ECx00U&EGx00U 系列

QuecOpen RTOS 开发指导

LTE Standard 系列

版本：1.0

日期：2021-10-12

状态：受控文件

上海移远通信技术股份有限公司（以下简称“移远通信”）始终以为客户提供最及时、最全面的服务为宗旨。如需任何帮助，请随时联系我司上海总部，联系方式如下：

上海移远通信技术股份有限公司
上海市闵行区田林路 1016 号科技绿洲 3 期（B 区）5 号楼　　邮编：200233
电话：+86 21 5108 6236　　邮箱：info@quectel.com

或联系我司当地办事处，详情请登录：http://www.quectel.com/cn/support/sales.htm。

如需技术支持或反馈我司技术文档中的问题，请随时登陆网址：
http://www.quectel.com/cn/support/technical.htm 或发送邮件至：support@quectel.com。

前言

移远通信提供该文档内容以支持客户的产品设计。客户须按照文档中提供的规范、参数来设计产品。同时，您理解并同意，移远通信提供的参考设计仅作为示例。您同意在设计您目标产品时使用您独立的分析、评估和判断。在使用本文档所指导的任何硬软件或服务之前，请仔细阅读本声明。您在此承认并同意，尽管移远通信采取了商业范围内的合理努力来提供尽可能好的体验，但本文档和其所涉及服务是在“可用”基础上提供给您的。移远通信可在未事先通知的情况下，自行决定随时增加、修改或重述本文档。

使用和披露限制

许可协议

除非移远通信特别授权，否则我司所提供硬软件、材料和文档的接收方须对接收的内容保密，不得将其用于除本项目的实施与开展以外的任何其他目的。

版权声明

移远通信产品和本协议项下的第三方产品可能包含受移远通信或第三方材料、硬软件和文档版权保护的相关资料。除非事先得到书面同意，否则您不得获取、使用、向第三方披露我司所提供的文档和信息，或对此类受版权保护的资料进行复制、转载、抄袭、出版、展示、翻译、分发、合并、修改，或创造其衍生作品。移远通信或第三方对受版权保护的资料拥有专有权，不授予或转让任何专利、版权、商标或服务商标权的许可。为避免歧义，除了正常的非独家、免版税的产品使用许可，任何形式的购买都不可被视为授予许可。对于任何违反保密义务、未经授权使用或以其他非法形式恶意使用所述文档和信息的违法侵权行为，移远通信有权追究法律责任。

商标

除另行规定，本文档中的任何内容均不授予在广告、宣传或其他方面使用移远通信或第三方的任何商标、商号及名称，或其缩略语，或其仿冒品的权利。

第三方权利

您理解本文档可能涉及一个或多个属于第三方的硬软件和文档（“第三方材料”）。您对此类第三方材料的使用应受本文档的所有限制和义务约束。

移远通信针对第三方材料不做任何明示或暗示的保证或陈述，包括但不限于任何暗示或法定的适销性或特定用途的适用性、平静受益权、系统集成、信息准确性以及与许可技术或被许可人使用许可技术相关的不侵犯任何第三方知识产权的保证。本协议中的任何内容都不构成移远通信对任何移远通信产品或任何其他硬软件、设备、工具、信息或产品的开发、增强、修改、分销、营销、销售、提供销售或以其他方式维持生产的陈述或保证。此外，移远通信免除因交易过程、使用或贸易而产生的任何和所有保证。

免责声明

1) 移远通信不承担任何因未能遵守有关操作或设计规范而造成损害的责任。

2) 移远通信不承担因本文档中的任何因不准确、遗漏、或使用本文档中的信息而产生的任何责任。

3) 移远通信尽力确保开发中功能的完整性、准确性、及时性，但不排除上述功能错误或遗漏的可能。除非另有协议规定，否则移远通信对开发中功能的使用不做任何暗示或法定的保证。在适用法律允许的最大范围内，移远通信不对任何因使用开发中功能而遭受的损害承担责任，无论此类损害是否可以预见。

4) 移远通信对第三方网站及第三方资源的信息、内容、广告、商业报价、产品、服务和材料的可访问性、安全性、准确性、可用性、合法性和完整性不承担任何法律责任。

文档历史

修订记录

版本	日期	作者	变更表述
-	2020-11-11	Kevin WANG	文档创建
1.0	2021-10-12	Kevin WANG	受控版本

目录

表格索引

图片索引

1 引言

移远通信 LTE Standard ECx00U 系列和 EGx00U 模块支持 QuecOpen® 方案；QuecOpen® 是基于 RTOS 的嵌入式开发平台，可简化 IoT 应用的软件设计和开发过程。有关 QuecOpen® 的详细信息，请参考 ***文档 [1]***。

本文档主要提供 QuecOpen® 方案下，移远通信 ECx00U 系列和 EGx00U 模块的 RTOS 开发应用指导。

1.1. 适用模块

表 1：适用模块

模块系列	模块
ECx00U	EC200U 系列
	EC600U 系列
EGx00U	EG500U-CN
	EG700U-CN

2 RTOS 简介

2.1. RTOS 定义

对计算机响应时间有要求的系统，通常称为临界系统或应用。为了满足临界系统或应用对计算机响应时间的要求，人们开发了实时操作系统（Real Time Operation System）。

实时操作系统指所有任务均在规定时间内完成的操作系统，同时任务完成必须满足时序可预测性。需要注意的是，实时操作系统的本质是其反应具有时序可预测性。得益于此，实时操作系统通常反应非常快。

实时操作系统的内核对象一般包括任务、信号量、互斥锁、消息队列、定时器、事件通知和软件看门狗等。下文中，实时操作系统统称为 RTOS。

2.2. 任务（Task）

在实际生活中处理一个大而复杂的问题时，惯用的方法是“分而治之”，即把一个大问题分解成多个相对简单、比较容易解决的小问题，小问题被解决了，大问题也就随之被解决了。同样，在设计一个较为复杂的应用程序时，也通常把一个大型任务分解成多个小任务，然后在计算机中运行这些小任务，最终达到完成大任务的目的。这种方法可以使操作系统并发地运行多个任务，从而提高处理器的利用率。

2.2.1. 任务及其内存结构

RTOS 的任务通常由三个部分组成：任务控制块、任务堆栈和任务代码。其中，任务控制块关联了任务程序代码，并记录任务的各个属性；任务堆栈用来保存任务的工作环境。下图是 RTOS 任务组成的示意图。

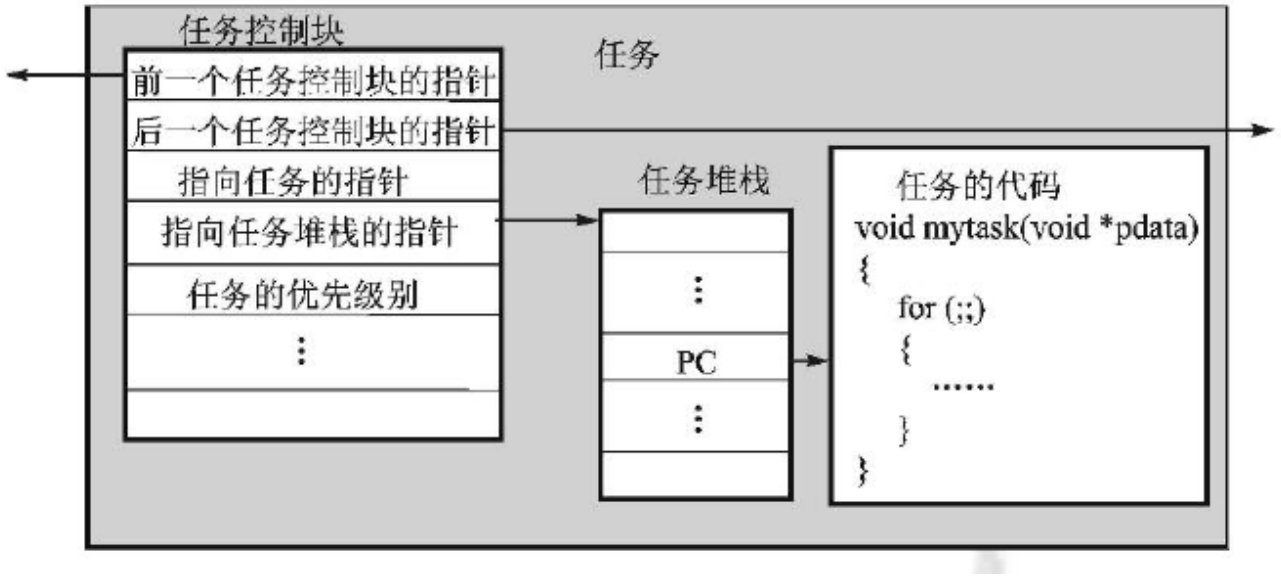

图 1：RTOS 任务组成

多个任务的控制块一般通过链表方式组织，任务控制块链表如下图所示：

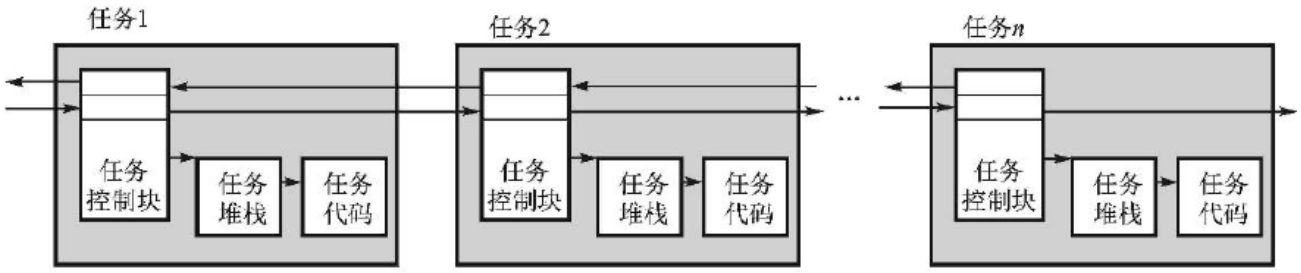

图 2：任务控制块链表

2.2.2. 任务状态

任务状态及描述如下表所示。

表 2：任务状态表

任务状态	描述
睡眠状态	任务只是以代码的形式驻留在程序空间（ROM 或 RAM），还没有交给操作系统管理时的情况叫做睡眠状态。简单地说，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态叫做任务的睡眠状态。
就绪状态	如果系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，则任务就具备了

	运行的充分条件，这时任务的状态叫做就绪状态。
运行状态	处于就绪状态的任务经调度器判断获得了 CPU 的使用权，则任务进入运行状态。 任何时刻只能有一个任务处于运行状态，就绪的任务只有当所有优先级高于本任务的任务都转为等待状态时，才能进入运行状态。
等待状态	正在运行的任务需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给其他任务而使任务进入等待状态。
中断服务状态	一个正在运行的任务一旦响应中断申请就会暂停运行而去执行中断服务程序，这时任务的状态就做中断服务状态。

一个任务可以在 5 个不同状态之间相互转换，转换关系如下图所示：

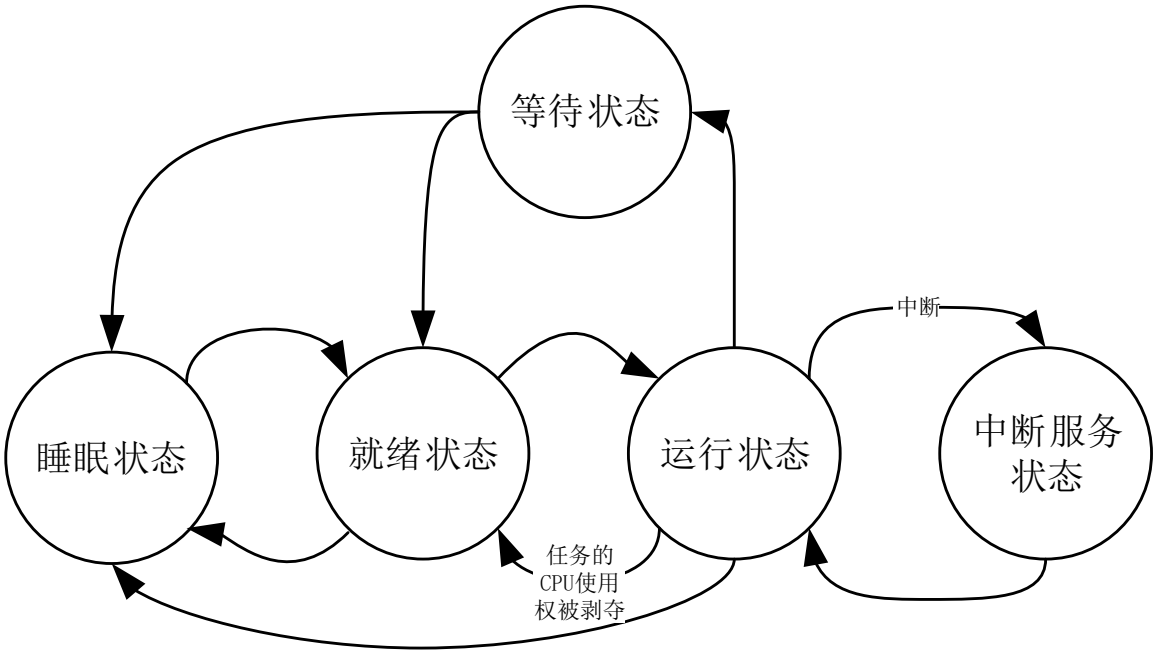

图 3：任务状态切换

2.2.3. 任务的创建和删除

模块提供的 RTOS 任务创建和删除接口如下，有关 API 详情，参见***第 3 章***。

```
//任务创建接口
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        */
);

//任务删除接口
QlOSStatus ql_rtos_task_delete
(
    ql_task_t taskRef       /* OS task reference    */
);
```

创建任务时需要传递的参数有：任务句柄指针、任务栈大小、任务优先级、任务名称、任务的入口函数及传递给任务的参数。其中，任务栈大小根据任务代码而定。

模块的任务优先级的范围是 0~30，30 是最高优先级。应用层的任务优先级请不要高于 30。

任务的代码结构一般遵循以下格式：

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

    ql_rtos_task_delete(NULL);
}
```

备注

1. 在任务中，如需退出 *while(1)*结构，退出后必须删除当前的任务，以释放任务控制块和任务栈所占用的资源。通过向 *ql_rtos_task_delete(ql_task_t taskRef)*传递 NULL 参数来删除当前正在运行的任务，即任务自身，实现任务的自杀。
2. *ql_rtos_task_delete(ql_task_t taskRef)*函数中传入的参数非 NULL 时，删除任务句柄指向的任务。

模块提供的 RTOS 任务管理相关的接口如下表所示，有关 API 详情，参见***第 3 章***。

表 3：任务管理接口

API 名称	描述
ql_rtos_task_create()	创建任务。

ql_rtos_task_create_default()	创建任务，默认事件数量为 23。
ql_rtos_task_delete()	删除任务，参数为 NULL 时，删除任务自身。
ql_rtos_task_suspend()	挂起任务。
ql_rtos_task_resume()	使任务退出挂起状态。
ql_rtos_task_yield()	释放 CPU 使用权。
ql_rtos_task_get_current_ref()	获取当前任务的任务句柄。
ql_rtos_task_change_priority()	切换任务优先级。
ql_rtos_task_get_status()	获取任务状态。
ql_rtos_task_sleep_ms()	设置任务休眠时间，休眠时间以毫秒为单位。
ql_rtos_task_sleep_s()	设置任务休眠时间，休眠时间以秒为单位。
ql_rtos_enter_critical()	进入临界区保护。
ql_rtos_exit_critical()	退出临界区保护。

2.3. 信号量（Semaphore）

一个复杂的应用程序拆分为多个小任务时，要求多个小任务之间能够很好的协调和同步。比如一个应用被拆分为任务 A 和任务 B，任务 B 需要等待任务 A 完成某个操作之后才能进行后续工作，此时任务 B 可以处于等待状态，直到任务 A 完成特定的操作后，向任务 B 发送一个信号，通知任务 B 可以执行后续工作。

信号量不仅具有信号的通知作用，还有量的概念。如果任务 A 向任务 B 连续发送 n 次信号，则任务 B 能够循环执行 n 次。信号量在多任务同步机制中十分重要。

下面的伪代码演示了如何使用信号量：

```
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);
        ...
    }
}
```

模块提供的 RTOS 信号量相关的接口如下表所示，有关 API 详情，参见***第 3 章***。

表 4：信号量接口

API 名称	描述
ql_rtos_semaphore_create()	创建信号量。
ql_rtos_semaphore_wait()	设置信号量等待时间，等待时间以毫秒为单位。
ql_rtos_semaphore_release()	释放信号量。
ql_rtos_semaphore_get_cnt()	获取信号量值。
ql_rtos_semaphore_delete()	删除信号量。

2.4. 互斥锁（Mutex）

互斥锁是一个二值型信号量，即发送出来的信号值最大为 1。它用于保护共享资源不被多个任务同时访问，以免造成共享资源的上下文在同一个任务访问前后不一致，产生非预期的结果，而这些非预期的结果有时是致命的。所以互斥锁在保护共享资源时尤为重要。

请看下面两段伪代码：

```
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();
    ...
}
```

上述代码中，如果 taskA 在执行完 lableA 前的 if 语句后产生了一次任务调度，CPU 使用权被切到 taskB，taskB 执行完全部 if 语句中的程序后，再调度到 taskA，此时 taskA 打印出的 bFlag 值将不再是我们期望的 TRUE，而是 FALSE，异常产生。

如果将两个任务的 if 代码块前后的互斥锁反注释，一个 if 代码块将是一组原子操作，将不会出现上述异常。

模块提供的 RTOS 互斥锁相关接口如下表所示，有关 API 详情，参见***第 3 章***。

表 5：互斥锁接口

API 名称	描述
ql_rtos_mutex_create()	创建互斥锁。
ql_rtos_mutex_lock()	获取互斥锁，等待时间用户可以根据需求进行自定义。
ql_rtos_mutex_try_lock()	尝试获得互斥锁，等待时间为永久等待。

ql_rtos_mutex_unlock()	释放互斥锁。
ql_rtos_mutex_delete()	删除互斥锁。

2.5. 消息队列（Queue）

多任务之间除了发送信号量的方式进行沟通交流之外，还可以通过消息队列进行消息传递。消息队列用于一个任务向另外一个任务传递消息。传递的消息方式是将待传递的消息根据消息的起始地址和消息大小，拷贝到消息队列中，然后向等待消息的任务发送信号；等待消息的线程将消息队列中的消息拷贝到本地的缓冲区中，然后将队列中的消息移除。

下面的伪代码演示了消息队列的用法：

```
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);
        ...
    }
}
```

模块提供的 RTOS 消息队列相关接口如下表所示，有关 API 详情，参见***第 3 章***。

表 6：消息队列接口

API 名称	描述
ql_rtos_queue_create()	创建消息队列。
ql_rtos_queue_wait()	等待队列中的消息，等待时间以毫秒为单位。
ql_rtos_queue_release()	释放消息队列。
ql_rtos_queue_get_cnt()	获取队列中的消息数量。
ql_rtos_queue_delete()	删除消息队列。

2.6. 定时器（Timer）

定时器，顾名思义，就是用来定时的。定时时间到达后，通过回调的形式通知到用户。模块的定时器既支持在系统服务中运行回调，也支持在指定的任务中运行回调。

模块提供的 RTOS 定时器相关接口如下表所示，有关 API 详情，参见***第 3 章***。

表 7：定时器接口

API 名称	描述
ql_rtos_timer_create()	创建定时器。
ql_rtos_timer_start()	启动定时器，时间精度为毫秒。定时器超时后，若系统处于休眠状态，将唤醒系统。
ql_rtos_timer_start_relaxed()	启动定时器，休眠模式下，由参数 *relax_time* 的值决定定时器超时后，是否唤醒系统。
ql_rtos_timer_start_us()	启动定时器，时间精度为微秒。定时器超时后，若系统处于休眠状态，将唤醒系统。
ql_rtos_timer_stop()	停止定时器。
ql_rtos_timer_is_running()	判断定时器是否正在计时。
ql_rtos_timer_delete()	删除定时器。

2.7. 事件通知（Event）

事件通知是模块的线程核心通讯方法。事件通知可以结合信号量、队列、事件标志等通讯方法的多数优点。模块的大部分 RTOS 功能，如软件看门狗、定时器的运行，也可以通过事件通知实现。因此建议如果线程间涉及数据交互，可以使用事件通知的方式实现。

事件通知的函数请参考***第 3 章***。任务的代码结构一般遵循以下格式：

```
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 == …(具体事件 id))
        {
            //具体事件句柄
        }
}
}
```

模块提供的 RTOS 任务管理相关的接口如下表所示，有关 API 详情，参见***第 3 章***。

表 8：事件通知接口

API 名称	描述
ql_rtos_event_send()	发送事件。
ql_event_wait()	等待事件，等待时间用户可以根据需求进行自定义。
ql_event_try_wait()	等待事件，等待时间为永久等待。

2.8. 软件看门狗（Software Watch Dog）

某个任务陷入死循环，或者一直占用 CPU，导致比这个任务优先级低的任务永远得不到调度，发生这种情况时，软件看门狗将重启模块。

用户创建任务后，可以调用软件看门狗绑定函数，将需要绑定的任务，以及对应的看门狗回调函数，绑定到操作系统。若此时软件看门狗未使能，可调用使能函数，配置好喂狗周期，以及最大喂狗丢失次数。配置完成后，系统将以使能时指定的喂狗周期运行，循环调用绑定时指定的看门狗回调函数，传入回调函数的参数为喂狗的事件 ID，以及需要进行喂狗的任务的任务句柄。用户可以在此回调函数中，通过发送事件，或者其他 RTOS 通信方式，通知需要喂狗的任务，调用喂狗函数进行喂狗。

下面代码演示了软件看门狗的用法：

```
void sw_dog_create()
{
    ql_rtos_swdog_register((ql_swdog_callback)feed_dog_callback, demo_task);
    ql_rtos_sw_dog_enable(5000, 3); //喂狗周期为 5 秒，最大丢失次数为 3
}
void feed_dog_callback(uint32 id_type, void *ctx) //软件看门狗的回调函数
{
    ql_event_t event;
    event.id = QUEC_KERNEL_FEED_DOG;
    ql_rtos_event_send(demo_task, &event); //给指定的任务发喂狗事件
}
void   demo_task_callback(void *argv)   //任务执行函数
{
    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(); //喂狗
    }
…
}
```

模块提供的软件看门狗 API，参见***第 3 章***。

表 9：软件看门狗接口

API 名称	描述
ql_rtos_swdog_register()	绑定软件看门狗。
ql_rtos_swdog_unregister()	解绑软件看门狗。
ql_rtos_sw_dog_enable()	使能软件看门狗功能。
ql_rtos_sw_dog_disable()	禁用软件看门狗功能。
ql_rtos_feed_dog()	给当前执行的任务喂狗，在需要喂狗的任务中调用。

3 RTOS 相关 API

3.1. 数据类型

3.1.1. QlOSStatus

```
typedef int QlOSStatus;     //RTOS API 返回值的数据类型
```

3.1.2. ql_task_t

```
typedef void * ql_task_t;     //任务句柄类型
```

3.1.3. ql_sem_t

```
typedef void * ql_sem_t;     //信号量句柄类型
```

3.1.4. ql_mutex_t

```
typedef void * ql_mutex_t;    //互斥锁句柄类型
```

3.1.5. ql_queue_t

```
typedef void * ql_queue_t;    //消息队列句柄类型
```

3.1.6. ql_timer_t

```
typedef void * ql_timer_t;     //定时器句柄类型
```

3.1.7. ql_wait_e

```
typedef enum
{
    QL_WAIT_FOREVER = 0xFFFFFFFF,
    QL_NO_WAIT            = 0
```

```
} ql_wait_e;
```

- **参数**

参数	描述
QL_WAIT_FOREVER	永久等待
QL_NO_WAIT	不等待

3.2. 头文件和参考示例

RTOS 相关 API 头文件为 *ql_api_osi*，位于 SDK 包 *components\ql-kernel\inc* 路径下。RTOS 相关 API 接口示例文件为 *osi_demo*，位于 SDK 包 *components\ql-application\osi* 路径下。

3.3. API 详解

3.3.1. ql_rtos_task_create

该函数用于创建任务。

- **函数原型**

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

- **参数**

taskRef：
[Out] 任务句柄。

stackSize：
[In] 任务栈大小，根据任务代码而定，最大值：128；单位：KB。

priority：
[In] 任务优先级，范围：0~30。

taskName：
[In] 任务名称，最大值：16；单位：字节。

taskStart：
[In] 任务的入口函数。

argv：
[In] 传递给任务的参数。

event_count：
[In] 任务中存放的最大待处理事件数量。

- **返回值**

QL_OSI_TASK_CREATE_FAIL：任务创建失败。
QL_OSI_TASK_NAME_INVALID：无效任务名。
QL_OSI_INVALID_TASK_REF：无效任务句柄。
QL_OSI_SUCCESS：任务创建成功。

3.3.2. ql_rtos_task_create_default

该函数用于创建任务，默认事件数量为 23。

- **函数原型**

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

- **参数**

taskRef：
[In] 任务句柄。

stackSize：
[In] 任务栈大小，根据任务代码而定，最大值：128；单位：KB。

priority：
[In] 任务优先级，范围：0~30。

taskName：
[In] 任务名称，最大值：16；单位：字节。

taskStart：
[In] 任务的入口函数。

argv：
[In] 传递给任务的参数。

- **返回值**

QL_OSI_TASK_CREATE_FAIL：任务创建失败。
QL_OSI_TASK_NAME_INVALID：无效任务名。
QL_OSI_INVALID_TASK_REF：无效任务句柄。
QL_OSI_SUCCESS：任务创建成功。

3.3.3. ql_rtos_task_delete

该函数用于删除任务。

- **函数原型**

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

- **参数**

taskRef：
[In] 任务句柄。参数为 NULL 时，删除任务自身；参数为非 NULL 时，删除任务句柄指向的任务。

- **返回值**

QL_OSI_INVALID_TASK_REF：无效任务句柄。
QL_OSI_SUCCESS：任务删除成功。

3.3.4. ql_rtos_task_suspend

该函数用于挂起任务。

- **函数原型**

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

- **参数**

taskRef：
[In] 任务句柄。

- **返回值**

QL_OSI_INVALID_TASK_REF：无效任务句柄。
QL_OSI_SUCCESS：任务挂起成功。

3.3.5. ql_rtos_task_resume

该函数用于使任务退出挂起状态。

- **函数原型**

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

- **参数**

taskRef：
[In] 任务句柄。

- **返回值**

QL_OSI_INVALID_TASK_REF：无效任务句柄。
QL_OSI_SUCCESS：任务退出挂起成功。

3.3.6. ql_rtos_task_yield

该函数用于释放 CPU 使用权。

- **函数原型**

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

- **参数**

无

- **返回值**

无

3.3.7. ql_rtos_task_get_current_ref

该函数用于获取当前任务的任务句柄。

- **函数原型**

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

- **参数**

taskRef：

[Out] 任务句柄。

- **返回值**

QL_OSI_TASK_GET_REF_FAIL：获取任务句柄失败。
QL_OSI_SUCCESS：获取任务句柄成功。

3.3.8. ql_rtos_task_change_priority

该函数用于切换任务优先级。

- **函数原型**

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

- **参数**

taskRef：
[In] 任务句柄。

new_priority：
[In] 任务的新优先级。

old_priority：
[In] 任务的原优先级。

- **返回值**

QL_OSI_TASK_GET_PRIO_FAIL：获取优先级失败。
QL_OSI_TASK_SET_PRIO_FAIL：设置优先级失败。
QL_OSI_SUCCESS：任务优先级切换成功。

3.3.9. ql_rtos_task_get_status

该函数用于获取任务状态。

- **函数原型**

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

- **参数**

taskref：
[In] 任务句柄。

status：
[Out] 任务状态。详情参见***第 3.3.9.1 章***。

- **返回值**

QL_OSI_INVALID_TASK_REF：任务句柄无效。
QL_OSI_SUCCESS：获取任务状态成功。

3.3.9.1. ql_task_status_t

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

- **参数**

类型	参数	描述
ql_task_t	*xHandle*	任务句柄。
const char	*pcTaskName*	任务名。
ql_task_state_e	*eCurrentState*	任务当前的状态。详情参见***第 3.3.9.2 章***。
unsigned long	*uxCurrentPriority*	任务运行优先级。
uint16	*usStackHighWaterMark*	从任务创建开始，该任务所剩余的最小堆栈空间数量。该值越接近零，任务堆栈将越接近溢出。

3.3.9.2. ql_task_state_e

```
typedef enum
{
  Running = 0,
  Ready,
  Blocked,
  Suspended,
  Deleted,
```

```
    Invalid
} ql_task_state_e;
```

- **参数**

参数	描述
Running	任务正在运行。
Ready	任务处于就绪状态。
Blocked	任务处于阻塞状态。
Suspended	任务处于挂起状态或无限期暂停状态。
Deleted	任务被删除，但未释放 TCB。
Invalid	无效状态。

3.3.10. ql_rtos_task_sleep_ms

该函数用于设置任务休眠时间。

- **函数原型**

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

- **参数**

ms：
[In] 任务休眠时间，单位：毫秒。

- **返回值**

无

3.3.11. ql_rtos_task_sleep_s

该函数用于设置任务休眠时间。

- **函数原型**

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

- **参数**

s：
[In] 任务休眠时间，单位：秒。

- **返回值**

无

3.3.12. ql_rtos_enter_critical

该函数用于进入临界区保护。

- **函数原型**

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

- **参数**

无

- **返回值**

临界区编号。

3.3.13. ql_rtos_exit_critical

该函数用于退出临界区保护。

- **函数原型**

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

- **参数**

critical：
[In] 临界区编号，由 *ql_rtos_enter_critical* 获得。

- **返回值**

无

3.3.14. ql_rtos_semaphore_create

该函数用于创建信号量。

- **函数原型**

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

- **参数**

semaRef：
[Out] 信号量。

initialCount：
[In] 初始信号量值。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SEMA_CREATE_FAILE：信号量创建失败。
QL_OSI_SUCCESS：创建成功。

3.3.15. ql_rtos_semaphore_wait

该函数用于设置信号量等待时间。

- **函数原型**

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

- **参数**

semaRef：
[In] 信号量。

timeout：
[In] 信号量等待时间；单位：毫秒。0xFFFFFFFF 表示永久等待。除以下取值外，用户还可自定义等待时间。
QL_WAIT_FOREVER 永久等待
QL_NO_WAIT 不等待

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SEMA_GET_FAIL：信号量获取失败。
QL_OSI_SUCCESS：信号量获取成功。

3.3.16. ql_rtos_semaphore_release

该函数用于释放信号量。

- **函数原型**

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

- **参数**

semaRef：
[In] 信号量。

- **返回值**

QL_OSI_SEMA_RELEASE_FAIL：信号量释放失败。
QL_OSI_SUCCESS：信号量释放成功。

3.3.17. ql_rtos_semaphore_get_cnt

该函数用于获取信号量值。

- **函数原型**

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

- **参数**

semaRef：
[In] 信号量。

cnt_ptr：
[Out] 输出的信号量值。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SEMA_GET_FAIL：信号量值获取失败。
QL_OSI_SUCCESS：信号量值获取成功。

3.3.18. ql_rtos_semaphore_delete

该函数用于删除信号量。

- **函数原型**

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

- **参数**

semaRef：
[In] 信号量。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SUCCESS：信号量删除成功。

3.3.19. ql_rtos_mutex_create

该函数用于创建互斥锁。

- **函数原型**

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

- **参数**

mutexRef：
[out] 互斥锁。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_MUTEX_CREATE_FAIL：互斥锁创建失败。
QL_OSI_SUCCESS：互斥锁创建成功。

3.3.20. ql_rtos_mutex_lock

该函数用于获取互斥锁，等待时间用户可以根据需求进行自定义。

- **函数原型**

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

- **参数**

mutexRef：
[In] 互斥锁。

timeout：
[In] 互斥锁等待时间；单位：毫秒。0xFFFFFFFF 表示永久等待。除以下取值外，用户还可自定义等待时间。
QL_WAIT_FOREVER 永久等待
QL_NO_WAIT 不等待

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_MUTEX_LOCK_FAIL：互斥锁获取失败。
QL_OSI_SUCCESS：互斥锁获取成功。

3.3.21. ql_rtos_mutex_try_lock

该函数用于尝试获得互斥锁，等待时间为永久等待。

- **函数原型**

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

- **参数**

mutexRef：
[In] 互斥锁。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_MUTEX_LOCK_FAIL：互斥锁获取失败。
QL_OSI_SUCCESS：互斥锁获取成功。

3.3.22. ql_rtos_mutex_unlock

该函数用于释放互斥锁。

- **函数原型**

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

- **参数**

mutexRef：
[In] 互斥锁。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SUCCESS：释放成功。

3.3.23. ql_rtos_mutex_delete

该函数用于删除互斥锁。

- **函数原型**

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

- **参数**

mutexRef：
[In] 互斥锁。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SUCCESS：删除成功。

3.3.24. ql_rtos_queue_create

该函数用于创建消息队列。

- **函数原型**

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

- **参数**

msgQRef：
[In] 消息队列。

maxSize：
[In] 消息队列支持的最大消息大小，单位：字节。

maxNumber：
[In] 消息队列中支持的最大消息条数。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SUCCESS：消息队列创建成功。
QL_OSI_QUEUE_CREATE_FAIL：消息队列创建失败。

3.3.25. ql_rtos_queue_wait

该函数用于等待队列中的消息。

- **函数原型**

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

- **参数**

msgQRef：
[In] 消息队列。

recvMsg：
[In] 接收消息指针。

size：
[In] 本参数无效，仅为兼容本司其他模块，该参数大小固定为创建队列时的 *maxSize* 大小。

timeout：
[In] 队列消息等待时间；单位：毫秒。0xFFFFFFFF 表示永久等待。除以下取值外，用户还可自定义等待时间。
QL_WAIT_FOREVER 永久等待
QL_NO_WAIT 不等待

- **返回值**

QL_OSI_TASK_PARAM_INVALID：参数无效。
QL_OSI_QUEUE_RECEIVE_FAIL：等待队列消息失败。
QL_OSI_SUCCESS：等待队列消息成功。

3.3.26. ql_rtos_queue_release

该函数用于释放消息队列。

- **函数原型**

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

- **参数**

msgQRef：
[In] 消息队列。

size：
[In] 消息大小，单位：字节。

msgPtr：
[In] 待发送数据的起始地址。

timeout：
[In] 队列释放等待时间；单位：毫秒。0xFFFFFFFF 表示永久等待。除以下取值外，用户还可自定义等待时间。
QL_WAIT_FOREVER 永久等待
QL_NO_WAIT 不等待

- **返回值**

QL_OSI_TASK_PARAM_INVALID：参数无效。
QL_OSI_QUEUE_RELEASE_FAIL：队列释放失败。
QL_OSI_SUCCESS：队列释放成功。

3.3.27. ql_rtos_queue_get_cnt

该函数用于获取队列中的消息数量。

- **函数原型**

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

- **参数**

msgQRef：
[In] 消息队列。

cnt_ptr：
[Out] 队列中保存的消息条数。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：参数无效。
QL_OSI_QUEUE_GET_CNT_FAIL：队列中的消息数量获取失败。
QL_OSI_SUCCESS：队列中的消息数量获取成功。

3.3.28. ql_rtos_queue_delete

该函数用于删除消息队列。

- **函数原型**

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

- **参数**

msgQRef：
[In] 消息队列。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：参数无效。
QL_OSI_SUCCESS：删除成功。

3.3.29. ql_rtos_timer_create

该函数用于创建定时器。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

taskRef：
[Out] 运行定时器回调函数的位置，取值如下：
NULL：表示在中断中运行回调函数，回调函数中不可阻塞、运行时间不可过长。
QL_TIMER_IN_SERVICE：该定时器的回调函数，在系统定时器服务任务中执行，回调函数中不可阻塞、运行时间不可过长。
指定任务句柄：在指定的任务中运行，要求该任务中有 *ql_event_wait()*或 *ql_event_try_wait()*接口。定时器定时结束，系统会发送 ID 为 1 的事件到该任务中，*ql_event_wait()*或 *ql_event_try_wait()*函数接收到该事件，会自动去运行定时器回调函数。建议用此方法运行定时器。

callBackRoutine：
[In] 定时器回调函数。

timerArgc：
[In] 定时器回调函数的参数。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_TIMER_CREATE_FAIL：创建失败。
QL_OSI_SUCCESS：创建成功。

3.3.30. ql_rtos_timer_start

该函数用于启动定时器，时间精度为毫秒。定时器超时后，若系统处于休眠状态，将唤醒系统。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

set_Time：
[In] 定时器超时时间，单位：毫秒。

cyclicalEn：
[In] 是否启用循环模式。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_TIMER_START_FAIL：定时器启动失败。
QL_OSI_SUCCESS：定时器启动成功。

3.3.31. ql_rtos_timer_start_relaxed

该函数用于启动定时器，休眠模式下，由参数 *relax_time* 的值决定定时器超时后，是否唤醒系统。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

set_Time：
[In] 定时器超时时间，单位：毫秒。

cyclicalEn：
[In] 是否启用循环模式。

relax_time：
[In] 定时器唤醒时间。
0：定时器启动函数的效果与 *ql_rtos_timer_start* 函数相同；
非 0：定时器超时时，若系统处于休眠状态，则不会唤醒系统，而是等待 *relax_time* 传入的时间后再唤醒系统；若在等待的过程中系统被其他唤醒源唤醒，则定时器回调函数会被立即执行。
QL_WAIT_FOREVER：定时器将不会唤醒系统。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_TIMER_START_FAIL：定时器启动失败。
QL_OSI_SUCCESS：定时器启动成功。

3.3.32. ql_rtos_timer_start_us

该函数用于启动定时器，时间精度为微秒。定时器超时后，若系统处于休眠状态，将唤醒系统。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

set_Time_us：
[In] 定时器超时时间，单位：微秒。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_TIMER_START_FAIL：定时器启动失败。
QL_OSI_SUCCESS：定时器启动成功。

3.3.33. ql_rtos_timer_stop

该函数用于停止定时器。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_TIMER_STOP_FAIL：定时器停止失败。
QL_OSI_SUCCESS：定时器停止成功。

3.3.34. ql_rtos_timer_is_running

该函数用于判断定时器是否正在计时。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

- **返回值**

1 定时器正在计时
0 定时器不在计时

3.3.35. ql_rtos_timer_delete

该函数用于删除定时器。

- **函数原型**

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

- **参数**

timerRef：
[In] 定时器。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SUCCESS：定时器删除成功。

3.3.36. ql_rtos_event_send

该函数用于发送事件。

- **函数原型**

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

- **参数**

task_ref：
[In] 任务句柄。

event：
[In] 事件结构体，不可为空。详情参见***第 3.3.36.1 章***。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_EVENT_SEND_FAIL：事件发送失败。
QL_OSI_SUCCESS：事件发送成功。

3.3.36.1. ql_event_t

```
typedef struct
{
```

```
uint32 id;
uint32 param1;
uint32 param2;
uint32 param3;
} ql_event_t
```

- **参数**

类型	参数	描述
uint32	id	事件标识符，不可小于 1000，为了保证标识符的唯一性，建议用 *ql_osi_def.h*（SDK 路径：*components\ql-kernel\inc*）中的方法来定义。
uint32	param1	第一个参数。
uint32	param2	第二个参数。
uint32	param3	第三个参数。

3.3.37. ql_event_wait

该函数用于等待事件，等待时间用户可以根据需求进行自定义。

- **函数原型**

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

- **参数**

event_strc：
[In] 事件结构体，不可为空。详情参见***第 3.3.36.1 章***。

timeout：
[In] 事件等待时间；单位：毫秒。0xFFFFFFFF 表示永久等待。除以下取值外，用户还可自定义等待时间。
QL_WAIT_FOREVER 永久等待
QL_NO_WAIT 不等待

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_EVENT_GET_FAIL：事件等待失败。
QL_OSI_SUCCESS：事件等待成功。

3.3.38. ql_event_try_wait

该函数用于等待事件，等待时间为永久等待。

- **函数原型**

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

- **参数**

event_strc：
[In] 事件结构体，不可为空。详情参见***第 3.3.36.1 章***。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_EVENT_GET_FAIL：事件等待失败。
QL_OSI_SUCCESS：事件等待成功。

3.3.39. ql_rtos_swdog_register

该函数用于为任务绑定软件看门狗。

- **函数原型**

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

- **参数**

callback：
[In] 软件看门狗回调函数，软件看门狗会在一个周期结束后调用此回调函数。详情参见***第 3.3.40.1 章***。

taskRef：
[In] 需要绑定任务的任务句柄。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SWDOG_REGISTER_FAIL：绑定失败。
QL_OSI_SUCCESS：绑定成功。

3.3.40. ql_rtos_swdog_unregister

该函数用于为任务解绑软件看门狗。

- **函数原型**

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

- **参数**

callback：
[In] 软件看门狗回调函数，即绑定时输入的回调函数。详情参见***第 3.3.40.1 章***。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SWDOG_UNREGISTER_FAIL：解绑失败。
QL_OSI_SUCCESS：解绑成功。

3.3.40.1. ql_swdog_callback

该函数为软件看门狗回调函数，软件看门狗会在一个周期结束后调用此回调函数。

- **函数原型**

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

- **参数**

id_type：
[Out] 喂狗标志符，固定传 *QUEC_KERNEL_FEED_DOG* 到应用层。

ctx：
[Out] 本次需要执行喂狗操作的任务的任务句柄。

- **返回值**

无

3.3.41. ql_rtos_sw_dog_enable

该函数用于使能软件看门狗功能。

- **函数原型**

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

- **参数**

period_ms：
[In] 软件看门狗运行周期，一个周期结束后，将会调用 *ql_rtos_swdog_register* 注册的每个回调函数。

missed_cnt：
[In] 最大丢失次数，即未喂狗的次数。

- **返回值**

QL_OSI_TASK_PARAM_INVALID：无效参数。
QL_OSI_SWDOG_ENABLE_FAIL：使能软件看门狗失败。
QL_OSI_SUCCESS：使能软件看门狗成功。

3.3.42. ql_rtos_sw_dog_disable

该函数用于禁用软件看门狗功能。

- **函数原型**

```
QlOSStatus ql_rtos_sw_dog_disable(void)
```

- **参数**

无

- **返回值**

QL_OSI_SWDOG_DISABLE_FAIL：禁用失败。
QL_OSI_SUCCESS：禁用成功。

3.3.43. ql_rtos_feed_dog

该函数用于给当前执行的任务喂狗，在需要喂狗的任务中调用。

- **函数原型**

```
QlOSStatus ql_rtos_feed_dog(void)
```

- **参数**

无

- **返回值**

QL_OSI_SWDOG_FEED_DOG_FAIL：喂狗失败。
QL_OSI_SUCCESS：喂狗成功。

3.3.44. ql_gettimeofday

该函数用于获取当前的详细时间。

- **函数原型**

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

- **参数**

timeval：
[Out] 当前时间。详情参见***第 3.3.44.1 章***。

- **返回值**

QL_OSI_SUCCESS：获取时间成功。

3.3.44.1. ql_timeval_t

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

- **参数**

类型	参数	描述
uint32	*sec*	1970 年 1 月 1 日到现在的总时间，单位：秒。
uint32	*usec*	1970 年 1 月 1 日到现在的总时间中的小数部分，小数部分代指微秒数；范围：0~99999。

3.3.45. ql_rtos_get_system_tick

该函数用于获取 RTOS 系统的时钟节拍数。

- **函数原型**

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

- **参数**

无

- **返回值**

大于 0：总时钟节拍。
小于 0：获取失败。

3.3.46. ql_assert

该函数用于强制使模块 dump。

- **函数原型**

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

- **参数**

无

- **返回值**

无

3.3.47. ql_rtos_up_time_ms

该函数用于获取自系统启动至当前的时间，该时间在休眠唤醒时不会停止，用于 littlevGL 的时间同步。

- **函数原型**

```
int64_t ql_rtos_up_time_ms();
```

- **参数**

无

- **返回值**

系统自系统启动至当前的时间。

4 附录 参考文档及术语缩写

表 10：参考文档

文档名称
[1] Quectel_ECx00U&EGx00U 系列_QuecOpen_CSDK 快速开发指导

表 11：术语缩写

缩写	英文全称	中文全称
API	Application Programming Interface	应用程序接口
CPU	Central Processing Unit	中央处理器
ID	Identifier	标识符
PC	Personal Computer	个人电脑
RAM	Random Access Memory	随机存取存储器
ROM	Read-Only Memory	只读存储器
RTOS	Real Time Operating System	实时操作系统
SDK	Software Development Kit	软件开发工具包
TCB	Task Control Block	任务控制块
OS	Operating System	操作系统