Analysis of FreeRTOS Source Code

Implementation of the Ready List in the Scheduler

TEAM ALGOXPERTS, CSE – E

NAME	ROLL
ARAVINDH	CB.EN.U4CSE21406
K Ganesh	CB.EN.U4CSE21426
Ramnaresh Ulaganathan	CB.EN.U4CSE21447
Shreyas Visweshwaran	CB.EN.U4CSE21455
N S Surya	CB.EN.U4CSE21461

Submitted to,
Dr. Vijay Kumar Sundar
Faculty – DSA

INTRODUCTION

Free RTOS is an open-source real time operating system designed primarily for being used in embedded systems and microcontrollers. It offers a compact, effective and portable kernel that enables synchronization, inter-task communication, task scheduling and multitasking.

Its key features include:

- Preemptive and cooperative scheduling
- Task Management
- Synchronization and communication
- Timers
- Memory Management
- Interrupt handling
- Portability
- Small Footprint

Its significance in Real time embedded systems include Real time task scheduling, Resource management, Portability, low overhead, comprehensive feature set etc.

Automotive, industrial automation, consumer electronics, medical devices, and other sectors and applications use FreeRTOS extensively.

The purpose of the report is to explain the working of the code, the various functions and macros used so that the reader will be able to understand the complex code quickly so that it will be helpful for future projects.

We specifically focus on the functions of the ready queue which manages and prioritizes tasks that are ready to be executed. The functions of ready list are task scheduling, Priority based execution, efficient task selection, preemption and context switching, system responsiveness etc. It allows the kernel to manage tasks effectively, ensuring that the right tasks are executed at the right time based on their priorities and system requirements.

We divide this report into a few parts such as overview of the scheduler, understanding the ready list, analyzing, detailed examination of ready list functions, integrating ready list into the scheduler. In this report we give a detailed explanation of how each function works, its time complexity and its importance to the free RTOS code. We have also made sure that we make the report more explanatory to the person by using a couple of pictures depicting how the scheduler works.

OVERVIEW OF FREE RTOS SCHEDULER

The FreeRTOS operating system's scheduler is a key component of task management and scheduling. Its main duty is to choose which task should be executed at any given moment and to allocate the system resources accordingly. Here are some key functions performed by the FreeRTOS scheduler:

1. **Task Creation**: The free RTOS scheduler manages the lifecycle of tasks in the system. It facilitates the creation of tasks by including the necessary resources.
2. **Task Scheduling**: The scheduler determines the order in which tasks are executed. It employs a priority-based scheduling algorithm, where each task is assigned a priority level that defines its relative importance.
3. **Task Switching**: The scheduler performs task switching to transition between different tasks. It saves the context of the currently running task and restores the context of the next task to be executed.
4. **Task Preemption**: The scheduler manages task priorities and enforces preemption if enabled, allowing higher-priority tasks to interrupt lower-priority tasks when necessary.
5. **Task Suspension and Resumption**: The scheduler provides mechanisms to suspend and resume tasks. Suspended tasks are temporarily inactive and do not participate in the scheduling process.
6. **Task Prioritization**: Higher-priority tasks are given precedence in execution, ensuring that critical tasks receive the necessary resources and are completed in a timely manner.
7. **Idle Task Management**: The scheduler includes an idle task, which is executed when no other tasks are ready to run. The idle task typically enters a low-power mode, conserving energy while waiting for a task to become ready.

Importance of ready list in scheduler's functions:

The ready list plays one of the most important roles in the scheduler's role as it is involved in task scheduling, priority management, efficient task lookup, Dynamic task management and real time responsiveness. By effectively organizing and prioritizing tasks, the Ready List helps optimize system performance and ensures the timely execution of tasks in a multitasking environment.

UNDERSTANDING THE READY LIST

In real-time operating systems (RTOS), a ready list is a data structure used to manage the execution of tasks or threads. An RTOS is designed to handle time-critical tasks and ensure that they are executed in a timely manner.

The ready list is the most important data structure in the free RTOS code as it deals with process scheduling and selection of processes to be executed by the CPU according to its priority.

The ready list is typically implemented as an **Array of Doubly Linked List.** It contains all the tasks or threads that are ready to run, meaning they have met their execution criteria and are waiting for the scheduler to allocate the CPU to them. Tasks in the ready list are usually in a runnable state and have all the necessary resources available for execution.

- **Adding Tasks to the Ready List:**

When a task becomes ready to run, meaning it has met its execution criteria and has all the necessary resources available, it is added to the ready list by the scheduler.

- **Task Selection and Execution:**

When it is time to select the next task for execution, the scheduler retrieves the task from the ready list. The task is typically selected based on the scheduling policy in use.

- **Updating the Ready List:**

As tasks progress and their states change, the ready list needs to be updated dynamically. When a task completes its execution or gets blocked, it is removed from the ready list. Conversely, when a task becomes ready to run, it is added to the ready list.

Performance Analysis and the Limitations of the ready List:

The performance of the ready List is evaluated based on several factors:

1)Task Scheduling Efficiency: The ready list plays an important role in deciding which task needs to be executed first based upon its priority.

2)Insertion and Removal Operations:How easy it is to insert and remove from the ready list.

3)Scalability: How the ready list is able to cope up when the number of tasks increases by a huge amount.

4)Memory Overhead:How the ready list minimizes the memory overhead by employing a compact data structure for the ready list.

Despite this the ready list has certain limitations:

1)Priority Inversion due to nested mutexes: Only jobs waiting for execution according to their priority are affected by the ready list's effect on priority inversion. Priority inversion can still happen if the priority ceiling or inheritance protocols are not used properly or if nested mutexes are present.

2)Task Lookup Time: The frequency of preemption in the system may have an effect on how effective the ready list is. The ready list maintenance and updating may incur additional expense due to frequent context shifts and task preemptions.

3)Memory usage The memory overhead of the ready list depends on factors such as the number of priority levels and the maximum number of tasks per priority level. In systems with limited memory resources, the ready list's memory usage may become a constraint.

4)Preemption Overhead: The ready list's efficiency may be impacted by the frequency of preemption in the system. Frequent context switches and task preemptions can introduce additional overhead in maintaining and updating the ready list.

ANALYZING THE IMPLEMENTATION OF THE READY LIST

In FreeRTOS, Ready List is represented by the identifier ***`pxReadyTasksList`***. The `pxReadyTasksLists` data structure is an array of `List_t` structures. It organizes tasks based on their priorities and uses doubly linked lists to store tasks with the same priority. It is indexed based on priority of the list of tasks, which vary from 0 to 9.

For example, `pxReadyTasksLists[2]` is used to retrieve a `List_t` object which is a Doubly Linked List containing pointers to tasks' `TCB` structure with priority set to 2. Hence ensuring a time complexity of `O(1)`.

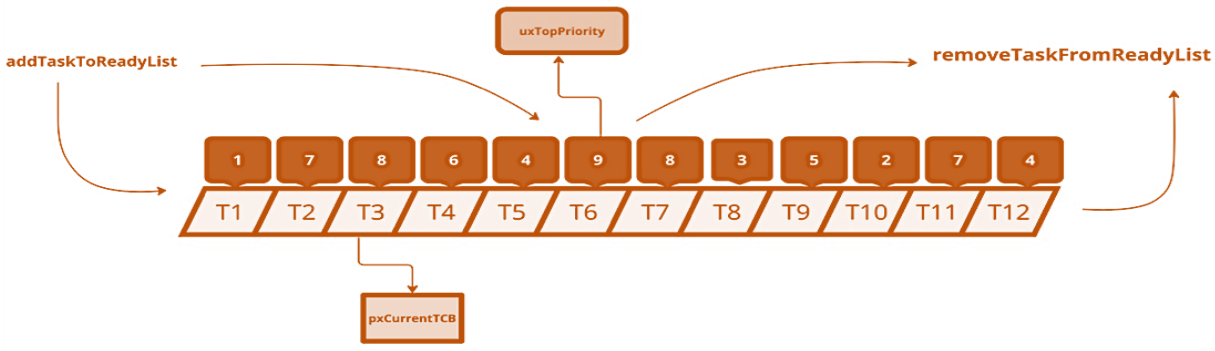
A detailed analysis on `TCB_t` data structure is provided in the later sections.

Rationale:

The choice for the `pxReadyTasksLists` to be implemented as an array of doubly linked lists is to optimize the performance of insertion, deletion and accessing lists of given priority in constant time complexity. In a DLL, insertion and deletion involves the change of pointing addresses of the `next` and `previous` pointers of the `ListItem_t` objects in the DLL `List_t`.

The Ready List is indexed by the priority of the Tasks grouped in the same DLL `List_t` by their priority. Hence, accessing the list of elements of a given priority is done in constant time, `O(1)`.

The space complexity of this implementation of Ready List is dependent on the number of priority levels configured. It's also worth mentioning that the amount of memory used by the ready list is affected by both the number of tasks and their distribution across various priority levels.



DETAILED EXAMINATION OF READY LIST OPERATIONS

Examination of Associated Macros

1. **`Portmacro:`** These macros provide a level of portability by allowing the RTOS to adapt to different data types based on the target platform's architecture and compiler.
2. **`TaskRecord_Ready_Priority`** and **`PortRecord_Ready_Priority:`** The purpose of these macros is to provide a mechanism for the RTOS to keep track of the ready priority of tasks.
3. **`traceMoved_task_to_ready_state:`** The purpose of this macro call is to update any data structures or records in the RTOS that are related to task priorities.
4. **`privileged_data:`** This macro is used to define variables or data structures that require privileged access or special memory attributes.
5. **`vtaskdelete:`** This macro includes freeing the resources associated with the task, removing it from any task lists, and potentially performing other cleanup operations specific to the RTOS.
6. **`configMax_Priorities:`** It is a configuration option that determines the maximum number of priority levels that can be assigned to tasks in the RTOS.
7. **`mtCoverage_Test_Marker:`** This macro in FreeRTOS is a placeholder that can be used to mark specific lines of code for code coverage testing.
8. **`ConfigAssert:`** This macro is a debugging macro used in FreeRTOS to perform assertions or checks during runtime.
9. **`vListInsertEnd:`** This macro is used to insert an item at the end of a list. It takes two arguments, the 1st argument is a pointer to the list, and the 2nd argument is a pointer to the item to be inserted.
10. **`List_t:`** This macro is a type of definition for a linked list. It takes one argument: the type of data items that will be stored in the list.
11. **`uxListRemove:`** this macro is used to remove an item from a list. It takes 2 arguments, 1st one is the list from which the item will be removed. 2nd one is the index of the item to be removed.
12. **`vListInitialise:`** This macro is used to initialize a list with a specified number of elements. It takes three arguments, 1st one is the list to be initialized, 2nd one is the number of elements to be initialized, 3rd one is the value to be assigned to each element.
13. **`vListInsert:`** This macro is used to insert an item into a list at a specified index. It takes three arguments, 1st one is the list into which the item will be inserted. 2nd one, is the index at which the item will be inserted. 3rd one, is the item to be inserted.

Function Analysis of `vTaskSwitchContext()`

Context switching involves saving the process or thread state to resume execution later and restoring a different, previously saved state. The function `vTaskSwitchContext()` is part of the FreeRTOS which is responsible for task scheduling and context switches. The function's details are as follows:

1. Suspension Handling and Task Context Switching

The function checks if the scheduler is suspended. If so, no context switch happens, and the pending yields of tasks are scheduled to execute. Otherwise, *`xYieldPending`* is set to *`pdFalse`* and context switching occurs. This guarantees tasks do not execute while the scheduler is suspended.

2. Run-Time Stats Calculation

If the OS is configured to generate run-time stats, then the run-time of the current task is calculated using the mechanism: $RunTimeCounter += TotalRunTime - TaskSwitchedInTime.$ This is then assigned to the Task's Task Control Block struct.

3. Stack Overflow Checks

Following the run-time statistics calculation, the function performs two stack overflow checks. It calls the `taskFIRST_CHECK_FOR_STACK_OVERFLOW()` and `taskSECOND_CHECK_FOR_STACK_OVERFLOW()` functions. This ensures the stability and integrity of the system by identifying and handling potential stack overflow issues.

4. Task Selection

After the stack overflow checks `taskSELECT_HIGHEST_PRIORITY_TASK()` is called to determine the next task to run. The selection function then calculates the `uxTopPriority` from the ready list of the tasks. If the ready task list contains tasks with the calculated priority, system interrupts are configured to occur and selection of next task entry in the prioritised ready list is done by the function `listGET_OWNER_OF_NEXT_ENTRY()`.

Function Analysis of `vTaskDelete()`

- The function takes a parameter ***xTaskToDelete.*** The *taskENTER_CRITICAL()* starts the critical section execution.
- The *pxTCB* is assigned to the return value of the function *prvGetTCBFromHandle(xTaskToDelete)*
- The *uxListRemove(&(pxTCB → xGenericListItem)) == 0* removes the task from the ready list with member address of *xGenericListItem*.
    - If the return value is 0, then the task was the only task at that priority level, so priority is reset using *taskRESET_READY_PRIORITY*.
- *listLIST_ITEM_CONTAINER(&(pxTCB->xEventListItem))!= NULL* checks if the task is waiting on an event by verifying if the *xEventListItem* member of the *pxTCB* is associated with list item container. If not *NULL*, then it is waiting for an event.
- If the task is waiting on an event, then the *uxListRemove(&(pxTCB->xEventListItem)* removes the task from the event list with the address of *xEventListItem*.
- *vListInsertEnd(&xTasksWaitingTermination, &(pxTCB->xGenericListItem))* inserts the task into the termination list by calling the function with address of *xGenericListItem*, which is a member of *TCB*.
- Finally, *uxTasksDeleted* is also incremented for deletion and *uxTaskNumber* is also updated to include awareness for kernel debuggers that the task list needs regeneration as the list has been modified.
- *taskEXIT_CRITICAL()* is called to exit the critical section and to signal that interrupts can function normal again. Job is not done yet!
- If, *xSchedulerRunning* is not *pdFalse* , (if the scheduler is running)
    - If *pxTCB == pxCurrentTCB* (if the task is current task)
        - The function ***portPRE_TASK_DELETE_HOOK(pxTCB, &xYieldPending)*** is called as it is primarily used for performing specific cleanup operations before a task is deleted.
        - Then, ***portYIELD_WITHIN_API()*** is called as in some situations after pre-delete hook functions, it may not be able to yield away from the task, so a pre-context switch is scheduled.
    - If, the running task is not the current task, then:
        - We enter back to critical section using *taskENTER_CRITICAL()*, we call *prvResetNextTaskUnblockTime()* (line #3602 `os_tasks.c`) to reset the next unblock time in case it referred to the task that has just been deleted.

- Then, it exits the critical section using *taskEXIT_CRITICAL().*

Time complexity of vTaskDelete():

- Time complexity of `prvGetTCBFromHandle` macro is O(1).
- Time complexity of `portRESET_READY_PRIORITY` is O(1) a s priority queue is implemented as list.
- Time complexity od `uxRemoveList` is O(1).
- Similarly, time complexity of `vListInsertEnd()` is also O(1).
- Hence, time complexity of `vListInsertEnd()` is constant – O(1).

Analysis of *uxTopReadyPriority*

- It is a variable or data structure that represents the priority of the highest ready task or thread in the system.
- In an RTOS, tasks or threads are typically assigned priorities based on their importance or urgency. The priority level determines the order in which tasks are scheduled for execution. The higher the priority, the more urgent or important the task is considered, and it will be scheduled to run before tasks with lower priorities.
- The *uxTopReadyPriority* variable or data structure is commonly used by the RTOS scheduler to keep track of the highest priority among the ready tasks or threads. It allows the scheduler to efficiently determine which task should be scheduled for execution next.
- By maintaining the highest priority, the scheduler can quickly identify the task with the most urgency and ensure that it gets CPU time as soon as possible.
- The *ux* prefix refers to unsigned *x* (unsigned integer with x bits) indicating that the variable stores an unsigned integer value representing the priority.

Where is uxTopReadyPriority used?

- It is first defined in line 268, which tracks the highest priority tasks.
- Line 331, a function to select highest priority task - *taskSELECT_HIGHEST_PRIORITY_TASK().*
- Line 351, a macro to reset priority to the task with high priority- *portRESET_READY_PRIORITY ().*

Function Analysis of `vTaskPlaceOnEventList()`

The function takes two parameters: *pxEventList*, which is a pointer to the event list where the task will be inserted, and *xTicksToWait*, which represents the duration in ticks the task should wait for the event.

- The *configASSERT* macro is used to assert that *pxEventList* is not *NULL*. If it evaluates to *pdFALSE* (0), the macro will trigger an assertion failure.
- The *vListInsert* function is called to insert the event list item of the Task Control Block (TCB) of the current task into the specified event list. The insertion is done in priority order.
- The task is removed from the ready list using the *uxListRemove* function. Once the blocking condition is satisfied, the task is moved from the blocked list back to the ready list, making it eligible for execution by the scheduler.
- If *(xTicksToWait == portMAX_DELAY)* which says indefinite delay then the task will be added to suspended task
- Else the tasks with a calculated delayed time using *ticks_to_wait* and *tick_count* are arranged in priority order of delay time in delayed tasks list.
- The overall time complexity of the vTaskPlaceOnEventList() function is O(1).
- The vTaskPlaceOnEventList() function only requires a pointer to the event list, so the space complexity is O(1).

Function Analysis of `xTaskRemoveFromEventList()`

- The function *xTaskRemoveFromEventList* takes a pointer to an event list (*pxEventList*) as a parameter and returns a *BaseType_t* value.
- The *xReturn* variable is declared to store the return value indicating whether the unblocked task has a higher priority than the calling task.
- The *listGET_OWNER_OF_HEAD_ENTRY* macro is used to obtain the owner of the head entry (highest priority task) in the event list. This owner is cast to *TCB_t** and assigned to the *pxUnblockedTCB* variable.
- The *configASSERT* macro is called to assert that the *pxUnblockedTCB* is not *NULL*, ensuring that a valid *TCB* is obtained.
- The *uxListRemove* function is used to remove the *xEventListItem* of the *pxUnblockedTCB* from the event list.
- If the scheduler is not suspended (*uxSchedulerSuspended* is *pdFALSE*), the task is removed from the generic list (ready list) using *uxListRemove* and added to the ready list using *prvAddTaskToReadyList*.

- If the scheduler is suspended, the task is inserted at the end of the *xPendingReadyList*, which holds tasks that are pending until the scheduler is resumed.
- If the unblocked task's priority is not higher than the calling task's priority, *xReturn* is set to *pdFALSE*.
- If tickless idle is enabled (*configUSE_TICKLESS_IDLE == 1*), the *prvResetNextTaskUnblockTime* function is called to reset the *xNextTaskUnblockTime* value.
- This ensures that if a task is blocked on a kernel object, the system enters sleep mode at the earliest possible time.
- The overall time complexity of the xTaskRemoveFromEventList() function is O(1).
- The xTaskRemoveFromEventList() function only requires a pointer to the event list, so the space complexity is O(1).

Functional Analysis of `prvaddtasktoreadylist()`

- The function takes the pointer to the *TCB* as its parameter, checks the readiness of the task based on a certain priority and invokes the *vListInsertEnd* function to add the task to end of the ready list based upon the priority of the task.
- By inserting the task at the end, it ensures that the task will be considered for execution after other tasks with same priority in the ready list.
- The free RTOS scheduler will select the task from the ready list based upon its priority.
- The time complexity of the function primarily depends on the time complexity of the *vListInsertEnd* which is the function called within it.In the worst-case scenario when the new list item is inserted at the end of the n items, the time complexity is *O(n)*.
- The *addTaskToReadyList()* is one of the most essential functions as it is used for various purposes. It is invoked when there is task initialization, task state transitions, priority changes when there is a priority inheritance or mutex ownership, task resumption from the suspended state.
- Task initialization is when a new task is created, task state transition is when the existing task is moved from a blocked or suspended state to a ready state, priority inheritance and mutex ownership occurs when a lower priority task captures the resource requires by a higher priority task and suspension of scheduler occurs when a certain critical section is supposed to be executed.

INTEGRATION OF THE READY LIST WITH THE SCHEDULER

Context Switch and Task Pre-emption

Context Switch and Task Pre-emption is done by the mechanism where Ready List's doubly linked lists runs its operations *`vListInsert()`*, *`vListDelete()`* and *`addCurrentTaskToDelay()`*. These are systematically and atomically invoked by the functions *`vTaskSwitchContext()`* and *`xTaskGenericCreate()`*. These involves list operations of insertions and deletions of list item, *`TCB`*, which represents the task and hence context switching and pre-empting the tasks.

The *`vTaskSwitchContext()`* function checks the next task in the Ready List with the highest priority. It also saves the run-time stats for the task that will be switched in its *`TCB`* struct.

The function *`xTaskGenericCreate()`* calls the macro *`taskYIELD_IF_USING_PREEMPTION()`*. This macro enables tasks to give up the processor voluntarily if pre-emption is enabled. This allows higher-priority tasks from the Ready Lists to run and ensures a responsive and predictable system behaviour.

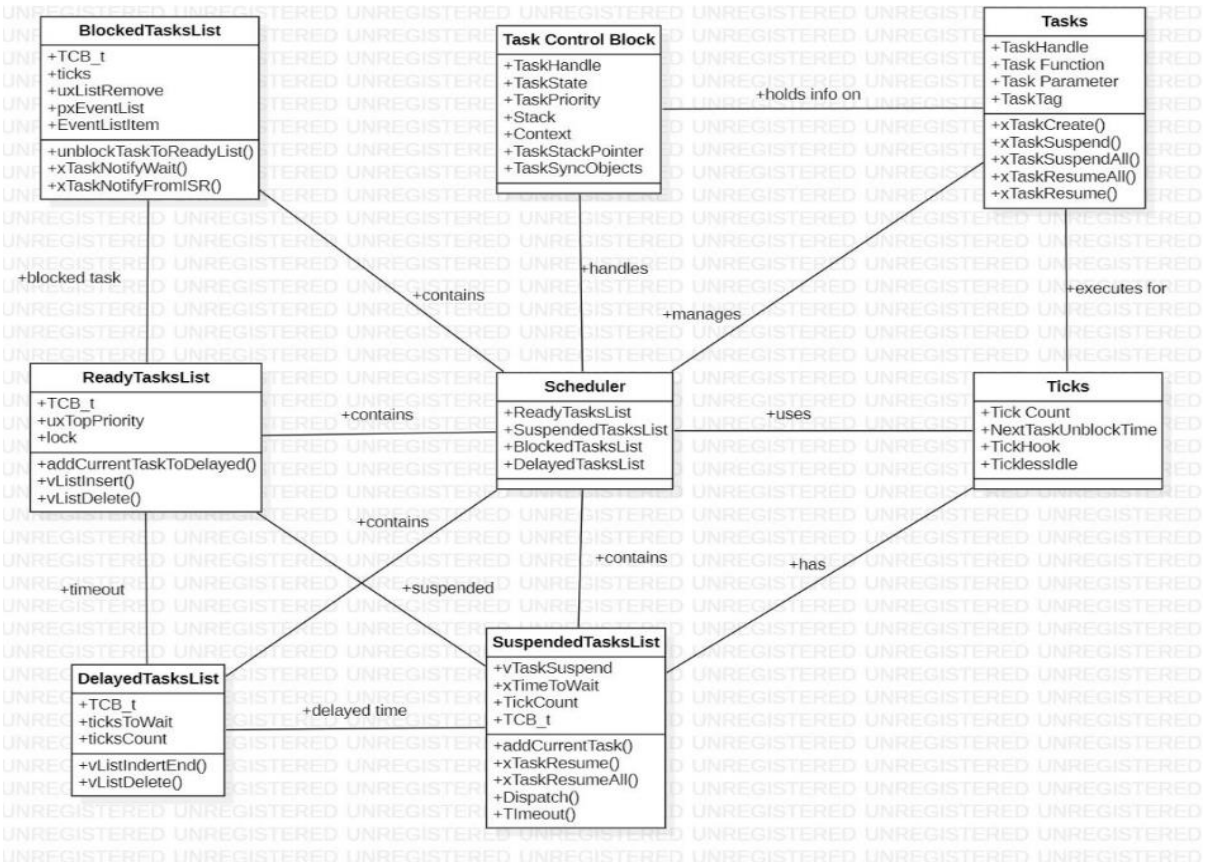


Figure 2: UML diagram representing the interaction between various data structures and the scheduler.

Synchronization Mechanisms

To make inter-task communication and coordination easier, FreeRTOS offers a variety of synchronization techniques:

1.Mutexes: Mutexes are used to protect shared resources from concurrent access. Tasks can acquire and release a mutex to ensure exclusive access to a resource.

2.Semaphores: Semaphores are used for signaling and synchronization between tasks. They can be used to indicate the availability of a resource.

3.Queues: Queues provide a way for tasks to send and receive messages or data items.

4.Event Flags: Event flags are used to communicate and synchronize between tasks based on specific event conditions. Tasks can wait for one or more event flags to be set, and other tasks can set or clear these flags to signal events.

5.Task Notifications: Task notifications are a lightweight form of inter-task communication. They allow tasks to send and receive simple notification values, which can be used for synchronization, event signaling, or communication purposes.

Conclusion:

As a real-time operating system, FreeRTOS provides effective mechanisms for task prioritisation, synchronisation, and scheduling, ensuring deterministic and predictable behaviour in time-critical applications. FreeRTOS is appropriate for embedded systems with limited resources due to its tiny memory footprint, low overhead, and adaptability to different hardware platforms. FreeRTOS is a dependable and adaptable option for developers due to its open-source nature, availability of comprehensive documentation, and active community. FreeRTOS offers a strong foundation for creating dependable and effective embedded applications, whether they are used in consumer electronics, industrial automation, medical devices, or automotive systems.

TOOLS AND RESOURCES

1. ChatGPT – for understanding code
2. Star UML – for UML diagrams
3. Miro Board – for diagrams
4. Notion AI – for better paraphrasing