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.

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.

When a task or thread becomes ready to run, it is added to the ready list by the scheduler. The scheduler is responsible for determining which task should be executed next based on the scheduling policy employed by the RTOS. The ready list allows the scheduler to quickly identify the next task to be executed without searching through all the tasks in the system.

MACROS:

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. By defining these macros in `portmacro.h`, the RTOS can ensure that the correct data types are used throughout its codebase, regardless of the underlying system's specific data type definitions.

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. The ready priority represents the priority of a task in the context of task scheduling and determines the order in which tasks are selected for execution when there are multiple ready tasks.

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. This step ensures that the system maintains an accurate record of the task's priority.

privileged_data and vtaskdelete

he `PRIVILEGED_DATA` macro is used to define variables or data structures that require privileged access or special memory attributes.

The behavior of the `vTaskDelete` 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.

configMax_Priorities

`configMAX_PRIORITIES` is a configuration option that determines the maximum number of priority levels that can be assigned to tasks in the RTOS. It allows for fine-grained control over task prioritization and enables developers to manage task scheduling based on their relative importance and urgency.

mtCOVERAGE_TEST_MARKER()

the **`mtCOVERAGE_TEST_MARKER`** macro in FreeRTOS is a placeholder that can be used to mark specific lines of code for code coverage testing. Its actual implementation and usage may vary depending on the specific version, port, and code coverage tool being utilized.

configAssert

The **`configASSERT`** macro is a debugging macro used in FreeRTOS to perform assertions or checks during runtime. It is commonly used to catch and handle conditions that should never occur or indicate a programming error.

vTaskPlaceOnEventList

The provided code is a part of FreeRTOS and specifically shows the implementation of the `vTaskPlaceOnEventList` function. Let's break down the code and explain its functionality:

1. 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.
2. 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.

1. A comment states that this function must be called with either interrupts disabled or the scheduler suspended, and the queue being accessed locked. This indicates that the function requires a specific execution context to ensure data integrity and synchronization.
2. 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, ensuring that the highest priority task is the first to be woken when the event occurs. The queue that contains the event list is locked to prevent simultaneous access from interrupts.
3. The task is removed from the ready list using the `uxListRemove` function. Since the same list item is used for both the ready list and the blocked list, the task needs to be removed from the ready list before adding it to the blocked list. If the return value of `uxListRemove` is 0, it means that the current task was in a ready list, and the `portRESET_READY_PRIORITY` macro is called to adjust the ready priority lists. If the return value is non-zero, it means that the current task was not in a ready list, and the `mtCOVERAGE_TEST_MARKER` macro is called (possibly for coverage testing purposes).
4. Depending on the configuration option `INCLUDE_vTaskSuspend`, different code blocks are executed. If `xTicksToWait` is `portMAX_DELAY`, indicating an indefinite wait, the task is added to the suspended task list using `vListInsertEnd`. Otherwise, the time at which the task should be woken is calculated by adding `xTicksToWait` to the current tick count (`xTickCount`). This result is passed to the `prvAddCurrentTaskToDelayedList` function, which adds the current task to the appropriate delayed task list based on the calculated wake time.

Overall, the `vTaskPlaceOnEventList` function is responsible for inserting the current task's event list item into the specified event list, removing the task from the ready list if it was in one, and either adding the task to the suspended task list or the appropriate delayed task list based on the value of `xTicksToWait`.

own words:-

3 task lists:-

→blocked

→suspended

→delayed

ticks is basically a unit of time which determines how long a task should wait before getting removed from the event list

event list is a list of tasks placed in a priority which would be invoked according to a particular event

the queue that contains the event list is locked to avoid simultaneous access

config_assert is used to determine that the event_list is not null

→ Tasks in the blocked list are waiting for specific events or resources to become available, such as waiting for a semaphore, a queue, or a time delay to elapse.

→ Blocked tasks are not considered for execution until the conditions they are waiting for are met, and they are typically removed from the ready list.(uxListRemove())

and inserted into event list using(vListinsert())

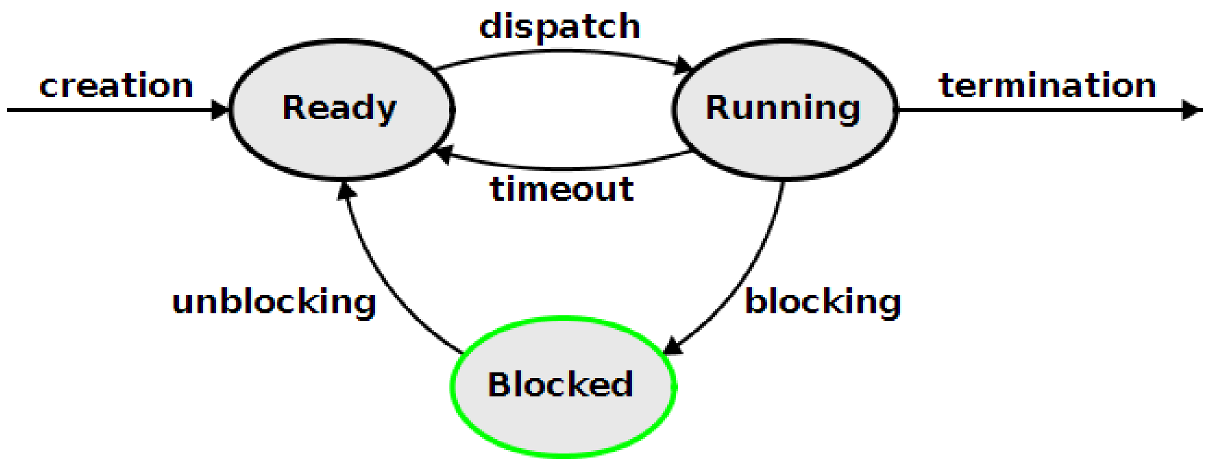
→ Once the blocking condition is satisfied, the task is moved from the blocked list back to the ready list, making it eligible for execution by the scheduler.

suspended and delayed task lists are different from blocked task list because they do not depend on some event but are dependent on the ticks.(it will be blocked indefinitely)

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

diagram:



xTaskRemoveFromEventList

The provided code shows the implementation of the `xTaskRemoveFromEventList` function in FreeRTOS. Let's break down the code and explain its functionality:

Here's a detailed explanation of the code:

1. The function `xTaskRemoveFromEventList` takes a pointer to an event list (`pxEventList`) as a parameter and returns a `BaseType_t` value.
2. The `pxUnblockedTCB` variable is declared as a pointer to the `TCB_t` (Task Control Block) structure, which will hold the TCB of the unblocked task.
3. The `xReturn` variable is declared to store the return value indicating whether the unblocked task has a higher priority than the calling task.
4. The comment emphasizes that this function must be called from a critical section to ensure data integrity and synchronization. It mentions that it can also be called from a critical section within an ISR (Interrupt Service Routine).
5. The comment explains that the event list is sorted in priority order, allowing the removal of the first task in the list since it is known to be the highest priority. It further states that if an event is for a locked queue, this function will not be called as the lock count on the queue will be modified instead. This guarantees exclusive access to the event list in this function. The comment assumes that a check has already been made to ensure that `pxEventList` is not empty.

6. 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.
7. The `configASSERT` macro is called to assert that the `pxUnblockedTCB` is not NULL, ensuring that a valid TCB is obtained.
8. The `uxListRemove` function is used to remove the `xEventListItem` of the `pxUnblockedTCB` from the event list.
9. 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`.
10. 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.
11. The code checks if the unblocked task's priority is higher than the calling task's priority. If so, `xReturn` is set to `pdTRUE`, indicating that the calling task should force a context switch. Additionally, the `xYieldPending` flag is set to `pdTRUE` to mark that a yield is pending.
12. If the unblocked task's priority is not higher than the calling task's priority, `xReturn` is set to `pdFALSE`.
13. 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.
14. Finally, the function returns the value stored in `xReturn`, indicating whether a context switch should occur.