RA3 RTOS

☎ +2001154784667
Cairo, Egypt
engaliyasser7@gmail.com

Table of Contents

Overview

RA3 RTOS is an open-source real-time operating system designed for embedded systems, providing a robust foundation for developing applications across various processor architectures. Currently, RA3 RTOS supports the ARM Cortex-M3 processor, making it an ideal choice for developers looking to create efficient and responsive applications in this environment.

The kernel of RA3 RTOS employs a preemptive priority round-robin scheduling algorithm, ensuring that tasks are managed effectively to meet real-time requirements. This approach allows for smooth task switching and high responsiveness, which are critical in embedded systems.

As part of our commitment to continuous improvement and community involvement, we invite developers to join us in enhancing RA3 RTOS. Future plans for the project include the integration of additional scheduling algorithms to cater to a wider range of application needs, as well as the development of an OSEK-compliant version, expanding its applicability in automotive and industrial automation contexts.

Whether you are a seasoned developer or new to embedded systems, RA3 RTOS provides the tools and flexibility needed to bring your projects to life. Join us in shaping the future of this Egyptian RTOS and contribute to its growth and success.

Meet the Team

Right now, the RA3 RTOS team consists solely of me, **Ali Yasser Ali Abdallah**. I am a passionate computer engineering student driven to contribute to the embedded systems industry by developing RA3 RTOS as an open-source, community-driven project. Through RA3, I aim to provide accessible, efficient solutions for embedded applications and offer a platform where developers can learn, innovate, and build together.

This team is open for expansion! If you share a vision for advancing embedded systems through open-source contributions, we welcome you to join and help enhance RA3 RTOS.

Getting Started

Installation Instructions

1. **Download the Latest Version**: Visit the Releases page to download the latest release of RA3 RTOS as a ZIP file.
2. **Unzip the File**: Extract the contents of the downloaded ZIP file into your project directory.
3. **Add to Project**: Link the RA3 RTOS files into your project setup.

Setting up the Development Environment

- **Compiler Requirements**: RA3 RTOS is currently built to support ARM Cortex-M3 processors. Use a compatible compiler (e.g., GCC for ARM).
- **Required Tools**: Ensure your development environment includes debugging tools, a JTAG or SWD interface, and any additional peripherals needed for your specific application.

Quick Start Guide

1. **Initialize the OS**: Call `OS_Init()` to initialize RA3 RTOS before creating tasks or semaphores.
2. **Create Tasks**: Use `OS_CreateTask()` to define and initialize tasks, setting each task's function, priority, and stack size.
3. **Start the OS**: Call `OS_StartOS()` to start the scheduler and allow tasks to begin running.

For full documentation, refer to the RA3 RTOS Guide.

Core Concepts

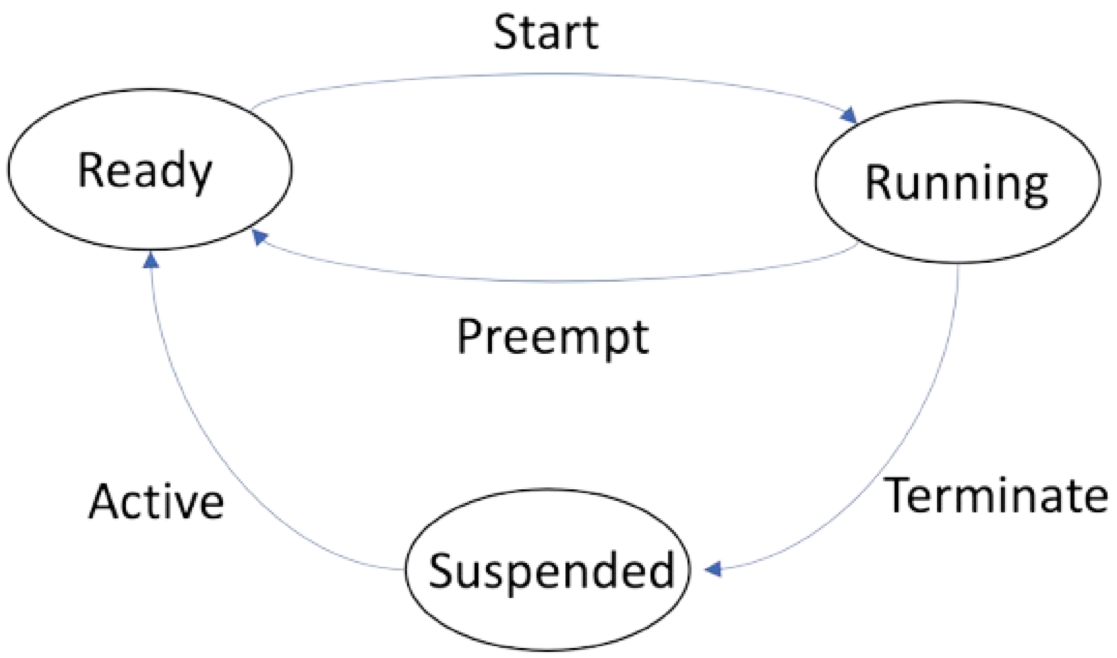
Understanding the foundational concepts of **RA3 RTOS** is essential for effectively utilizing its features and developing applications. This section outlines the key components and mechanisms that drive the operating system.

1. Task Management

At the heart of RA3 RTOS is its task management system, which allows for the creation, scheduling, and control of tasks. The **Task Control Block (TCB)** is a crucial data structure that defines each task within the system. It contains essential information, including:

- **Priority**: Determines the scheduling order of tasks, with lower values indicating higher priority.
- **Task Name**: A descriptive identifier for the task, aiding in debugging and management.
- **Stack Size**: Specifies the memory allocated for the task’s stack, which is essential for local variable storage and function calls.
- **Task Function Pointer**: A reference to the function that the task will execute.

Tasks can exist in various states, including **Ready**, **Running** and **Suspended**, which are managed by the scheduler to ensure efficient execution.



2. Memory Management

RA3 RTOS employs a flexible memory management strategy to allocate stacks for tasks dynamically. This approach allows for efficient use of memory resources, accommodating varying stack sizes based on the needs of different tasks. Key components include:

- **Main Stack**: A dedicated stack used by the operating system for handling interrupts and task switching.
- **Task Stacks**: Individual stacks allocated for each task, enabling isolated execution environments.

Efficient memory management is crucial for embedded systems, where resources are often limited.

3. Inter-task Communication

Effective communication between tasks is vital for synchronizing operations and sharing resources. RA3 RTOS provides several mechanisms for inter-task communication, including:

- **Semaphores**: Used to manage access to shared resources. A semaphore can signal when a resource is available or when a task should wait.
- **Mutexes**: Similar to semaphores, mutexes provide mutual exclusion for tasks needing to access a shared resource, ensuring that only one task can hold the lock at a time.

These mechanisms facilitate coordinated interactions between tasks, enhancing the overall functionality of applications.

4. Scheduling

RA3 RTOS employs a **preemptive priority round-robin** scheduling algorithm, which dynamically allocates CPU time based on task priorities. The scheduler continually evaluates task states and prioritizes execution based on their assigned priority levels. This approach allows high-priority tasks to preempt lower-priority tasks, ensuring timely execution of critical operations.

As part of our future roadmap, we aim to introduce additional scheduling algorithms and an OSEK-compliant version to broaden the applicability of RA3 RTOS across various domains.

API Reference

This section provides a detailed reference for the API functions available in **RA3 RTOS**. Each function is described with its purpose, parameters, return values, and an example of usage.

Task Management APIs

API Function: OS_CreateTask

Header File: `task.h`

```
OS_ErrorStatus OS_CreateTask(OS_TCB* Task);
```

Creates a new task and adds it to the list of tasks that are ready to run. Each task requires RAM that is used to hold the task state and is utilized by the task as its stack. The task's stack size and function are specified in the `OS_TCB` structure.

Parameters:

- **`Task`**
  Pointer to an instance of the `OS_TCB` structure that defines the task's properties, including its priority, name, stack size, task function, and auto-start option.

Returns:

- **`OS_OK`**
  If the task was created successfully.
- **`TASK_CREATION_ERROR`**
  If there was an error during task creation.

Example Usage:

```
// Task function to be created.
void MyTask(void) {
    for(;;) {
        // Task code goes here.
    }
}

void CreateMyTask(void) {
    OS_TCB myTask;
    myTask.Priority = 1;               // Task priority
    strncpy((char*)myTask.TaskName, "MyTask", sizeof(myTask.TaskName)); //
Task name
    myTask.StackSize = 100;            // Stack size in words
    myTask.func = MyTask;              // Task function

    OS_ErrorStatus status = OS_CreateTask(&myTask); // Create the task
    if (status == OS_OK) {
        // Task was created successfully
    }
}
```

API Function: OS_ActivateTask

Header File: `task.h`

```
OS_ErrorStatus OS_ActivateTask(OS_TCB* Task);
```

Activates a previously created task and makes it ready to run. The task's state is changed to ready, allowing it to be scheduled by the RTOS.

Parameters:

- **`Task`**
  Pointer to the `OS_TCB` structure of the task to be activated.

Returns:

- **`OS_OK`**
  If the task was activated successfully.
- **`TASK_CREATION_ERROR`**
  If the task could not be activated.

Example Usage:

```
void ActivateMyTask(void) {
    OS_ErrorStatus status = OS_ActivateTask(&myTask); // Activate the task
    if (status == OS_OK) {
        // Task was activated successfully
    }
```

```
}
```

API Function: OS_TerminateTask

Header File: `task.h`

```
OS_ErrorStatus OS_TerminateTask(OS_TCB* Task);
```

Terminates a running task and removes it from the task scheduler. The task cannot be activated again after termination.

Parameters:

- **`Task`**
  Pointer to the `OS_TCB` structure of the task to be terminated.

Returns:

- **`OS_OK`**
  If the task was terminated successfully.
- **`TASK_CREATION_ERROR`**
  If the task could not be terminated.

Example Usage:

```
void TerminateMyTask(void) {
    OS_ErrorStatus status = OS_TerminateTask(&myTask); // Terminate the task
    if (status == OS_OK) {
        // Task was terminated successfully
    }
}
```

API Function: OS_DelayTask

Header File: `task.h`

```
OS_ErrorStatus OS_DelayTask(OS_TCB* Task, uint32_t NoOfTicks);
```

Delays the execution of the specified task for a given number of ticks.

Parameters:

- **`Task`**
  Pointer to the `OS_TCB` structure of the task to be delayed.
- **`NoOfTicks`**
  Number of ticks to delay the task.

Returns:

- `OS_OK`
  If the task was delayed successfully.
- `TASK_CREATION_ERROR`
  If the task could not be delayed.

Example Usage:

```
void DelayMyTask(void) {
    OS_ErrorStatus status = OS_DelayTask(&myTask, 10); // Delay the task for 10 ticks
    if (status == OS_OK) {
        // Task was delayed successfully
    }
}
```

Initialization and OS Control APIs

API Function: OS_Init

Header File: `<Your_RTOS_Header_File>.h` (Specify your RTOS main header file here)

```
OS_ErrorStatus OS_Init(void);
```

Initializes the G RTOS kernel by setting up essential data structures, initializing resources, and preparing the system for task scheduling and synchronization. This function should be called before any other RTOS functions to ensure that the RTOS kernel is correctly initialized and ready for operation.

Parameters:

- **None**

Returns:

- **OS_OK**: If the initialization was successful.
- **OS_ERROR**: If an error occurred during initialization.

Example Usage:

```
void main(void) {
    OS_ErrorStatus status = OS_Init();  // Initialize the RTOS
    if (status == OS_OK) {
        // Initialization succeeded, proceed with task creation and scheduling
        OS_StartScheduler();  // Start task scheduling
    } else {
        // Handle initialization error
```

```
    }
}
```

API Function: OS_StartOS

Header File: `task.h`

```
OS_ErrorStatus OS_StartOS();
```

Starts the operating system scheduler. This function must be called after creating and activating all tasks.

Returns:

- **`OS_OK`**
  If the OS started successfully.
- **`TASK_CREATION_ERROR`**
  If there was an error starting the OS.

Example Usage:

```
void StartMyRTOS(void) {
    OS_ErrorStatus status = OS_StartOS(); // Start the RTOS scheduler
    if (status == OS_OK) {
        // RTOS started successfully
    }
}
```

Hook APIs

API Function: OS_SetSysTickHook

Header File: `YourHeaderFile.h`

```
void OS_SetSysTickHook(OS_SysTickHook callback);
```

Sets a custom callback function to be executed at each SysTick interrupt. This function allows the user to define specific actions or behaviors that should occur periodically with every SysTick event.

Parameters:

- `callback`: A pointer to a function of type `OS_SysTickHook` that defines the callback to be executed on each SysTick interrupt. The function must take no parameters and return `void`.

Returns:

- None.

Example Usage:

```
// Custom SysTick callback function
void MySysTickHandler(void) {
    // Custom actions to perform at each SysTick
}

// Setting the SysTick hook to the custom handler
void InitSysTickHook(void) {
    OS_SetSysTickHook(MySysTickHandler); // Assign MySysTickHandler to the
SysTick hook
}
```

API Function: OS_RegisterIdleHook

Header File: `task.h`

```
void OS_RegisterIdleHook(OS_IdleHookCallback callback);
```

Registers a callback function that is called when the system is idle. This is useful for low-power modes or executing background tasks.

Parameters:

- **`callback`**
  Pointer to the function to be called when the system is idle.

Returns:
None

Example Usage:

```
void MyIdleTask(void) {
    // Code to run when the system is idle
}

void RegisterMyIdleHook(void) {
    OS_RegisterIdleHook(MyIdleTask); // Register the idle task
}
```

Semaphore APIs

API Function: OS_InitSemaphore

Header File: `semaphore.h`

```
OS_SemaphoreState OS_InitSemaphore(OS_Semaphore* semaphore, uint8_t
initialCount);
```

Initializes a semaphore with a specified initial count. This count represents the number of available resources that the semaphore can manage.

Parameters:

- **`semaphore`**
  Pointer to the `OS_Semaphore` structure that will be initialized.
- **`initialCount`**
  The initial count of the semaphore, representing the number of resources available. It must be greater than or equal to zero.

Returns:

- **`OS_SEMAPHORE_INIT_OK`**
  If the semaphore was initialized successfully.
- **`OS_SEMAPHORE_ALREADY_ACQUIRED`**
  If there was an error initializing the semaphore.

Example Usage:

```
void InitMySemaphore(void) {
    OS_Semaphore mySemaphore;
    OS_SemaphoreState state = OS_InitSemaphore(&mySemaphore, 1); //
Initialize semaphore with 1 resource
    if (state == OS_SEMAPHORE_INIT_OK) {
        // Semaphore initialized successfully
    }
}
```

API Function: OS_AcquireSemaphore

Header File: `semaphore.h`

```
OS_SemaphoreState OS_AcquireSemaphore(OS_Semaphore* semaphore, OS_TCB* task);
```

Attempts to acquire the specified semaphore. If the semaphore is available, the task becomes the owner, and the semaphore count is decremented. If it is busy, the task is added to the waiting queue.

Parameters:

- **`semaphore`**
  Pointer to the `OS_Semaphore` structure that the task is attempting to acquire.
- **`task`**
  Pointer to the `OS_TCB` structure of the task attempting to acquire the semaphore.

Returns:

- **`OS_SEMAPHORE_AVAILABLE`**
  If the semaphore was acquired successfully.
- **`OS_SEMAPHORE_BUSY`**
  If the semaphore is currently busy and the task is added to the waiting queue.
- **`OS_SEMAPHORE_ALREADY_ACQUIRED`**
  If the task already owns the semaphore.

Example Usage:

```
void AcquireMySemaphore(OS_TCB* myTask) {
    OS_SemaphoreState state = OS_AcquireSemaphore(&mySemaphore, myTask); //
Attempt to acquire the semaphore
    if (state == OS_SEMAPHORE_AVAILABLE) {
        // Semaphore acquired successfully
    } else if (state == OS_SEMAPHORE_BUSY) {
        // Semaphore is busy; task is now waiting
    }
}
```

API Function: OS_ReleaseSemaphore

Header File: `semaphore.h`

```
OS_SemaphoreState OS_ReleaseSemaphore(OS_Semaphore* semaphore);
```

Releases the semaphore, making it available for other tasks. If there are tasks waiting for the semaphore, one is woken up and allowed to acquire the semaphore.

Parameters:

- **`semaphore`**
  Pointer to the `OS_Semaphore` structure that is to be released.

Returns:

- **`OS_SEMAPHORE_AVAILABLE`**
  If the semaphore was released successfully.
- **`OS_SEMAPHORE_ALREADY_ACQUIRED`**
  If the semaphore was not owned by any task at the time of release.

Example Usage:

```
void ReleaseMySemaphore(void) {
    OS_SemaphoreState state = OS_ReleaseSemaphore(&mySemaphore); // Release
the semaphore
    if (state == OS_SEMAPHORE_AVAILABLE) {
        // Semaphore released successfully
    }
}
```

API Function: OS_InitMutex

Header File: `Mutex.h`

`OS_MutexState OS_InitMutex(OS_Mutex* mutex);`

Initializes a mutex, setting its initial state to unlocked. This function prepares the mutex for use by a task.

Parameters:

- **`mutex`**
  Pointer to the `OS_Mutex` structure that will be initialized.

Returns:

- **`OS_MUTEX_INIT_OK`**
  If the mutex was initialized successfully.

Example Usage:

```
void InitMyMutex(void) {
    OS_Mutex myMutex;
    OS_MutexState state = OS_InitMutex(&myMutex); // Initialize the mutex
    if (state == OS_MUTEX_INIT_OK) {
        // Mutex initialized successfully
    }
}
```

API Function: OS_AcquireMutex

Header File: `Mutex.h`

`OS_MutexState OS_AcquireMutex(OS_Mutex* mutex, OS_TCB* task);`

Attempts to acquire the specified mutex for the given task. If the mutex is already locked and owned by another task, the calling task will be blocked and added to the waiting queue.

Parameters:

- **`mutex`**
  Pointer to the `OS_Mutex` structure that the task is attempting to acquire.
- **`task`**
  Pointer to the `OS_TCB` structure of the task attempting to acquire the mutex.

Returns:

- **`OS_MUTEX_AVAILABLE`**
  If the mutex was acquired successfully.
- **`OS_MUTEX_BUSY`**
  If the mutex is currently busy, and the task has been added to the waiting queue.
- **`OS_MUTEX_ALREADY_ACQUIRED`**
  If the task already owns the mutex.

Example Usage:

```
void AcquireMyMutex(OS_TCB* myTask) {
    OS_MutexState state = OS_AcquireMutex(&myMutex, myTask); // Attempt to acquire the mutex
    if (state == OS_MUTEX_AVAILABLE) {
        // Mutex acquired successfully
    } else if (state == OS_MUTEX_BUSY) {
        // Mutex is busy; task is now waiting
    }
}
```

API Function: OS_ReleaseMutex

Header File: `Mutex.h`

```
OS_MutexState OS_ReleaseMutex(OS_Mutex* mutex);
```

Releases the mutex from the current owner, making it available for other tasks. If there are tasks waiting for the mutex, one of them is woken up and allowed to acquire the mutex.

Parameters:

- **`mutex`**
  Pointer to the `OS_Mutex` structure that is to be released.

Returns:

- **`OS_MUTEX_AVAILABLE`**
  If the mutex was released successfully.

Example Usage:

```
void ReleaseMyMutex(void) {
    OS_MutexState state = OS_ReleaseMutex(&myMutex); // Release the mutex
    if (state == OS_MUTEX_AVAILABLE) {
        // Mutex released successfully
    }
}
```

Event group APIs

API Function: OS_InitEventGroup

Header File: `EventGroup.h`

```
void OS_InitEventGroup(OS_EventGroup* eventGroup);
```

Initializes an event group, clearing all event bits and preparing it for use by tasks.

Parameters:

- **`eventGroup`**
  Pointer to the `OS_EventGroup` structure that will be initialized.

Returns:

None.

Example Usage:

```
void InitMyEventGroup(void) {
    OS_EventGroup myEventGroup;
    OS_InitEventGroup(&myEventGroup);  // Initialize the event group
}
```

API Function: `OS_WaitForEventBits`

Header File: `EventGroup.h`

```
uint8_t OS_WaitForEventBits(OS_EventGroup* eventGroup, OS_EventGroupBits
eventBits, uint8_t waitForAllBits, uint32_t timeout);
```

Waits for specific bits in an event group to be set. If the bits are not set, the task waits until either they are set or the timeout period expires.

Parameters:

- **`eventGroup`**
  Pointer to the `OS_EventGroup` structure to check.
- **`eventBits`**
  Bit pattern specifying the event bits to wait for.
- **`waitForAllBits`**
  Flag indicating whether to wait for all specified bits (1) or any of them (0).
- **`timeout`**
  Maximum number of ticks to wait; set to 0 for no timeout.

Returns:

- `OS_EVENT_GROUP_OK`
  If the requested event bits were set.
- `OS_EVENT_GROUP_TIMEOUT`
  If the timeout period expired.
- `OS_EVENT_GROUP_ERROR`
  If an error occurred.

Example Usage:

```
void WaitForMyEventBits(void) {
    OS_EventGroup myEventGroup;
    uint8_t result = OS_WaitForEventBits(&myEventGroup, 0x03, 1, 100); //
Wait for bits 0 and 1
    if (result == OS_EVENT_GROUP_OK) {
        // Event bits set successfully
    } else if (result == OS_EVENT_GROUP_TIMEOUT) {
        // Timeout occurred
    }
}
```

API Function: `OS_SetEventBits`

Header File: `EventGroup.h`

```
void OS_SetEventBits(OS_EventGroup* eventGroup, OS_EventGroupBits eventBits);
```

Sets specified bits in an event group, potentially releasing tasks waiting for these bits.

Parameters:

- **`eventGroup`**
  Pointer to the `OS_EventGroup` structure where bits are to be set.
- **`eventBits`**
  Bit pattern specifying the event bits to set.

Returns:

None.

Example Usage:

```
void SetMyEventBits(void) {
    OS_EventGroup myEventGroup;
    OS_SetEventBits(&myEventGroup, 0x01); // Set bit 0
}
```

API Function: `OS_ClearEventBits`

Header File: `EventGroup.h`

```
void OS_ClearEventBits(OS_EventGroup* eventGroup, OS_EventGroupBits
eventBits);
```

Clears specified bits in an event group, effectively resetting them.

Parameters:

- **`eventGroup`**
  Pointer to the `OS_EventGroup` structure where bits are to be cleared.
- **`eventBits`**
  Bit pattern specifying the event bits to clear.

Returns:

None.

Example Usage:

```
void ClearMyEventBits(void) {
    OS_EventGroup myEventGroup;
    OS_ClearEventBits(&myEventGroup, 0x01); // Clear bit 0
}
```

Configuration for RA3 RTOS

Header File: **`config.h`**

This configuration file allows you to customize RA3 RTOS for specific system needs by setting key parameters. These configurations define the stack size, timing, priority levels, CPU frequency, and more to optimize the RTOS for various embedded applications.

Configuration Parameters in `config.h`

PREEMPTION

```
// Define macro for OS preemption control
#define OS_PREEMPTION_ENABLED                1
```

Description: Enables or disables OS preemption. If set to 1, preemption is enabled, allowing the OS to switch tasks based on priority. If set to 0, the PendSV trigger for task switching will be removed, disabling preemption.

MAIN_STACK_SIZE

```
// Size of the main stack in bytes
#define OS_MAIN_STACK_SIZE           3072
```

Description: Sets the size of the main stack used by the RTOS in bytes. Adjust this according to your system’s memory requirements, especially if the main stack handles intensive or recursive tasks.

DEFAULT_TASK_STACK_SIZE

```
// Default stack size for tasks in bytes
#define OS_DEFAULT_TASK_STACK_SIZE    1024
```

Description: Specifies the default stack size for tasks. Increase this value if tasks require more stack space (e.g., tasks involving deep function calls or large local variables).

TICK_TIME_IN_MS

```
// Time duration of each tick in milliseconds
#define OS_TICK_TIME_IN_MS           1
```

Description: Sets the duration of each OS tick. This tick frequency controls the RTOS's task switching and timing functions. A lower tick duration means more frequent task switching but higher CPU load.

CPU_CLOCK_FREQ_IN_HZ

```
// CPU clock frequency in hertz
#define OS_CPU_CLOCK_FREQ_IN_HZ     7200000
```

Description: Defines the CPU clock frequency in MHz. Accurate setting is essential for precise timing in the RTOS scheduler. Ensure this matches your processor’s actual frequency for optimal performance.

OS_LOWEST_PRIORITY

```
// Lowest priority level for tasks
#define OS_LOWEST_PRIORITY            255
```

Description: Specifies the lowest priority level available for tasks, where higher numbers represent lower priorities. Setting this higher allows for a more granular priority system.

OS_HIGHEST_PRIORITY

```
// Highest priority level for tasks
#define OS_HIGHEST_PRIORITY           0
```

Description: Sets the highest priority level for tasks. Lower numbers represent higher priorities in RA3 RTOS. Set this to define the top priority available for critical tasks.

IDLE_TASK_HOOK_ENABLED

```
// Enable/disable the idle task hook
#define OS_IDLE_TASK_HOOK_ENABLED     1
```

Description: Enables or disables the idle task hook. If enabled (`1`), the RTOS will execute a user-defined callback whenever the idle task runs, allowing low-priority background operations.

TICK_HOOK_ENABLED

```
// Enable/disable the tick task hook
#define OS_TICK_HOOK_ENABLED 1
```

Description: Enables or disables the tick hook. If enabled (1), the RTOS will execute a user-defined callback function at each tick, useful for periodic background tasks.

Note: Ensure you review and adjust these parameters based on the specific needs of your embedded system for optimal performance. Misconfiguration can lead to unexpected behavior in task timing, scheduling, and memory utilization.

Contributing to RA3 RTOS

We welcome contributions from the community to help improve RA3 RTOS! If you're interested in contributing, please follow these guidelines:

1. Fork the Repository

Start by forking the RA3 RTOS repository to your own GitHub account. This creates a copy of the repository where you can make your changes.

- Navigate to the RA3 RTOS GitHub Repository.
- Click on the **Fork** button in the top right corner of the page.

2. Clone Your Fork

Clone your forked repository to your local machine to begin making changes.

```
git clone https://github.com/ENGaliyasser/RA3-RTOS.git
cd RA3-RTOS
```

3. Create a New Branch

Before making changes, create a new branch for your feature or bug fix. This keeps your modifications organized and separate from the main codebase.

```
git checkout -b feature/my-new-feature
```

4. Implement Your Changes

Make the necessary changes or add the new feature. Please adhere to the following coding rules to ensure consistency across the project:

- Follow the existing coding style (indentation, naming conventions).
- Comment your code adequately for clarity.
- Keep your changes focused on a single feature or bug fix.

5. Test Your Changes

Before submitting your contributions, test your code to ensure it works as expected. Ensure that you run any existing tests and add new tests if applicable.

6. Commit Your Changes

Once you're satisfied with your changes, commit them with a clear and descriptive message.

```
git add .
git commit -m "Add my new feature"
```

7. Push Your Changes

Push your changes to your forked repository.

```
git push origin feature/my-new-feature
```

8. Create a Pull Request

Navigate to the original RA3 RTOS repository and click on the **Pull Requests** tab. Then, click on the **New Pull Request** button.

- Select your branch from the dropdown.
- Provide a descriptive title and description for your pull request, explaining the changes you've made and why they should be merged.

9. Address Feedback

Once you submit your pull request, the maintainers will review your changes. Be prepared to discuss your code and make adjustments based on feedback.

10. Celebrate Your Contribution!

After your pull request is merged, you’ll be recognized as a contributor to RA3 RTOS! Thank you for your efforts to improve the project!