# Analysis of FreeRTOS Source Code

# Analysing the importance of ready list in efficient EXECUTING OF TASKS

GROUP NAME : **DSA DYNAMOS**

GROUP MEMBERS :

|  |  |
| --- | --- |
| NAME | ROLL NUMBER |
| DASARI SAI SAMRAT | CB.EN.U4CSE21415 |
| GUDIVADA ROHAN LAL | CB.EN.U4CSE21420 |
| KARRI AMMI REDDY | CB.EN.U4CSE21428 |
| SAGIRAJU HARINADHA RAJU | CB.EN.U4CSE21451 |
| VINAYAKAN V S | CB.EN.U4CSE21467 |

Introduction:

FreeRTOS is a real-time operating system kernel for embedded devices that has been ported to 35 Microcontroller platforms.It is designed to be small and simple also it uses C language for it’s implementation to make it easy to port and maintain. Using an RTOS means you can run multiple tasks concurrently, bringing in the basic connectivity, privacy, security, and so on as and when you need them. An RTOS allows you to create an optimized solution for the specific requirements of your project. FreeRTOS is widely used in various industries, including automotive, aerospace, industrial automation, medical devices, and more.

The purpose of this report is to analyse the implementation of the Ready List in the FreeRTOS scheduler. The Ready List is crucial within the scheduler as it keeps track of processes which are ready for execution. The deeper understanding about this Ready List helps us to get a clear picture on FreeRTOS implementation and how the tasks are managed within.

An overview of the structure of the report:

1. Introduction
2. Overview of FreeRTOS Scheduler
3. Understanding the Ready List
4. Analyzing the Implementation of the Ready List
5. Detailed Examination of Ready List Operations
6. Integration of the Ready List with the Scheduler
7. Conclusion

Overview of FreeRTOS Scheduler :

The FreeRTOS scheduler is responsible for managing and scheduling tasks in Real Time Operating System based on priorities assigned to each task. It ensures that tasks are run efficiently and successfully based on resources available. The higher level priority task is considered to be executed first.

The Ready List is a crucial data structure. It is a Priority Based Linked List which holds the tasks according to its priority and also coordinates the efficient task management. The Ready List enable efficient task switching and scheduling. When the current task is preempted or completes its execution, the scheduler looks at the Ready List to determine the highest-priority task that is ready to run. This task is then selected for execution, and the scheduler context switches to that task.

The Ready List allows the scheduler to quickly identify and select the most critical task to ensure that the system meets its real-time requirements. It ensures that high-priority tasks are promptly executed and that low-priority tasks do not starve for CPU time.

The Ready List is dynamic and can change as tasks are created, deleted, or their priorities are modified during runtime. The scheduler continuously scans the Ready List to maintain an up-to-date view of the system's task readiness, allowing it to make optimal scheduling decisions based on the available tasks and their priorities.

The Ready List allows the scheduler to quickly identify the highest-priority task that should be scheduled for execution. When a task becomes ready to run, such as when it completes a waiting state or is newly created, it is added to the Ready List. The scheduler then selects the task with the highest priority from the Ready List and switches the processor's execution context to that task.

In Conclusion The Ready List is the one which plays a vital role in the whole task scheduling and task context switching and also maintaining a good timely atmosphere for efficient running of the processes according to priorities assigned , and contributing to the overall real-time responsiveness of the system.

Understanding the Ready List:

1. States of a Task in FreeRTOS:

• Ready : Tasks that are eligible to run but waiting for the CPU.

• Blocked : Tasks that are waiting for an event or a resource.

• Running : The currently executing task.

2. The Ready List:

• The Ready List is a data structure used by the task scheduler to maintain the details of tasks that are in the ready state.

• The purpose of the Ready List is to efficiently organize and prioritize the tasks for scheduling in the appropriate order.

3. Algorithm Overview:

• The algorithm uses a preemptive scheduling policy where tasks with higher priorities preempt lower-priority tasks and lower priority task will be added to Ready list.

4. Ready List Management:

• The Ready List can be implemented as a priority based linked list, where tasks are sorted based on their priority.

• Each task in the Ready List is represented by a data structure that includes information such as its priority, state, and context.

5. Task Creation:

• When a task is created, it is initially placed in the Ready List with its initial state set as ready.

• The tasks will be placed in the Ready List based on their priority order.

6. Task Execution:

• The scheduler selects the task with the highest priority from the Ready List for execution.

• The selected task is marked as running, and its context is loaded onto the CPU for execution.

8. Task Blocking:

• If a task needs to wait for a resource, it enters into a blocked state.

• Blocked tasks are removed from the Ready List and put into a separate data structure.

9. Task Unblock:

• When the resource becomes available, a blocked task is unblocked. Then the task is moved from the blocked state back to the ready state and inserted into the Ready List based on its priority.

10. Task Preemption:

• If a higher-priority task becomes ready while a lower-priority task is running, the higher-priority task preempts the lower-priority task.

• The running task is moved back to the Ready List, and the higher-priority task takes its place on the CPU for execution.

The Ready List plays a crucial role in task scheduling by efficiently managing the states of tasks. It allows the scheduler to prioritize and select tasks based on their priorities, ensuring that higher-priority tasks get executed before lower-priority tasks. By maintaining the Ready List, the scheduler can quickly identify ready tasks and transition them to the running state while efficiently managing blocked tasks in separate data structures. This approach enables efficient multitasking and resource utilization in FreeRTOS.

Analyzing the Implementation of the Ready List :

The Ready List is commonly implemented in FreeRTOS utilising a Priority-based linked list data structure. Each priority level has its own linked list, and jobs are organised within these lists according to their priority levels. Several factors, including performance and memory utilisation considerations, influence the choice of a priority-based linked list data structure for the Ready List in FreeRTOS. The following are the benefits of this approach:

1. Effective Task Selection: The linked list structure enables effective task selection of the task with the highest priority. Since tasks are listed in order of importance, the item with the highest priority is always at the top of the list. This task is simple for the scheduler to access, which cuts down on the time needed for task selection. This method offers quick and deterministic scheduling, which is essential for real-time applications.

2. Dynamic Priority Changes: Task priorities can vary on the fly,thanks to the linked list implementation. A task can be efficiently relocated inside the linked list to the appropriate location based on the new priority when its priority changes. Fine-grained scheduling control and the avoidance of priority inversion are made possible by this flexibility.

3. Constant-Time Insertion and Removal: Using a linked list, adding or removing a job from the Ready List is a constant-time operation. Based on its priority, a job can be added to the relevant linked list as soon as it is complete. Similar to that, a job can be swiftly purged from the list if it blocks or becomes unready. These activities provide effective task state transition management since they are independent of job quantity or priority levels.

4. Memory Efficiency: The linked list structure has good memory performance. In order to hold the necessary pointers and metadata, each job in the Ready List needs a modest amount of extra memory. For embedded systems with limited resources, the memory overhead is acceptable because it is inversely related to the number of jobs in the system rather than the number of priority levels.

5. Scalability: The linked list approach scales effectively with different numbers of tasks and priority levels. The linked list can be readily changed without adding a lot of overhead as tasks are added or removed. This scalability enables effective handling of a large number of jobs and makes it adaptable to various system configurations.

6. Task Delaying Support: The linked list implementation effectively supports task delaying capabilities. A job can be transferred to a different list connected to the delayed tasks when it is delayed. The scheduler can therefore concentrate on the tasks that are ready, delaying processing of activities that are postponed until their delay time has passed.

Using a priority-based linked list data structure for the Ready List in FreeRTOS brings several benefits to the system's performance, responsiveness, and resource efficiency. It enables efficient task selection, allowing the scheduler to quickly identify the highest-priority task. The structure also supports dynamic changes in task priorities, facilitating fine-grained control over scheduling. Insertion and removal operations are constant-time, ensuring efficient task management. Additionally, the linked list implementation optimizes memory usage, scales well with varying task and priority levels, and supports task delaying functionality. These advantages collectively enhance the overall performance, responsiveness, and resource utilization of the FreeRTOS scheduler.

Detailed Examination of Ready List Operations :

Function 1: vTaskSwitchContext() :

Summary about the Function: This Functions starts with checking if the scheduler is suspended or it is active by (**uxSchedulerSuspended != pdFALSE**). If scheduler is suspended then it sets the value of **xYieldPending** to TRUE. By setting value to true it means that context switching is awaiting and not to be performed as the scheduler is suspended. If scheduler is not suspended then it sets the value of **xYieldPending** to False which tells that there is no context switch awaiting.Then **traceTASK\_SWITCHED\_OUT()** this is called which keeps track of tasks that has been switched out that is removed out from CPU. Then if configuration is started in thus **configGENERATE\_RUN\_TIME\_STATS** is set to 1 so that it gets information of the runtime of current task using **portALT\_GET\_RUN\_TIME\_COUNTER\_VALUE.** The runtime is calculated by subtracting the previously recorded **ulTaskSwitchedInTime** from the current **ulTotalRunTime** and adds it to the task's **ulRunTimeCounter**. Then the code performs the stack overflow checks to analyze the stack usage of the current task and take appropriate action if a stack overflow is detected through the functions **taskFIRST\_CHECK\_FOR\_STACK\_OVERFLOW()** and **taskSECOND\_CHECK\_FOR\_STACK\_OVERFLOW().** Then **taskSELECT\_HIGHEST\_PRIORITY\_TASK()** function is called to know Highest priority task and also determines task with greater priority which is ready to run. **This is a selection process.** After selection of the process another function is called which is **traceTASK\_SWITCHED\_IN()** where it tracks when a new task is scheduled to CPU(Switching). Finally **configUSE\_NEWLIB\_REENTRANT** is set to 1 so that switches Newlib's **\_impure\_ptr** variable to point to the **\_reent** structure specific to the current task. This ensures that each task has its own separate Newlib reentrant structure, providing thread-safety when using Newlib functions. In summary, the **vTaskSwitchContext()** function in FreeRTOS handles the context switching between tasks. It determines whether a context switch is allowed based on the scheduler's state, performs runtime statistics calculations (if enabled), checks for stack overflow, selects the next task to run, and handles any necessary bookkeeping or configuration changes related to context switching.­ Time complexity analysis for context switching is that there is a comparison process for finding out the highest priority task in this if the index where there is highest priority has ready tasks then it is O(1) otherwise it has an iteration to perform to get the highest priority task in the worst case if there are m priorities then time taken is O(m). Finally for the whole function this selection effects the time complexity and it is O(m).

Function 2: prvAddTaskToReadyList () :

This starts by calling the function **traceMOVED\_TASK\_TO\_READY\_STATE()** which records when a task is moved to the ready state. Then **taskRECORD\_READY\_PRIORITY()** is called in which it has the parameter **uxPriority** which represents of a task that is being added to Ready List. It checks if the **uxPriority** is greater than the current highest priority(**uxTopReadyPriority**). If the **uxPriority** is greater then it updates the value of **uxTopReadyPriority** to new highest priority. This ensures that the system keeps track of the highest priority among the tasks that are ready to run.

Then another **vListInsertEnd()** is called which inserts the task’s **xGenericListItem** (a member of the task's control block structure **pxTCB**) to end of Ready List corresponding to the task's priority level. The ready list is an array of linked lists (**pxReadyTasksLists**) indexed by priority. Each priority level has its own linked list to hold the tasks ready to run at that priority level. By using **vListInsertEnd()**, the task is added to the end of its priority level list.

In summary, **prvAddTaskToReadyList()** is used to add a task to Ready List by checking its priority and recording it . Also it is used when a task transitions from a blocked or suspended state to the ready state, indicating that it is ready to be scheduled and executed by the FreeRTOS scheduler.

Time complexity depends on how the Ready List is implemented if it is done using Linked list or priority queue then for adding task to a Ready List the operation takes a time complexity of O(1) which will be the most efficient way of implementing the Ready List.

Function 3: uxListRemove() :

This function begins with obtaining the pointer to the list (**pxList**) that contains the item (**pxItemToRemove**). This is achieved by accessing the **pvContainer** member of the **pxItemToRemove** structure, which points to the container list. The following lines update the next and previous neighbour in the list to remove **pxItemToRemove** from list. By changing next and previous pointers we achieve unlinking the item from the list effectively. Then it checks the **pxIndex** pointer if it is pointing towards **pxItemToRemove** if so it updates the **pxIndex** to point previous item in the list so that the index will be valid all the time. **mtCOVERAGE\_TEST\_MARKER()** , it serves as a placeholder or marker that can be used during coverage analysis. It indicates a specific point in the code where the coverage tool can log the fact that this line has been reached. By placing the **mtCOVERAGE\_TEST\_MARKER()** at strategic points in the code, developers can track and analyze the coverage data generated during testing to identify untested or low-coverage areas. **pvContainer** member of the **pxItemToRemove** is set to NULL which disconnect it from the list. **uxNumberOfItems** is decremented by one reflecting that one item is removed from the list. In Summary, uxListRemove() is responsible for removing an item in the list also updates the neighbours and updates the items in the list. This function is used for managing lists of items efficiently, such as ready lists or event queues.

Time complexity depends on how the Ready List is implemented if it is done using Linked list or priority queue then for adding task to a Ready List the operation takes a time complexity of O(1) which will be the most efficient way of implementing the Ready List.

4: uxTopReadyPriority :

This is a variable that keeps track of highest priority task among the tasks which are ready to run. It is used by the Scheduler to determine the next task to be scheduled efficiently. Whenever a task is added or removed from the ready list, the scheduler updates the **uxTopReadyPriority** variable accordingly. If a task with a higher priority becomes ready, **uxTopReadyPriority** is updated to reflect this change. Similarly, if the task with the highest priority is removed from the ready list, the scheduler checks for the new highest priority among the remaining tasks. Maintaining **uxTopReadyPriority** allows the scheduler to quickly identify the highest-priority task without the need for exhaustive searching through all the ready tasks. It is used multiple times in the code whenever there is a need of finding Highest Priority Task.

5: pxCurrentTCB :

This is a pointer which points to Task Control Block(TCB) of currently executing task. TCB contains each and every detail of the task allowing kernel to manage and schedule tasks efficiently. The TCB holds various pieces of information related to the task, including its stack pointer, state, priority, runtime statistics, and more. It serves as a central data structure that the kernel references to track and control the execution of tasks.

The pxCurrentTCB provides convenient way for kernel to access TCB. By dereferencing this pointer, the kernel can access and modify the task's attributes and state, perform context switches, update runtime statistics, and make scheduling decisions based on the task's priority and other factors.

This pointer gets updated by scheduler whenever there is a context switching. When there is task scheduled then the scheduler makes this Pointer to point towards this task in TCB by this the kernel gets easy to collect or update any information of that particular task being performed.

Integration of the Ready List with the Scheduler:

In FreeRTOS, the Ready List interacts with other scheduler elements like context switching and task preemption to provide efficient task management and scheduling. To guarantee thread safety and avoid race scenarios, synchronisation techniques and algorithms are used. Let's go over these points in greater detail:

1. Interaction with Context Switching: The process of saving one task's current execution context and restoring the execution context of another task is known as context switching. Due to the Ready List's ability to offer the scheduler a selection of ready jobs, it plays a significant part in context switching. The scheduler checks the Ready List to find the next task to run when a task is interrupted or willingly yields the CPU. The selected task's context is restored and the current task's context is saved, enabling seamless continuation of its execution. In order to facilitate effective task scheduling, the Ready List makes sure that jobs in the ready state are easily accessible for context switching.

2. Interaction with Task Preemption: Job preemption is the process through which a task with a higher priority can halt the execution of a job with a lower priority. Based on its priority, a task should be preempted or not according to the Ready List. The Ready List enables task preemption when a task with a higher priority becomes ready and the task that is currently being executed has a lower priority. Using the data from the Ready List, the scheduler pauses the lower-priority activity, preserves its context, and switches to the higher-priority task. This enables the system to give priority to important processes and make sure they get CPU time as needed.

Algorithms and Synchronisation Mechanisms:

To guarantee thread safety and avoid race circumstances in multi-threaded settings, synchronisation methods are crucial. In order to guarantee the reliability of the Ready List and its operations, FreeRTOS uses a variety of synchronisation techniques and algorithms. Several typical mechanisms include:

- Mutexes: Mutexes are used to enable mutual exclusion, limiting the number of tasks that can simultaneously access the Ready List. A job must acquire the mutex before it can conduct an operation on the Ready List, preventing another task from concurrently changing the list. Race circumstances are avoided, and thread safety is guaranteed.

- Atomic Operations: On shared data structures, atomic operations are utilised for low-level, indivisible operations. To ensure that these activities are carried out atomically and without interruption in the context of the Ready List, atomic operations may be used when updating task state or changing pointers.

- Interrupt Disable/Enable: FreeRTOS may temporarily disable interrupts when carrying out crucial Ready List activities. The scheduler ensures that no interrupt handler or higher priority task can preempt the operation by suppressing interrupts, hence ensuring the consistency and accuracy of the Ready List.

The integrity of the Ready List is preserved by these synchronisation techniques and algorithms, race circumstances are avoided, and the scheduler functions appropriately and dependably in a multi-threaded environment.

In conclusion, the Ready List interacts with other scheduler elements, like context switching and task preemption, to facilitate effective task management and scheduling. To maintain thread safety and avoid race circumstances when gaining access to and making changes to the Ready List, synchronisation mechanisms including mutexes, atomic operations, and interrupt disable/enable are utilised. In managing multiple activities at once, the scheduler is more reliable and effective overall because to these interactions and synchronisation methods.

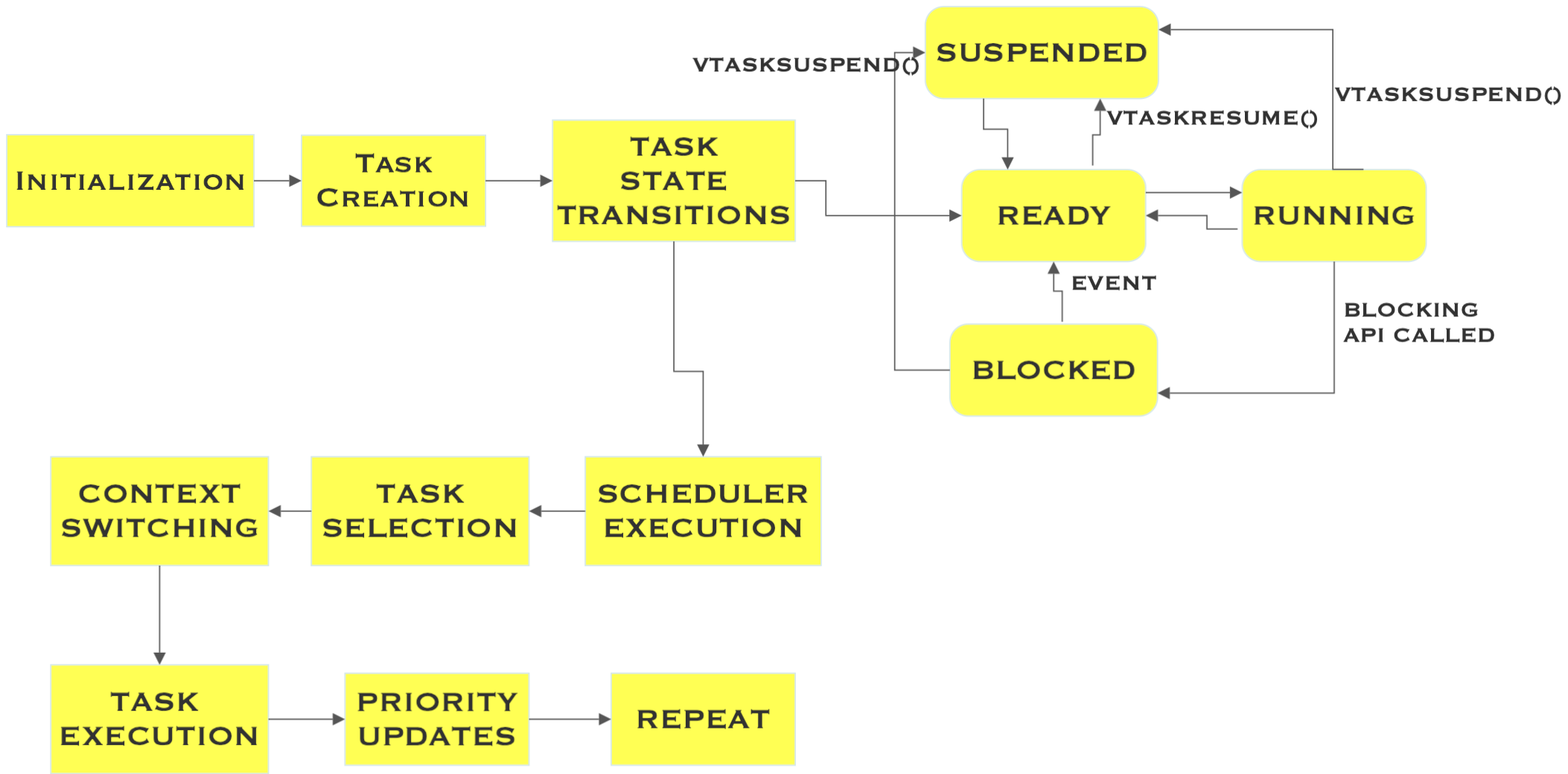
Conclusion:

The analysis of implementation of Ready List in FreeRTOS show that Ready List is a data structure which organizes tasks based on priorities and also tasks are added to this when they are ready to run and removed when the task’s state are suspended, blocked or deleted.

Ready List interacts with all the important components of scheduling like Context switching,Task pre-emption. It serves as central data structure which ensures a proper efficient executing of tasks without any race conditions and also ensures thread safety.

Studying FreeRTOS source code provides a clear understanding of importance of efficient task scheduling and proper use of data structures in executing and managing tasks efficiently.

# FLOWCHART OF STAGES A TASK IS EXECUTED :



REFERENCES :

<https://chat.openai.com>

<https://www.freertos.org>

<https://www.wikipedia.org>