EC200U&EG91xU Series QuecOpen RTOS Development Guide

LTE Standard Module Series

Version: 1.1

Date: 2023-08-16

Status: Released

At Quectel, our aim is to provide timely and comprehensive services to our customers. If you require any assistance, please contact our headquarters:

Quectel Wireless Solutions Co., Ltd.
Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai 200233, China
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local offices. For more information, please visit:
http://www.quectel.com/support/sales.htm.

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/technical.htm.
Or email us at: support@quectel.com.

About the Document

Revision History

Version	Date	Author	Description
-	2021-12-02	Kevin WANG	Creation of the document
1.0	2022-01-04	Kevin WANG	First official release
1.1	2023-08-16	Kevin WANG	Added applicable modules EG912U-GL and EG915U series.

Contents

Table Index

Figure Index

1 Introduction

Quectel EC200U series and EG91xU family modules support QuecOpen® solution. QuecOpen® is an embedded development platform based on RTOS, which is intended to simplify the design and development of IoT applications. For more information on QuecOpen®, see ***document [1]***.

This document explains how to develop RTOS on EC200U series and EG91xU family modules in QuecOpen® solution.

1.1. Applicable Modules

Table 1: Applicable Modules

Module Family	Module
-	EC200U Series
EG91xU	EG912U-GL
	EG915U Series

2 Brief Introduction on RTOS

2.1. RTOS Definition

Real Time Operating System (hereinafter referred to as RTOS) is an operating system that serves real-time applications in processing data as it comes without buffer delays. It is a time-bound system with well-defined and fixed time constraints. RTOS responds very fast and the processing is done within the specified time.

RTOS kernel objects: task, semaphore, mutex, message queue, timer, event notification and software watchdog.

2.2. Task

In daily life, when we manage a serious problem, we generally divide it into several parts to solve them one by one. Similarly, in a complex program development, we usually divide a big task into multiple small tasks, and process them on the computer step by step until the development of the complex program is finally completed. This solution is to make the operating system run multiple tasks simultaneously to improve processor utilization.

2.2.1. Task and Its Memory Structure

A RTOS task usually consists of three parts: task control block, task stack and task code. Among them, the task control block is associated with the task code, and records each task attribute. The task stack saves the task environment. RTOS task composition is illustrated in the figure below.

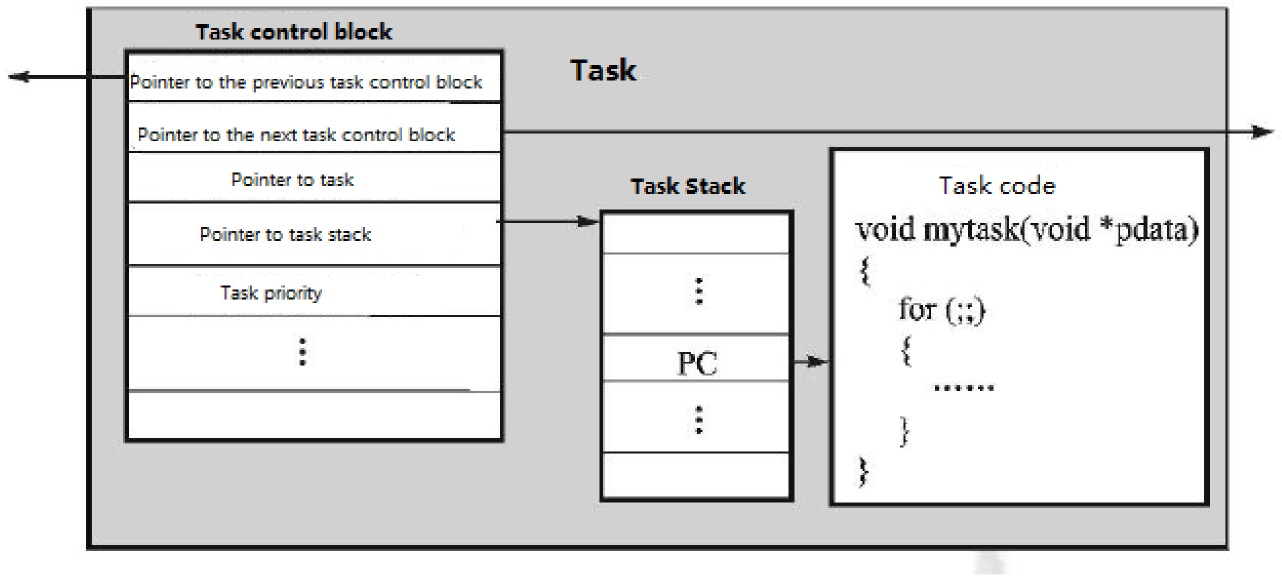

Figure 1: RTOS Task Composition

The control blocks of multiple tasks are generally organized in a linked list. The linked list is shown in the figure below:

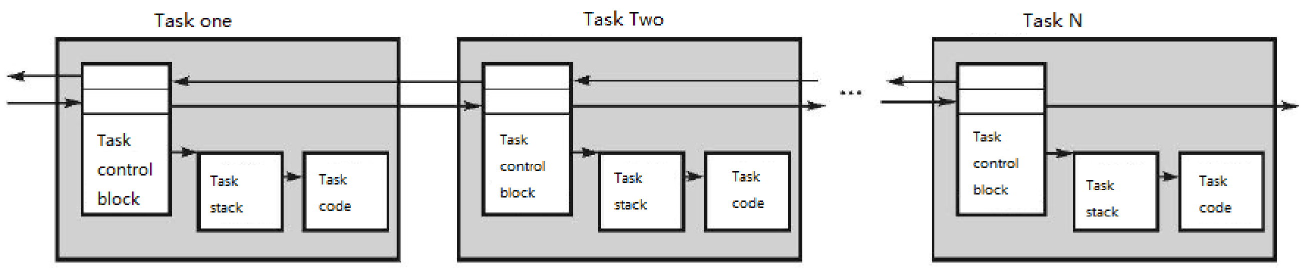

Figure 2: Linked List of Task Control Block

2.2.2. Task State

Task states are described below:

Table 2: Task State Description

State	Description
Sleep	The task resides in the program space (ROM or RAM) in the form of code and has not yet been delivered to the operating system for management. In short, if the task has not been assigned to a task control block or has been deprived of a task control block, the task is in the sleep state.

Ready	If the system assigns a task control block to the task and registers the task in the task ready table for its further running, the task is in the ready state.
Running	The scheduler determines the task in the ready state to obtain the right to use the CPU, then the task enters the running state. Only one task can be running at a time. Among the ready tasks, only when all tasks with a priority higher than this task are turned into the waiting state, can this task enter the running state.
Waiting	If a running task is waiting for an interval to pass or an event to occur before running, the task transfers the right to use the CPU to other tasks to enter the waiting state.
Interrupt Service	Once a running task responds to the interrupt request, it will suspend running and execute the interrupt service. At this time, the task is in the interrupt service state.

A task can switch between 5 different states. The switching process is shown in the figure below:

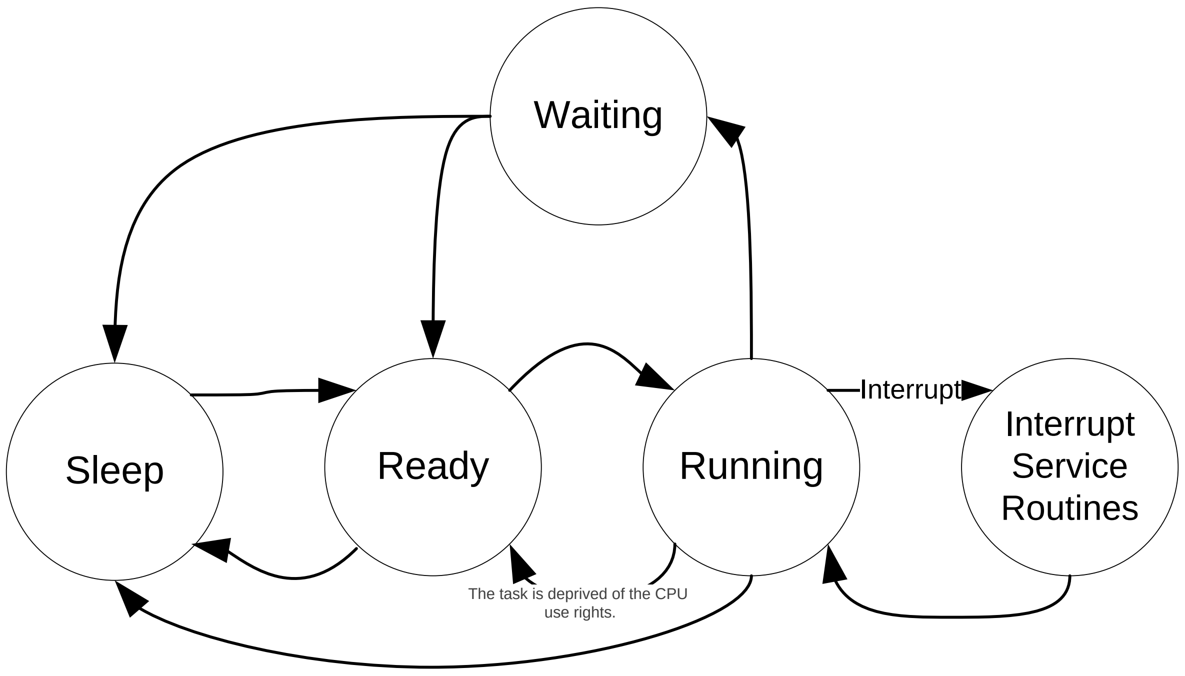
Waiting
Sleep
Ready
Running
Interrupt
Interrupt Service Routines
The task is deprived of the CPU use rights.

Figure 3: Task State Switching

2.2.3. Task Creation and Deletion

The RTOS APIs for task creation and deletion provided by the module are listed below. See ***Chapter 3*** for details.

```
//For task creation
QlOSStatus ql_rtos_task_create
(
    ql_task_t      *taskRef,              /* OS task reference                 */
    uint32               stackSize,            /* number of bytes in task stack area    */
    uint8                priority,             /* task priority                     */
    char            *taskName,            /* task name                         */
    void            (*taskStart)(void*),  /* pointer to task entry point       */
    void*           argv                  /* task entry argument pointer       */
);

//For task deletion
QlOSStatus ql_rtos_task_delete
(
    ql_task_t taskRef      /* OS task reference    */
);
```

The parameters that need to be passed for task creation are task handle pointer, task stack size, task priority, task name, task entry function and parameters passed to the task. Among them, the task stack size depends on the task code.

Module task priority range: 0 to 30. 30 means the highest task priority. The task priority of the application layer should not be higher than 30.

Task code structure:

```
void user_task(void * argv)
{
    while(1)
    {
        //process your task
    }

    ql_rtos_task_delete(NULL);
}
```

NOTE

1. In a task, after exiting the *while(1)* structure, the current task must be deleted to release the resources occupied by the task control block and task stack. Passing NULL to *ql_rtos_task_delete(ql_task_t taskRef)* can delete the current task.
2. If the parameter passed into *ql_rtos_task_delete(ql_task_t taskRef)* is not NULL, the task pointed to by the task handle is deleted.

The task management APIs are listed in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 3: Task Management API Overview

Function	Description
ql_rtos_task_create()	Creates a task.
ql_rtos_task_create_default()	Creates a task. The default number of events is 23.
ql_rtos_task_delete()	Deletes a task (If the parameter is NULL, delete the current task).
ql_rtos_task_suspend()	Suspends a task.
ql_rtos_task_resume()	Takes the task out of the suspended state.
ql_rtos_task_yield()	Releases CPU usage.
ql_rtos_task_get_current_ref()	Gets the current task handle.
ql_rtos_task_change_priority()	Changes the priority of target task.
ql_rtos_task_get_status()	Gets the task state.
ql_rtos_task_sleep_ms()	Sets task sleep time in milliseconds.
ql_rtos_task_sleep_s()	Sets task sleep time in seconds.
ql_rtos_enter_critical()	Enters the critical section protection.
ql_rtos_exit_critical()	Exits the critical section protection.

2.3. Semaphore

When a complex task is divided into several small tasks, these small tasks should be arranged and processed in an orderly manner. For example, an application is divided into task A and task B. Task B should wait for task A to finish the corresponding operation for further execution of task B. At this time, task B is in the waiting state until task A completes its operation. Then, task A sends a signal to task B to notify task B that it can now be executed.

Semaphore is not only used for notification, but it can also measure the number of task executions. If task A continuously sends signals to task B for several times, task B can be executed cyclically. Semaphore plays an important role in the multi-task synchronization mechanism.

Below is the pseudo code related to semaphore.

```
sem_t sem;

void taskA(void * argv)
{
    while(1)
    {
        ...
        sem_post(&sem);
        sleep(1);
        ...
    }
}

void taskB(void * argv)
{
    int cnt = 0;
    while(1)
    {
        ...
        sem_wait(&sem);
        printf(“got sem for %d time(s)\n”, ++cnt);
        ...
    }
}
```

The semaphore APIs are shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 4: Semaphore API Overview

Function	Description
ql_rtos_semaphore_create()	Creates a semaphore.
ql_rtos_semaphore_wait()	Sets the waiting time in milliseconds for a semaphore.
ql_rtos_semaphore_release()	Releases a semaphore.
ql_rtos_semaphore_get_cnt()	Gets the semaphore value.
ql_rtos_semaphore_delete()	Deletes a semaphore.

2.4. Mutex

Mutex, a binary semaphore, whose maximum value is 1, is used to protect shared resources from being accessed by multiple tasks at the same time, so as to avoid inconsistencies in the context of shared resources accessed by the same task before and after, resulting in some unexpected situations (that may be serious). Therefore, mutex plays an important role in protecting shared resources.

Below is the pseudo code related to mutex.

```
bool bFlag = TRUE;

void taskA(void * argv)
{
    ...
    //lock();
    if(bFlag == TRUE)
    {
    lableA:
        printf("bFlag value is: %d\n", bFlag);
        bFlag = FALSE;
    }
    //unlock();
    ...
}

void taskB(void * argv)
{
    ...
    //lock();
    if(bFlag == TRUE)
    {
    lableB:
        printf("bFlag value is: %d\n", bFlag);
        bFlag = FALSE;
    }
    //unlock();
    ...
}
```

In the code above, if task scheduling appears after *taskA* finishes the *if* sentence before *lableA*, CPU usage is switched to *taskB*. After *taskB* finishes all *if* sentences, task scheduling is switched to *taskA*. At this time, *bFlag* printed by *taskA* will be FALSE, not be TRUE as you wish.

If you uncomment the mutex before and after the *if* code blocks the two tasks. One *if* code block represents an array of atomic operations and an unexpected situation will not occur.

The mutex APIs are shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 5: Mutex API Overview

Function	Description
ql_rtos_mutex_create()	Creates a mutex.
ql_rtos_mutex_lock()	Locks a mutex. You can set the waiting time according to your requirements.
ql_rtos_mutex_try_lock()	Tries to lock a mutex.
ql_rtos_mutex_unlock()	Unlocks a mutex.
ql_rtos_mutex_delete()	Deletes a mutex.

2.5. Message Queue

In addition to sending semaphores for interaction, multiple tasks can also deliver messages through message queues. The message queue delivers messages from one task to another. Copy the message to the message queue according to the start address and message size, and then send a semaphore to the task waiting for the message. The thread waiting for the message copies the message in the message queue to the local buffer, and then removes the message in the queue.

The pseudo code below shows how to run the message queue.

```
queue_t queue;

void taskA(void * argv)
{
    while(1)
    {
        ...
        queue_post(&queue, &msg, msg_size);
        sleep(1);
        ...
    }
}

void taskB(void * argv)
{
    int cnt = 0;
    while(1)
    {
        ...
        queue_wait(&queue, &msg, msg_size);
        printf("got msg for %d time(s): %s\n", ++cnt, msg);
        ...
    }
}
```

The message queue APIs are shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 6: Message Queue API Overview

Function	Description
ql_rtos_queue_create()	Creates a message queue.
ql_rtos_queue_wait()	Waits for messages in the queue. The waiting time is in milliseconds.
ql_rtos_queue_release()	Releases a message queue.
ql_rtos_queue_get_cnt()	Gets the number of messages in the queue.
ql_rtos_queue_delete()	Deletes a message queue.

2.6. Timer

A timer is used for timing. After the specified timer arrives, it will inform the user about the expiration through callback function. The timer of the module supports running callback in system service and the specified task.

The timer APIs are shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 7: Timer API Overview

Function	Description
ql_rtos_timer_create()	Creates a timer.
ql_rtos_timer_start()	Starts a timer. The time precision is in milliseconds. After the timer expires, the system will be awakened from sleep mode.
ql_rtos_timer_start_relaxed()	Starts a timer. The value of *relax_time* determines whether the system will be awakened from sleep mode after the timer expires.
ql_rtos_timer_start_us()	Starts a timer. The time precision is in microseconds. After the timer expires, the system will be awakened from sleep mode.
ql_rtos_timer_stop()	Stops a timer.
ql_rtos_timer_is_running()	Determines whether the timer is running.
ql_rtos_timer_delete()	Deletes a timer.

2.7. Event Notification

Event notification is a way of module core thread communication, it can integrate the advantages of communication methods such as semaphore, message queue, and event flag. Most of the RTOS features of the module, such as software watchdog and timer, can be realized through event notification. Therefore, if data interaction is involved between threads, it is recommended to use event notification to achieve communication.

The code structure of the tasks is shown below:

```
void user_task(void * argv)
{
        ql_event_t event = {0};

        while(1)
```

```
    {
        if(ql_event_wait(&test_event, QL_WAIT_FOREVER) != 0)
        {
            continue;
        }
        if(test_event.id == …(Specific event id))
        {
            //Specific event handler
        }
}
}
```

The event notification APIs are shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 8: Event Notification API Overview

Function	Description
ql_rtos_event_send()	Sends an event.
ql_event_wait()	Waits for an event. You can set the waiting time according to your requirements.
ql_event_try_wait()	Tries to wait for an event.

2.8. Software Watchdog

If a task falls into an infinite loop or occupies the CPU continuously, other tasks with lower priority than this task will not be scheduled. At this time, the module is rebooted by the software watchdog.

After a task is created, the software watchdog registration function can be called to register the task and the specific watchdog callback function to the operating system. If the watchdog is not enabled at this time, the enabling function can be called to configure the feeding period and the maximum number of lost feedings. After the configuration is completed, the system will run in the watchdog-feeding cycle specified when it was enabled, circularly call the specified watchdog callback function at registration and pass the parameter of the callback function as the watchdog-feeding event ID and the task handle of the task that needs to feed the watchdog. You can send an event or other RTOS communication methods to notify the watchdog-feeding task and call the watchdog-feeding function to feed the watchdog in this callback.

The code below shows how to use the software watchdog:

```
void sw_dog_create()
{
        ql_rtos_swdog_register((ql_swdog_callback)feed_dog_callback, demo_task);
        ql_rtos_sw_dog_enable(5000, 3); //The feeding cycle is 5 s, and the maximum number of lost
                                                                        feedings is 3.
}
void feed_dog_callback(uint32 id_type, void *ctx) //The callback function of software watchdog.
{
        ql_event_t event;
        event.id = QUEC_KERNEL_FEED_DOG;
        ql_rtos_event_send(demo_task, &event); //Feed a watchdog for the specified task.
}
void    demo_task_callback(void *argv)    //Task execution function.
{
        ql_event_t event ;
        if(ql_event_try_wait(&test_event) != 0)
                {
                continue;
        }
        …
        if(test_event.id == QUEC_KERNEL_FEED_DOG)
        {
                ql_rtos_feed_dog(); //Feed a watchdog.
        }
…
}
```

The software watchdog APIs is shown in the following table. See ***Chapter 3.3*** for details on these APIs.

Table 9: Software Watchdog API Overview

Function	Description
ql_rtos_swdog_register()	Registers a software watchdog.
ql_rtos_swdog_unregister()	Unregisters a software watchdog.
ql_rtos_sw_dog_enable()	Enables a software watchdog.
ql_rtos_sw_dog_disable()	Disables a software watchdog.
ql_rtos_feed_dog()	Feeds a watchdog for the current task and is called in the task that needs to feed the watchdog.

3 RTOS APIs

3.1. Data Type

3.1.1. QlOSStatus

```
typedef int QlOSStatus;     //Data type of RTOS API return value
```

3.1.2. ql_task_t

```
typedef void * ql_task_t;     //Type of task handle
```

3.1.3. ql_sem_t

```
typedef void * ql_sem_t;     //Type of semaphore handle
```

3.1.4. ql_mutex_t

```
typedef void * ql_mutex_t;    //Type of mutex handle
```

3.1.5. ql_queue_t

```
typedef void * ql_queue_t;    //Type of message queue handle
```

3.1.6. ql_timer_t

```
typedef void * ql_queue_t,    //Type of timer handle
```

3.1.7. ql_wait_e

```
typedef enum
{
    QL_WAIT_FOREVER = 0xFFFFFFFF,
    QL_NO_WAIT           = 0
} ql_wait_e;
```

- **Member**

Member	Description
QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

3.2. Header File and Example File

ql_api_osi, the header file of RTOS API, is in the *components\ql-kernel\inc* directory of the SDK. *osi_demo*, the example file of RTOS API is in the *components\ql-application\osi* directory of the SDK.

3.3. API Description

3.3.1. ql_rtos_task_create

This function creates a task.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_create (ql_task_t *taskRef, uint32 stackSize, uint8 priority, char
*taskName, void (*taskStart)(void*), void* argv, uint32 event_count);
```

- **Parameter**

taskRef:
[Out] Task handle.

stackSize:
[In] Task stack size depending on task code. Maximum value: 128 KB. Unit: byte.

priority:
[In] Task priority. Range: 0–30.

taskName:
[In] Task name. Maximum value: 16. Unit: byte.

taskStart:
[In] The task entry function.

argv:
[In] The parameters passed to the task.

event_count:
[In] The maximum number of pending events stored in the task.

- **Return Value**

QL_OSI_TASK_CREATE_FAIL	Failed execution
QL_OSI_TASK_NAME_INVALID	Invalid task name
QL_OSI_INVALID_TASK_REF	Invalid task handle
QL_OSI_SUCCESS	Successful execution

3.3.2. ql_rtos_task_create_default

This function creates a task. The default number of events is 23.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_create_default (ql_task_t *taskRef, uint32 stackSize, uint8 priority,
char *taskName, void (*taskStart)(void*), void* argv,);
```

- **Parameter**

taskRef:
[In] Task handle.

stackSize:
[In] Task stack size depending on task code. Maximum value: 128 KB. Unit: byte.

priority:
[In] Task priority. Range: 0–30.

taskName:
[In] Task name. Maximum value: 16. Unit: byte.

taskStart:
[In] The task entry function.

argv:
[In] The parameters passed to the task.

- **Return Value**

QL_OSI_TASK_CREATE_FAIL	Failed execution
QL_OSI_TASK_NAME_INVALID	Invalid task name
QL_OSI_INVALID_TASK_REF	Invalid task handle
QL_OSI_SUCCESS	Successful execution

3.3.3. ql_rtos_task_delete

This function deletes a task.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_delete (ql_task_t taskRef);
```

- **Parameter**

taskRef:
[In] Task handle. If the parameter is NULL, delete the current task. Otherwise, delete the task pointed to by the task handle.

- **Return Value**

QL_OSI_INVALID_TASK_REF	Invalid task handle
QL_OSI_SUCCESS	Successful execution

3.3.4. ql_rtos_task_suspend

This function suspends a task.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_suspend (ql_task_t taskRef);
```

- **Parameter**

taskRef:
[In] Task handle.

- **Return Value**

QL_OSI_INVALID_TASK_REF	Invalid task handle
QL_OSI_SUCCESS	Successful execution

3.3.5. ql_rtos_task_resume

This function takes the task out of the suspended state.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_resume(ql_task_t taskRef);
```

- **Parameter**

taskRef:
[In] Task handle.

- **Return Value**

QL_OSI_INVALID_TASK_REF	Invalid task handle
QL_OSI_SUCCESS	Successful execution

3.3.6. ql_rtos_task_yield

This function releases CPU usage.

- **Prototype**

```
extern void ql_rtos_task_yield (void);
```

- **Parameter**

None

- **Return Value**

None

3.3.7. ql_rtos_task_get_current_ref

This function gets the current task handle.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_get_current_ref (ql_task_t * taskRef)
```

- **Parameter**

taskRef:
[Out] Task handle.

- **Return Value**

QL_OSI_INVALID_TASK_REF	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.8. ql_rtos_task_change_priority

This function changes the priority of target task.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_change_priority (ql_task_t taskRef, uint8 new_priority, uint8
*old_priority);
```

- **Parameter**

taskRef:
[Out] Task handle.

new_priority:
[In] New priority of the task.

old_priority:
[In] Original priority of the task.

- **Return Value**

QL_OSI_TASK_GET_PRIO_FAIL	Failed to get the priority
QL_OSI_TASK_SET_PRIO_FAIL	Failed to set the priority
QL_OSI_SUCCESS	Successful execution

3.3.9. ql_rtos_task_get_status

This function gets the task state.

- **Prototype**

```
extern QlOSStatus ql_rtos_task_get_status (ql_task_t task_ref, ql_task_status_t *status);
```

- **Parameter**

taskref:
[In] Task handle.

status:
[Out] Task state. See ***Chapter 3.3.9.1*** for details.

- **Return Value**

QL_OSI_INVALID_TASK_REF	Failed to get the task handle
QL_OSI_SUCCESS	Successful execution

3.3.9.1. ql_task_status_t

The structure of task status:

```
typedef struct
{
  ql_task_t          xHandle;
  const char *       pcTaskName;
  ql_task_state_e    eCurrentState;
  unsigned long      uxCurrentPriority;
  uint16             usStackHighWaterMark;
} ql_task_status_t;
```

- **Parameter**

Type	Parameter	Description
ql_task_t	*xHandle*	Task handle.
const char	*pcTaskName*	Task name.
ql_task_state_e	*eCurrentState*	The current task state. See ***Chapter 3.3.9.2*** for details.

unsigned long	*uxCurrentPriority*	Task running priority.
uint16	*usStackHighWaterMark*	The minimum amount of stack space left by the task from the beginning of the task creation. The closer the value is to zero, the closer the task stack will be to overflow.

3.3.9.2. ql_task_state_e

The enumeration of the current task state:

```
typedef enum
{
  Running = 0,
  Ready,
  Blocked,
  Suspended,
  Deleted,
  Invalid
} ql_task_state_e;
```

- **Member**

Member	Description
Running	Task is running.
Ready	Task is ready.
Blocked	Task is blocked.
Suspended	Task is suspended definitively or indefinitely.
Deleted	Task is deleted but TCB is not released.
Invalid	Task is invalid.

3.3.10. ql_rtos_task_sleep_ms

This function sets task sleep time in milliseconds.

- **Prototype**

```
extern void ql_rtos_task_sleep_ms (uint32 ms);
```

- **Parameter**

ms:
[In] Task sleep time. Unit: millisecond.

- **Return Value**

None

3.3.11. ql_rtos_task_sleep_s

This function sets task sleep time in seconds.

- **Prototype**

```
extern void ql_rtos_task_sleep_s (uint32 s);
```

- **Parameter**

s:
[In] Task sleep time. Unit: second.

- **Return Value**

None

3.3.12. ql_rtos_enter_critical

This function enters the critical section protection.

- **Prototype**

```
extern uint32_t ql_rtos_enter_critical(void);
```

- **Parameter**

None

- **Return Value**

The number of critical sections.

3.3.13. ql_rtos_exit_critical

This function exits the critical section protection.

- **Prototype**

```
extern void ql_rtos_exit_critical(uint32_t critical);
```

- **Parameter**

critical:
[In] The number of critical sections, obtained by *ql_rtos_enter_critical()*.

- **Return Value**

None

3.3.14. ql_rtos_semaphore_create

This function creates a semaphore.

- **Prototype**

```
extern QlOSStatus ql_rtos_semaphore_create (ql_sem_t *semaRef, uint32 initialCount);
```

- **Parameter**

semaRef:
[Out] Semaphore.

initialCount:
[In] Initial semaphore.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SEMA_CREATE_FAILE	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.15. ql_rtos_semaphore_wait

This function sets the waiting time in milliseconds for a semaphore.

- **Prototype**

```
extern QlOSStatus ql_rtos_semaphore_wait (ql_sem_t semaRef, uint32 timeout);
```

- **Parameter**

semaRef:
[In] Semaphore.

timeout:
[In] The waiting time for the semaphore. Unit: millisecond. In addition to the following values, you can also set the waiting time according to your requirements.

QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SEMA_GET_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.16. ql_rtos_semaphore_release

This function releases a semaphore.

- **Prototype**

```
extern QlOSStatus ql_rtos_semaphore_release (ql_sem_t semaRef);
```

- **Parameter**

semaRef:
[In] Semaphore.

- **Return Value**

QL_OSI_SEMA_RELEASE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.17. ql_rtos_semaphore_get_cnt

This function gets the semaphore value.

- **Prototype**

```
extern QlOSStatus ql_rtos_semaphore_get_cnt (ql_sem_t semaRef, uint32 * cnt_ptr);
```

- **Parameter**

semaRef:
[In] Semaphore.

cnt_ptr:
[Out] The output semaphore.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SEMA_GET_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.18. ql_rtos_semaphore_delete

This function deletes a semaphore.

- **Prototype**

```
extern QlOSStatus ql_rtos_semaphore_delete (ql_sem_t semaRef );
```

- **Parameter**

semaRef:
[In] Semaphore.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution

3.3.19. ql_rtos_mutex_create

This function creates a mutex.

- **Prototype**

```
extern QlOSStatus ql_rtos_mutex_create (ql_mutex_t *mutexRef);
```

- **Parameter**

mutexRef:
[Out] Mutex.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_MUTEX_CREATE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.20. ql_rtos_mutex_lock

This function locks a mutex. You can set the waiting time according to your requirements.

- **Prototype**

```
extern QlOSStatus ql_rtos_mutex_lock (ql_mutex_t mutexRef, uint32 timeout);
```

- **Parameter**

mutexRef:
[In] Mutex.

timeout:
[In] Mutex waiting time. Unit: millisecond. In addition to the following values, you can also set the waiting time according to your requirements.

QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_MUTEX_LOCK_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.21. ql_rtos_mutex_try_lock

This function tries to lock a mutex.

- **Prototype**

```
extern QlOSStatus ql_rtos_mutex_try_lock (ql_mutex_t mutexRef);
```

- **Parameter**

mutexRef:
[In] Mutex.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_MUTEX_LOCK_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.22. ql_rtos_mutex_unlock

This function unlocks a mutex.

- **Prototype**

```
extern QlOSStatus ql_rtos_mutex_unlock (ql_mutex_t mutexRef);
```

- **Parameter**

mutexRef:
[In] Mutex.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution

3.3.23. ql_rtos_mutex_delete

This function deletes a mutex.

- **Prototype**

```
extern QlOSStatus ql_rtos_mutex_delete (ql_mutex_t mutexRef);
```

- **Parameter**

mutexRef:
[In] Mutex.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution

3.3.24. ql_rtos_queue_create

This function creates a message queue.

- **Prototype**

```
extern QlOSStatus ql_rtos_queue_create (ql_queue_t *msgQRef, uint32 maxSize, uint32
maxNumber);
```

- **Parameter**

msgQRef:
[In] Message queue.

maxSize:
[In] The maximum message size supported by the message queue. Unit: byte.

maxNumber:
[In] The maximum number of messages in the message queue.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution
QL_OSI_QUEUE_CREATE_FAIL	Failed execution

3.3.25. ql_rtos_queue_wait

This function waits for messages in the queue.

- **Prototype**

```
extern QlOSStatus ql_rtos_queue_wait (ql_queue_t msgQRef, uint8 *recvMsg, uint32 size, uint32
timeout);
```

- **Parameter**

msgQRef:
[In] Message queue.

recvMsg:
[In] The pointer to receive the message.

size:
[In] Invalid parameter and it is only compatible with other Quectel modules. The value is equal to *maxSize* when the queue is created.

timeout:
[In] The waiting time of the message queue. Unit: millisecond. In addition to the following values, you can also set the waiting time according to your requirements.

QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_QUEUE_RECEIVE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.26. ql_rtos_queue_release

This function releases a message queue.

- **Prototype**

```
extern QlOSStatus ql_rtos_queue_release (ql_queue_t msgQRef, uint32 size, uint8 *msgPtr, uint32
timeout);
```

- **Parameter**

msgQRef:
[In] Message queue.

size:
[In] Message size. Unit: byte.

msgPtr:
[In] The starting address of the data to be sent.

timeout:
[In] The waiting time of the message queue. Unit: millisecond. In addition to the following values, you can also set the waiting time according to your requirements.

QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_QUEUE_RELEASE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.27. ql_rtos_queue_get_cnt

This function gets the number of messages in the queue.

- **Prototype**

```
extern QlOSStatus ql_rtos_queue_get_cnt (ql_queue_t msgQRef, uint32 *cnt_ptr);
```

- **Parameter**

msgQRef:
[In] Message queue.

cnt_ptr:
[Out] The number of messages saved in the queue.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_QUEUE_GET_CNT_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.28. ql_rtos_queue_delete

This function deletes a message queue.

- **Prototype**

```
extern QlOSStatus ql_rtos_queue_delete (ql_queue_t msgQRef);
```

- **Parameter**

msgQRef:
[In] Message queue.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution

3.3.29. ql_rtos_timer_create

This function creates a timer.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_create(ql_timer_t* timerRef, ql_task_t taskRef,void
(*callBackRoutine)(void *), void *timerArgc)
```

- **Parameter**

timerRef:
[In] Timer.

taskRef:
[Out] The location to run the timer callback function, the values are as below:

NULL: The timer callback function runs in the interrupt and cannot be blocked. The running time should be as short as possible.

QL_TIMER_IN_SERVICE: The timer callback function runs in the system timer tasks and cannot be blocked. The running time should be as short as possible.

Specified task handle: The timer callback function runs in the specified task. However, there should be *ql_event_wait()* or *ql_event_try_wait()* in this task. When the timer expires, the system sends an event with ID 1 to the task. After receiving the event, the *ql_event_wait()* or *ql_event_try_wait()* will automatically run the timer callback function. It is recommended to use this method to run the timer.

callBackRoutine:
[In] Timer callback function.

timerArgc:
[in] The parameter of the timer callback function.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_TIMER_CREATE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.30. ql_rtos_timer_start

This function starts a timer. The time precision is in milliseconds. After the timer expires, the system will be awakened from sleep mode.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_start (ql_timer_t timerRef, uint32 set_Time, unsigned char cyclicalEn);
```

- **Parameter**

timerRef:
[In] Timer.

set_Time:
[In] Timer timeout. Unit: millisecond.

cyclicalEn:
[In] Whether to enable timer cycle mode.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_TIMER_START_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.31. ql_rtos_timer_start_relaxed

This function starts a timer. The value of *relax_time* determines whether the system will be awakened from sleep mode after the timer expires.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_start_relaxed (ql_timer_t timerRef, uint32 set_Time, unsigned char cyclicalEn, uint32 relax_time);
```

- **Parameter**

timerRef:
[In] Timer

set_Time:
[In] Timer expiration time. Unit: millisecond.

cyclicalEn:
[In] Whether to enable the timer cycle mode.

relax_time:
[In] Timer wake-up time.

> *0*: The effect of the timer startup function is the same as that of *ql_rtos_timer_start*.
> Other non-zero parameters: When the timer expires, the system will not be awakened from sleep mode, but it will be awakened after the time passes in *relax_time*. If the system is awakened by other wakeup sources during waiting, the timer callback function will be executed immediately.
> *QL_WAIT_FOREVER*: The system will not be awakened by the timer.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_TIMER_START_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.32. ql_rtos_timer_start_us

This function starts a timer and the time precision is in microseconds. After the timer expires, the system will be awakened from sleep mode.

- **Prototype**

```
QlOSStatus ql_rtos_timer_start_us (ql_timer_t timerRef, uint32 set_Time_us);
```

- **Parameter**

timerRef:
[In] Timer

set_Time_us:
[In] Timer expiration time. Unit: microsecond.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_TIMER_START_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.33. ql_rtos_timer_stop

This function stops a timer.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_stop (ql_timer_t timerRef);
```

- **Parameter**

timerRef:
[In] Timer.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_TIMER_STOP_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.34. ql_rtos_timer_is_running

This function determines whether the timer is running.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_is_running (ql_timer_t timerRef);
```

- **Parameter**

timerRef:
[In] Timer.

- **Return Value**

1	The timer is running.
0	The timer is not running.

3.3.35. ql_rtos_timer_delete

This function deletes a timer.

- **Prototype**

```
extern QlOSStatus ql_rtos_timer_delete (ql_timer_t timerRef );
```

- **Parameter**

timerRef:
[In] Timer

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SUCCESS	Successful execution

3.3.36. ql_rtos_event_send

This function sends an event.

- **Prototype**

```
extern QlOSStatus ql_rtos_event_send(ql_task_t task_ref, ql_event_t *event);
```

- **Parameter**

task_ref:
[In] Task handle.

event:
[In] The structure of events. It should not be NULL. See ***Chapter 3.3.36.1*** for details.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_EVENT_SEND_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.36.1. ql_event_t

The structure of events:

```
typedef struct
{
uint32 id;
uint32 param1;
uint32 param2;
uint32 param3;
} ql_event_t
```

- **Parameter**

Type	Parameter	Description
uint32	*id*	Event identifier. It should be at least 1000. To ensure identifier uniqueness, it is recommended to use the methods in *ql_osi_def.h* in *components\ql-kernel\inc* directory in SDK to define it.
uint32	*param1*	Parameter 1.
uint32	*param2*	Parameter 2.
uint32	*param3*	Parameter 3.

3.3.37. ql_event_wait

This function waits for an event. You can set the waiting time according to your requirements.

- **Prototype**

```
extern QlOSStatus ql_event_wait(ql_event_t* event_strc, uint32 timeout );
```

- **Parameter**

event_strc:
[In] The structure of events. It should not be NULL. See ***Chapter 3.3.36.1*** for details.

timeout:
[In] Event waiting time. Unit: millisecond. In addition to the following values, you can also set the waiting time according to your requirements.

QL_WAIT_FOREVER	Keep waiting
QL_NO_WAIT	Do not wait

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_EVENT_GET_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.38. ql_event_try_wait

This function tries to wait for an event.

- **Prototype**

```
extern QlOSStatus ql_event_try_wait (ql_event_t *event_strc);
```

- **Parameter**

event_strc:
[In] The structure of events. It should not be NULL. See ***Chapter 3.3.36.1*** for details.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_EVENT_GET_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.39. ql_rtos_swdog_register

This function registers a software watchdog for a task.

- **Prototype**

```
QlOSStatus ql_rtos_swdog_register(ql_swdog_callback callback, ql_task_t taskRef);
```

- **Parameter**

callback:
[In] Software watchdog callback function. This callback function will be called by the software watchdog upon the completion of one feeding cycle. See ***Chapter 3.3.40.1*** for details

taskRef:
[In] The task handle that needs to register tasks.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SWDOG_REGISTER_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.40. ql_rtos_swdog_unregister

This function unregisters the software watchdog for a task.

- **Prototype**

```
QlOSStatus ql_rtos_swdog_unregister(ql_swdog_callback callback)
```

- **Parameter**

callback:
[In] Software watchdog callback function, i.e., the callback function at registration. See ***Chapter 3.3.40.1*** for details.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SWDOG_UNREGISTER_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.40.1. ql_swdog_callback

This function is software watchdog callback function, which will be called by the software watchdog upon the completion of one feeding cycle.

- **Prototype**

```
typedef void (*ql_swdog_callback)(uint32 id_type, void *ctx);
```

- **Parameter**

id_type:
[Out] Dog-feeding Identifier, sending *QUEC_KERNEL_FEED_DOG* to the application layer.

ctx:
[Out] The task handle of the task that needs to feed the dog this time.

- **Return Value**

None

3.3.41. ql_rtos_sw_dog_enable

This function enables the software watchdog.

- **Prototype**

```
QlOSStatus ql_rtos_sw_dog_enable(uint32 period_ms, uint32 missed_cnt)
```

- **Parameter**

period_ms:
[In] Software watchdog running cycle. After a cycle ends, each callback function registered by *ql_rtos_swdog_register* will be called.

missed_cnt:
[In] The maximum number of times that the watchdog has not been fed.

- **Return Value**

QL_OSI_TASK_PARAM_INVALID	Invalid parameter
QL_OSI_SWDOG_ENABLE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.42. ql_rtos_sw_dog_disable

This function disables the software watchdog.

- **Prototype**

```
QlOSStatus ql_rtos_sw_dog_disable(void)
```

- **Parameter**

None

- **Return Value**

QL_OSI_SWDOG_DISABLE_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.43. ql_rtos_feed_dog

This function feeds the software watchdog for the current task and is called in the task that needs to feed the watchdog.

- **Prototype**

```
QlOSStatus ql_rtos_feed_dog(void)
```

- **Parameter**

None

- **Return Value**

QL_OSI_SWDOG_FEED_DOG_FAIL	Failed execution
QL_OSI_SUCCESS	Successful execution

3.3.44. ql_gettimeofday

This function gets the exact time.

- **Prototype**

```
QlOSStatus ql_gettimeofday (ql_timeval_t *timeval);
```

- **Parameter**

timeval:
[Out] Exact time. See ***Chapter 3.3.44.1*** for details.

- **Return Value**

QL_OSI_SUCCESS	Successful execution

3.3.44.1. ql_timeval_t

The structure of the exact time:

```
typedef struct
{
  uint32 sec;
  uint32 usec;
}ql_timeval_t;
```

- **Parameter**

Type	Parameter	Description
uint32	*sec*	The total time from January 1, 1970, to the present. Unit: second.
uint32	*usec*	The fractional part of the total time from January 1, 1970, to the present, represented by microseconds. Range: 0–99999.

3.3.45. ql_rtos_get_system_tick

This function gets the tick count of RTOS.

- **Prototype**

```
uint32 ql_rtos_get_system_tick(void);
```

- **Parameter**

None

- **Return Value**

Greater than *0*	Total tick count
Less than *0*	Failed execution

3.3.46. ql_assert

This function forces the module to dump.

- **Prototype**

```
void ql_assert(void);
```

- **Parameter**

None

- **Return Value**

None

3.3.47. ql_rtos_up_time_ms

This function obtains the current time since the system was started. This time does not stop when the system is awakened from deep sleep. It is used for time synchronization of littlevGL.

- **Prototype**

```
int64_t ql_rtos_up_time_ms();
```

- **Parameter**

None

- **Return Value**

The current time since the system was started.

4 Appendix References

Table 10: Related Document

Document Name
[1] Quectel_EC200U&EG91xU_Series_QuecOpen_CSDK_Quick_Start_Guide

Table 11: Terms and Abbreviations

Abbreviation	Description
API	Application Programming Interface
CPU	Central Processing Unit
ID	Identifier
PC	Personal Computer
RAM	Random Access Memory
ROM	Read-Only Memory
RTOS	Real Time Operating System
SDK	Software Development Kit
TCB	Task Control Block
OS	Operating System