems.

Since it is an Open Source ISA, therefore, a large number of community contribute to the overall development process of ISA that ranges from hardware design, SOCs, ISA extension development to documentation about RISC-V.

RISC-V operated in 3 modes ,namely U mode, S mode and M mode. U mode is for User Space applications where U-mode is for user processes such as applications defines by users or User Space applications, S-mode is for kernel and device drivers and M-mode is for boot loaders and

firmwares. Each mode have specific CSR (Control and Status Registers) and higher privilege mode can access CSR of lower privilege mode.

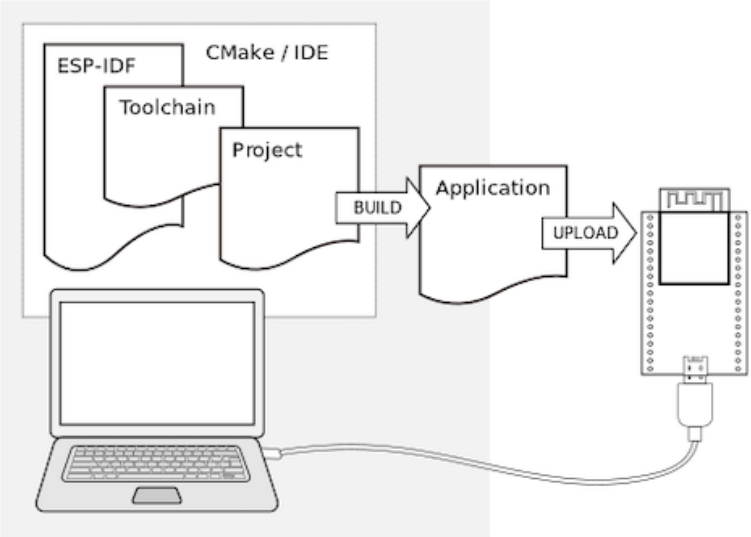
Programming environments:

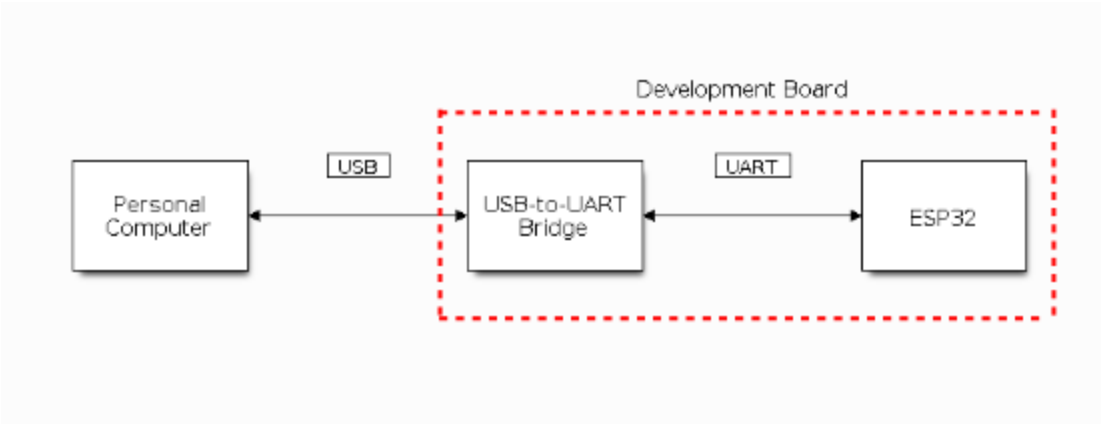
ESP32 can be programmed using various languages and environments. These includes Arduino IDE (C / C++) , Thonny IDE (micro python) , Lua programming language as well as JavaScript (Espruino framework).

Espressif provides ESP-IDF (IoT Development Framework) that can be used in various IDEs such as Visual Studio Code (vsCode) along with platformIO (for board management). However, the official supported compiler is for C/C++ by Espressif and other supported languages and environments are third party or open source.

ESP32 supports transferring application through serial comm as well as OTA (Over The Air) update, however, OTA requires initial sketch to be uploaded before-hand in order to use OTA for firmware/application update.

A typical workflow using ESP-ISF.





ESP32 and Real time Capabilities

ESP32 supports utilization of different RTOS (open source as well as closed/proprietary) to achieve Real-Time capabilities in applications. Espressif provides ESP-IDF that already includes the System API for FreeRTOS “ported”to ESP32 based modules, however, other RTOS can be ported to ESP32 Eco-System as well. Some of the RTOS that are widely used are listed below.

FreeRTOS:

- Lightweight and widely used open-source RTOS.
- Provides task scheduling, inter-task communication, and synchronization mechanisms.
- Supports multiple architectures, including the ESP32.
- Offers a large community and extensive documentation.
- Well-suited for resource-constrained devices and applications with real-time requirements.

RIOT OS:

- Lightweight and energy-efficient open-source RTOS.

- Designed specifically for IoT devices and supports a wide range of platforms, including the ESP32.
- Provides various network protocols and a modular architecture.
- Offers a low memory footprint and efficient energy management.
- Suitable for IoT applications with constrained resources and real-time requirements.

NuttX:

- Real-time embedded operating system with a small footprint.
- Supports a variety of architectures, including the ESP32.
- Provides a POSIX-like interface and a rich set of features.
- Offers a scalable design with optional components.
- Suitable for applications requiring real-time capabilities and POSIX compatibility.

Zephyr:

- Scalable open-source RTOS designed for resource-constrained systems.
- Supports multiple architectures, including the ESP32.
- Provides a wide range of drivers, libraries, and protocols.
- Offers a modular and customizable architecture.
- Suitable for applications requiring real-time capabilities, connectivity, and scalability.

Mongoose OS:

- Lightweight and efficient RTOS with a focus on IoT applications.
- Supports multiple platforms, including the ESP32.
- Provides built-in networking protocols, device management, and cloud integration.
- Offers real-time capabilities for time-sensitive applications.
- Suitable for IoT applications requiring real-time responsiveness and connectivity.

FreeRTOS

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements. It allows applications to be organized as a collection of independent threads of execution. On a processor that has only one core, only a single thread can be executed at any one time. The kernel decides which thread should be executed by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

In FreeRTOS, each thread of execution is called a ‘task’. There is no consensus on terminology within the embedded community, but I prefer ‘task’ to ‘thread,’ as thread can have a more specific meaning in some fields of application.

Why Use a Real-time Kernel?

Abstracting away timing information
The kernel is responsible for execution timing and provides a time-related API to the application. This allows the structure of the application code to be simpler, and the overall code size to be smaller

Maintainability/Extensibility
Abstracting away timing details results in fewer interdependencies between modules, and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.

Modularity
Tasks are independent modules, each of which should have a well-defined purpose.

Team development
Tasks should also have well-defined interfaces, allowing easier development by teams. Easier testing If tasks are well-defined independent modules with clean interfaces, they can be tested in isolation.

Code reuse
Greater modularity and fewer interdependencies results in code that can be reused with less effort.

Improved efficiency
Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done. Counter to the efficiency saving is the need to process the RTOS tick interrupt, and to switch execution from one task to another. However, applications that don't make use of an RTOS normally include some form of tick interrupt anyway.

Idle time
The Idle task is created automatically when the scheduler is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

Power Management

The efficiency gains that are obtained by using an RTOS allow the processor to spend more time in a low power mode. Power consumption can be decreased significantly by placing the processor into a low power state each time the Idle task runs. FreeRTOS also has a special tick-less mode. Using the tick-less mode allows the processor to enter a lower power mode than would otherwise be possible, and remain in the low power mode for longer.

Flexible interrupt handling
Interrupt handlers can be kept very short by deferring processing to either a task created by the application writer, or the FreeRTOS daemon task.

Mixed processing requirements
Simple design patterns can achieve a mix of periodic, continuous and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities

What is FreeRTOS ?

FreeRTOS can be thought of as a library that provides multi-tasking capabilities to what would otherwise be a bare metal application. FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port. Build the source files as part of your project to make the FreeRTOS API available to your application. To make this easy for you, each official FreeRTOS port is provided with a demo application. The demo application is pre-configured to build the correct source files, and include the correct header files.

FreeRTOS has the following standard features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Flexible, fast and light weight task notification mechanism
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive Mutexes
- Software timers
- Event groups

- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering.
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications
- Software managed interrupt stack when appropriate (this can help save RAM)

FreeRTOSConfig.h

FreeRTOS is configured by a header file called FreeRTOSConfig.h. FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example, FreeRTOSConfig.h contains constants such as configUSE_PREEMPTION, the setting of which defines whether the co-operative or pre-emptive scheduling algorithm will be used1 . As FreeRTOSConfig.h contains application specific definitions, it should be located in a directory that is part of the application being built, not in a directory that contains the FreeRTOS source code.

FreeRTOS Source Files Common to All Ports The core FreeRTOS source code is contained in just two C files that are common to all the FreeRTOS ports. These are called tasks.c, and list.c, and they are located directly in the FreeRTOS/Source directory .In addition to these two files, the following source files are located in the same directory:

queue.c

provides both queue and semaphore services, as described later in this book. queue.c is nearly always required.

timers.c

timers.c provides software timer functionality, as described later in this book. It need only be included in the build if software timers are actually going to be used.

event_groups.c

event_groups.c provides event group functionality, as described later in this book. It need only be included in the build if event groups are actually going to be used. Croutine.c

croutine.c

Croutine.c implements the FreeRTOS co-routine functionality. It need only be included in the build if co-routines are actually going to be used. Co-routines were intended for use on very small microcontrollers, are rarely used now, and are therefore not maintained to the same level as other FreeRTOS features

Data Types Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two port specific data types: TickType_t and BaseType_t. These data types are described as follows:

TickType_t

FreeRTOS configures a periodic interrupt called the tick interrupt. The number of tick interrupts that have occurred since the FreeRTOS application started is called the tick count. The tick count is used as a measure of time. The time between two tick interrupts is called the tick period. Times are specified as multiples of tick periods. TickType_t is the data type used to hold the tick count value, and to specify times. TickType_t can be either an unsigned 16-bit type, or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is set to 1, then TickType_t is defined as uint16_t. If configUSE_16_BIT_TICKS is set to 0 then TickType_t is defined as uint32_t. Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit architectures, but severely limits the maximum block period that can be specified. There is no reason to use a 16-bit type on a 32-bit architecture.

BaseType_t

This is always defined as the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture. BaseType_t is generally used for return types that can take only a very limited range of values, and for pdTRUE/pdFALSE type Booleans.

Variable Names

Variables are prefixed with their type: ‘c’ for char, ‘s’ for int16_t (short), ‘l’ int32_t (long), and ‘x’ for BaseType_t and any other non-standard types (structures, task handles, queue handles, etc.). If a variable is unsigned, it is also prefixed with a ‘u’. If a variable is a pointer, it is also prefixed with a ‘p’. For example, a variable of type uint8_t will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.

Function Names

Functions are prefixed with both the type they return, and the file they are defined within. For example:

vTaskPrioritySet() returns a void and is defined within task.c
xQueueReceive() returns a variable of type BaseType_t and is defined within queue.c
pvTimerGetTimerID() returns a pointer to void and is defined within timers.c.

File scope (private) functions are prefixed with ‘prv’

Macro Names

Most macros are written in upper case, and prefixed with lower case letters that indicate where the macro is defined

port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Note that the semaphore API is written almost entirely as a set of macros, but follows the function naming convention, rather than the macro naming convention

Dynamic Memory Allocation and its Relevance to FreeRTOS

From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time: Following chapters of this book will introduce kernel objects such as tasks, queues, semaphores and event groups. To make FreeRTOS as easy to use as possible, these kernel objects are not statically allocated at compile-time, but dynamically allocated at run-time; FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each

time a kernel object is deleted. This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

Dynamic memory allocation is a C programming concept, and not a concept that is specific to either FreeRTOS or multitasking. It is relevant to FreeRTOS because kernel objects are allocated dynamically, and the dynamic memory allocation schemes provided by general purpose compilers are not always suitable for real-time applications. Memory can be allocated using the standard C library malloc() and free() functions, but they may not be suitable, or appropriate, for one or more of the following reasons:

- They are not always available on small embedded systems.
- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call.
- They can suffer from fragmentation1 .
- They can complicate the linker configuration.
- They can be the source of difficult to debug errors if the heap space is allowed to grow into memory used by other variables.

FreeRTOS now treats memory allocation as part of the portable layer (as opposed to part of the core code base). This is in recognition of the fact that different embedded systems have varying dynamic memory allocation and timing requirements, so a single dynamic memory allocation algorithm will only ever be appropriate for a subset of applications. Also, removing dynamic memory allocation from the core code base enables application writer's to provide their own specific implementations, when appropriate.

When FreeRTOS requires RAM, instead of calling malloc(), it calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the kernel calls vPortFree(). pvPortMalloc() has the same prototype as the standard C library malloc() function, and vPortFree() has the same prototype as the standard C library free() function. pvPortMalloc() and vPortFree() are public functions, so can also be called from application code

Task Management

- How FreeRTOS allocates processing time to each task within an application.
- How FreeRTOS chooses which task should execute at any given time.
- How the relative priority of each task affects system behavior.
- The states that a task can exist in.
- How to implement tasks.
- How to create one or more instances of a task.
- How to use the task parameter.
- How to change the priority of a task that has already been created.
- How to delete a task.

- How to implement periodic processing using a task
- When the idle task will execute and how it can be used.

Task Functions

Tasks are implemented as C functions. The only thing special about them is their prototype, which must return void and take a void pointer parameter.
Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit.FreeRTOS tasks must not be allowed to return from their implementing function in any way— they must not contain a ‘return’ statement and must not be allowed to execute past the end of the function. If a task is no longer required, it should instead be explicitly deleted.
A single task function definition can be used to create any number of tasks—each created task being a separate execution instance, with its own stack and its own copy of any automatic (stack) variables defined within the task itself

Task States

An application can consist of many tasks. If the processor running the application contains a single core, then only one task can be executed at any given time. This implies that a task can exist in one of two states, Running and Not Running. This simplistic model is considered first—but keep in mind that it is an over simplification.When a task is in the Running state the processor is executing the task’s code. When a task is in the Not Running state, the task is dormant, its status having been saved ready for it to resume execution the next time the scheduler decides it should enter the Running state. When a task resumes execution, it does so from the instruction it was about to execute before it last left the Running state

A task transitioned from the Not Running state to the Running state is said to have been ‘switched in’ or ‘swapped in’. Conversely, a task transitioned from the Running state to the Not Running state is said to have been ‘switched out’ or ‘swapped out’. The FreeRTOS scheduler is the only entity that can switch a task in and out.

Creating Tasks

The xTaskCreate() API Function and e xTaskCreateStatic() function.
Tasks are created using the FreeRTOS xTaskCreate() API function. This is probably the most complex of all the API functions, so it is unfortunate that it is the first encountered, but tasks must be mastered first as they are the most fundamental component of a multitasking system.

A **prototype** for creating a xTASKCreate() API is

BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, uint16_t usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask);

pvTaskCode

Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function that implements the task (in effect, just the name of the function).

pcName

A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a task by a human readable name is much simpler than attempting to identify it by its handle

usStackDepth

Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack.

The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32-bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes). The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type uint16_t. The size of the stack used by the Idle task is defined by the application defined constant configMINIMAL_STACK_SIZE1 . The value assigned to this constant in the FreeRTOS demo application for the processor architecture being used is the minimum recommended for any task. If your task uses a lot of stack space, then you must assign a larger value.

There is no easy way to determine the stack space required by a task. It is possible to calculate, but most users will simply assign what they think is a reasonable value, then use the features provided by FreeRTOS to ensure that the space allocated is indeed adequate, and that RAM is not being wasted unnecessarily. Section 12.3, Stack Overflow, contains information on how to query the maximum stack space that has actually been used by a task.

pvParameters

Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task

uxPriority

Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority. configMAX_PRIORITIES is a user defined constant.Passing a uxPriority value above (configMAX_PRIORITIES – 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value.

pxCreatedTask

pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

Returned value

There are two possible return values:
pdPASS: This indicates that the task has been created successfully.
pdFAIL: This indicates that the task has not been created because there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

Task Priorities

The uxPriority parameter of the xTaskCreate() API function assigns an initial priority to the task being created. The priority can be changed after the scheduler has been started by using the vTaskPrioritySet() API function. The maximum number of priorities available is set by the application-defined configMAX_PRIORITIES compile time configuration constant within FreeRTOSConfig.h. Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible. Therefore, the range of available priorities is 0 to (configMAX_PRIORITIES – 1). Any number of tasks can share the same priority—ensuring maximum design flexibility

The FreeRTOS scheduler can use one of two methods to decide which task will be in the Running state. The maximum value to which configMAX_PRIORITIES can be set depends on the method used:

Generic Method

The generic method is implemented in C, and can be used with all the FreeRTOS architecture ports. When the generic method is used, FreeRTOS does not limit the maximum value to which configMAX_PRIORITIES can be set. However, it is always advisable to keep the configMAX_PRIORITIES value at the minimum necessary, because the higher its value, the more RAM will be consumed, and the longer the worst case execution time will be. The generic method will be used if configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0 in FreeRTOSConfig.h, or if configUSE_PORT_OPTIMISED_TASK_SELECTION is left undefined, or if the generic method is the only method provided for the FreeRTOS port in use

Architecture Optimized Method

The architecture optimized method uses a small amount of assembler code, and is faster than the generic method. The configMAX_PRIORITIES setting does not affect the worst case execution time. If the architecture optimized method is used then configMAX_PRIORITIES cannot be greater than 32. As with the generic method, it is advisable to keep configMAX_PRIORITIES at the minimum necessary, as the higher its value, the more RAM will be consumed. The architecture optimized method will be used if configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1 in FreeRTOSConfig.h. Not all FreeRTOS ports provide an architecture optimized method.

The FreeRTOS scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn.

Using vTASKdelay() and vTASKDelayUntil() API functions.both have different utilizations. vTASKDelay() function is relative to the time it is called. It can be used to convert tasks from polling to event driven approach. For Example , assume we have 2 tasks with different priorities, high priority tasks will always be running and low priority tasks will be starving of processing time. To avoid this behavior , we can use event driver approaches that will avoid this starvation problem. Event driven approaches can be of two types, temporal and synchronized which can be further divided into more types of events.

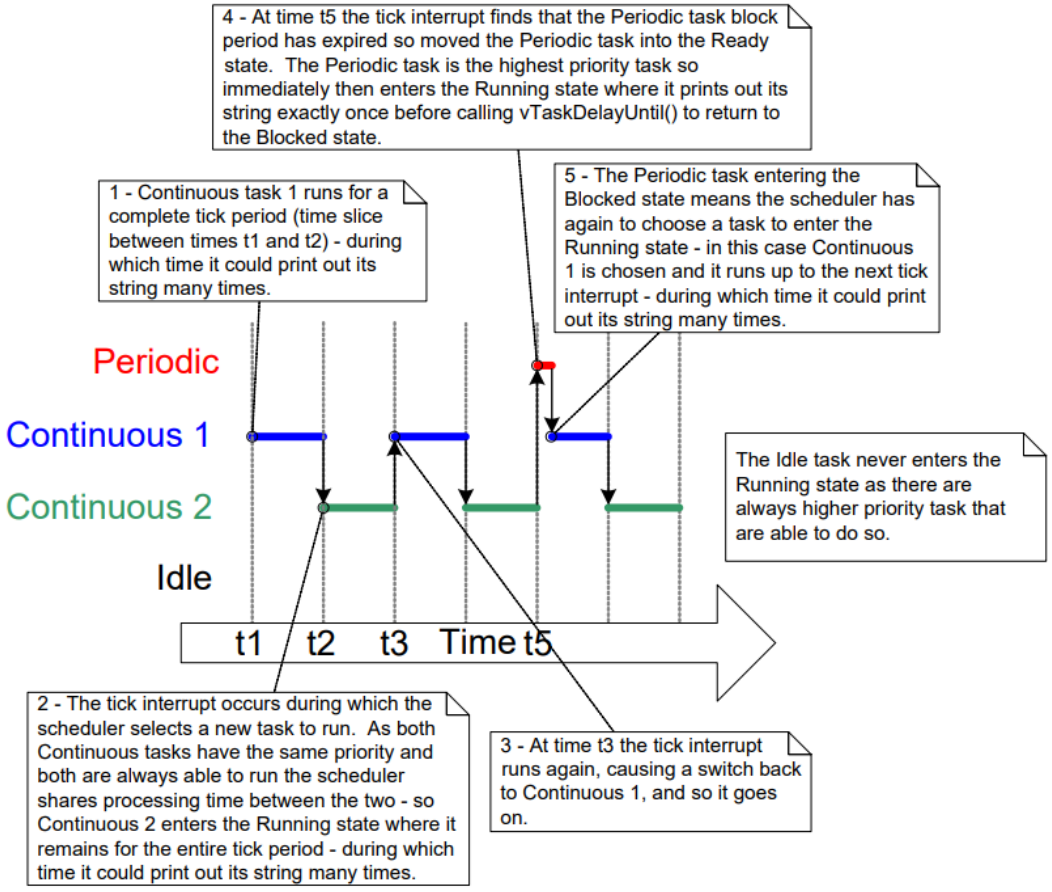
Difference between vTaskDelay() and vTASKDelayUntil() is that vTASkDelay() is relative whereas vTASKDelayUntil() is absolute. Let's assume a function called a VTaskDelay() and uses 10 ticks as parameter, and the function is called at 1200 ticks. Then the calling task is moved from the blocked state to the ready state at 1210 ticks. On the other hand, if vTaskDelayUntil() API is called with some tick value or time value through another conversion API , the task will be executed at ticks (or time) specified at absolute count.

Combining blocking and non-blocking tasks:

Previous examples have examined the behavior of both polling and blocking tasks in isolation. This example re-enforces the stated expected system behavior by demonstrating an execution sequence when the two schemes are combined, as follows.

Two tasks are created at priority 1. These do nothing other than continuously print out a string.
These tasks never make any API function calls that could cause them to enter the Blocked state, so are always in either the Ready or the Running state. Tasks of this nature are called ‘continuous processing’ tasks, as they always have work to do (albeit rather trivial work, in this case)

A third task is then created at priority 2, so above the priority of the other two tasks. The third task also just prints out a string, but this time periodically, so uses the vTaskDelayUntil() API function to place itself into the Blocked state between each print
Iteration
Expected behavior



Changing the Priority of a Task

The vTaskPrioritySet() API function can be used to change the priority of any task after the scheduler has been started. Note that the vTaskPrioritySet() API function is available only when INCLUDE_vTaskPrioritySet is set to 1 in FreeRTOSConfig.h.
Parameters to this API function are as follows

pxTask

The handle of the task whose priority is being modified (the subject task)—see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.
A task can change its own priority by passing NULL in place of a valid task handle

uxNewPriority

The priority to which the subject task is to be set. This is capped automatically to the maximum available priority of (configMAX_PRIORITIES – 1), where configMAX_PRIORITIES is a compile time constant set in the FreeRTOSConfig.h header file.

Schedulers

Round Robin for all tasks with same priority: does not guarantee equal time sharing between tasks.

‘Fixed Priority Pre-emptive Scheduling with Time Slicing’, which is the scheduling algorithm used by most small RTOS applications. This can be selected by configuring FreeRTOS_config.h file in following way.

configUSE_PREEMPTION 1
configUSE_TIME_SLICING 1
Explanation of terms in name
Fixed Priority Scheduling algorithms described as ‘Fixed Priority’ do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority, or that of other tasks

Pre-Emptive

Pre-emptive scheduling algorithms will immediately ‘pre-empt’ the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state.

Time Slicing

Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using ‘Time Slicing’ will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Prioritized Pre-Emptive Scheduling (without Time Slicing)

Prioritized Preemptive Scheduling without time slicing maintains the same task selection and pre-emption algorithms as described in the previous section, but does not use time slicing to share processing time between tasks of equal priority.
configUSE_PREEMPTION 1
configUSE_TIME_SLICING 0
if time slicing is used, and there is more than one ready
state task at the highest priority that is able to run, then the scheduler will select a new task to enter the Running state during each RTOS tick interrupt (a tick interrupt marking the end of a time slice). If time slicing is not used, then the scheduler will only select a new task to enter the Running state when either:

- A higher priority task enters the Ready state.
- The task in the Running state enters the Blocked or Suspended state

There are fewer task context switches when time slicing is not used than when time slicing is used. Therefore, turning time slicing off results in a reduction in the scheduler’s processing overhead. However, turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time, a scenario demonstrated by Figure 29. For this reason, running the scheduler without time slicing is considered an advanced technique that should only be used by experienced user

Co-operative Scheduling

configUSE_PREEMPTION 0
configUSE_TIME_SLICING Any value

When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling taskYIELD(). Tasks are never pre-empted, so time slicing cannot be used. systems will be less responsive when the co-operative scheduler is used than when the pre-emptive scheduler is used

- When the pre-emptive scheduler is used the scheduler will start running a task immediately that the task becomes the highest priority Ready state task. This is often essential in real-time systems that must respond to high priority events within a defined time period.
- When the co-operative scheduler is used a switch to a task that has become the highest priority Ready state task is not performed until the Running state task enters the Blocked state or calls taskYIELD()

Queue Management

Queue provides task-to-task , task-to-interrupt and interrupt-to-task communication mechanisms.

A queue can hold a finite number of fixed size data items. The maximum number of items a the queue can be called its ‘length’. Both the length and the size of each data item are set when the queue is created.
Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue.

There are two ways in which queue behavior could be implemented:
1. Queue by copy
Queuing by copy means the data sent to the queue is copied byte for byte into the queue.
2. Queue by reference
Queuing by reference means the queue only holds pointers to the data sent to the queue, not the data itself.

FreeRTOS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queueing by reference because:

- Stack variable can be sent directly to a queue, even though the variable will not exist after the function in which it is declared has exited.

- Data can be sent to a queue without first allocating a buffer to hold the data, and then copying the data into the allocated buffer

- The sending task can immediately re-use the variable or buffer that was sent to the queue

- The sending task and the receiving task are completely de-coupled—the application designer does not need to concern themselves with which task ‘owns’ the data, or which task is responsible for releasing the data

Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue Instead
The RTOS takes complete responsibility for allocating the memory used to store data

In a memory protected system, the RAM that a task can access will be restricted. In that case queueing by reference could only be used if the sending and receiving task could both access the RAM in which the data was stored. Queuing by copy does not impose that restriction; the kernel always runs with full privileges, allowing a queue to be used to pass data across memory protection boundaries

Access by Multiple Tasks

Queues are objects in their own right that can be accessed by any task or ISR that knows of their existence. Any number of tasks can be written to the same queue, and any number of tasks can read from the same queue. In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

Blocking on Queue Reads

When a task attempts to read from a queue, it can optionally specify a ‘block’ time. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty. A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue. The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.
Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data. If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked.

Blocking on Queue Writes

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full. Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task that is waiting for space. If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked.

Using Queue
Queue can be created using an API function provided by RTOS. Queues are referenced by their handle name which is created during creation of Queues. RTOS provides 2 APIs to create Queues. xQueueCreate() API Function creates a Queue and assigns memory from heap memory. Second API provided by RTOS is xQueueCreateStatic() API Function which provides memory required to Queue during compile time and therefore does not require any heap memory to be initialized. A Queue function prototype is given as follows:

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
Explanation of terms used in queue is given as follows
uxQueueLength
The maximum number of items that the queue being created can hold at any one time
uxItemSize
The size in bytes of each data item that can be stored in the queue.
Return Value
If NULL value is returned, it means the heap does not have enough storage and therefore , a queue cannot be created. Non zero return value means that Queue is created. The returned value should be stored as the handle to the created queue.

After a queue has been created the xQueueReset() API function can be used to return the queue to its original empty state

The xQueueSendToBack() and xQueueSendToFront() API Functions
As might be expected, xQueueSendToBack() is used to send data to the back (tail) of a queue, and xQueueSendToFront() is used to send data to the front (head) of a queue. xQueueSend() is equivalent to, and exactly the same as, xQueueSendToBack()

xQueueSendToFront() , xQueueSendToBack() and xQueueSend() , all three have same definition or prototype :

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
 const void * pvItemToQueue,
```

TickType_t xTicksToWait);

xQueue
The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvItemToQueue
A pointer to the data to be copied into the queue.

xTicksToWait
The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full. Both xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full.

ThxQueueReceive() is used to receive (read) an item from a queue. The item that is received is removed from the queue xQueueReceive() API Function. Function prototype is as follows

BaseType_t xQueueReceive(QueueHandle_t xQueue, void * const pvBuffer,
TickType_t xTicksToWait);

xQueue
The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvBuffer
A pointer to the memory into which the received data will be copied.

xTicksToWait
The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty

Receiving Data From Multiple Sources

It is common in FreeRTOS designs for a task to receive data from more than one source. The receiving task needs to know where the data came from to determine how the data should be processed. An easy design solution is to use a single queue to transfer structures with both the value of the data and the source of the data contained in the structure's fields.

Queuing Pointers

If the size of the data being stored in the queue is large, then it is preferable to use the queue to transfer pointers to the data, rather than copy the data itself into and out of the queue byte by byte. Transferring pointers is more efficient in both processing time and the amount of RAM required to create the queue. However, when queuing pointers, extreme care must be taken
The owner of the RAM being pointed to is clearly defined.
The RAM being pointed to remains valid

Using a Queue to Create a Mailbox

There is no consensus on terminology within the embedded community, and ‘mailbox’ will mean different things in different RTOSes. Here, the term mailbox is used to refer to a queue that has a length of one. A queue may get described as a mailbox because of the way it is used in the application, rather than because it has a functional difference to a queue:

- A queue is used to send data from one task to another task, or from an interrupt service routine to a task. The sender places an item in the queue, and the receiver removes the item from the queue. The data passes through the queue from the sender to the receiver.
- A mailbox is used to hold data that can be read by any task, or any interrupt service routine. The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten. The sender overwrites the value in the mailbox. The receiver reads the value from the mailbox, but does not remove the value from the mailbox.

FreeRTOS provides 2 APIs to use Queue as a mailbox. These are

The xQueueOverwrite() API Function

Like the xQueueSendToBack() API function, the xQueueOverwrite() API function sends data to a queue. Unlike xQueueSendToBack(), if the queue is already full, then xQueueOverwrite() will overwrite data that is already in the queue. xQueueOverwrite() should only be used with queues that have a length of one. That restriction avoids the need for the function’s implementation to make an arbitrary decision as to which item in the queue to overwrite, if the queue is full.

Prototype of API

BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void * pvItemToQueue);

xQueue
The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvItemToQueue
A pointer to the data to be copied into the queue

The xQueuePeek() API Function
xQueuePeek() is used to receive (read) an item from a queue without the item being removed from the queue. xQueuePeek() receives data from the head of the queue, without modifying the data stored in the queue, or the order in which data is stored in the queue.xQueuePeek() has the same function parameters and return value as xQueueReceive().

```
BaseType_t xQueuePeek( QueueHandle_t xQueue,
 void * const pvBuffer,
 TickType_t xTicksToWait )
```

Simulation Analysis

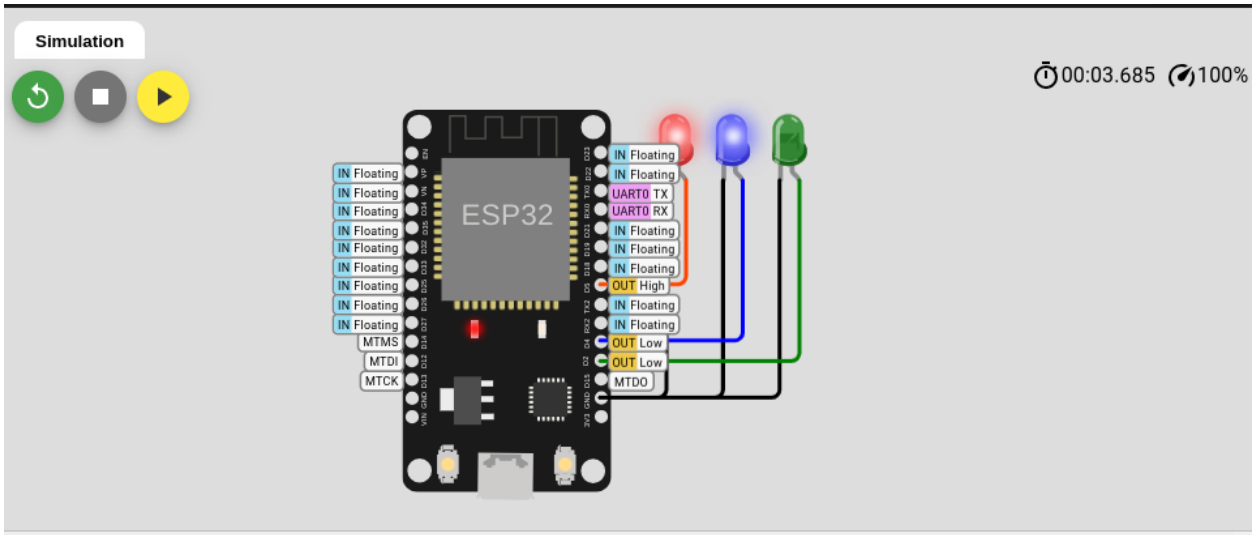
Simulation platforms such as Wokwi provide a convenient way to evaluate performance of MCUs without actually using hardware. There is no doubt that these simulations can deviate from actual results of hardware but these simulations can serve as baseline or starting point for evaluation of MCUs in general or FreeRTOS in particular.

One example of how Wokwi platform can be used to perform analysis on ESP32 S3 MCU using Wokwi can be done. Below example gives a sneak peak of how it can be done

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define LED_GREEN GPIO_NUM_4
#define LED_RED GPIO_NUM_5
#define LED_BLUE GPIO_NUM_2

struct led_task_parameter_t {
gpio_num_t led_gpio ;
TickType_t blink_time;
char led_name[6] ;
} ;
```

```
static led_task_parameter_t red_led_parameter = {LED_RED , 100,"red"};
static led_task_parameter_t green_led_parameter = {LED_GREEN ,
200,"green"};
static led_task_parameter_t blue_led_parameter = {LED_BLUE , 300,"blue"};

void led_task(void *pvParameter) {
printf("entry point for task running led is test and stack memory
remaining is %d\n",uxTaskGetStackHighWaterMark(NULL));
TickType_t blink = ((led_task_parameter_t *) pvParameter) -> blink_time;
gpio_num_t led_gpio = ((led_task_parameter_t *) pvParameter) -> led_gpio;
char *led_name = ((led_task_parameter_t *) pvParameter) -> led_name;
uint8_t led_value = 0;
gpio_reset_pin(led_gpio);
gpio_set_direction(led_gpio,GPIO_MODE_OUTPUT);
printf("entry point for task running led is %s and stack memory remaining
is %d\n",led_name,uxTaskGetStackHighWaterMark(NULL));
//printf("abcd");
while(1) {
printf("before task running while led is %s and stack memory remaining is
%d\n", led_name,uxTaskGetStackHighWaterMark(NULL));
gpio_set_level(led_gpio , led_value);
led_value = !led_value;
printf("after task running led is %s and stack memory remaining is %d\n",
led_name,uxTaskGetStackHighWaterMark(NULL));
vTaskDelay(pdMS_TO_TICKS(blink));
}
vTaskDelete(NULL);
}


extern "C" void app_main(){


printf("before creating task \n");


xTaskCreate(&led_task,"red led task",2048,&red_led_parameter,2,NULL);
```

```
xTaskCreate(&led_task,"green led task",2048,&green_led_parameter,2,NULL);

xTaskCreate(&led_task,"blue led task",2048,&blue_led_parameter,2,NULL);

}
```




```
entry 0x400805dc
before creating taskentry point for  task running led is test and stack memory remaining  is
1752
entry point for  task running led is test and stack memory remaining  is 1596
entry point for  task running led is red and stack memory remaining  is 308
before task running  while led is red and stack memory remaining  is 308
entry point for  task running led is green and stack memory remaining  is 428
before task running  while led is green and stack memory remaining  is 428
after task running led is green and stack memory remaining  is 428
entry point for  task running led is test and stack memory remaining  is 1699
entry point for  task running led is blue and stack memory remaining  is 432
after task running led is red and stack memory remaining  is 308
before task running  while led is blue and stack memory remaining  is 432
after task running led is blue and stack memory remaining  is 432
before task running  while led is red and stack memory remaining  is 308
after task running led is red and stack memory remaining  is 308
before task running  while led is green and stack memory remaining  is 428
after task running led is green and stack memory remaining  is 428
before task running  while led is red and stack memory remaining  is 308
after task running led is red and stack memory remaining  is 308
before task running  while led is red and stack memory remaining  is 308
after task running led is red and stack memory remaining  is 308
```

Note: Reason for different stack size when we call uxPortGetSackHighWaterMark() API function: Stack is responsible to save all variables initialized as well as execution state during “context switch”, therefore, it is possible to have different stack consumption for functions having same implementation due to different states during context switching.

Estimation of timing Analysis of Queues.

Queues play an important role for Inter Process Communication as they provide a mechanism for processes or in FreeRTOS context,Tasks,to transfer data safely. Queues provide different API functions that can be used to perform or implement functionality such as waiting for Queue to have space available so that Task can put data in it. In general , Queues work in FIFO mechanism where the first element put inside Queue is placed in front of Queue and is the first one to be taken out of Queue.

Queues in FreeRTOS work by copying data in Queue instead for referencing data. This allows Running Task to access data placed in Queue without worrying about the state of Task from where data was originated. Queues can hold any type of data or structure as long as enough space is provided to Queue to hold data during initialization of Queues.

An Example where Queues can be used to transfer data is given below.

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define LED_RED GPIO_NUM_2
#define LED_GREEN GPIO_NUM_4

QueueHandle_t xQueue;

void getQueueValue(void *pvParameter)
{
  int32_t value_to_recieve;
  BaseType_t xStatusReceive;
  gpio_num_t red_led = LED_RED ;
  gpio_set_direction(red_led, GPIO_MODE_OUTPUT);


  for(;;){
```

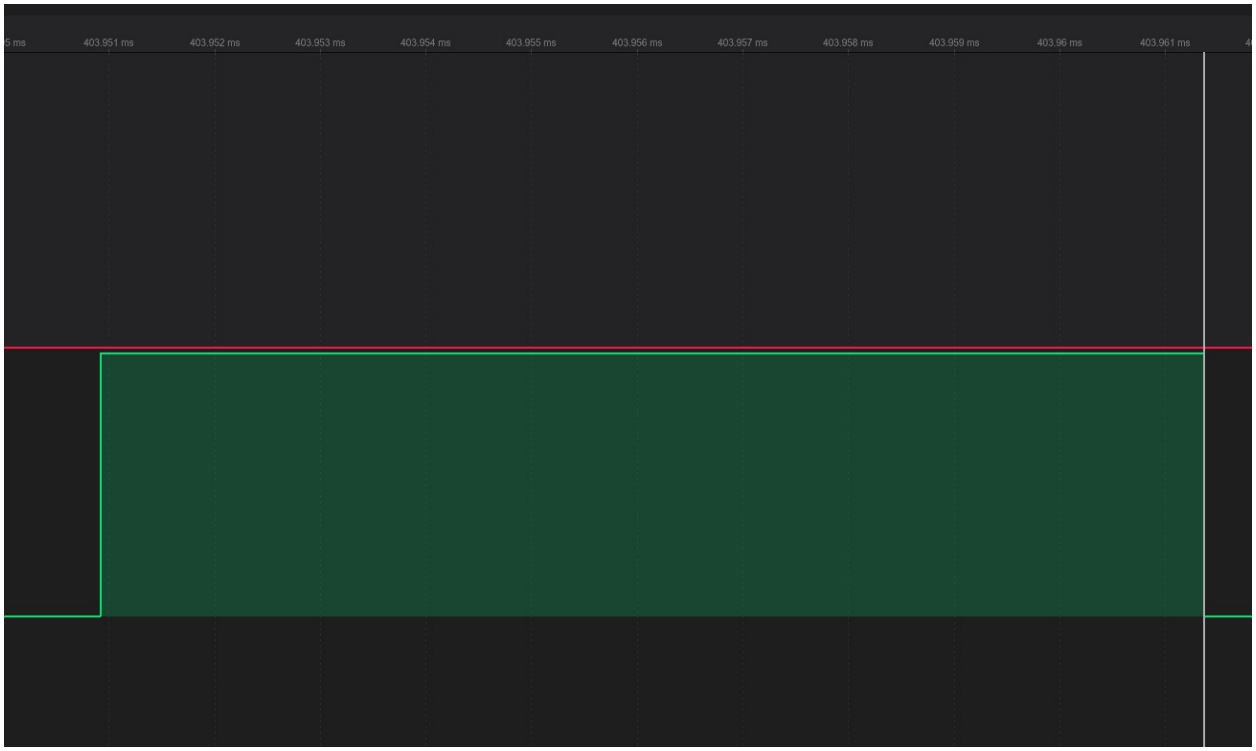
```
    gpio_set_level(red_led, 1);

    xStatusReceive = xQueueReceive(xQueue, &value_to_recieve, 0);

    gpio_set_level(red_led, 0);
    vTaskDelay(pdMS_TO_TICKS(200));
  }
}

void putQueueValue(void *pvParameters) {
int32_t value_to_send = (int32_t ) pvParameters;
BaseType_t xStatusSend;
gpio_num_t green_led = LED_GREEN ;
gpio_set_direction(green_led, GPIO_MODE_OUTPUT);

for(;;){
  gpio_set_level(green_led, 1);
  xStatusSend = xQueueSendToBack(xQueue,&value_to_send,0);
  gpio_set_level(green_led, 0);
  vTaskDelay(pdMS_TO_TICKS(1000));
  }

}


extern "C" void app_main()
{
  xQueue = xQueueCreate(1,sizeof(int32_t));
  if (xQueue != NULL){
  xTaskCreate(putQueueValue,"Sender",2048,(void*)100 , 1,NULL);
  xTaskCreate(getQueueValue,"Receiver",2048,NULL,1,NULL);
  }
  else{
    printf("not enough space");
  }
}
```

To analyze . We can attach an LED to GPIO 2 and 4 to visualize when Receiver and Sender tasks are running. We can use Logic Analyzer as well to get a timing behavior to analyze time taken for one Task to complete its execution from start to end. These simulated results can deviate from actual values at microsecond scale but these values can be used to estimate performance and behavior .

Test :

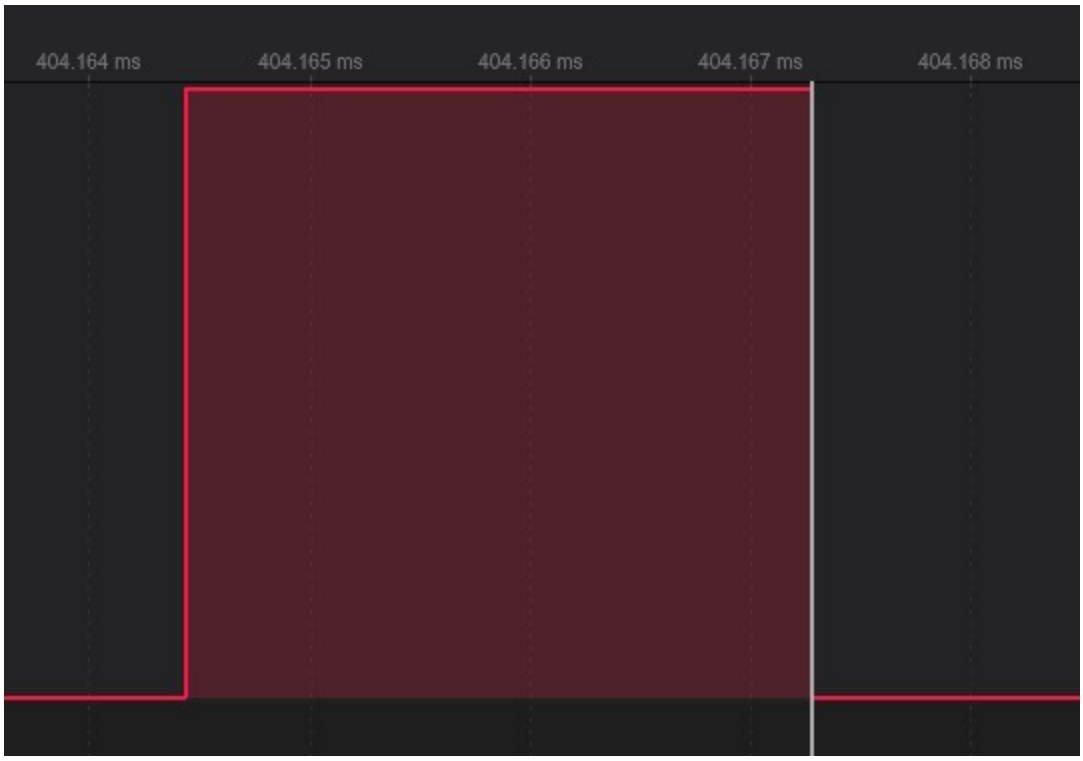
Queue of size 1, where Sender and Receiver having same priority. Sender Task is initiated first which means first value is placed in Queue and than Receiver Task is executed. This ensures that there is always one value since scheduler is round robin.

Sender Task is assigned Green LED



Total time for Sender Task:
403.951ms - 403.961ms = 0.009ms.
Here , it is important to mention that Receiver and Sender task consist of turning LED ON , putting/receiving value from Queue and turning LED OFF. from other experiments , it is estimated that on average , LED takes 0.004ms to turn ON and OFF

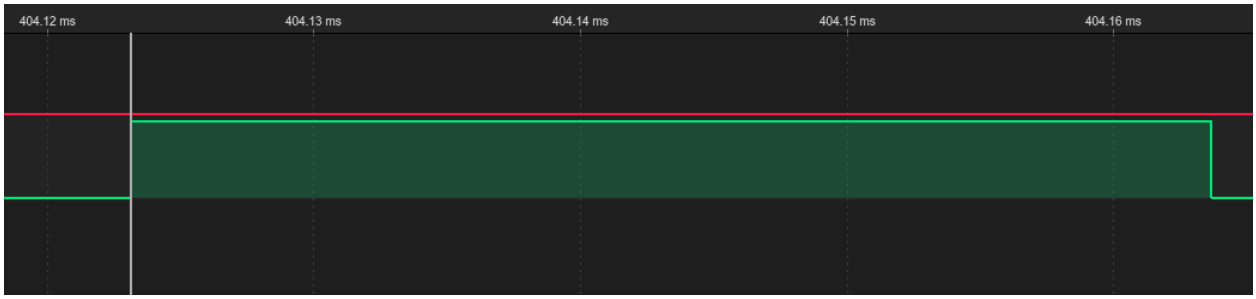
Receiver is Assigned Red LED



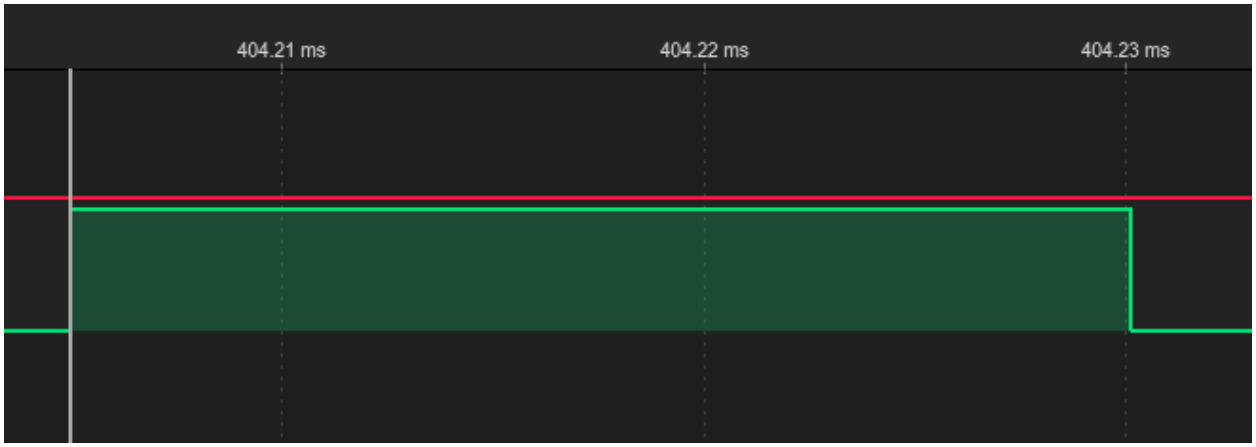
Total time for Receiver Task Execution : 404.168ms - 404.164ms = 0.004 ms
Here , it is important to mention that Receiver and Sender task consist of turning LED ON , putting/receiving value from Queue and turning LED OFF. from other experiments , it is estimated that on average , LED takes 0.004ms to turn ON and OFF.

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

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.

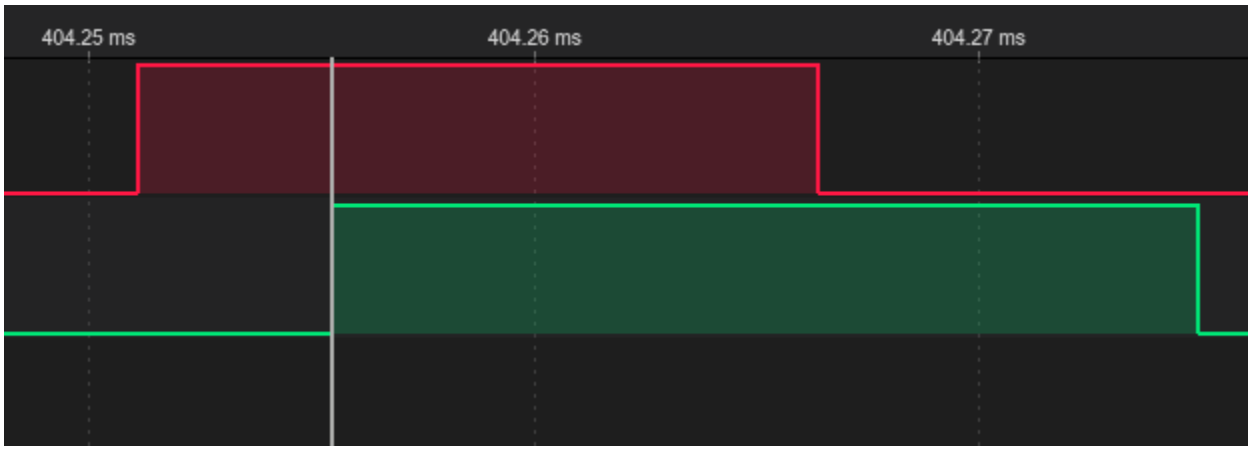


Time taken for execution : 404.164 - 404.126 = 0.038ms



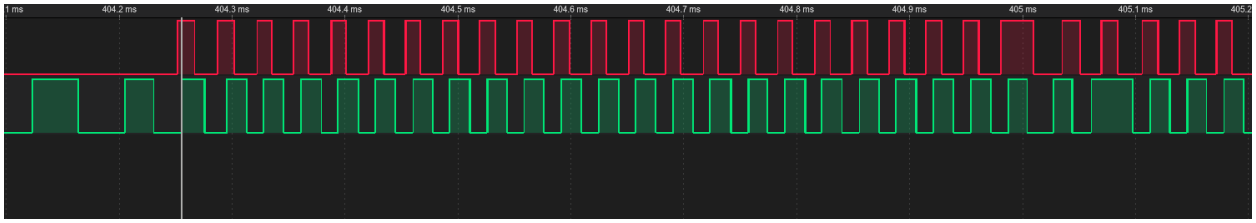
Total execution time for Sender Task in second iteration : 404.230 - 404.205 = 0.030ms .
From previous test, it was estimated that time taken for putting one value in Queue and LED operation took 0.009ms , there , putting 5 values took approx 0.035ms

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 5 values only 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.
Time for First Preemption : 404.255 - 404.251 = 0.004ms (this time is similar to time in the previous test).

Preemption pattern looks like this



Software timers

Software timers can be used to perform a certain task when a specified time expires. Which means when we add a software timer in our application , it starts a background task. When the scheduler is started ,this creates a timer service task. This task maintains a list of all timers and associated handlers or function calls.
This timer does not continuously run and blocks itself; when the timer expires, it is activated and calls the associated function.
Callback function have same priority as software timer function , therefore, it is usually treated similar to ISR but differs in a way that ISRs are generally considered hardware based Interrupts and can be executed quicker as compared to timers and have a

Software timers, timing analysis

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define LED_RED GPIO_NUM_5
#define LED_GREEN GPIO_NUM_2
#define LED_BLUE GPIO_NUM_4


static TimerHandle_t led1 , led2 ,led3= NULL ;




void TimerLedControl(TimerHandle_t xtimer){
  uint32_t led_timer_id = (uint32_t) pvTimerGetTimerID(xtimer) ;
```

```
  //red led
  if (led_timer_id == 0){
  gpio_set_level(LED_RED,!gpio_get_level(LED_RED));
  }
  //green led
  if (led_timer_id == 1){
  gpio_set_level(LED_GREEN,!gpio_get_level(LED_GREEN));
  }
  //blue led
  if (led_timer_id == 2){
  gpio_set_level(LED_BLUE,!gpio_get_level(LED_BLUE));
  }
  printf("inside timer\n");

}

extern "C" void app_main()
{


  //setting GPIO Direction (INPUT , OUTPUT)

  gpio_set_direction(LED_RED, GPIO_MODE_OUTPUT);
  gpio_set_direction(LED_GREEN, GPIO_MODE_OUTPUT);
  gpio_set_direction(LED_BLUE, GPIO_MODE_OUTPUT);

  //timer tasks

  led1 = xTimerCreate("LED Red", pdMS_TO_TICKS(100),pdTRUE,(void *)
0,TimerLedControl);
  led2 = xTimerCreate("LED Green", pdMS_TO_TICKS(100),pdTRUE,(void *)
1,TimerLedControl);
  led3 = xTimerCreate("LED Blue", pdMS_TO_TICKS(100),pdTRUE,(void *)
2,TimerLedControl);
  xTimerStart(led1,0);
  xTimerStart(led2,0);
  xTimerStart(led3,0);
}
```

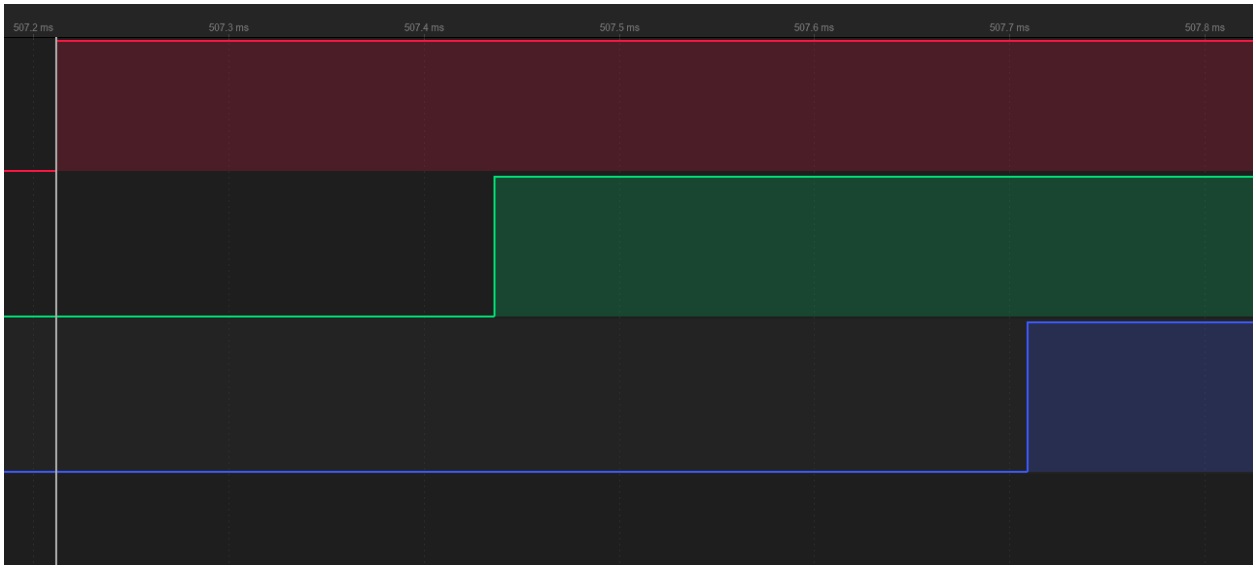
Timing analysis (vcd file: software-timer-1)

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 turn LED ON and OFF based on timer ID.

Task1: red led toggle based on expiring timer
task2: green led toggle based on expiring timer
task3: blue led toggle based on expiring timer

Behavior :
1 cycle
Rising Edge (LED turn on) (First call to timer expire callback)

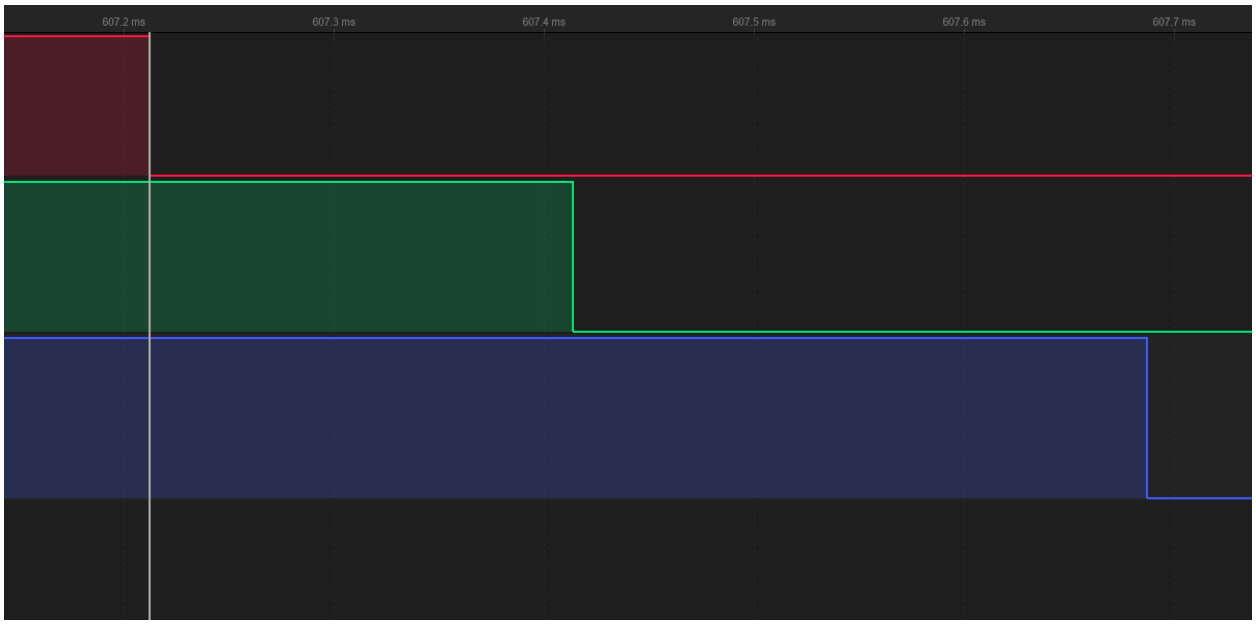


Time to start task1 : 507.212 ms
Time to start task2 : 507.436 ms
Time to start task3 : 507.709

Difference between execution of 2 timer callbacks
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, we have higher execution time.

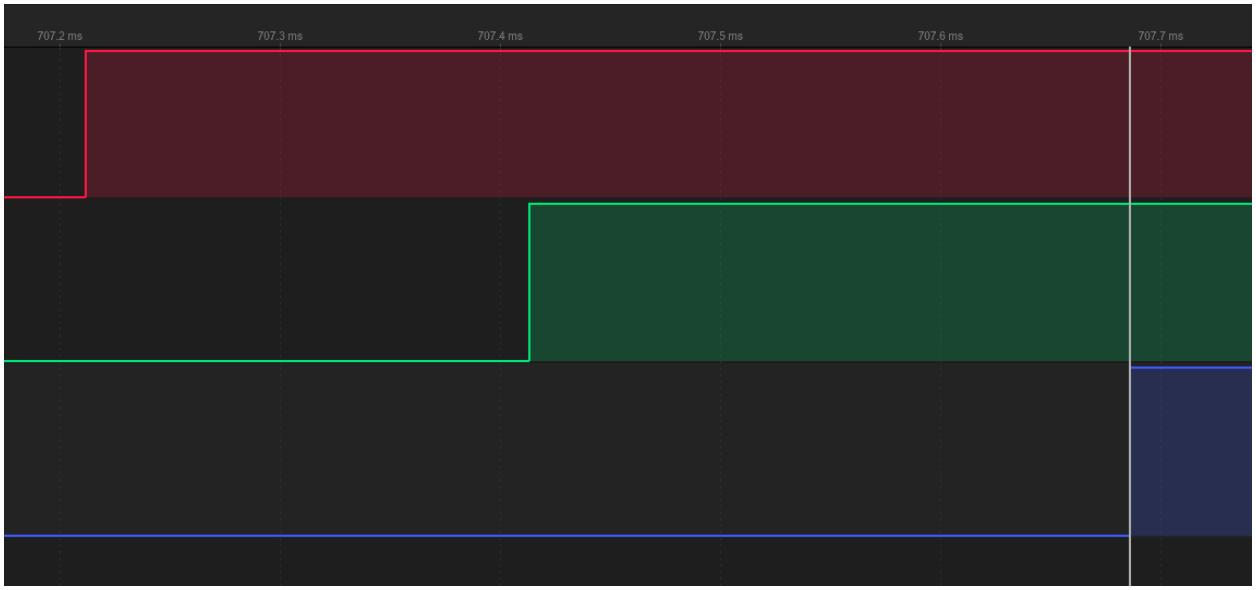
Falling Edge (LED turn off) (second call to timer expire callback)



Time to start task1 : 607.212 ms
Time to start task2 : 607.414 ms
Time to start task3 : 607.687 ms
difference
Task2 - task1 = 0.202 ms
Task 3 - task 2 = 0.273 ms

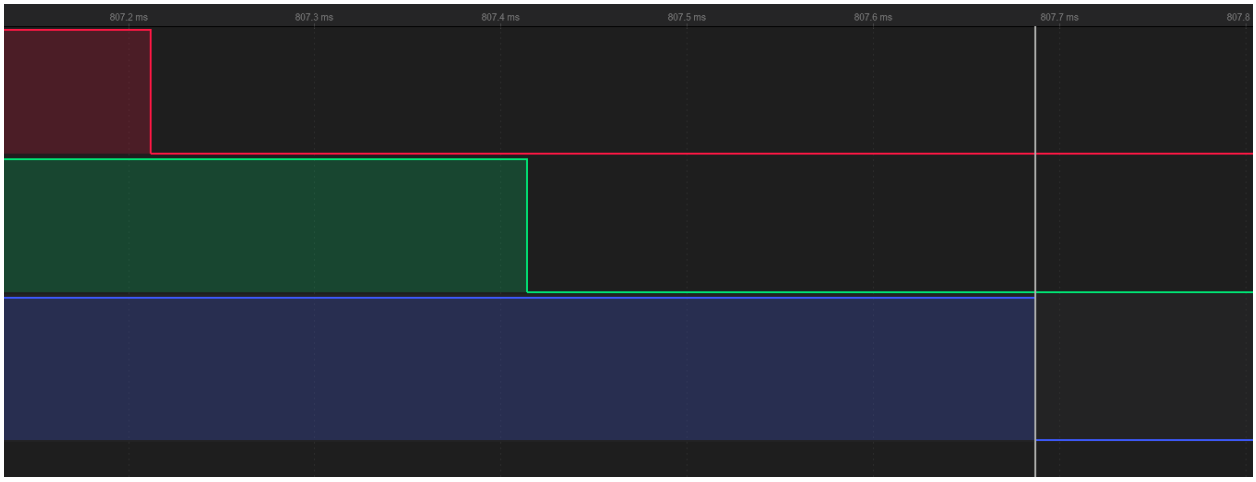
2 cycle

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



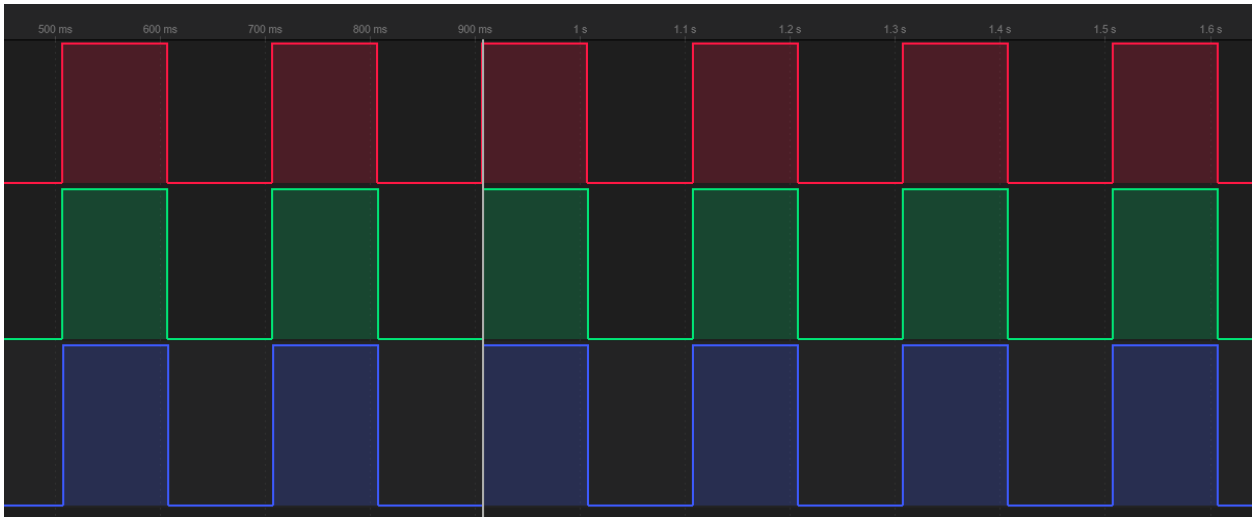
Time to start task1 : 707.212 ms
Time to start task2 : 707.413 ms
Time to start task3 : 707.686 ms
Difference
Task2 - task1 = 0.201 ms
Task 3 - task 2 = 0.273 ms

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



Time to start task1 : 807.212 ms
Time to start task2 : 807.414 ms
Time to start task3 : 807.687 ms
difference
Task2 - task1 = 0.202 ms
Task 3 - task 2 = 0.273 ms

All cycles overview



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

TEST:

Changing function calls

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define LED_RED GPIO_NUM_5
#define LED_GREEN GPIO_NUM_2
#define LED_BLUE GPIO_NUM_4


static TimerHandle_t led1 , led2 ,led3= NULL ;


void TimerLedControlRed(TimerHandle_t xtimer){
```

```
 // uint32_t led_timer_id = (uint32_t) pvTimerGetTimerID(xtimer) ;
  //red led

  gpio_set_level(LED_RED,!gpio_get_level(LED_RED));
 printf(" from RED %d\n",ESP.getFreeHeap());


  //printf("inside timer\n");
}
void TimerLedControlGreen(TimerHandle_t xtimer){
 // uint32_t led_timer_id = (uint32_t) pvTimerGetTimerID(xtimer) ;
  gpio_set_level(LED_GREEN,!gpio_get_level(LED_GREEN));
  printf(" from GREEN %d\n",ESP.getFreeHeap());

  //printf("inside timer\n");
}

void TimerLedControlBlue(TimerHandle_t xtimer){
 // uint32_t led_timer_id = (uint32_t) pvTimerGetTimerID(xtimer) ;
  //red led
  gpio_set_level(LED_BLUE,!gpio_get_level(LED_BLUE));
printf(" from BLUE %d\n",ESP.getFreeHeap());


  //printf("inside timer\n");
}

extern "C" void app_main()
{


  //setting GPIO Direction (INPUT , OUTPUT)
 printf(" from main 1 %d\n",ESP.getFreeHeap());

  gpio_set_direction(LED_RED, GPIO_MODE_OUTPUT);
  gpio_set_direction(LED_GREEN, GPIO_MODE_OUTPUT);
  gpio_set_direction(LED_BLUE, GPIO_MODE_OUTPUT);

  //timer tasks
 printf(" from main 2 %d\n",ESP.getFreeHeap());
```

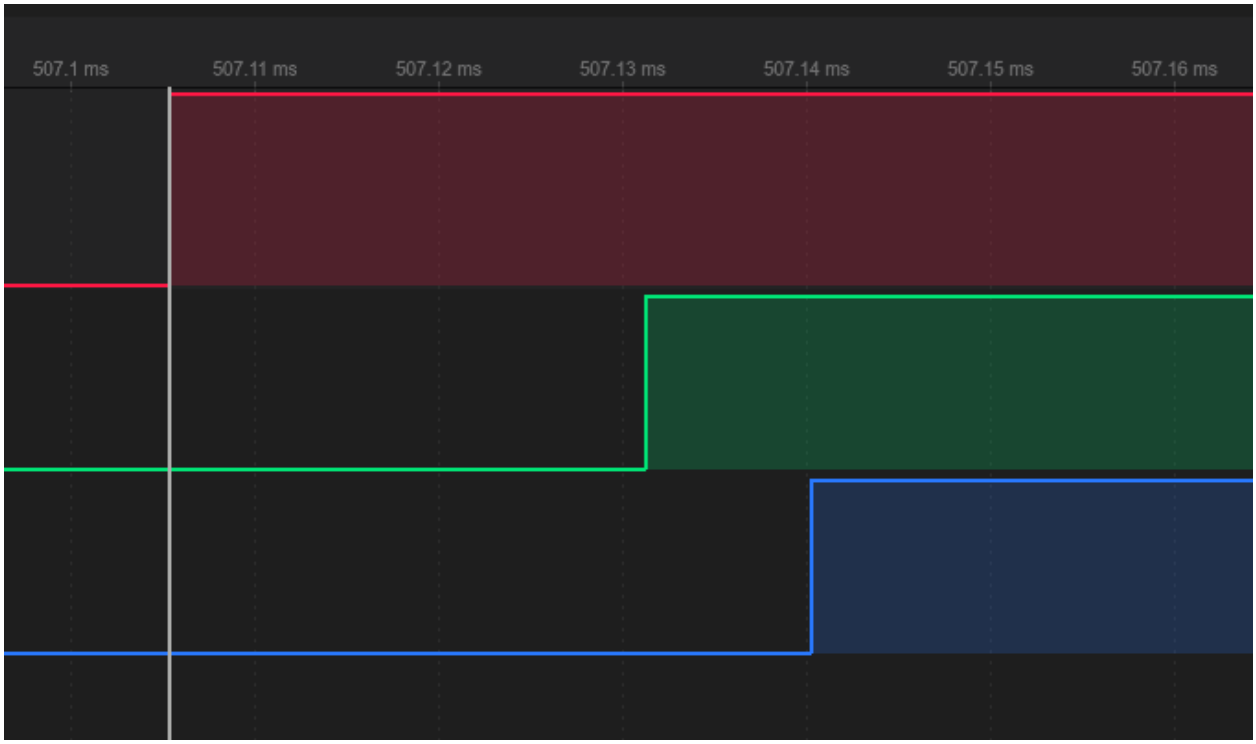
```
  led1 = xTimerCreate("LED Red", pdMS_TO_TICKS(100),pdTRUE,(void *)
0,TimerLedControlRed);
  led2 = xTimerCreate("LED Green", pdMS_TO_TICKS(100),pdTRUE,(void *)
1,TimerLedControlGreen);
  led3 = xTimerCreate("LED Blue", pdMS_TO_TICKS(100),pdTRUE,(void *)
2,TimerLedControlBlue);
  xTimerStart(led1,0);
  xTimerStart(led2,0);
  xTimerStart(led3,0);
 printf(" from main 3 %d\n",ESP.getFreeHeap());


}
```

Each timer has its own callback function. Callback function is responsible for toggling LED connected to Pinouts.

Timing analysis (vcd file: software-timer-2)
All timer callbacks have same time to expire : 100 ms
Experimental setup: 3 timer based tasks. Each timer has its own callback function. Callback function is responsible for toggling LED connected to Pinouts.

Task1: red led toggle based on expiring timer
task2: green led toggle based on expiring timer
task3: blue led toggle based on expiring timer

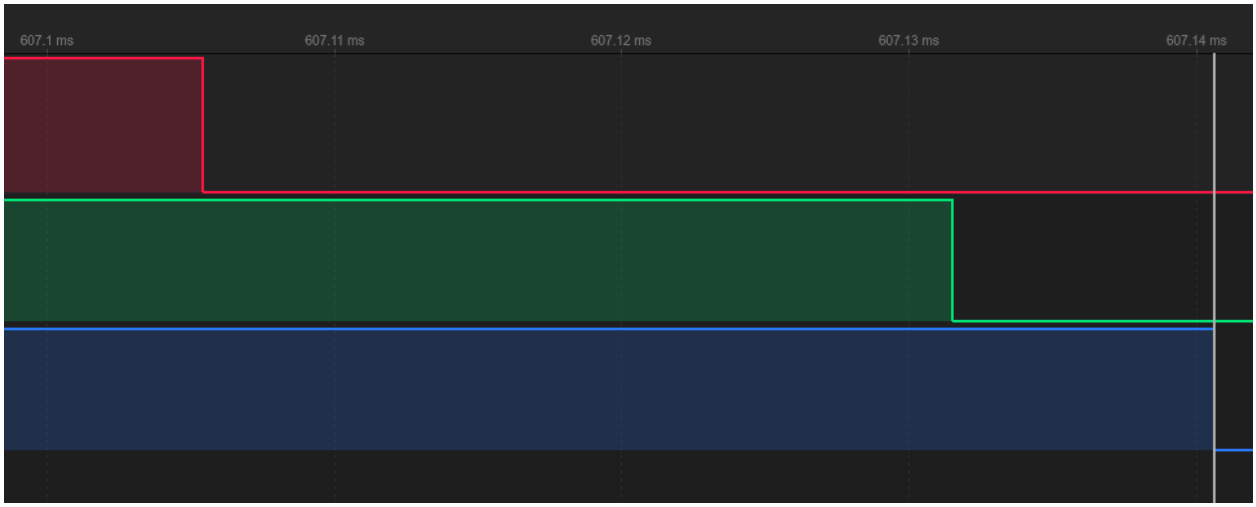
Behavior :
1 cycle
Rising Edge (LED turn on) (First call to timer expire callback)



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

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

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

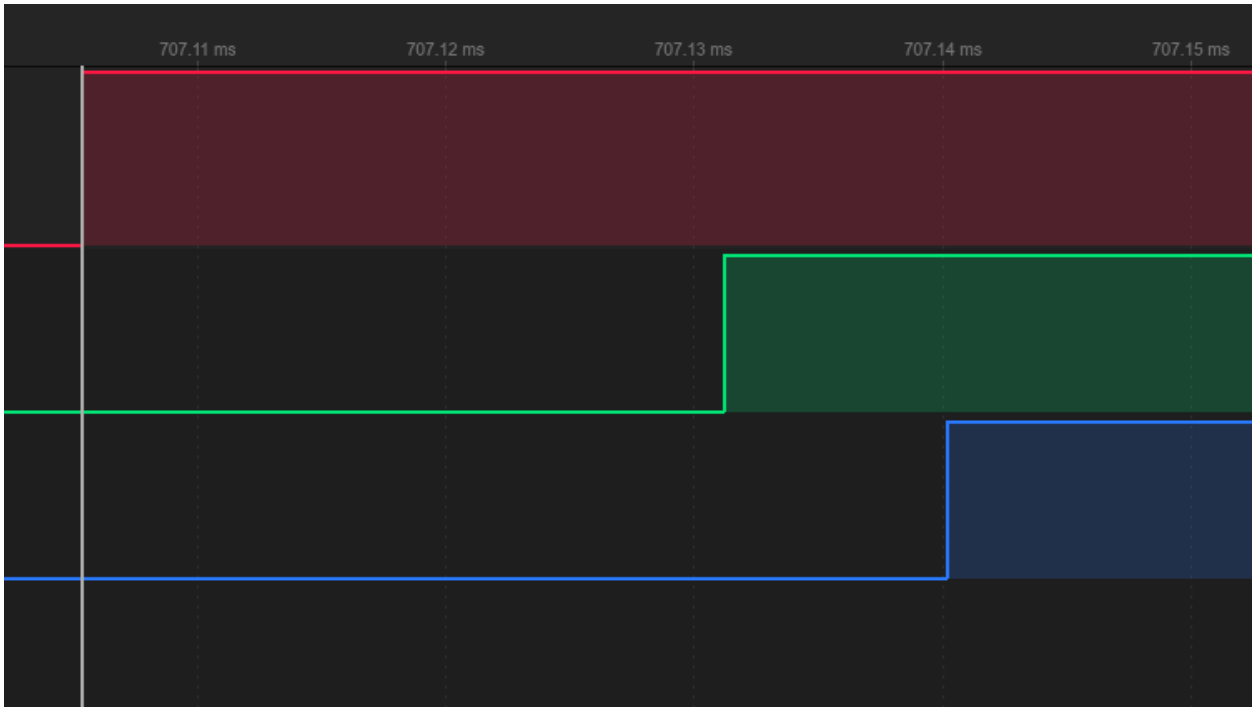


Time to start task1 : 607.105 ms
Time to start task2 : 607.131 ms
Time to start task3 : 607.141 ms

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

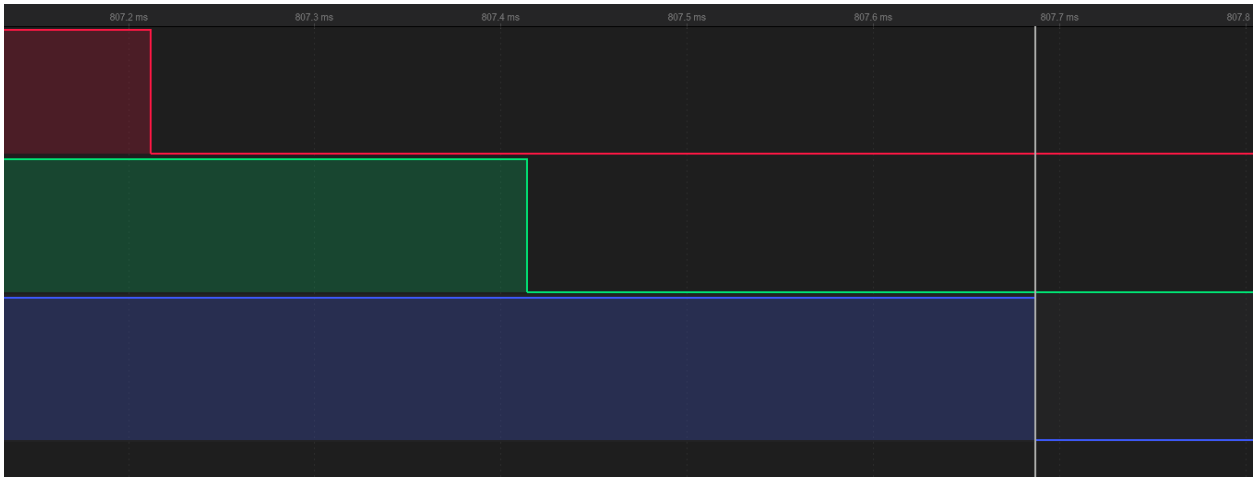
2 cycle

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



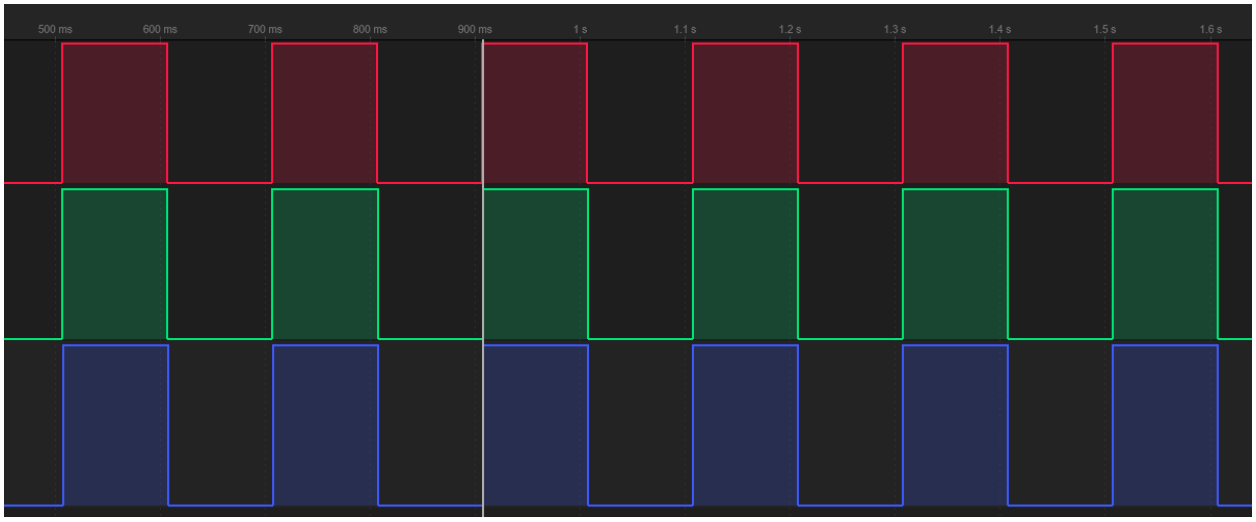
Time to start task1 : 707.105 ms
Time to start task2 : 707.131 ms
Time to start task3 : 707.140 ms
Difference
Task2 - task1 = 0.026 ms
Task 3 - task 2 = 0.009 ms

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



Time to start task1 : 807.105 ms
Time to start task2 : 807.131 ms
Time to start task3 : 807.141 ms
difference
Task2 - task1 = 0.026 ms
Task 3 - task 2 = 0.010 ms

All cycles overview



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 Red Led callback
From GREEN 365340 // inside Green Led Callback
From BLUE 365340 //inside Blue Led Callback

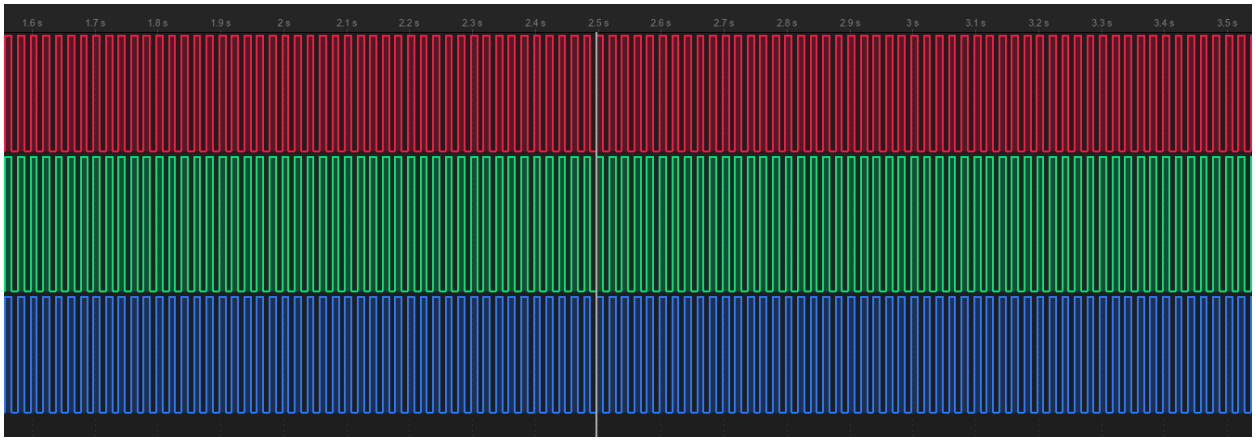
Test: 10 ms gap in all timer expire

T1 417.105
T2 417.131
T3 417.140
T2-t1 = 0.026
t3-t2= 0.009

T1 427.105
T2 427.131
T3 427.141

T1 437.105
T2 437.131
T3 437.141

No visible exceptional pattern in 1800+ samples



Observations:

- Giving same callback function to all timers had smaller memory footprint but took significant large time for performing required operation which is toggling LEDs.
- Giving individual callback function to each timer reduced execution time to significantly small but increased memory consumptions.
- In both experimental setup , strict timer behavior is observed.

Hardware Analysis

ESP32 and its variations such as ESP32-S3 , ESP32-C3 can be programmed in various ways. Most common ways include using Arduino IDE and ESP-IDF. Hardware configuration is as follows

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

ESP-IDF has its own variation of FreeRTOS that differs from original or “Vanilla FreeRTOS” in some ways. One prominent difference is, the ESP-IDF variation of FreeRTOS allows tasks to be scheduled on specific cores (in dual core systems). Instead of the xTaskCreate() macro , the xTaskCreatePinnedToCore() macro is used to create an affinity of tasks to a specific core. Usually, cores of CPU are used for some specific tasks such as application or process , but not mandatory. Therefore, sometimes , core 0 is referred to as proc core and core 1 is referred to as app core.

Other than that , ESP-IDF allows total control over partitioning of memory by using partition tables. Typical or built-in partition table contains following fields
Nvs
Phy_init
Factory

A custom partition table can be generated in csv format to add partitions for OTA and OTA data. Moreover, ESP-IDF provides md5 checksum at runtime to check integrity of partition table(s).

ESP-IDF provides menuconfig to configure SDK for enabling components, configuring components and enabling secure boot options.

As mentioned earlier, ESP-IDF provides an official “port” of FreeRTOS as part of ESP-IDF SDK. This consists of standard APIs provided by FreeRTOS as well as some additional APIs specific to ESP32 in general and specific ESP32 modules. ESP-IDF uses Cmake for building projects as well as some utility tools to ease development and flashing processes.

Installing ESP-IDF

ESP-IDF can be installed from the Espressif website by choosing the appropriate OS platform. Espressif toolchain and ESP-IDF can be used on Windows, Linux based systems and MacOS. To install SDK on windows, simply download the installer and choose the SDK version during installation. This process will set up the toolchain as well as create a Shell that can be used to build projects , flash into chips and monitor serial ports connected to chips.

ESP-IDF provides example projects that can be used to interact with peripherals such as WIFI , Bluetooth as well as provide examples for partition table creation and custom bootloader.

ESP-IDF variation of FreeRTOS has default configuration of TICK_RATE_HZ to be 100 Hz. This defines the “tick” period at which task will be preempted or removed from running state to ready or blocked state. This can be configured using menuconfig and changing configuration parameter in “sdkconfig” file generated during build process of application.

To configure a project or build a project using “esptools”, navigate to the project directory containing source code and Cmake file from the terminal having the ESP-IDF environment loaded and use following command(s).
To configure project using menuconfig:

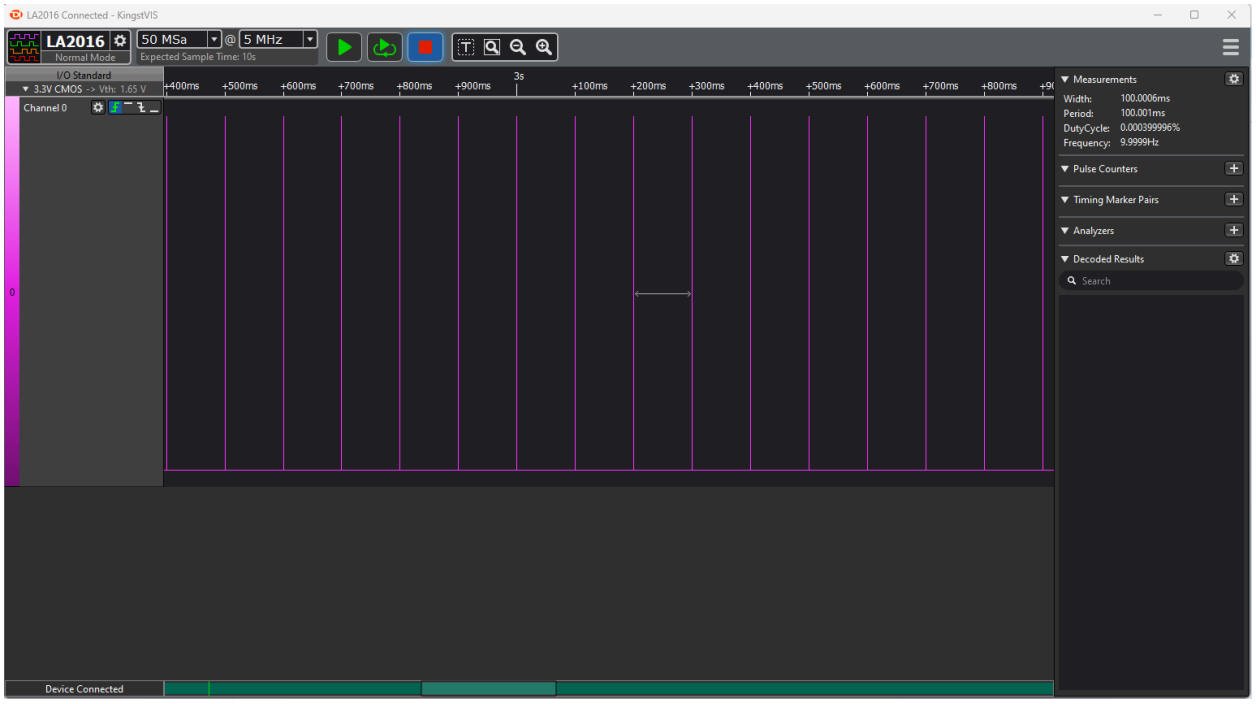
- Idf.py menuconfig
  To set target chip
- Idf.py set-target <chip type such as esp32s3 , esp32c3>
- To build project
- Idf.py build
- To flash project into chip
- Idf.py -p <serial port connected to esp32 board> flash
- To monitor port
- Idf.py -p <serial port connected to esp32 board> monitor

Analysis:

By keeping FreeRTOS at default TICK_RATE_HZ (100 Hz) , this will result in a period of 10 ms if 1 tick is provided as a wait time for each task. If all tasks have the same priority, each task will take turns and be executed for 10 ms and then removed from execution for another task to run which is referred to as the Round Robin method.

By providing different tick time in vTaskDelay() API , we can control the execution time for each task. Following are some observations with various time periods or number of ticks

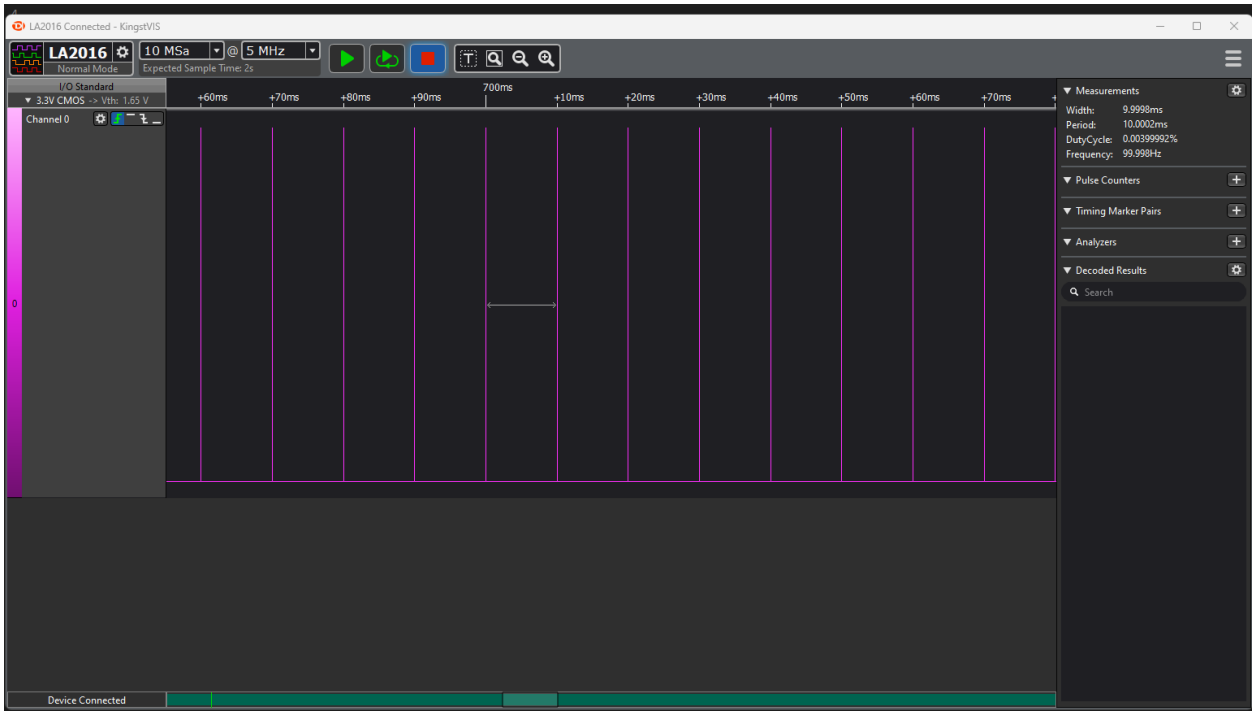
With 100 milliseconds delay between task scheduling



Here , we can observe that GPIO is toggled every 100 ms however tasks does not seem to follow strict 100 ms time period as is variation of time period and width
Width : 100.0006 ms
Period : 100.001 ms
Frequency : 9.9999 Hz

These parameters show variation in timing behavior throughout execution showing that this Task Scheduling is not following a strict deadline of 100 ms.

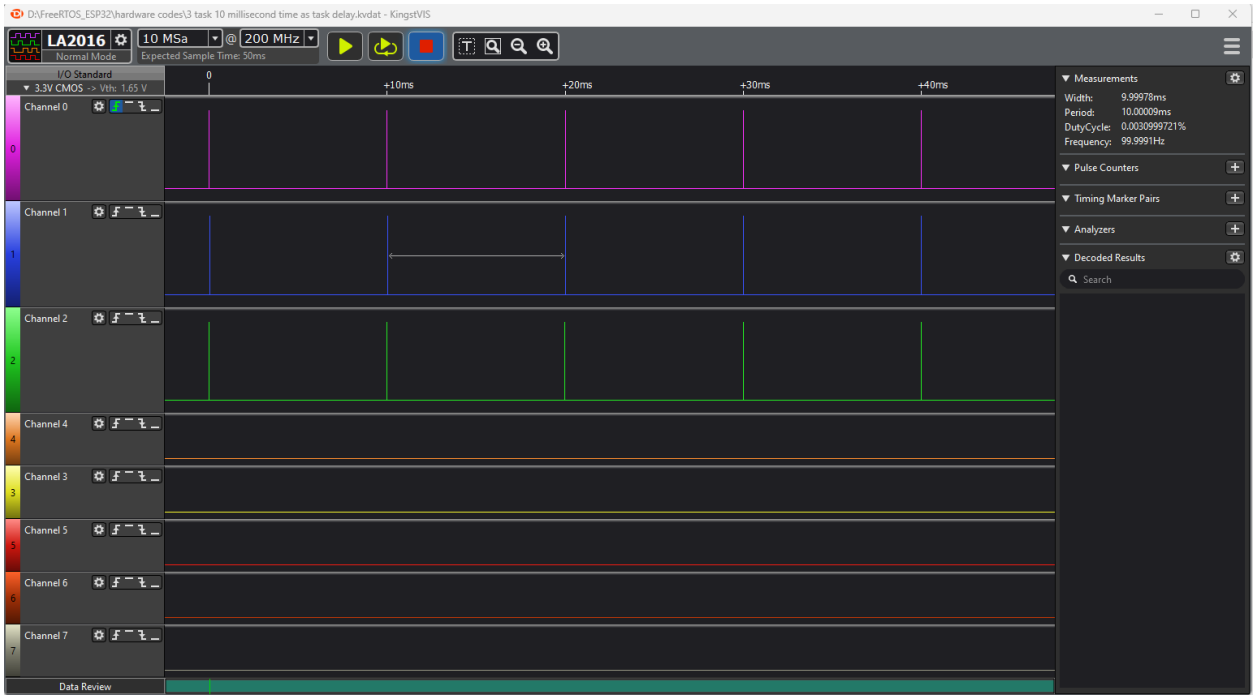
With 10 ms as time delay



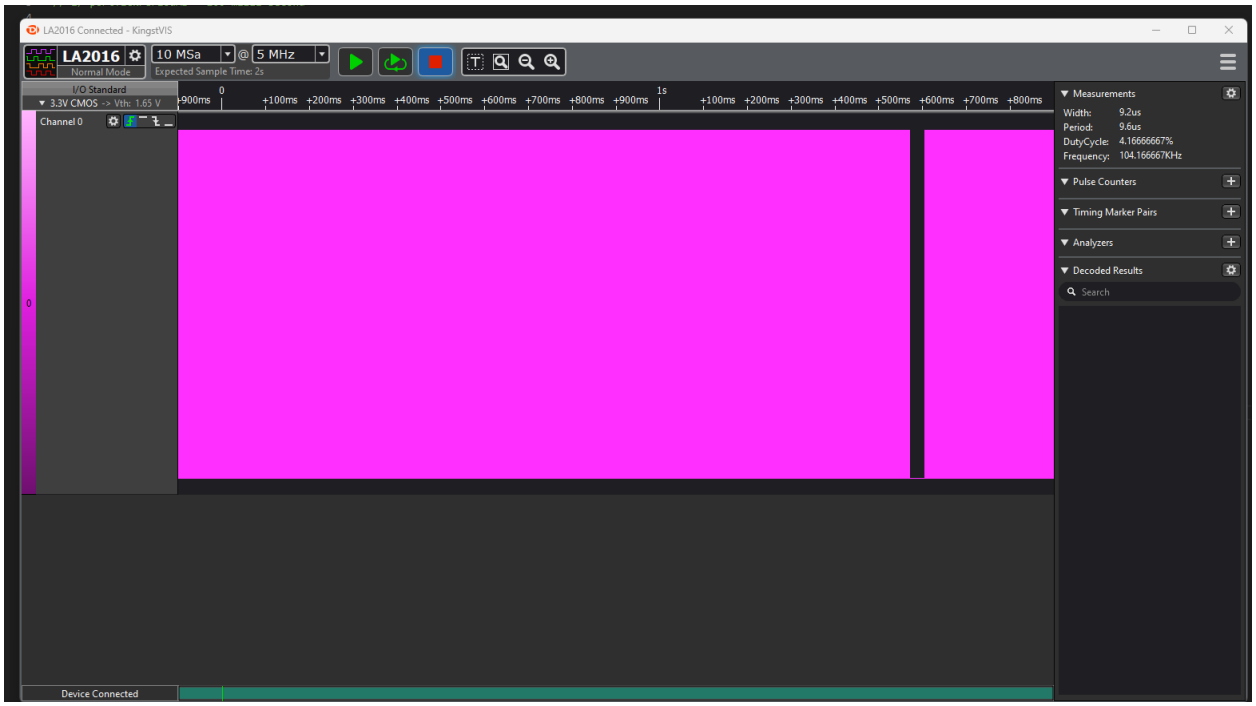
Here , we can observe that GPIO is toggled every 10 ms however tasks does not seem to follow strict 100 ms time period as is variation of time period and width
Width : 9.9998 ms
Period : 10.0002 ms
Frequency : 99.998 Hz

These parameters show variation in timing behavior throughout execution showing that this Task Scheduling is not following a strict deadline of 10 ms.

Similarly, with 3 tasks with same priority and 10 ms delay time , variation in width, period and frequency can be observed



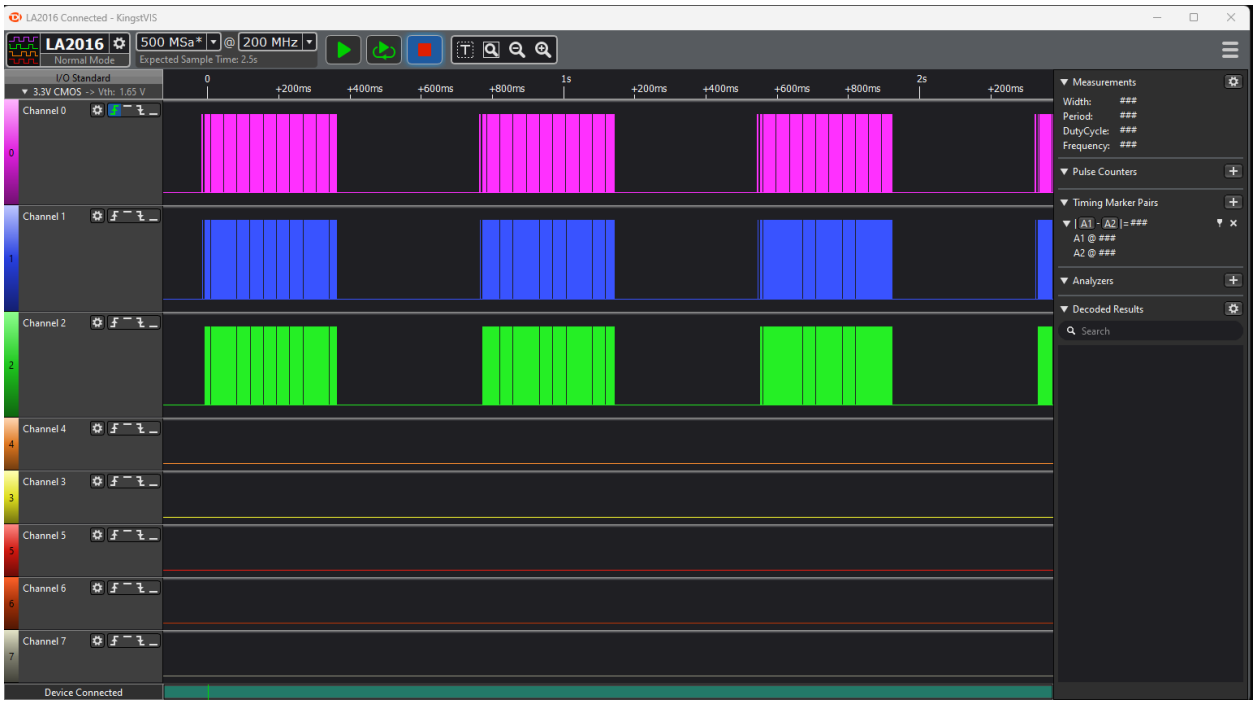
1 Task with 1 ms as delay (defualt Tick Rate is 100 Hz which means 10 ms period).



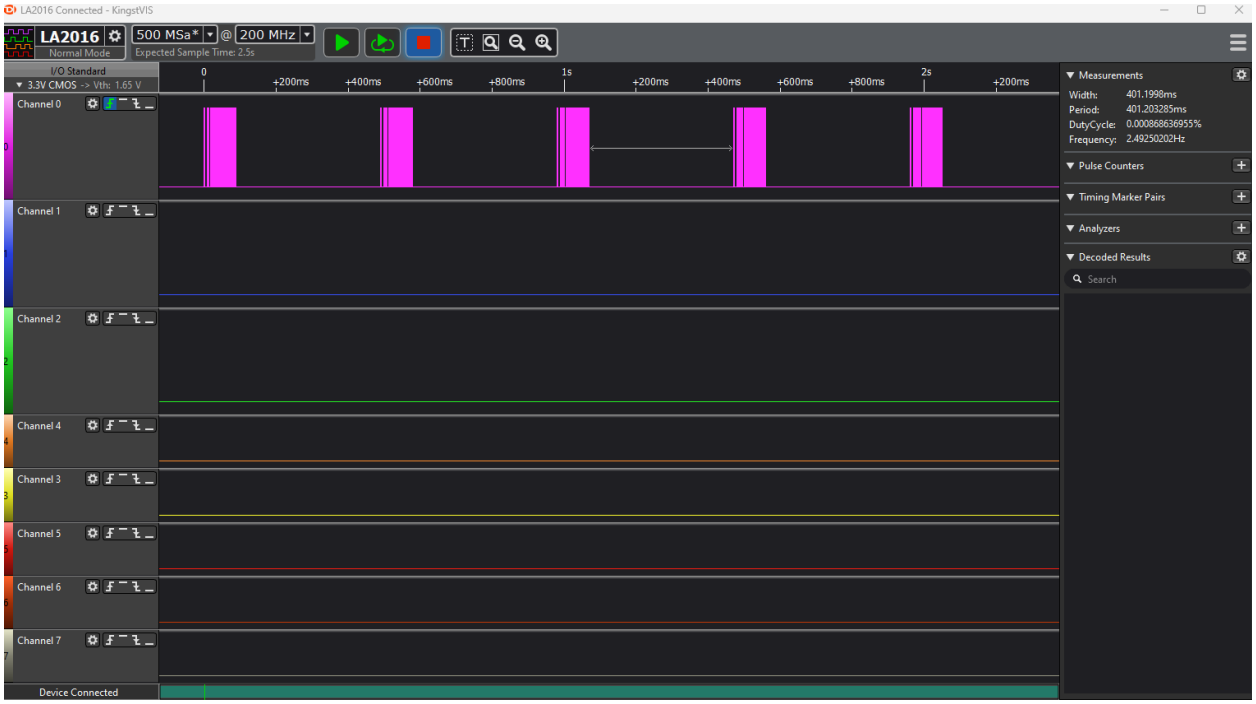
Here we can observe that there is no delay between task execution or scheduling. Width and period parameters are follows
Width: 9.2 microseconds
Period : 9.6 micro seconds

NOTE: Interesting thing to observe here is that the period is 9.6 microseconds which is roughly equal to the time taken by the scheduler to switch tasks for execution, therefore , it can be concluded that theoretically there is no delay between task execution. There are multiple reasons for this behavior such as TICK PERIOD is defined at 100 Hz (10 ms period) and Tick is defined as integer, therefore, anything in decimal value with 0 as significant number will eventually type casted to integer, hence , 0 time delay.

1 ms time delay with 3 tasks having same priority

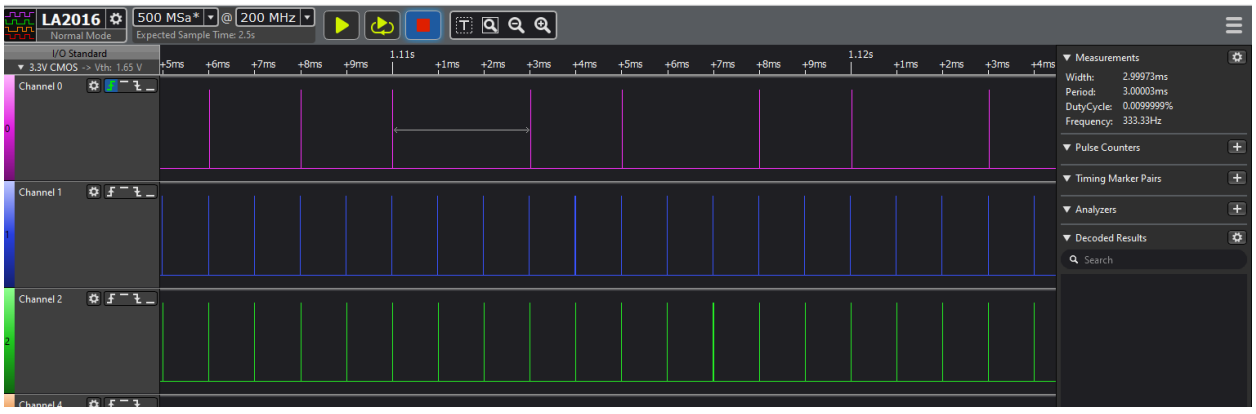


0 ms as time delay , 1 task



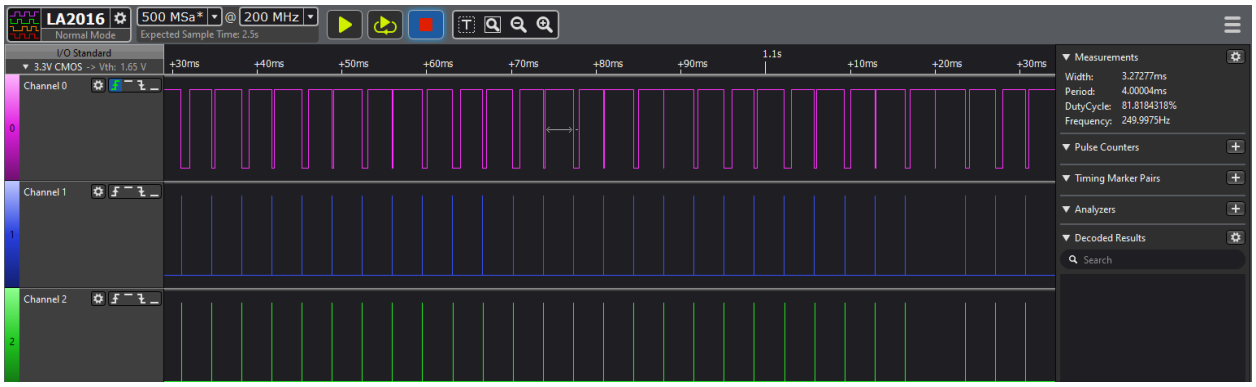
Setting TICK_PERIOD_HZ to 1000 through SDKCONFIGfor project,

3 task, same priority , 1 ms delay time



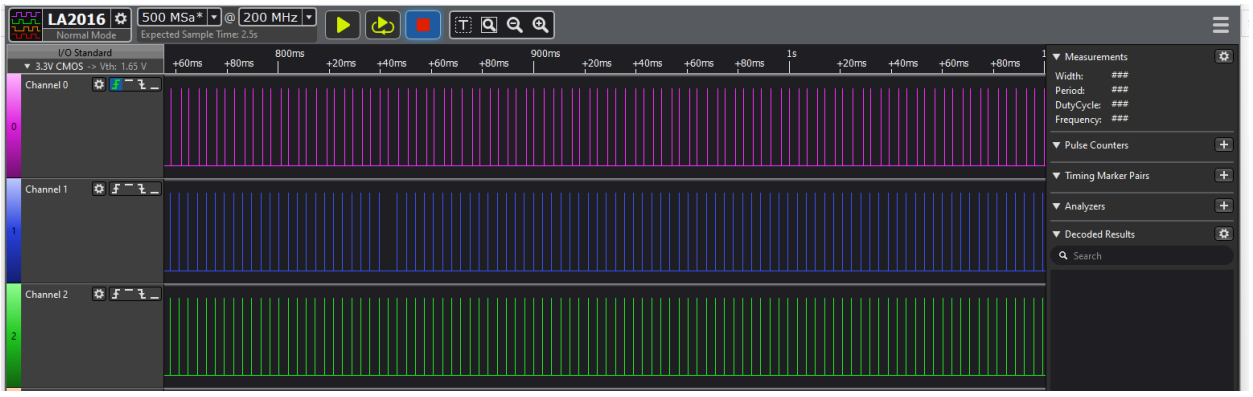
Here , it can be observed that task 1 (channel 0) has a different behavior as compared to other 2 tasks. This shows that with this small delay time , a small change such as logging info from a task can affect the behavior of task execution to a great extent.

1 millisecond delay, 3 tasks, 2 tasks same priority, 1 task higher priority - 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



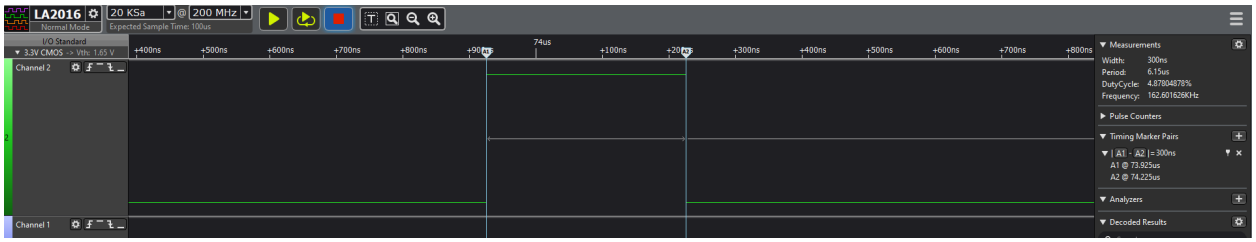
This shows even small change in sequence of instructions can affect final outcome behavior

1 millisecond delay, 3 tasks, 2 tasks same priority, 1 task higher priority - 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



This shows even small change in sequence of instructions can affect final outcome behavior

Time taken by ESP32 S3 to toggle GPIO running at 240MHz frequency using “Super Loop Method” is as follows



Time taken: 300 nano seconds

Observations:

- Time taken by FreeRTOS scheduler : 9.1 microseconds to 11.5 micro seconds depending on Stack size. With 2048 words stack size, scheduler took 9.1 micro seconds and when stack size was increased to 3096 words.
- FreeRTOS is not following strict deadlines for task execution.
- Decreasing Tick Rate to 1000 Hz (1 ms time period) requires careful and precise design of application as small change or order of code statement can cause change in behavior

Conclusion And Findings

For Soft Real Time Deadlines, FreeRTOS seems to be a good option because of a mature ecosystem and active feature development backed by community as well as Amazon. For Hard Real Time Deadlines, it appears that FreeRTOS is not an ideal option as we can observe deviation in the time period of tasks.

Wokwi is a great tool for prototyping and testing “Logical Functionality” of an application by simulating different I/Os, however, for more accurate results, testing on hardware is better to have closer to reality results.

Performing analysis on hardware poses some challenges and can easily add contamination or noise in results. One such reason can be due to Electronics of MCU or equipment used for analysis such Kingst LA 2016 Logic Analyzer, therefore, these facts should be considered in evaluations and steps should be taken to minimize such contaminations.