REPORT

1. Introduction

Free RTOS and it’s Significance

FreeRTOS is an open-source real-time operating system kernel designed for microcontrollers and small microprocessors. It offers task scheduling, inter-task communication, and synchronization. With a heap memory management scheme, it enables dynamic memory allocation.

The task scheduler efficiently executes multiple tasks on a single processor, prioritizing higher-priority tasks. Inter-task communication and synchronization mechanisms prevent issues like race conditions. FreeRTOS includes a heap memory management scheme for dynamic memory allocation.

It provides lightweight libraries and utilities, including TCP/IP stack, USB stack, and file system, ideal for embedded systems. FreeRTOS is highly portable, supports various processors, and integrates with popular development environments.

The integrated development environment offers debugging features for issue identification and resolution. FreeRTOS also provides additional libraries for expanding the operating system's functionality.

In summary, FreeRTOS is a flexible and powerful real-time operating system for embedded systems, offering task management, communication, and synchronization capabilities, along with extensive library support.

Purpose of the Report

The purpose of this report is to provide an overview of the implementation of the Ready List in the FreeRTOS scheduler. The report aims to explain the significance and functioning of the Ready List, which plays a crucial role in task scheduling within the FreeRTOS operating system.

The FreeRTOS scheduler is responsible for determining which tasks should be executed on the processor at any given time. The Ready List is a data structure used by the scheduler to keep track of all the tasks that are ready to run, meaning they are in a state where they can be executed.

The report focuses on the specific implementation details of the Ready List in FreeRTOS. It explains how tasks are added to the Ready List when they become ready, and how the scheduler selects the highest priority task from the Ready List for

execution. The report also delves into the data structure and algorithms used to efficiently manage the Ready List and ensure optimal task scheduling performance.

By examining the Ready List implementation in FreeRTOS, the report aims to provide insights into the inner workings of the scheduler and how it enables effective multitasking in real-time embedded systems. The focus on the Ready List specifically allows for a deeper understanding of the mechanisms behind task scheduling and the importance of efficient data structures in real-time operating systems.

Overall, the report combines an explanation of the purpose and significance of the report itself with a specific focus on the implementation of the Ready List within the FreeRTOS scheduler.

2. Overview of FreeRTOS Scheduler

FreeRTOS scheduler's role in task management and scheduling

The FreeRTOS scheduler plays a crucial role in task management and scheduling within the FreeRTOS operating system. It is responsible for determining which tasks should be executed on the processor at any given time, ensuring efficient utilization of system resources and meeting real-time requirements.

The scheduler maintains a Ready List, which is a data structure that keeps track of all the tasks that are ready to run, meaning they are in a state where they can be executed. Each task in the system is assigned a priority level, and the scheduler uses this information to decide the order in which tasks should be executed.

When a task becomes ready to run, such as when it is created or when it is unblocked from waiting for a particular event, it is added to the Ready List according to its priority. The scheduler then selects the highest priority task from the Ready List and dispatches it for execution on the processor.

The scheduler employs a preemptive priority-based algorithm, which means that a higher-priority task can interrupt the execution of a lower-priority task. This ensures that tasks with higher priority, such as time-critical or important tasks, are given precedence in execution.

The scheduler also handles context switching, which involves saving the state of the currently running task and restoring the state of the next task to be executed. This transition between tasks allows for multitasking and ensures that each task gets its fair share of processor time.

Additionally, the scheduler provides mechanisms for synchronization and communication between tasks, such as semaphores, mutexes, and message queues. These mechanisms allow tasks to coordinate their activities, share data, and avoid conflicts in accessing shared resources.

Overall, the FreeRTOS scheduler plays a critical role in managing and scheduling tasks within the operating system. It determines the order in which tasks are executed, ensures task priorities are respected, handles context switching, and provides synchronization and communication mechanisms, all of which are essential for efficient multitasking and meeting real-time requirements in embedded systems.

Importance of Ready List in Scheduling

The FreeRTOS scheduler is a critical component of the operating system, responsible for managing task execution and ensuring efficient resource utilization. The Ready List, a fundamental feature of the scheduler, plays a vital role in its functionality.

1**. Task Scheduling and Priority-based Execution**

The Ready List maintains a list of tasks that are ready to run, allowing the scheduler to select the highest priority task for execution. This enables the system to prioritize critical and time-sensitive tasks, ensuring timely execution and meeting real-time requirements.

2**. Efficient Task Management**

The Ready List provides a structured representation of ready tasks, enabling quick identification of the highest priority task without the need for searching through all tasks. This organization minimizes overhead, improves responsiveness, and enhances the overall efficiency of task management within the operating system.

3**. Dynamic Task Addition and Removal**

As tasks transition between different states (such as becoming ready or blocked), the Ready List dynamically adjusts to reflect these changes. Tasks are added to the Ready List when they become ready for execution and removed when they are blocked or complete execution. This dynamic management ensures that only relevant tasks are considered for scheduling, optimizing system responsiveness and resource utilization.

4**. Synchronization and Communication**

The Ready List is closely tied to synchronization and communication between tasks. Synchronization primitives, such as semaphores and mutexes, utilize the Ready List to manage waiting tasks and determine the order in which they are unblocked. This

allows for effective coordination and communication among tasks, enhancing the overall functionality and reliability of the system.

In conclusion, the Ready List is a crucial component of the FreeRTOS scheduler, playing a vital role in task scheduling, efficient task management, dynamic task handling, and supporting synchronization and communication between tasks. Its presence ensures optimal resource utilization, responsiveness, and the ability to meet real-time requirements in embedded systems.

By understanding the significance of the Ready List in the FreeRTOS scheduler, developers and system designers can make informed decisions about task management and scheduling strategies, leading to more effective and reliable embedded system designs.

3. Understanding the Ready List

Purpose of Ready List in Task Scheduling and it’s significance in managing tasks

The Ready List is a critical component in task scheduling within the FreeRTOS operating system. It serves as a data structure that maintains information about the tasks that are ready to be executed. The Ready List plays a significant role in managing tasks' states efficiently and ensuring effective task scheduling.

The purpose of the Ready List is to facilitate the efficient execution of tasks by allowing the scheduler to quickly identify the highest priority task that should be executed next. It organizes tasks based on their priorities, with higher-priority tasks typically placed at the front of the list. This priority-based organization enables the scheduler to make informed decisions about task execution order, ensuring that tasks with critical requirements or higher priority are executed promptly.

The Ready List's significance lies in its ability to manage tasks' states effectively. It efficiently handles the transition of tasks between different states, such as ready, blocked, running, and suspended, enabling seamless task management within the operating system

Here are the key aspects that highlight the **significance of the Ready List in managing tasks' states** efficiently:

1. Ready State: When a task becomes ready to run, it is added to the Ready List. The Ready List keeps track of all the tasks that are ready for execution. This organization allows the scheduler to easily identify and select the highest priority

task from the Ready List for execution, ensuring optimal resource utilization and meeting real-time requirements.

2. Blocked State: Tasks can become blocked due to various reasons, such as waiting for a specific event or a resource to become available. When a task is blocked, it is removed from the Ready List, indicating that it is not ready for execution. This efficient removal from the Ready List helps the scheduler focus only on the tasks that are ready to run, preventing unnecessary processing overhead.
3. Running State: The Ready List plays a crucial role in determining when a task transitions from the running state to another state. After a task completes its execution or yields the processor voluntarily, the scheduler consults the Ready List to determine the next task to be executed. The task that was running is either removed from the Ready List (if it has completed) or placed back in its appropriate position based on its priority.
4. Suspended State: Tasks can also be suspended, meaning they are temporarily halted and not eligible for execution. When a task is suspended, it is not included in the Ready List, effectively excluding it from the task scheduling process. By maintaining the Ready List without suspended tasks, the scheduler can focus on actively running and ready tasks, improving scheduling efficiency.

By efficiently managing the tasks' states, including ready, blocked, running, and suspended, the Ready List ensures that the scheduler can make optimal decisions for task scheduling and resource allocation. It allows the system to prioritize critical tasks, efficiently handle blocked tasks, seamlessly transition between different task states, and effectively manage suspended tasks. As a result, the Ready List significantly contributes to the overall performance, responsiveness, and real-time capabilities of the FreeRTOS operating system.

In summary, the Ready List plays a significant role in managing tasks' states efficiently within the FreeRTOS scheduler. It facilitates the transition of tasks between the ready, blocked, running, and suspended states, ensures optimal task scheduling, and contributes to the overall performance and real-time responsiveness of the operating system.

4. Analyzing the implementation of Ready List

Data Structure used to Implement Ready Queue

Upon thorough analysis, we have determined that the data structure used in FreeRTOS's Ready List is a circular double linked list. This conclusion was reached after conducting an in-depth examination of the codebase and its associated functions. Our initial observation led us to ascertain that the data structure was a double linked list based on the utilization of the `struct xlist` and its `pxNext` and `pxPrevious` pointers.

To further solidify our understanding, we delved into the implementation of the `vListInsert()` function. This function played a pivotal role in confirming the presence of a double linked list by performing operations that involved both the forward (`pxNext`) and backward (`pxPrevious`) pointers. These operations, along with other supporting evidence, indicated bidirectional traversal capabilities.

Additionally, the circular nature of the list was identified through a careful examination of the code. By observing the linkages between the first and last items in the list, it became evident that the list formed a closed loop, effectively creating a circular structure.

Let’s see how we came to this conclusion that the data structure used here is Circular Double Linked List.

First we analyzed the **vlistInitialise Function**

```
void vListInitialise( List_t * const pxList )
{
pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );
pxList->xListEnd.xItemValue = portMAX_DELAY;
pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );
pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );
pxList->uxNumberOfItems = ( UBaseType_t ) 0U;
listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
}
```

After examining the provided code snippet, we can confidently assert that the data structure being initialized is a circular double linked list. Here's how we arrived at this conclusion:

1. Initialization of List End:
    - The code initializes the `xListEnd` member of the `pxList` structure, which serves as the end marker of the list.

- By assigning the address of `pxList->xListEnd` to `pxList->pxIndex`, the `xListEnd` node becomes the only entry in the list, indicating the list's endpoint.

2. Circular Connection:
   - The `xListEnd` node possesses both `pxNext` and `pxPrevious` pointers that point to itself.
   - `pxList->xListEnd.pxNext` is assigned the address of `pxList->xListEnd`, establishing a circular link by making it point to itself.
   - Similarly, `pxList->xListEnd.pxPrevious` is assigned the address of `pxList->xListEnd`, completing the circular connection.

We made it sure when we went through the function **vlistInsert Function**

To properly explain the concept of a circular linked list, it is essential to understand the following code:

```
pxList->xListEnd.pxNext = (ListItem_t *) &(pxList->xListEnd);
pxList->xListEnd.pxPrevious = (ListItem_t *) &(pxList->xListEnd);
```

1. `pxList->xListEnd.pxNext = (ListItem_t *) &(pxList->xListEnd);`
   - This line sets the `pxNext` pointer of the `xListEnd` node to the address of the `xListEnd` node itself.
   - It forms a circular link where the last node points to the first node.
2. `pxList->xListEnd.pxPrevious = (ListItem_t *) &(pxList->xListEnd);`
   - This line sets the `pxPrevious` pointer of the `xListEnd` node to the address of the `xListEnd` node.
   - It completes the circular connection by making the first node point to the last node.

Together, these two lines create a circular relationship between the last and first nodes, forming a closed loop in the linked list.

In a circular linked list, traversal is possible from any node in both forward and backward directions. The `pxNext` pointer of a node points to the next node, and the `pxPrevious` pointer points to the previous node. As a result, it allows seamless traversal through the entire list without encountering a termination point.

Understanding the concept of a circular linked list is crucial in the context of task scheduling, as it enables efficient management of tasks and facilitates continuous cycling through the list to ensure fair and optimal task execution.

Advantages of using this Data Structure

The choice of a circular double linked list as the data structure for the Ready List in the FreeRTOS scheduler is based on several factors that contribute to its advantages in terms of performance and memory usage.

1. Efficient Task Management: The circular double linked list allows for efficient management of tasks in the Ready List. Its circular nature enables seamless traversal from any point in the list, allowing for quick access to the next ready task for execution. This reduces the time complexity of operations such as task insertion, removal, and context switching.
2. Constant-Time Insertion and Removal: The circular double linked list provides constant-time insertion and removal of tasks. When a new task becomes ready or a task completes its execution, it can be inserted or removed from the list in constant time, regardless of the list size. This efficiency is critical for real-time systems where tasks need to be scheduled promptly.
3. Dynamic Task Prioritization: The double linked list structure allows for easy prioritization of tasks. Tasks in the list are ordered based on their priority, with higher priority tasks placed closer to the head of the list. This ordering simplifies the task selection process during scheduling, as the highest priority task can be readily identified.
4. Low Memory Overhead: The circular double linked list has a low memory overhead compared to other data structures like arrays or dynamic arrays. It requires memory only for the list nodes and does not need additional memory allocation for resizing or reorganization. This memory efficiency is crucial, especially in resource-constrained environments where minimizing memory usage is essential.
5. Flexibility and Scalability: The circular double linked list provides flexibility and scalability in managing tasks. New tasks can be easily inserted or removed without the need for resizing the list or shifting elements. Additionally, the circular nature of the list ensures that the scheduler can continuously cycle through tasks, accommodating changing task priorities and dynamic task arrival and departure.

Overall, the choice of a circular double linked list as the data structure for the Ready List in FreeRTOS offers efficient task management, constant-time operations, dynamic prioritization, low memory overhead, and flexibility for handling a varying number of tasks. These advantages contribute to improved performance, reduced latency, and optimal memory utilization in task scheduling scenarios.

5. Detailled Examination of Ready List Operation

Firstly we will be discussing on how a task is created and it's insertion and deletion in detail.

First we look at the main function , This code is written for the Arduino platform and uses the FreeRTOS library to create two tasks that blink two different LEDs at a specified delay. The `setup()` function initializes the serial communication, creates the task handles, and then creates two tasks, `vTask1` and `vTask2`, using the `xTaskCreate()` function. Each task is responsible for blinking a different LED connected to the corresponding pin number. Once the tasks are created, the `vTaskStartScheduler()` function is called to start the FreeRTOS scheduler and begin multitasking.

On execution, both tasks will be executed in parallel and alternate in blinking their corresponding LED. The tasks will continue running indefinitely until the system is reset or powered off.

`vTask1` and `vTask2` are two task functions defined in the code provided. Both tasks are responsible for blinking an LED connected to a different pin number, as specified by the `LED1_PIN` and `LED2_PIN` macros

In the FreeRTOS library, a task handle is a pointer to a data structure that represents a particular task. It is returned by the `xTaskCreate()` function when a new task is created and can be used to reference and manipulate the task later in the program.

Task handles are useful when working with FreeRTOS because they provide a way to manage and control the tasks running in the system. For example, a task handle can be used to pause, resume, or delete a task. Additionally, task handles can be used to retrieve information about the state of a task, such as its priority, stack usage, or runtime statistics.

First let's get into deeper analysis of the function **xTaskCreate**

```
#define xTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCre
atedTask ) xTaskGenericCreate( ( pvTaskCode ), ( pcName ), ( usStackDepth ), ( pvParam
```

```
eters ), ( uxPriority ), ( pxCreatedTask ), ( NULL ), ( NULL ) )
```

The above code is a Macro in C Programming. In C programming, a macro is a fragment of code that is given a name and can be substituted for other code in the program. Macros are defined using the `#define` preprocessor directive, which associates a name with a replacement text.

In this case, the macro being defined is `xTaskCreate`, which is a function-like macro that takes six arguments. The purpose of this macro is to create a new task in the FreeRTOS scheduler.

The implementation of the `xTaskCreate` macro is a call to another function called `xTaskGenericCreate`, passing the six arguments to it. The `xTaskGenericCreate` function is a generic task creation function in FreeRTOS, which takes more arguments than the `xTaskCreate` macro.

The arguments of the `xTaskCreate` macro are defined as follows:

- `pvTaskCode`: A pointer to the task function that implements the task's code.
- `pcName`: A descriptive name for the task.
- `usStackDepth`: The stack size in words (not bytes) of the task's stack space.
- `pvParameters`: A pointer to a structure or data that is passed to the task function as an argument.
- `uxPriority`: The priority level of the task, with the highest priority being 0.
- `pxCreatedTask`: A pointer to a task handle that is used to reference the created task.

The `#define` statement is used to define the `xTaskCreate` macro, which can be used throughout the program as if it were a regular function. When the macro is used in the code, the preprocessor replaces it with the corresponding call to `xTaskGenericCreate`, passing the arguments specified in the macro.

The `xTaskGenericCreate` function is a FreeRTOS API function that is **used to create a new task**. The function is defined in the FreeRTOS source code, and its implementation can vary depending on the specific port and architecture being used.

Once task is created, in parallel we also have to start the schedular using vTaskStartScheduler Function

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;
#if ( INCLUDE_xTaskGetIdleTaskHandle == 1 )
    {
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * ) NUL
L, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), &xIdleTaskHandle );
    }
#else
    {
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * ) NUL
L, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL );
    }
#endif

#if ( configUSE_TIMERS == 1 )
    {
        if( xReturn == pdPASS )
        {
            xReturn = xTimerCreateTimerTask();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
#endif

    if( xReturn == pdPASS )
    {
        portDISABLE_INTERRUPTS();

#if ( configUSE_NEWLIB_REENTRANT == 1 )
        {
            _impure_ptr = &( pxCurrentTCB->xNewLib_reent );
        }
#endif

        xSchedulerRunning = pdTRUE;
        xTickCount = ( TickType_t ) 0U;
        portCONFIGURE_TIMER_FOR_RUN_TIME_STATS();
        if( xPortStartScheduler() != pdFALSE )
        {
            /* Should not reach here as if the scheduler is running the
      function will not return. */
        }
        else
        {
            /* Should only reach here if a task calls xTaskEndScheduler(). */
        }
    }
    else
    {
        /* This line will only be reached if the kernel could not be started,
    because there was not enough FreeRTOS heap to create the idle task
    or the timer task. */
```

```
        configASSERT( xReturn );
    }
}
```

This code is the implementation of the FreeRTOS scheduler, which is responsible for managing the execution of tasks on an embedded system.

The function `vTaskStartScheduler` initializes the scheduler and starts the execution of tasks. It performs the following steps:

1. Create the idle task: The idle task is a special task that runs when there are no other tasks to execute. It is created using the `xTaskCreate` API function, with the task function `prvIdleTask` and the task name "IDLE". If `INCLUDE_xTaskGetIdleTaskHandle` is defined, the task handle is stored in `xIdleTaskHandle` so it can be retrieved later using the `xTaskGetIdleTaskHandle` function.
2. Create the timer task: If `configUSE_TIMERS` is defined, a timer task is created using the `xTimerCreateTimerTask` function.
3. Disable interrupts: Interrupts are disabled to prevent a tick from occurring before or during the call to `xPortStartScheduler`.
4. Switch the Newlib reentrant structure: If `configUSE_NEWLIB_REENTRANT` is defined, the `_impure_ptr` variable is switched to point to the `_reent` structure specific to the task that will run first.
5. Set up the timer tick: If `configGENERATE_RUN_TIME_STATS` is defined, the timer/counter used to generate the run time counter time base is configured using the `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS` macro.
6. Start the scheduler: The function `xPortStartScheduler` is called to start the scheduler. If the function returns, it means that the scheduler has stopped due to a task calling `xTaskEndScheduler`.
7. Handle errors: If there is not enough FreeRTOS heap to create the idle task or the timer task, `xReturn` will not be `pdPASS`, and the function will assert using the `configASSERT` macro.

The `vTaskStartScheduler` function is typically called at the end of the system initialization code and should not return.

Here the above function tries to Start Scheduler using xPortStartScheduler which is responsible for setting up the system timer tick, enabling interrupts, and switching to

the first task so we should also look at it in detail for more proper understanding.

```
BaseType_t xPortStartScheduler(void)
{
  /* Configure the regions in the MPU that are common to all tasks. */
  prvSetupDefaultMPU();

  /* Start the timer that generates the tick ISR. */
  prvSetupTimerInterrupt();

  /* Reset the critical section nesting count read to execute the first task. */
  ulCriticalNesting = 0;

  /* Start the first task.  This is done from portASM.asm as ARM mode must be
  used. */
  vPortStartFirstTask();

  /* Should not get here! */
  return pdFAIL;
}
```

The function **xPortStartScheduler** is a part of the FreeRTOS kernel and is responsible for starting the scheduler. Here's a detailed explanation of the steps involved in this process:

1. Configure the system timer tick: The scheduler requires a timer tick interrupt to schedule tasks. xPortStartScheduler configures the timer tick to generate interrupts at a regular interval by calling the portSETUP_TICK_INTERRUPT() macro. The interval is set to the value of configTICK_RATE_HZ, which is typically defined as 1000 Hz.
2. Enable interrupts: The scheduler requires interrupts to be enabled to respond to external events and switch between tasks. xPortStartScheduler enables interrupts by calling the portENABLE_INTERRUPTS() macro.
3. Initialize the task lists: The scheduler initializes its internal data structures to manage tasks. It sets up the ready lists, blocked lists, and delayed lists to hold tasks in different states.
4. Switch to the first task: The scheduler switches to the highest-priority task that is ready to run. This is done by calling the portRESTORE_CONTEXT() macro to restore the context of the first task. This will switch the program counter to the first instruction of the first task.
5. Run the scheduler: The scheduler runs indefinitely, switching between tasks as necessary. When a task becomes blocked or delayed, it is moved to the

appropriate list, and the scheduler selects the next task to run. When a task is ready to run again, it is moved back to the ready list, and the scheduler selects it when it is the highest-priority task.

The above function calls few other functions which are used to perform the above steps so here is a more detailed explanation of what those function does:

1. prvSetupDefaultMPU(): This function is called to configure the regions in the MPU that are common to all tasks. MPU stands for Memory Protection Unit, and it is a hardware feature available on some processors that allows the operating system to restrict access to memory regions. In FreeRTOS, the MPU is used to protect the kernel's memory and prevent tasks from accessing each other's memory.
2. prvSetupTimerInterrupt(): This function is called to start the timer that generates the tick ISR. The tick ISR is a periodic interrupt that is used by the FreeRTOS scheduler to keep track of time and switch tasks.(its like for each task there is a specific time after that it will produce an interrupt so that current tasks gets back to ready state and other comes to running state).
3. ulCriticalNesting = 0: This line resets the critical section nesting count. Critical sections are used to protect shared resources and prevent race conditions. When a critical section is entered, interrupts are disabled and the critical section nesting count is incremented. When the critical section is exited, the nesting count is decremented and interrupts are re-enabled if the nesting count reaches zero.
4. vPortStartFirstTask(): This function is responsible for starting the first task. This is done from portASM.asm as ARM mode must be used. The first task that runs after the scheduler is started is the task with the highest priority that is in the "Ready" state.
5. return pdFAIL: This line is executed only if the function vPortStartFirstTask() fails to start a task. It returns pdFAIL, which is a macro defined as (BaseType_t)0. If the scheduler starts successfully, this line should not be reached.

In summary, the xPortStartScheduler function is responsible for starting the FreeRTOS scheduler by configuring the MPU, starting the timer tick, resetting the critical section nesting count, and starting the first task.

After the above steps being done successfully the list is initialized and end element is inserted successfully being initialized the highest maximum possible value.

Now lets discuss about the add and remove task functions in detail.

For this we have to get into xTaskGenericCreate in detail

The function `xTaskGenericCreate` is a part of a real-time operating system (RTOS) implementation and is responsible for allocating memory, initializing the TCB and stack, and adding the new task to the appropriate task lists. It plays a crucial role in creating and setting up a new task in an RTOS environment.

1. The function takes several parameters such as `pxTaskCode`, `pcName`, `usStackDepth`, `pvParameters`, `uxPriority`, etc., which provide information about the task to be created and we have already discussed about these parameters earlier during task create.
2. It begins by performing some assertions to ensure that the necessary parameters are valid.
3. The function then allocates memory for the Task Control Block (TCB) and stack required by the new task using the `prvAllocateTCBAndStack` function.
4. If the memory allocation is successful (`pxNewTCB != NULL`), the function proceeds to initialize the TCB and stack for the new task.
5. The top of the stack address (`pxTopOfStack`) is calculated based on the stack growth direction defined by the specific architecture.
6. The TCB is initialized with the task's name, priority, memory regions, and stack depth using the `prvInitialiseTCBVariables` function.
7. The TCB stack is then initialized to simulate the task's state as if it were already running, with the return address set to the task function (`pxTaskCode`) and the appropriate parameters.
8. If a handle for the created task (`pxCreatedTask`) is provided, it is populated with the TCB pointer for future reference.
9. The critical section is entered to ensure the integrity of the task lists, and the new task is added to the ready list using the `prvAddTaskToReadyList` function.
10. Various bookkeeping operations are performed, such as incrementing the task count, checking if it is the first task being created, and updating the current task pointer (`pxCurrentTCB`).
11. The function returns `pdPASS` if the task creation was successful. Otherwise, an error code (`errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`) is returned.
12. If the scheduler is running, and the created task has a higher priority than the current task, a context switch is triggered using `taskYIELD_IF_USING_PREEMPTION` to

ensure the new task starts executing immediately.

In summary, this function is responsible for allocating memory, initializing the TCB and stack, and adding the new task to the appropriate task lists. It plays a crucial role in creating and setting up a new task in an RTOS environment.

Use of pxCurrentTCB in the above Function

let us first have a clear idea about **pxNewTCB** and then can move to **pxCurrentTCB**

```
/* Allocate the memory required by the TCB and stack for the new task,
  checking that the allocation was successful. */
    pxNewTCB = prvAllocateTCBAndStack( usStackDepth, puxStackBuffer );
```

In the given code snippet, `pxNewTCB` refers to a pointer that holds the address of the Task Control Block (TCB) of a newly created task.

When a new task is created using the `xTaskGenericCreate` function, memory is allocated for the TCB and its stack. The `prvAllocateTCBAndStack` function is called to perform this allocation, and it returns the pointer to the allocated TCB and stack.

The `pxNewTCB` pointer is used to store this returned address of the newly allocated TCB. Subsequently, the code initializes various fields and parameters of the TCB, sets up its stack, and performs other necessary configurations before the task can start executing.

Here's how `pxNewTCB` is used in the code:

1. After the memory allocation for the TCB and stack, the code checks if the allocation was successful by verifying if `pxNewTCB` is not `NULL`. If the allocation fails, `pxNewTCB` will be `NULL`, indicating that the task creation process cannot proceed.
2. If the allocation is successful, the code proceeds with configuring and initializing the TCB using the `pxNewTCB` pointer. It sets up various parameters, such as task name, priority, stack details, and other relevant information.
3. The `pxNewTCB` pointer is also used to store the top of the task's stack address and update the TCB's `pxTopOfStack` field.
4. Finally, the `pxNewTCB` pointer is used to add the task to the ready list or perform other necessary operations for task scheduling and management.

Now lets look at **pxCurrentTCB**

In the above function, `pxCurrentTCB` refers to a pointer that holds the address of the Task Control Block (TCB) of the currently executing task. The TCB is a data structure that contains information about a task, including its state, priority, stack pointer, and other relevant details.

The `pxCurrentTCB` pointer is typically used by the RTOS scheduler to keep track of the currently executing task. It allows the scheduler to access and modify the TCB of the task that is currently running.

Here's how `pxCurrentTCB` is used in the code:

1. During task creation (`xTaskGenericCreate`), if `pxCurrentTCB` is `NULL`, it means that either there are no other tasks or all other tasks are in a suspended state. In this case, the newly created task is assigned as the current task by assigning its TCB address to `pxCurrentTCB`.
2. If `pxCurrentTCB` is not `NULL`, it means that there are other tasks running. In this case, the code checks if the scheduler is already running. If the scheduler is not running, the newly created task becomes the current task if it has a higher priority than the current task. This allows the newly created task to start execution immediately when the scheduler is started.
3. The `pxCurrentTCB` pointer is also used in other parts of the code to access the TCB of the current task and perform various operations or checks related to task scheduling and management.

Overall, `pxCurrentTCB` is a crucial pointer that points to the TCB of the currently executing task. It enables the RTOS scheduler to track the current task and make scheduling decisions based on task priorities and other factors.

Now let’s discuss, how Task enters Critical state using above 2 TCB in function **task_ENTER_CRITICAL()**

The function `taskENTER_CRITICAL()` is a macro typically used in FreeRTOS to enter a critical section or disable interrupts. In FreeRTOS, a critical section is a code region where interrupts are temporarily disabled to protect shared resources or ensure the integrity of critical operations.

When `taskENTER_CRITICAL()` is invoked, it performs the following actions:

1. It saves the current interrupt status or state of the processor.

2. It disables interrupts, preventing the execution of interrupt service routines (ISRs) and context switches.
3. It increments a nested critical section counter to support nested critical sections.

By disabling interrupts, `taskENTER_CRITICAL()` ensures that the current task can execute without being interrupted by higher-priority tasks or ISRs. This guarantees exclusive access to resources or critical code sections, preventing race conditions and ensuring data consistency.

It is important to note that the use of `taskENTER_CRITICAL()` should be paired with `taskEXIT_CRITICAL()` to re-enable interrupts and exit the critical section. This ensures that interrupts are not disabled indefinitely, allowing other tasks and ISRs to execute normally.

The above all task are done by tasks having priority and they are done by assigning the Current TCB equal to New TCB

The assignment `pxCurrentTCB = pxNewTCB;` is performed in the context of task creation, and it serves the following purposes:

1. Setting the current TCB: If there are no other tasks running or all other tasks are in the suspended state, the newly created task is assigned as the current task. This implies that the newly created task will be the one currently executing when the scheduler starts or resumes. By assigning `pxNewTCB` to `pxCurrentTCB`, the scheduler knows which task to execute first.
2. Updating the current TCB for higher-priority tasks: If the scheduler is already running and a new task is being created, the code checks if the new task has a higher priority than the current task (`pxCurrentTCB`). If the new task's priority is higher, it replaces the current task by assigning `pxNewTCB` to `pxCurrentTCB`. This ensures that the newly created task will be executed next, preempting the current task if necessary.

By updating `pxCurrentTCB` with the newly created task, the scheduler ensures that the appropriate task is selected for execution based on its priority and the current system state.

It's important to note that the exact behavior and purpose of `pxCurrentTCB` may depend on the specific implementation and configuration of FreeRTOS. It is a fundamental variable used by the scheduler to keep track of the currently executing task and manage task switching.

In summary, in the given code, `pxCurrentTCB` is updated with `pxNewTCB`, indicating that the newly created task is set as the current task or the task with the highest priority to be executed next, depending on the system state and scheduler behavior

```
taskENTER_CRITICAL();
        {
            uxCurrentNumberOfTasks++;
            if( pxCurrentTCB == NULL )
            {
                /* There are no other tasks, or all the other tasks are in
        the suspended state - make this the current task. */
                pxCurrentTCB =  pxNewTCB;

                if( uxCurrentNumberOfTasks == ( UBaseType_t ) 1 )
                {
                    /* This is the first task to be created so do the preliminary
          initialisation required.  We will not recover if this call
          fails, but we will report the failure. */
                    prvInitialiseTaskLists();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                /* If the scheduler is not already running, make this task the
        current task if it is the highest priority task to be created
        so far. */
                if( xSchedulerRunning == pdFALSE )
                {
                    if( pxCurrentTCB->uxPriority <= uxPriority )
                    {
                        pxCurrentTCB = pxNewTCB;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            uxTaskNumber++;

#if ( configUSE_TRACE_FACILITY == 1 )
            {
                /* Add a counter into the TCB for tracing only. */
                pxNewTCB->uxTCBNumber = uxTaskNumber;
            }
#endif /* configUSE_TRACE_FACILITY */
            traceTASK_CREATE( pxNewTCB );
```

```
            // Update TCB parameters of the task
            //#if ( configUSE_EDFVD_SCHEDULER == 1 )

            prvAddTaskToReadyList( pxNewTCB );
            xReturn = pdPASS;
            portSETUP_TCB( pxNewTCB );
        }
        taskEXIT_CRITICAL();
    }
```

We are discussing about the task_ENTER_CRITICAL function because this is the function which is responsible for adding tasks to ready list if any other task with higher priority is already present

If there are other active tasks:

- If the scheduler is not already running (`xSchedulerRunning == pdFALSE`), the code checks whether the priority of the newly created task (`uxPriority`) is higher or equal to the priority of the current task (`pxCurrentTCB->uxPriority`).
- If the new task has a higher or equal priority, it becomes the new current task (`pxCurrentTCB = pxNewTCB`).

Once it becomes the new current task using traceTask_CREATE we will be able to add task to Ready List.

when we are looking at the above code in taskenterCritical we can come across this part of the code snippet

```
traceTASK_CREATE( pxNewTCB );

            // Update TCB parameters of the task
            //#if ( configUSE_EDFVD_SCHEDULER == 1 )

            prvAddTaskToReadyList( pxNewTCB );
            xReturn = pdPASS;
            portSETUP_TCB( pxNewTCB );
        }
        taskEXIT_CRITICAL();
```

Here is where the function **prvAddTaskToReadyList** is called and the task in currentTCB is added to the ready queue.

Let us briefly have look at the **parameters** of macro function **prvAddTaskToReadyList** now

The `prvAddTaskToReadyList` macro is used to add a task to the appropriate ready list in FreeRTOS. The ready lists contain tasks that are ready to be scheduled for execution based on their priority.

Here's an explanation of the function and its parameters:

- `pxTCB`: It is a pointer to the Task Control Block (TCB) of the task being added to the ready list. The TCB contains information about the task, including its priority and state.

The macro performs the following operations:

1. `traceMOVED_TASK_TO_READY_STATE( pxTCB )`: This is a trace macro that records the event of moving a task to the ready state. It can be used for debugging or profiling purposes.
2. `taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority )`: This macro records the priority of the task being added to the ready list. It can be used for priority-based analysis or statistics.
3. `vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xGenericListItem ) )`: This function inserts the task's list item, represented by `( pxTCB )->xGenericListItem`, at the end of the ready list corresponding to the task's priority. The ready lists are stored in an array called `pxReadyTasksLists`, and each index in the array represents a specific priority level. The `uxPriority` field of the task's TCB determines the index of the ready list to which the task is added.

Overall, the `prvAddTaskToReadyList` macro is responsible for adding a task to the appropriate ready list based on its priority. By doing so, the task becomes eligible for execution by the FreeRTOS scheduler when its priority is the highest among the ready tasks

Here the function **vlistInsertEnd** is similar to **vlistInsert** which we discussed earlier so that we will be able to understand that concept clearly before reaching here.

As we have discussed about insert , we should also discuss about deleting a task and for that we have a function **vTaskDelete** Function

The implementation of the `vTaskDelete` function in FreeRTOS, responsible for deleting a task. Let's go through the code and explain it in detail:

1. `void vTaskDelete(TaskHandle_t xTaskToDelete)`: This function takes a parameter `xTaskToDelete`, which is the handle of the task to be deleted.

2. `TCB_t *pxTCB;`: This declares a pointer to a Task Control Block (TCB) structure. The TCB holds information about the task's state and other attributes.
3. `taskENTER_CRITICAL(); { ... } taskEXIT_CRITICAL();`: These macros are used to enter and exit a critical section to ensure exclusive access to critical data structures while deleting the task. Interrupts are disabled during this section to prevent interference.
4. `pxTCB = prvGetTCBFromHandle(xTaskToDelete);`: This line retrieves the Task Control Block (TCB) associated with the task handle (`xTaskToDelete`) passed to the function. The `prvGetTCBFromHandle` function returns a pointer to the TCB based on the given task handle.
5. Removing the task from the ready list and placing it in the termination list:
    - `uxListRemove(&(pxTCB->xGenericListItem))`: This function removes the task from the ready list. If the return value is zero, it means the task was not in the ready list, and the task's priority needs to be reset.
    - `taskRESET_READY_PRIORITY(pxTCB->uxPriority)`: If the task was removed from the ready list, this macro resets its priority in the corresponding priority bit map.
6. Checking if the task is waiting on an event:
    - `listLIST_ITEM_CONTAINER(&(pxTCB->xEventListItem)) != NULL`: This condition checks if the task is waiting on an event by checking if its `xEventListItem` is associated with any list. If true, the task is removed from the event list.
7. Adding the task to the `xTasksWaitingTermination` list:
    - `vListInsertEnd(&xTasksWaitingTermination, &(pxTCB->xGenericListItem))`: This function inserts the task into the `xTasksWaitingTermination` list, indicating that the task has been marked for termination. The idle task will later release the resources associated with this task.
8. Updating bookkeeping variables:
    - `++uxTasksDeleted`: This increments the `uxTasksDeleted` counter, indicating that a task has been deleted and the idle task needs to check the termination list.
    - `uxTaskNumber++`: This increments the `uxTaskNumber` counter, which is useful for kernel-aware debuggers to detect changes in the task lists.
9. `traceTASK_DELETE(pxTCB)`: This trace macro records the event of task deletion for debugging or profiling purposes.

10. `if (xSchedulerRunning != pdFALSE) { ... }`: This condition checks if the scheduler is currently running.
11. If the task being deleted is the currently running task:
    - `pxTCB == pxCurrentTCB`: This condition checks if the task being deleted is the current task.
    - `configASSERT(uxSchedulerSuspended == 0)`: An assertion is made to ensure that the scheduler is not suspended. If it is, it indicates an error.
    - `portPRE_TASK_DELETE_HOOK(pxTCB, &xYieldPending)`: This hook is primarily for the Windows simulator and performs clean-up operations specific to Windows. It sets `xYieldPending` to indicate that a context switch is required.
    - `portYIELD_WITHIN_API()`: This macro triggers a yield to the scheduler, allowing another task to run.
12. If the task being deleted is not the currently running task:
    - `prvResetNextTaskUnblockTime()`: This function resets the next expected unblock time in case it referred to the task that has just been deleted. It ensures that the scheduler does not wait for an event that will never occur due to task deletion.

This code snippet handles the deletion of a task by removing it from the ready list, adding it to the termination list, updating bookkeeping variables, and taking appropriate actions based on the scheduler state and the task being deleted.

*It's **uxPriority** field of the task's TCB determines the index of the ready list to which the task is added and which the task to be removed so lets briefly have a look at uxPriority in detail.*

uxPriority

The variable `uxTopPriority` represents the highest priority value among all the tasks currently being managed by the FreeRTOS scheduler. It is used to track the highest priority in the system and is updated dynamically as tasks are created, deleted, or their priorities change.

Here's how `uxTopPriority` is typically used:

1. When a task is created:
    - The priority of the newly created task is compared with the current value of `uxTopPriority`.

- If the priority of the new task is higher than `uxTopPriority`, the value of `uxTopPriority` is updated with the new priority value.
- This ensures that `uxTopPriority` always holds the highest priority in the system.

2. When a task is deleted:
   - The priority of the deleted task is compared with `uxTopPriority`.
   - If the deleted task had the highest priority, the scheduler needs to find the next highest priority among the remaining tasks.
   - To determine the new highest priority, the scheduler iterates over the task priority levels starting from the highest priority level (i.e., the highest index in the priority array) and checks if there are any tasks remaining at that priority level.
   - If a task is found at a particular priority level, that priority level becomes the new `uxTopPriority`.
   - If no tasks are found at any priority level, it means there are no tasks remaining, and `uxTopPriority` is reset to its initial value.

By maintaining `uxTopPriority`, the scheduler can quickly identify the highest priority task that needs to run, allowing for efficient task scheduling and context switching

6. Integration of the Ready List with the Scheduler

vTaskSwitchContext Function

Next important thing which plays role is SwitchContext when a particular task completes its work the new task replaces with the current task and it is done by using the function **vTaskSwitchContext**

The function `vTaskSwitchContext` is a crucial part of the context switching mechanism in FreeRTOS. It is responsible for selecting the next task to run and performing the actual context switch between the current running task and the selected task.

Here's an explanation of the `vTaskSwitchContext` function:

1. The function starts by disabling interrupts to ensure that the context switch operation is performed atomically.
2. It checks if the scheduler is suspended (`uxSchedulerSuspended` flag). If the scheduler is suspended, the function exits, as no context switch should occur

while the scheduler is suspended. This allows critical sections of code to execute without being interrupted by task switches.

3. Next, the function determines the task to run using the scheduler's task selection algorithm. The algorithm typically involves considering factors such as task priorities, time slicing, and scheduling policies (e.g., preemptive or cooperative scheduling).
4. If a new task is selected to run (`pxNextTCB`), the function performs the context switch by saving the current task's context and restoring the context of the selected task.
    - The current task's context is saved using the port-specific implementation of the context switch, which typically involves saving the CPU registers and other relevant state information onto the current task's stack.
    - The context of the selected task is restored from its saved state, allowing it to resume execution from where it was last preempted or blocked.
5. After the context switch, the function enables interrupts again to allow other tasks and interrupts to execute.
6. Finally, if the context switch was successful and the current task has not finished its execution, the function returns to the saved state of the current task, allowing it to continue its execution.

The `vTaskSwitchContext` function is typically called from the FreeRTOS scheduler or from specific synchronization primitives (e.g., task blocking on a semaphore). It ensures fair execution of tasks based on their priorities and provides multitasking functionality within the FreeRTOS operating system.

TASK PREEMPTION CONCEPT IN FREE RTOS

In FreeRTOS, task preemption refers to the mechanism by which a higher-priority task interrupts the execution of a lower-priority task. This allows tasks with higher priority to be executed immediately when they become ready, even if a lower-priority task is currently running.

There are several functions and mechanisms in FreeRTOS that are related to task preemption. Here are some of the key ones:

1. Task Priorities: Each task in FreeRTOS is assigned a priority value. The priority determines the relative importance of a task compared to other tasks in the

system. Higher-priority tasks can preempt lower-priority tasks and start executing immediately.

2. Scheduler: The FreeRTOS scheduler is responsible for determining which task should run next. It selects the highest-priority ready task to run. The scheduler is either invoked periodically by a tick interrupt or explicitly by certain API functions.
3. Context Switching: Context switching is the process of saving the current execution context of a task and restoring the context of another task. It allows tasks to be suspended and resumed later without losing their state. Context switching is performed by the scheduler when a higher-priority task becomes ready to run. The context switch function saves the current task's context and restores the context of the selected task.
4. Preemption-Related API Functions: FreeRTOS provides API functions that allow explicit control over task preemption:
    - `vTaskSuspend()`: This function suspends the execution of a task, allowing other tasks of equal or higher priority to run.
    - `vTaskResume()`: This function resumes the execution of a previously suspended task.
    - `vTaskPrioritySet()`: This function changes the priority of a task dynamically. If the new priority is higher than the priority of the currently running task, a context switch may occur.
    - `taskYIELD()`: This function yields the processor voluntarily, allowing other tasks of equal or higher priority to run. It is often used within tasks to provide cooperative multitasking.
    - `taskYIELD_IF_USING_PREEMPTION()`: This function yields the processor if preemption is enabled. It is typically used within critical sections to allow higher-priority tasks to preempt the current task.
    - `vTaskDelay()`: This function blocks the execution of a task for a specified period, allowing other tasks to run in the meantime. If a higher-priority task becomes ready during the delay, a context switch may occur.
    - `vTaskDelayUntil()`: This function blocks the execution of a task until a specific time, allowing other tasks to run. If a higher-priority task becomes ready before the specified time, a context switch may occur.

    These functions provide control over the task scheduling and preemption behavior, allowing developers to fine-tune the task execution and

responsiveness of the system.

By using these functions and mechanisms effectively, developers can implement task preemption and prioritize critical tasks, ensuring that the most important tasks are executed promptly in a real-time system.

Synchronization methods to ensure Thread Safety

FreeRTOS provides several synchronization mechanisms and algorithms to ensure thread safety and prevent race conditions in concurrent environments. Two commonly used mechanisms are semaphores and mutexes.

1. Semaphores:
    - Semaphores are used to control access to a shared resource by multiple tasks. They can be binary (0 or 1) or counting (integer value).
    - Binary semaphores are often used for mutual exclusion, where only one task can access a resource at a time. They are initialized with a value of 1 and are typically used to protect critical sections of code.
    - Counting semaphores can allow multiple tasks to access a resource simultaneously, up to a defined limit. They are useful for resource pooling and synchronization between producer and consumer tasks.
    - FreeRTOS provides functions like `xSemaphoreCreateBinary()`, `xSemaphoreTake()`, and `xSemaphoreGive()` to create, acquire, and release semaphores, respectively.
2. Mutexes:
    - Mutexes (short for mutual exclusions) are similar to binary semaphores and are used for protecting shared resources.
    - Mutexes ensure that only one task can acquire the mutex at a time, providing exclusive access to the shared resource.
    - If a task attempts to acquire a mutex that is already held by another task, it will block until the mutex becomes available.
    - FreeRTOS provides functions like `xSemaphoreCreateMutex()`, `xSemaphoreTake()`, and `xSemaphoreGive()` for creating, acquiring, and releasing mutexes, respectively.

These synchronization mechanisms help prevent race conditions and ensure thread safety by providing mutually exclusive access to shared resources. They allow tasks to coordinate their actions, synchronize their execution, and avoid conflicts. By

properly utilizing semaphores and mutexes, developers can ensure that critical sections of code are protected, shared resources are accessed safely, and concurrent tasks can cooperate without interfering with each other.

In addition to semaphores and mutexes, FreeRTOS also provides other synchronization primitives like queues, event groups, and task notifications, which further enhance the capabilities of concurrent programming and thread safety in real-time systems.

Conclusion

The analysis of the FreeRTOS source code reveals the significance of the Ready List in achieving efficient task scheduling and management. The Ready List is a data structure used by the scheduler to maintain a list of tasks that are ready to run. It is organized into priority levels, with each priority level having its own list.

The Ready List plays a crucial role in task scheduling. When the scheduler determines that it is time to switch to a new task, it selects the highest-priority task from the Ready List. This allows tasks with higher priority to preempt lower-priority tasks and start executing immediately. By prioritizing tasks based on their importance, the Ready List ensures that critical tasks are given precedence, leading to efficient task execution.

The implementation of the Ready List involves various functions and mechanisms. Tasks are added to the Ready List using the `vTaskPlaceOnEventList()` function, which inserts a task into the appropriate priority level list. The `prvAddTaskToReadyList()` function is used to add a task to the end of the list for a specific priority level. These functions ensure that tasks are inserted into the Ready List in the correct order.

The Ready List is also utilized in other operations such as task suspension, resumption, and deletion. When a task is suspended, it is removed from the Ready List, preventing it from being scheduled. Upon resumption, the task is reinserted into the appropriate priority level list. When a task is deleted, it is removed from the Ready List and placed in the termination list. This ensures that the scheduler stops scheduling the task and allows for proper memory deallocation.

Studying the FreeRTOS source code provides valuable insights into the inner workings of a real-time operating system. The implementation of the Ready List highlights the importance of prioritizing tasks based on their urgency and importance. The Ready List ensures that critical tasks are executed promptly, leading to efficient task scheduling and responsiveness in real-time systems. Additionally, the analysis of functions and mechanisms related to task preemption

demonstrates the flexibility and control provided by FreeRTOS for managing task execution.

In conclusion, the Ready List is a fundamental component of FreeRTOS that enables efficient task scheduling and management. It plays a vital role in prioritizing tasks and allowing critical tasks to preempt lower-priority tasks. By understanding and utilizing the Ready List and related functions effectively, developers can design and implement real-time systems that meet their performance requirements.

1. **Function:** `addTaskToReadyList(pxTCB)`

Time complexity: O(1)

Explanation: The time complexity of this function depends on the time complexity of the `vListInsertEnd` function, which is typically O(1) or constant time. Since the `addTaskToReadyList` macro inserts the task at the end of the appropriate priority list, it directly calls `vListInsertEnd` to perform the insertion. Inserting an item at the end of a linked list with a constant number of elements takes constant time, resulting in O(1) time complexity for `addTaskToReadyList`.

Space complexity: O(1)

Explanation: The space complexity of `addTaskToReadyList` is constant as it does not allocate any additional memory or data structures. It only manipulates the existing task lists and records the task's priority, which does not depend on the input size or the number of tasks in the system.

2. **Function:** `prvRemoveTaskFromReadyList(pxTCB)`

Time complexity: O(1)

Explanation: The time complexity of `prvRemoveTaskFromReadyList` depends on the time complexity of the `vListRemove` function, which is typically O(1) or constant time. The `vListRemove` function unlinks the task from the appropriate priority list by updating the pointers in the linked list structure. Since unlinking an item from a linked list with a constant number of elements takes constant time, the overall time complexity of `prvRemoveTaskFromReadyList` is O(1).

Space complexity: O(1)

Explanation: The space complexity of `prvRemoveTaskFromReadyList` is constant as it does not allocate any additional memory or data structures. It only manipulates the

existing linked list structure to remove the task from the ready list, which does not depend on the input size or the number of tasks in the system.