RL-ARM User's Guide

Оглавление

RL-RTX

This chapter describes the powerful Keil RTX Real-Time Operating System (RTX-RTOS) designed for microcontrollers based on ARM7™TDMI, ARM9™, and Cortex™-M CPU cores. The RTX kernel can be used for creating applications that perform multiple functions or tasks simultaneously.

While it is certainly possible to create real-time applications without an RTOS (by executing one or more tasks in a loop), there are numerous scheduling, maintenance, and timing issues that can be solved better with an RTOS. For example, an RTOS enables flexible scheduling of system resources like CPU and memory, and offers methods to communicate between tasks.

RTX programs are written using standard C constructs and are compiled with the RealView® Compiler. The **RTX.H** header file defines the RTX functions and macros that allow declaring tasks and accessing all RTOS features easily.

This chapter contains the following sections:

Overview

Provides an overview about the RL-RTX basic functions, inter-process communication and technical specifications.

Theory of Operation

Describes the resources and their management, such as scheduling, task and stack management, interrupts, and timers.

Configuring the RTX Kernel

Describes the basic and advanced RTX configuration options, and lists the RL-RTX library files.

Using the RTX Kernel

Provides instructions for writing and debugging applications.

Function Reference

Describes the RTX functions.

The topic **Create New RTX Application** provides a step-by-step introduction that explains you how to create RTX applications.

Overview

The RTX kernel is an easy to use **R**eal **T**ime e**X**ecutive (RTX) providing a set of C functions and macros for building real-time applications which are using tasks running quasi-parallel on the CPU.

This section includes the topics:

- Product Description - RTX provides functions to synchronize different tasks, manage common resources (like peripherals or memory regions), and pass complete messages between tasks.
- Product Specification - Lists technical and timing specifications.
- Advantages - Lists some of the advantages when using the RTX kernel.

Product Description

The RTX kernel provides basic functionality to start and stop concurrent tasks (processes). RTX provides additional functions for inter-process communication. Use the communication functions to synchronize tasks, manage common resources (like peripherals or memory regions), and pass complete messages between tasks (round-robin scheduling).

Developers can assign execution priorities to tasks. When more tasks exists in the ready list, the RTX kernel uses the execution priorities to select the next task to run (preemptive scheduling).

The RTX kernel provides several ways for inter-process communication. These are:

- **Event flags**
  Event flags are the primary instrument for implementing task synchronization. Each task has 16 event flags assigned to it. Hence, task can selectively wait for 16 different events at the same time. In this case, the task can wait for all the selected flags (AND-connection), or wait for any one of the selected flags (OR-connection).

  Event flags can be set either by tasks or by ARM **interrupt functions**. Synchronize an external asynchronous event to an RTX kernel task by making the ARM interrupt function set a flag that the task is waiting for.

- **Semaphores**
  If more than one task needs to access a common resource, special means are required in a real time multitasking system. Otherwise, accesses by different tasks might interfere and lead to inconsistent data, or a peripheral element might function incorrectly.

  Semaphores are the primary means of avoiding such access problems. Semaphores (binary semaphores) are software objects containing a **virtual token**. The kernel gives the token to the first task that requests it. No other task can obtain the token until it is released back into the semaphore. Since only the task that has the token can access the common resource, it prevents other tasks from accessing and interfering with the common resource.

  The RTX kernel puts a task to sleep if the requested token is not available in the semaphore. The kernel wakes-up the task and puts it in the ready list as soon as the token is returned to the semaphore. It is posible to use a time-out to ensure the task does not sleep indefinitely.

- **Mutexes**
  Mutual exclusion locks (mutexs) are an alternative to avoid synchronization and memory access problems. Mutexes are software objects that a task can use to lock the common resource. Only the task that locks the mutex can access the common resource. The kernel blocks all other tasks that request the mutex until the task that locked the mutex unlocks it.

- **Mailboxes**
  Tasks can pass **messages** between each other using mailboxes. This is usually the case when implementing various high level protocols like TCP-IP, UDP, and ISDN.

  The message is simply a **pointer** to the block of memory containing a protocol message or frame. It is the programmer's responsibility to dynamically allocate and free the memory block to prevent memory leaks.

  The RTX kernel puts the waiting task to sleep if the message is not available. The kernel wakes the task up as soon as another task sends a message to the mailbox.

Product Specification

RTX Kernel Library can be used with devices based on:

- ARM7 and ARM9
- Cortex-M0/M1, Cortex-M3, Cortex-M4, and Cortex-R4

Cortex-M processor-based devices have extended RTOS features, which allows more robust and fail-proof RTX kernel implementation. The main concept differences are:

- ARM7 and ARM9 versions use the system task manager to control task switches of all user tasks. Tasks are executed in System Mode.
- Cortex-M versions use system calls that are all executed as **SVC** System Supervisor Calls.

In addition, refer to:

- Technical Data - lists hardware requirements, RAM and code space, number of possible tasks, ...
- Timing Specifications - lists timing measurements (cycles) for various RTX functions.

Technical Data

Description	ARM7™/ARM9™	Cortex™-M
Defined Tasks	Unlimited	Unlimited
Active Tasks	250 max	250 max
Mailboxes	Unlimited	Unlimited
Semaphores	Unlimited	Unlimited
Mutexes	Unlimited	Unlimited
Signals / Events	16 per task	16 per task
User Timers	Unlimited	Unlimited
Code Space	<4.2 Kbytes	<4.0 Kbytes
RAM Space for Kernel	300 bytes + 80 bytes User Stack	300 bytes + 128 bytes Main Stack
RAM Space for a Task	TaskStackSize + 52 bytes	TaskStackSize + 52 bytes
RAM Space for a Mailbox	MaxMessages * 4 + 16 bytes	MaxMessages * 4 + 16 bytes
RAM Space for a Semaphore	8 bytes	8 bytes
RAM Space for a Mutex	12 bytes	12 bytes
RAM Space for a User Timer	8 bytes	8 bytes
Hardware Requirements	One on-chip timer	SysTick timer
User task priorities	1 - 254	1 - 254
Context switch time	<5.3 µsec @ 60 MHz	<2.6 µsec @ 72 MHz
Interrupt lockout time	<2.7 µsec @ 60 MHz	Not disabled by RTX

Note

- **Unlimited** means that the RTX kernel does not impose any limitations on the number. However, the available system memory resources limit the number of items you can create.
- The default configuration of the RTX kernel allows 10 tasks and 10 user timers. It also disables stack checking by default.
- In the RTX kernel, **Event** is simply another name for signal.
- RAM requirements depend on the number of concurrently running tasks.
- The code and RAM size was calculated for **MicoLib** runtime library.

Timing Specifications

Function	ARM7™/ARM9™ (cycles)	Cortex™-M (cycles)
Initialize system (os_sys_init), start task	1721	1147
Create task (no task switch)	679	403
Create task (switch task)	787	461
Delete task (os_tsk_delete)	402	218
Task switch (by os_tsk_delete_self)	458	230
Task switch (by os_tsk_pass)	321	192
Set event (no task switch)	128	89
Set event (switch task)	363	215
Send semaphore (no task switch)	106	72
Send semaphore (switch task)	364	217
Send message (no task switch)	218	117
Send message (switch task)	404	241
Get own task identifier (os_tsk_self)	23	65
Interrupt latency	<160	0

Note

- The table for **ARM7™/ARM9™** RTX Kernel library is measured on **LPC2138** (ARM7), code execution from internal flash with zero-cycle latency.
- The table for **Cortex™-M** RTX Kernel library is measured on **LPC1768** (Cortex-M3), code execution from internal flash with zero-cycle latency.
- The RTX Kernel for the test is configured for 10 tasks, 10 user timers and stack checking disabled.
- Interrupt latency in **ARM7™/ARM9™** includes the ISR prolog generated by the compiler.
- The RTX Kernel library for **Cortex™-M3/M4** does not disable interrupts. Interrupt latency is the same as without the RTX kernel. Interrupt latency for **Cortex™-M0/M1** is <20 cycles.

Related Knowledgebase Articles

- RL-ARM: RTX KERNEL MODE USED IN ARM CPU

Advantages

The RTX kernel is based on the idea of parallel tasks (processes). In the RTX kernel, the task that a system must fulfill is split up into several smaller tasks that run concurrently. There are many advantages to using the RTX kernel:

- Real world processes consist, usually, of several concurrent activities. This pattern can be represented in software by using the RTX kernel.
- You can make different activities occur at different times, for example, just at the moment when they are needed. This is possible because each activity is packed into a separate task, which can be executed on its own.
- Tasks can be prioritized.
- It is easier to understand and manage smaller pieces of code than one large piece of software.
- Splitting up the software into independent parts reduces the system complexity, the number of errors, and even facilitates testing.
- The RTX kernel is scalable. Additional tasks can be added easily at a later time.
- The RTX kernel offers services needed in many real-time applications, for example, good handling of interrupts, periodical activation of tasks, and time-limits on wait functions.

Your First RTX Application

This section demonstrates an example of using the RTX kernel for a simple application. The example is located in the folder **\Keil\ARM\RL\RTX\Examples\RTX_ex1**. The application must perform two activities. The first activity must continuously repeat 50 ms after the second activity completes. The second activity must repeat 20 ms after the first activity completes.

Hence, you can implement these activities as two separate tasks, called task1 and task2:

1. Place the code for the two activities into two separate functions (task1 and task2). Declare the two functions as tasks using the keyword **__task** (defined in RTL.H) which indicates a RTX task.

```
__task void task1 (void) {
 .... place code of task 1 here ....
}
```

```
__task void task2 (void) {
  .... place code of task 2 here ....
}
```

2. When the system starts up, the RTK kernel must start before running any task. To do this, call the **os_sys_init** function in the C **main** function. Pass the function name of the first task as the parameter to the **os_sys_init** function. This ensures that after the RTX kernel initializes, the task starts executing rather than continuing program execution in the **main** function.

   In this example, task1 starts first. Hence, task1 must create task2. You can do this using the **os_tsk_create** function.

```
__task void task1 (void) {
 os_tsk_create (task2, 0);
 .... place code of task 1 here ....
}
```

```
__task void task2 (void) {
  .... place code of task 2 here ....
}
```

```
void main (void) {
  os_sys_init (task1);
}
```

3. Now implement the timing requirements. Since both activities must repeat indefinitely, place the code in an endless loop in each task. After the task1 activity finishes, it must send a signal to task2, and it must wait for task2 to complete. Then it must wait for 50 ms before it can perform the activity again. You can use the **os_dly_wait** function to wait for a number of system intervals. The RTX kernel starts a system timer by programming one of the on-chip hardware timers of the ARM processors. By default, the system interval is 10 ms and timer 0 is used (this is configurable).

   You can use the **os_evt_wait_or** function to make task1 wait for completion of task2, and you can use the **os_evt_set** function to send the signal to task2. This examples uses bit 2 (position 3) of the event flags to inform the other task when it completes.

   task2 must start 20 ms after task1 completes. You can use the same functions in task2 to wait and send signals to task1. The listing below shows all the statements required to run the example:

```
/* Include type and function declarations for RTX */
#include "rtl.h"

/* id1, id2 will contain task identifications at run-time */
OS_TID id1, id2;

/* Forward declaration of tasks. */
__task void task1 (void);
__task void task2 (void);

__task void task1 (void){
  /* Obtain own system task identification number */
  id1 = os_tsk_self();

  /* Create task2 and obtain its task identification number */
  id2 = os_tsk_create (task2, 0);

  for (;;) {
    /* ... place code for task1 activity here ... */

    /* Signal to task2 that task1 has compelted */
    os_evt_set(0x0004, id2);

    /* Wait for completion of task2 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 50 ms before restarting task1 activity. */
    os_dly_wait(5);
  }
}

__task void task2 (void) {
  for (;;) {
    /* Wait for completion of task1 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 20 ms before starting task2 activity. */
    os_dly_wait(2);

    /* ... place code for task2 activity here ... */

    /* Signal to task1 that task1 has compelted */
    os_evt_set(0x0004, id1);
  }
}

void main (void) {
  /* Start the RTX kernel, and then create and execute task1. */
  os_sys_init(task1);
}
```

4. Finally, to compile the code and link it with the RTX library, you must select the RTX operating system for the project. From the main menu, select **Project —> Options for Target**. Select the **Target** tab. Select RTX Kernel for the Operating system. Build the project to generate the absolute file. You can run the object file output from the linker either on your target or on the µVision Simulator.

Theory of Operation

The RTX kernel uses and manages your target system's resources. This section describes the resources and how they are managed by the RTX kernel.

Many aspects of the RTX kernel can be configured on a project by project basis. This is mentioned where applicable.

Timer Tick Interrupt

The RTX kernel for ARM7™ and ARM9™ uses one of the standard ARM timers to generate a periodic interrupt. RTX kernel for Cortex™-M uses common SysTick timer. This interrupt is called the RTX kernel timer tick. For some of the RTX library functions, you must specify timeouts and delay intervals in number of RTX kernel timer ticks.

The parameters for the RTX kernel timer are selected in the **RTX_Config.c** configuration file. Each ARM microcontroller family provides a different peripherals that are supported with different **RTX_Config.c** files.

For example, the **RTX_Config.c** file for NXP LPC2100/LPC2200 allows to use Timer 0 or Timer 1 for the RTX kernel timer.

The **timer clock value** specifies the input clock frequency and depends on CPU clock and APB clock. For a device with CPU clock 60 MHz and VPB divider 4 the peripheral clock is 15MHz and therefore the value is set to 15000000.

The **time tick value** specifies the interval of the periodic RTX interrupt. The value 10000 us configures timer tick period of 0.01 seconds.

System Task Manager

The task manager is a system task that is executed on every **timer tick interrupt**. It has the highest assigned priority and does not get preempted. This task is basically used to switch between user tasks.

The RTX tasks are not really executed concurrently. They are **time-sliced**. The available CPU time is divided into time slices and the RTX kernel assigns a time slice to each task. Since the time slice is short (default time slice is set to 10 ms) it appears as though tasks execute simultaneously.

Tasks execute for the duration of their time-slice unless the task's time-slice is given up explicitly by calling the **os_tsk_pass** or one of the **wait** library functions. Then the RTX Kernel switches to the next task that is ready to run. You can set the duration of the time-slice in the **RTX_Config.c** configuration file.

The **task manager** is the system tick timer task that manages all other tasks. It handles the task's delay timeouts and puts waiting tasks to sleep. When the required event occurs, it puts the waiting tasks back again into the ready state. This is why the tick timer task must have the highest priority.

The task manager runs not only when the timer tick interrupt occurs, but also when an interrupt calls one of the **isr_** functions. This is because interrupts cannot make the current task wait, and therefore interrupts cannot perform task switching. However, interrupts can generate the event, semaphore or mailbox message (using an **isr_** library function) that a higher priority task is waiting for. The higher priority task must preempt the current task, but can do so only after the interrupt function completes. The interrupt therefore forces the timer tick interrupt, which runs when the current interrupt finishes. The forced tick timer interrupt starts the **task manager** (clock task) scheduler. The task scheduler process all the tasks and then puts the highest ready task into the running state. The highest priority task can then continue with its execution.

Note

- The tick timer task is an RTX system task and is therefore created by the system.
- The RTX library for **Cortex™-M** uses extended RTOS features of Cortex™-M devices. All RTX system functions are executed in **svc mode**.

Task Management

Each RTX task is always in exactly one state, which tells the disposition of the task.

State	Description
RUNNING	The task that is currently running is in the **RUNNING** state. Only one task at a time can be in this state. The **os_tsk_self()** returns the Task ID (TID) of the currently executing task.
READY	Tasks which are ready to run are in the **READY** state. Once the running task has completed processing, RTX selects the next ready task with the **highest priority** and starts it.
WAIT_DLY	Tasks which are waiting for a delay to expire are in the **WAIT_DLY** State. Once the delay has expired, the task is switched to the **READY** state. The **os_dly_wait()** function is used to place a task in the **WAIT_DLY** state.
WAIT_ITV	Tasks which are waiting for an interval to expire are in the **WAIT_ITV** State. Once the interval delay has expired, the task is switched back to the **READY** State. The **os_itv_wait()** function is used to place a task in the **WAIT_IVL** State.
WAIT_OR	Tasks which are waiting for at least one event flag are in the **WAIT_OR** State. When the event occurs, the task is switched to the **READY** state. The **os_evt_wait_or()** function is used to place a task in the **WAIT_OR** state.
WAIT_AND	Tasks which are waiting for all the set events to occur are in the **WAIT_AND** state. When all event flags are set, the task is switched to the **READY** state. The **os_evt_wait_and()** function is used to place a task in the **WAIT_AND** state.
WAIT_SEM	Tasks which are waiting for a semaphore are in the **WAIT_SEM** state. When the token is obtained from the semaphore, the task is switched to the **READY** state. The **os_sem_wait()** function is used to place a task in the **WAIT_SEM** state.
WAIT_MUT	Tasks which are waiting for a free mutex are in the **WAIT_MUT** state. When a mutex is released, the task acquire the mutex and switch to the READY state. The **os_mut_wait()** function is used to place a task in the **WAIT_MUT** state.
WAIT_MBX	Tasks which are waiting for a mailbox message are in the **WAIT_MBX** state. Once the message has arrived, the task is switched to the **READY** state. The **os_mbx_wait()** function is used to place a task in the **WAIT_MBX** state. Tasks waiting to send a message when the mailbox is full are also put into the **WAIT_MBX** state. When the message is read out from the mailbox, the task is switched to the **READY** state. In this case the **os_mbx_send()** function is used to place a task in the **WAIT_MBX** state.
INACTIVE	Tasks which have not been started or tasks which have been deleted are in the **INACTIVE** state. The **os_tsk_delete()** function places a task that has been started (with **os_tsk_create()**) into the **INACTIVE** state.

Idle Task

When no task is ready to run, the RTX kernel executes the idle task **os_idle_demon**. The idle task is simply an endless loop. For example:

```
for (;;);
```

ARM devices provide an **idle mode** that reduces power consumption by halting program execution until an interrupt occurs. In this mode, all peripherals, including the interrupt system, still continue to work.

The RTX kernel initiates idle mode in the os_idle_demon task (when no other task is ready for execution). When the RTX kernel **timer tick interrupt** (or any other interrupt) occurs, the microcontroller resumes program execution.

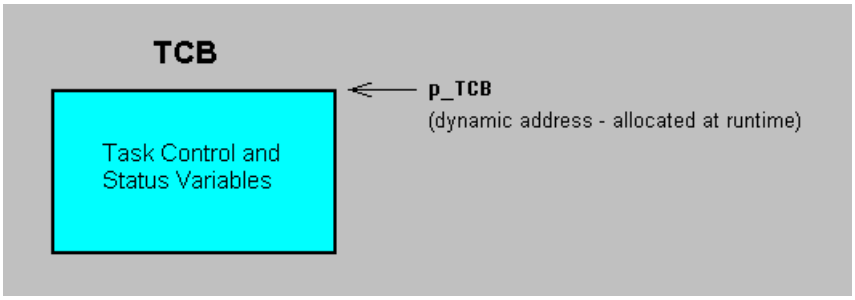
You can add your own code to the os_idle_demon task. The code executed by the idle task can be configured in the **RTX_Config.c** configuration file.

Note

- The idle task **os_idle_demon** is an RTX kernel system task and is therefore created by the system.
- Do not use **idle mode** if you are using the JTAG interface for debugging. Some ARM devices may stop communicating over the JTAG interface when idle.

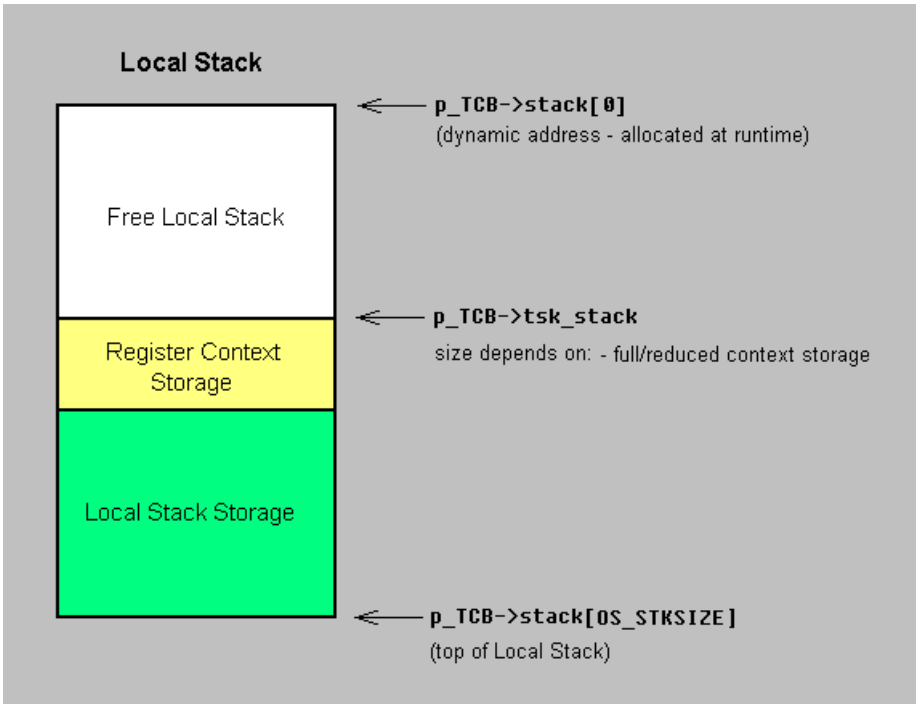
System Resources

RTX kernel tasks are identified by their Task Control Block (TCB). This is a dynamically allocated block of memory where all task control and status variables are located. TCB is allocated at runtime when the task is created with the **os_tsk_create** or **os_tsk_create_user** function call.



The size of the TCB memory pool is defined in the **RTX_Config.c** configuration file, and it depends on the number of concurrent running tasks. This is not necessarily the number of defined tasks since **multiple instances** of a task are supported by the RTX kernel.

The RTX kernel also allocates the task its own **stack**. The stack is allocated at runtime after the TCB has been allocated. The pointer to the stack memory block is then written into the TCB.



Scheduling Options

RTX allows you to build an application with three different kernel-scheduling options. These are:

- **Pre-emptive scheduling**
  Each task has a **different priority** and will run until it is pre-empted or has reached a blocking OS call.

- **Round-Robin scheduling**
  Each task has the **same priority** and will run for a fixed period, or time slice, or until has reached a blocking OS call.

- **Co-operative multi-tasking**
  Each task has the **same priority** and the Round-Robin is disabled. Each task will run until it reached a blocking OS call or uses the **os_tsk_pass()** call.

The default scheduling option for RTX is Round-Robin Pre-emptive. For most applications, this is the most useful option.

Pre-emptive Scheduling

RTX is a **pre-emptive** multitasking operating system. If a task with a higher priority than the currently running task becomes ready to run, RTX **suspends** the currently running task.

A preemptive task switch occurs when:

- the task scheduler is executed from the system **tick timer interrupt**. Task scheduler processes the **delays** of tasks**.** If the delay for a task with a higher priority has expired, then the higher priority task starts to execute instead of currently running task.
- an **event** is set for a higher priority task by the currently running task or by an interrupt service routine. The currently running task is suspended, and the higher priority task starts to run.
- a token is returned to a **semaphore**, and a higher priority task is waiting for the semaphore token. The currently running task is suspended, and the higher priority task starts to run. The token can be returned by the currently running task or by an interrupt service routine.
- a **mutex** is released and a higher priority task is waiting for the mutex. The currently running task is suspended, and the higher priority task starts to run.
- a message is posted to a **mailbox**, and a higher priority task is waiting for the mailbox message. The currently running task is suspended, and the higher priority task starts to run. The message can be posted by the currently running task or by an interrupt service routine.
- a **mailbox** is **full**, and a higher priority task is waiting to post a message to a mailbox. As soon as the currently running task or an interrupt service routine takes a message out from the mailbox, the higher priority task starts to run.
- the **priority** of the currently running task is reduced. If another task is ready to run and has a higher priority than the new priority of the currently running task, then the current task is suspended immediately, and the higher priority task resumes its execution.

The following example demonstrates one of the task switching mechanisms. Task job1 has a higher priority than task job2. When job1 starts, it creates task job2 and then enters the **os_evt_wait_or** function. The RTX kernel suspends job1 at this point, and job2 starts executing. As soon as job2 sets an event flag for job1, the RTX kernel suspends job2 and then resumes job1. Task job1 then increments counter cnt1 and calls the **os_evt_wait_or** function, which suspends it again. The kernel resumes job2, which increments counter cnt2 and sets an event flag for job1. This process of task switching continues indefinitely.

```
#include <rtl.h>

OS_TID tsk1,tsk2;
int    cnt1,cnt2;
__task void job1 (void);
__task void job2 (void);

__task void job1 (void) {
  os_tsk_prio (2);
  tsk1 = os_tsk_self ();
  os_tsk_create (job2, 1);
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    cnt1++;
  }
}

__task void job2 (void) {
  while (1) {
    os_evt_set (0x0001, tsk1);
    cnt2++;
  }
}

void main (void) {
  os_sys_init (job1);
```

```
  while (1);
}
```

Round-Robin Scheduling

RTX can be configured to use Round-Robin Multitasking (or task switching). Round-Robin allows quasi-parallel execution of several tasks. Tasks are not really executed concurrently but are **time-sliced** (the available CPU time is divided into time slices and RTX assigns a time slice to each task). Since the time slice is short (only a few milliseconds) it appears as though tasks execute simultaneously.

Tasks execute for the duration of their time-slice (unless the task's time slice is given up). Then, RTX switches to the next task that is **ready** to run and has the **same priority**. If no other task with the same priority is ready to run, the currently running task resumes it execution. The duration of a time slice can be defined in the **RTX_config.c** configuration file.

The following example shows a simple RTX program that uses Round-Robin Multitasking. The two tasks in this program are counter loops. RTX starts executing task 1, which is the function named **job1.** This function creates another task called **job2.** After **job1** executes for its time slice, RTX switches to **job2.** After **job2** executes for its time slice, RTX switches back to **job1.** This process repeats indefinitely.

```
#include <rtl.h>

int counter1;
int counter2;

__task void job1 (void);
__task void job2 (void);

__task void job1 (void) {
  os_tsk_create (job2, 0);    /* Create task 2 and mark it as ready */
  while (1) {                 /* loop forever */

    counter1++;               /* update the counter */
  }
}

__task void job2 (void) {
  while (1) {                 /* loop forever */
    counter2++;               /* update the counter */
  }
}

void main (void) {
  os_sys_init (job1);         /* Initialize RTX Kernel and start task 1 */
  for (;;);
}
```

Note

- Rather than wait for a task's time slice to expire, you can use one of the **system wait** functions or the **os_tsk_pass** function to signal to the RTX kernel that it can switch to another task. The system wait function suspends the current task (changes it to the **WAIT_xxx** state) until the specified event occurs. The task is then changed to the **READY** state. During this time, any number of other tasks can run.

Cooperative Multitasking

If you disable Round-Robin Multitasking you must design and implement your tasks so that they work **cooperatively.** Specifically, you must call the system wait function like the **os_dly_wait()** function or the **os_tsk_pass()** function somewhere in each task. These functions signal the RTX kernel to switch to another task.

The following example shows a simple RTX program that uses Cooperative Multitasking. The RTX kernel starts executing task 1. This function creates task 2. After counter1 is incremented once, the kernel switches to task 2. After counter2 is incremented once, the kernel switches back to task 1. This process repeats indefinitely.

```
#include <rtl.h>

int counter1;
int counter2;

__task void task1 (void);
__task void task2 (void);

__task void task1 (void) {
  os_tsk_create (task2, 0);  /* Create task 2 and mark it as ready */
  for (;;) {                 /* loop forever */
    counter1++;              /* update the counter */
    os_tsk_pass ();          /* switch to 'task2' */
  }
}

__task void task2 (void) {
  for (;;) {                 /* loop forever */
    counter2++;              /* update the counter */
    os_tsk_pass ();          /* switch to 'task1' */
  }
}

void main (void) {
  os_sys_init(task1);        /* Initialize RTX Kernel and start task 1 */
  for (;;);
}
```

The difference between the system wait function and os_tsk_pass is that the system wait function allows your task to wait for an event, while os_tsk_pass switches to another ready task immediately.

Note

- If the next ready task has a lower priority than the currently running task, then calling **os_tsk_pass** does not cause a task switch.

Priority Inversion

The RTX Real Time Operating system employs a priority-based preemptive scheduler. The RTX scheduler assings each task a unique priority level. The scheduler ensures that *of those tasks that are ready to run*, the one with the highest priority is always the task that is actually running.

Because tasks share resourcs, events outside the scheduler's control can prevent the highest priority ready task from running when it should. If this happens, a critical deadline could be missed, causing the system to fail. **Priority inversion** is the term of a scenario in which the highest-priority ready task fails to run when it should.

Resource sharing

Tasks need to share resources to communicate and process data. Any time two or more tasks share a recource, such as a memory buffer or a serial port, one of them will usually have a higher priority. The higher-priority task expects to be run as soon as it is ready. However, if the lower-priority task is using their shared resource when the higher-priority task becomes ready to run, the higher-priority task must wait for the lower-priority task to finish with it.

Priority inheritance

To prevent priority inversions, the RTX Real Time OS employs a **Priority inheritance** method. The lower-priority task **inherit** the priority of any higher-priority task pending on a resource they share. For a short time, the lower-priority task runs at a priority of a higher-priority pending task. The priority change takes place as soon as the high-priority task begins to pend. When the lower-priority task stops using a shared resource, it's priority level returns to normal.

The RTX **mutex** objects (Mutual Exclusive Lock objects) employ the Priority inheritance.

Stack Management

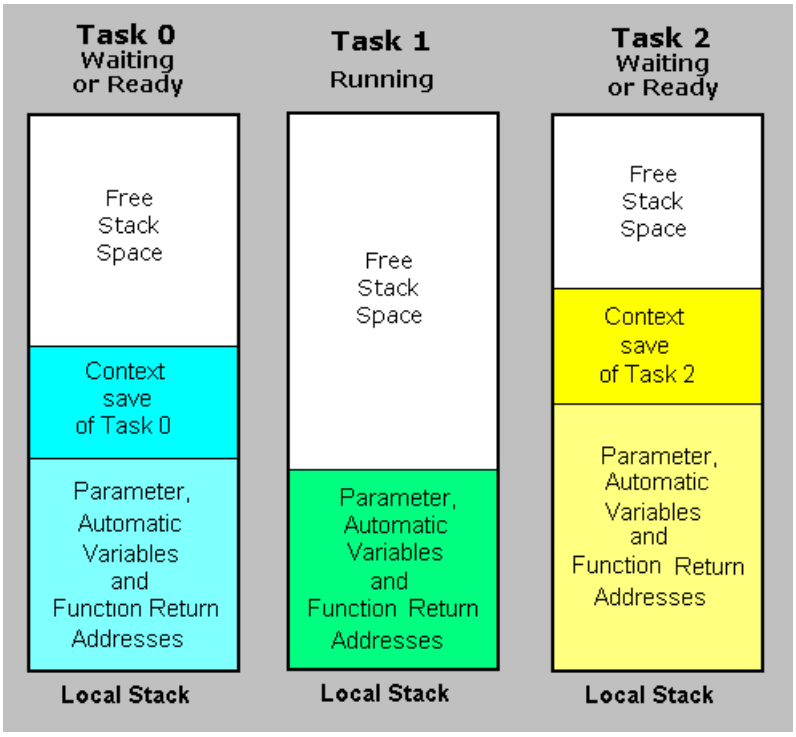
The Stack Management of the RTX kernel is designed for optimal memory usage. The RTX kernel system needs one stack space for the task that is currently in the **RUNNING** state:

- **Local Stack**: stores parameters, automatic variables, and function return addresses. On the ARM device, this stack can be anywhere. However, for performance reasons, it is better to use the on-chip RAM for the local stack.

When a task switch occurs:

- the context of the currently running task is stored on the local stack of the current task
- the stack is switched to that of the next task
- the context of the new task is restored
- the new task starts to run.

The Local Stack also holds the task context of waiting or ready tasks.



The other stack spaces need to be configured from the ARM **startup** file. All tasks run in **user mode**. The task scheduler switches the user/system mode stack between tasks. For this reason, the **default user/system** mode **stack** (which is defined in the startup file) is used until the first task is created and started. The default stack requirements are very small, so it is optimal to set the user/system stack in the startup file to **64 bytes**.

User Timers

User Timers are simple timer blocks that count down on every system timer tick. They are implemented as single shot timers. This means you cannot pause and restart these timers. However, you can create and kill the user timers dynamically at runtime. If you do not kill a user timer before it expires, the RTX kernel calls the user provided callback function, **os_tmr_call()**, and then deletes the timer when it expires.

A timeout value is defined when the timer is created by the **os_tmr_create()** function.

The RTX kernel calls the callback function with the argument **info.** The user provides this argument when creating the user timer. The RTX kernel stores the argument in the timer control block. When the timer expires, this argument is passed back to the user in the **os_tmr_call()** function. If the user kills the timer before the timeout value expires, the RTX kernel does not call the callback function.

You can customize the callback function **os_tmr_call()** in the **RTX_Config.c** configuration file.

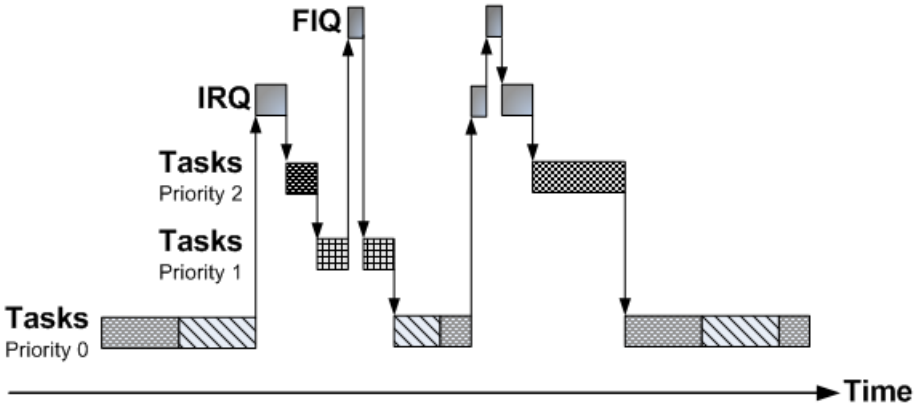
Note

- The callback function, **os_tmr_call**, is called from the system task scheduler. It is recommended to make your **os_tmr_call()** function as small and fast as possible because the callback function **blocks** the RTX task scheduler for the length of time it executes.
- The function **os_tmr_call** behaves the same way as standard interrupt functions. It is allowed to call the **isr_** functions to set an event, send a semaphore, or send a message to other tasks. You cannot call the **os_** library functions from **os_tmr_call()**.

Interrupt Functions

RTX can work with interrupt functions in **parallel.** However, it is better to avoid IRQ nesting. Good programming techniques use short interrupt functions that send signals or messages to RTOS tasks. With this practice, interrupt nesting becomes unimportant. This avoids common problems with nested interrupts where the user mode stack usage becomes unpredictable.

The following figure shows how interrupts should be handled with tasks in the RTX kernel system. An IRQ function can send a signal or message to start a high priority task.



Interrupt functions are added to an ARM application in the same way as in any other non-RTX projects.

Note

- The **FIQ** interrupts are never disabled by the RTX kernel.
- You cannot call the **isr_** library functions from the **FIQ** interrupt function.

The following example shows how to use interrupts with the RTX kernel. The interrupt function, **ext0_int**, sends an **event** to *process_task* and exits. The task *process_task* processes the external interrupt event. In this example, *process_task* is simple and only counts the number of interrupt events.

```
#define EVT_KEY 0x0001

OS_TID pr_task;
int    num_ints;

/*----------------------------------------------------------------------------
 *   External 0 Interrupt Service Routine
 *---------------------------------------------------------------------------*/
void ext0_int (void) __irq  {
  isr_evt_set (EVT_KEY, pr_task);      /* Send event to 'process_task'       */
  EXTINT      = 0x01;                  /* Acknowledge Interrupt              */
  VICVectAddr = 0;
}

/*----------------------------------------------------------------------------
 *   Task 'process_task'
 *---------------------------------------------------------------------------*/
__task void process_task (void) {
  num_ints = 0;
  while (1) {
    os_evt_wait_or (EVT_KEY, 0xffff);
    num_ints++;
```

```
  }
}

/*--------------------------------------------------------------------------
 *   Task 'init_task'
 *-------------------------------------------------------------------------*/
__task void init_task (void) {
  PINSEL1 &= ~0x00000003;              /* Enable EINT0                       */
  PINSEL1 |=  0x00000001;
  EXTMODE  = 0x03;                     /* Edge triggered lo->hi transition   */
  EXTPOLAR = 0x03;

  pr_task = os_tsk_create (process_task, 100);

  VICVectAddr14  = (U32)eint0_int;     /* Task started, Enable interrupts    */
  VICVectCntl14  = 0x20 | 14;

  os_tsk_delete_self ();               /* Terminate this task                */
}
```

Configuring RTX Kernel

The RTX kernel is easy to customize for each application you create. This section describes how you can configure the RTX kernel's features for your applications. It contains:

- Basic RTX Configuration
- Advanced RTX Configuration.

Basic RTX Configuration

The RTX kernel must be configured for the embedded applications you create. All configuration settings are found in the **RTX_Config.c** file, which is located in the **\Keil\ARM\Startup** directory. **RTX_Config.c** is configured differently for the different ARM devices. Configuration options in **RTX_Config.c** allow you to:

- Specify the number of concurrent running tasks
- Specify the number of tasks with user-provided stack
- Specify the stack size for each task
- Enable or disable the stack checking
- Enable or disable running tasks in privileged mode
- Specify the CPU timer number used as the system tick timer
- Specify the input clock frequency for the selected timer
- Specify the timer tick interval
- Enable or disable the round-robin task switching
- Specify the time slice for the round-robin task switching
- Define idle task operations
- Specify the number of user timers
- Specify code for the user timer callback function
- Specify the FIFO Queue size
- Specify code for the runtime error function

There is no default configuration in the RL-RTX library. Hence, you must add the **RTX_Config.c** configuration file to each project you create.

To customize the RTX kernel's features, you must change the configurable settings in **RTX_Config.c**.

Tasks

The following **#define**(s) specify how the RTX kernel tasks are configured:

- **OS_TASKCNT** specifies the maximum number of tasks that can be active at the same time. This includes tasks in any state (running, waiting, or ready) other than the INACTIVE state.

  This information is used by the RTX kernel to reserve the memory pool for the task control variables. This number can be higher than the number of defined tasks in your application (one task can run in **multiple instances**) or lower if it is guaranteed that the number of created and running tasks will never exceed OS_TASKCNT.

```
#define OS_TASKCNT     6
```

- **OS_PRIVCNT** specifies the number of tasks with **user-provided** stack.

  By default, the RTX kernel allocates a fixed size stack to each task. However, the stack requirement can vary widely between tasks. For example, if a task's local variables include large buffers, arrays, or complex structures, then the task requires a lot more stack. If such a task tries to use more stack than the allocated stack, it might overwrite the stack of neighboring tasks. This is because the fixed size stacks of the tasks are part of the common system stack and are contiguous. This leads to malfunctioning of the RTX kernel and is likely to cause a **system crash**. An intuitive solution to this problem is to increase the fixed stack size. However, this increases the stack size of every other task that might not need the extra stack. To avoid this wastage of valuable resource, a better solution is to allocate a separate user-provided stack for tasks that require a lot more stack.

  The term **user-provided**, in this case, means that the memory space for the task's stack is provided by the user when the **task is created**. It is not automatically assigned by the kernel. The RTX kernel uses OS_PRIVCNT to **optimize** the memory usage. The kernel will not reserve stack space for the tasks with a user-provided stack.

```
#define OS_PRIVCNT     0
```

Note

- In addition to OS_TASKCNT user tasks, the system creates one system task **os_idle_demon**. This task is always required by the RTX kernel. Total number of concurrent running tasks is **OS_TASKCNT+1** (number of user tasks plus one system task).

Stack Size

The **stack usage** of a particular task depends on its amount of local automatic variables and the number of subroutine levels. Interrupt functions do not use the stack of the interrupted task.

- **OS_STKSIZE** specifies the amount of RAM allocated for the stack of each task. Stack size is defined in U32 (unsigned int). However, Configuration Wizard converts the specified size and displays it in bytes. Stack size with the following define is 400 bytes.

```
#define OS_STKSIZE    100
```

- On the full context task switch, the RTX kernel stores all ARM registers on the stack. Full task context storing requires **64 bytes** of stack.

Note

- The **Cortex-M4** with Hardware Floating Point, needs additional **136 bytes** on stack for storing VFP registers. This means the total size of the full context store for Cortex-M4 with FP is **200 bytes**.
- The **Cortex-M4** tasks, where the Floating Point arithmetics is not used, do not store the additional VFP registers on context save. This means, they do not need additional 136 bytes on the stack. The full context store for tasks with no Floating Point calculations is still **64 bytes**.

Stack Checking

It is possible that the stack is exhausted because of many nested subroutine calls or an extensive use of large automatic variables.

If Stack Checking is enabled, the kernel can detect the stack exhaustion problem and execute the **system error** function. The application will hang in an endless loop inside the error function with parameter *err_code set to OS_ERR_STK_OVF*. You can identify the task id with **isr_tsk_get()** function. Check the **Active Tasks** debug dialog for the task name.

The solution to this problem is to increase the stack size for all tasks in the **configuration** file. If only one task requires a big stack and RAM is limited, you can **create** this task with a user-provided stack space.

- **OS_STKCHECK** enables the Stack Checking algorithm. It must be set to **1** to enable it or **0** to disable it. It is enabled by default.

```
#define OS_STKCHECK    1
```

Note

- Enabled Stack Checking slightly **decreases** the kernel performance because on every task switch the kernel needs to execute additional code for stack checking.
- On stack overflow a runtime **system error** function is called.

Run in Privileged Mode

RTX Library version for **Cortex™-M** devices allows to select the running mode of all user tasks. User tasks may run in two modes:

- **Unprivileged** - Protected mode or
- **Privileged** - Unprotected mode.

In **privileged** mode user may access and configure the system and control registers like NVIC interrupt controller etc. This is however not allowed from **unprivileged** mode. An access to NVIC registers from unprivileged mode will result in Hard Fault.

- **OS_RUNPRIV** enables running of all tasks in Privileged mode. It must be set to **1** to enable it or **0** to disable it. It is disabled by default.

```
#define OS_RUNPRIV   1
```

You can enable the **privileged mode** for old projects. The existing code will run without any modifications when RTX_Config.c configuration file is replaced with a new one and a project is recompiled for a new Cortex™-M RTX Kernel library. Tasks are not protected in privileged mode and you may configure the system for example the interrupts from any task.

Privileged mode is disabled by default. This allows all tasks to run in protected mode. The tasks are not allowed to change system settings, change interrupts etc. The user has two options:

- run the configuration code in privileged mode as **__svc** function from the task
- run the configuration code before the kernel is started when the device is still running in privileged mode.

Note

- The RTX Kernel library for **ARM7™/ARM9™** does not allow this option because of a different concept.

Hardware Timer

The following **#defines** specify how the RTX kernel's hardware timer is configured:

- **OS_TIMER** specifies the on-chip timer used as a time-base for the real-time system. It delivers a periodic interrupt that wakes up a time-keeping system task. The user can choose which timer serves this purpose. Use 0 for Timer 0, or 1 for Timer 1.

```
#define OS_TIMER        1
```

- **OS_CLOCK** specifies the input clock frequency for the selected timer. This value is calculated as: f(xtal) / VPBDIV. Example is for 15 MHz at @ 60 MHz CPU clock and VPBDIV = 4

```
#define OS_CLOCK        15000000
```

- **OS_TICK** specifies the timer tick interval in µsec. Recommended values are 1000 to 100000. The resulting interval is from 1 ms to 100 ms. Default configuration is for 10 ms.

```
#define OS_TICK         10000
```

Note

- Hardware Timer configuration is required only for **ARM7™** and **ARM9™** RTX Library version. The**Cortex™-M** version uses a common **SysTick** timer for all Cortex™-M device variants.

Round-Robin Multitasking

The following **#define** specify how the RTX kernel **Round-Robin Multitasking** is configured:

- **OS_ROBIN** enables the Round-Robin Multitasking. It must be set to **1** to enable it or **0** to disable it. It is enabled by default.

```
#define OS_ROBIN       1
```

- **OS_ROBINTOUT** specifies the Round-Robin Timeout. This is the time-slice assigned to the currently running task. After this time-slice expires, the currently running task is suspended and the next task ready to run is resumed. It is specified in number of system timer ticks.

```
#define OS_ROBINTOUT   5
```

User Timers

You can create and kill **user timers** at runtime. You must specify the maximum number of running user timers and also the code for the **os_tmr_call**() function.

- **OS_TIMERCNT** specifies the number of user timers that can be created and started. If user timers are not used, set this value to 0. This information is used by RTX to reserve the memory resources for timer control blocks.

```
#define OS_TIMERCNT    5
```

- The **callback** function **os_tmr_call()** is called when the user timer expires. It is provided in the **RTX_Config.c** configuration file as an empty function. You must modify it to suit your needs.

  Parameter **info** is the parameter passed to the **os_tmr_create()** function when the timer was created.

```
/*--------------------------- os_tmr_call -----------------------------------*/

void os_tmr_call (U16 info) {
  /* This function is called when the user timer has expired.               */
  /* Parameter "info" is the parameter defined when the timer was created.  */
  /* HERE: include here optional user code to be executed on timeout.       */
  info = info;
}
```

FIFO Queue Buffer

The **isr_** library function, when called from the interrupt handler, stores the request type and optional parameter to the **ISR FIFO Queue buffer** to be processed later, after the interrupt handler exits.

The **task manager** is activated immediately after the IRQ handler has finished its execution to process the requests stored to the FIFO Queue buffer. The size of this buffer needed, depends on the number of **isr**_ functions, that are called within the interrupt handler.

For example, if there is only one interrupt handler in your project and calls one **isr_evt_set()**, the FIFO Queue buffer size of 4 entries is sufficient. If there are more interrupts used in the project that use the **isr**_ communication with RTX kernel or, one interrupt handler that calls several **isr**_ library functions, the FIFO Queue buffer size needs to be increased. The interrupts, that do not use **isr**_ library functions are not counted here.

Default FIFO Queue buffer size is 16 entries. This should be enough for a typical RTX project.

- **OS_FIFOSZ** specifies the number of entries that can be stored to the FIFO Queue buffer. Default size is 16 entries.

```
#define OS_FIFOSZ    16
```

Note

- On FIFO Queue buffer overflow a runtime **system error** function is called.
- See the **Rtx_Config.c** configuration file for the possible configuration settings.

Error Function

Some system error conditions can be detected during runtime. If RTX kernel detects a **runtime error**, it calls the **os_error()** runtime error function.

```
void os_error (U32 err_code) {
  /* This function is called when a runtime error is detected. */
  OS_TID err_task;

  switch (err_code) {
    case OS_ERR_STK_OVF:
      /* Identify the task with stack overflow. */
      err_task = isr_tsk_get();
      break;
    case OS_ERR_FIFO_OVF:
      break;
    case OS_ERR_MBX_OVF:
      break;
  }
  for (;;);
}
```

The **error code** is passed to this function as a parameter *err_code*:

Error Code	Description
OS_ERR_STK_OVF	The stack checking has detected a stack overflow for the currently running task.
OS_ERR_FIFO_OVF	The ISR FIFO Queue buffer overflow is detected.
OS_ERR_MBX_OVF	The mailbox overflow is detected for isr_mbx_send() function.

The runtime error function must contain an **infinite loop** to prevent further program execution. You can use an emulator to step over infinite loop and trace into the code introducing a runtime error. For the overflow errors this means you need to increase the size of the object causing an overflow.

Related Knowledgebase Articles

- RL-ARM: TRANSITION FROM OS_STK_OVERFLOW() TO OS_ERROR()
- RL-ARM: ERROR: L6218E: UNDEFINED SYMBOL OS_ERROR

Idle_Task

When no tasks are ready to run, the RTX kernel executes the idle task with the name **os_idle_demon()**. By default this task is an empty end-less loop that does nothing. It only waits until another task becomes ready to run.

You may change the code of **os_idle_demon()** to put the CPU into a power-saving or idle mode. Most **RTX_Config.c** files define the macro _idle_() that contains the code to put the CPU into a power-saving mode.

Example:

```
/*--------------------------- os_idle_demon ---------------------------------*/

__task void os_idle_demon (void) {
  /* The idle demon is a system task. It is running when no other task is   */
  /* ready to run (idle situation). It must not terminate. Therefore it     */
  /* should contain at least an endless loop.                               */

  for (;;) {
    _idle_();   /* enter low-power mode */
  }
}
```

Note

- On some devices, the IDLE blocks debugging via the JTAG interface. Therefore JTAG debuggers such as ULINK may not work when you are using CPU power-saving modes.
- For using power-saving modes, some devices may require additional configuration (such as clock configuration settings).

Related Knowledgebase Articles

- RL-ARM: USING IDLE MODE ON STR9 WITH RTX KERNEL

Advanced RTX Configuration

RL-ARM provides several versions of the **RTX_Config.c** file for ARM7™/ARM9™ RTX Kernel library. Each one configures the RTX kernel for a specific ARM device variant that RL-ARM supports. However the ARM family of devices is growing quickly, and it is possible that RL-ARM does not contain the configuration file for the device you use. In this case, you can take the **RTX_Config.c** file for the NXP device as a template and modify it for your particular ARM device. This file is located in the**\Keil\ARM\Startup\Philips** directory.

All hardware dependent definitions are extracted from the code and defined with macros. This makes it possible to customize the configuration without modifying the code.

Note

- RTX Kernel library for Cortex™-M has only one configuration file which is common for all Cortex™-M device variants.

HW Resources Required

In order to run the RTX kernel, the following hardware resources are required from an ARM device:

- **Peripheral Timer** for generating periodic ticks. It is better to use a peripheral timer with an auto-reload function. RTX also supports timers with manual timer (or counter) reload. However, this can generate jitter and inaccuracy in the long run. The RTX kernel needs a **count-up** timer. If the timer used is a count-down timer, you need to convert the timer value.
- **Timer Interrupts** to interrupt the execution of a task and to start the system task scheduler.
- **Forced Interrupts** to force a timer interrupt when **isr_** functions are used. If an **isr_** function is called, the kernel forces the timer interrupt immediately after the interrupt ends. The forced timer interrupt activates the task scheduler. It is possible that a task has become ready. If this task has a higher priority than the currently running task, a task switch must occur.

Configuration Macros

All hardware related configuration options are described with configuration macros. Some of the macros (for example **OS_TID_** and **OS_TIM_**) are used only to simplify the code. They are not used in all of the configuration files. Using configuration macros allows easy customization of the configuration for the different peripheral timers supported by an ARM device.

The following configuration macros are introduced:
(examples are for Philips LPC21xx devices - Timer 0)

- **OS_TRV** macro specifies the timer reload value for the peripheral timer. Peripheral timer counts up to a reload value, then then overflows to 0, and then generates a tick interrupt. The reload value should be calculated to generate the desired interval length (for example 10 ms).

```
#define OS_TRV      ((U32)(((double)OS_CLOCK*(double)OS_TICK)/1E6)-1)
```

- **OS_TVAL** macro is used to read the current timer value for a **count-up** timer. The RTX kernel uses it to check whether a timer interrupt is a periodic timer interrupt or a software forced interrupt.

```
#define OS_TVAL     T0TC              /* Timer Value */
```

  For a countdown timer, you must convert the return value. This is an example for a 16-bit count-down timer:

```
#define OS_TVAL     (0xFFFF - T0VAL)  /* Timer Value */
```

- **OS_TOVF** specifies a timer overflow flag. The RTX kernel uses it together with the OS_TVAL macro to differentiate between the periodic timer interrupts and forced timer interrupts.

```
#define OS_TOVF     (T0IR & 1)        /* Overflow Flag */
```

- **OS_TREL()** macro specifies a code sequence to reload a peripheral timer on overflow. When a peripheral timer has an auto-reload functionality, this macro is left empty.

```
#define OS_TREL()   ;                 /* Timer Reload */
```

- **OS_TFIRQ()** specifies a code sequence to force a timer interrupt. This must be a software triggered interrupt if the peripheral timer does not allow manual setting of an overflow flag. If manual setting is possible, this macro should set a peripheral timer overflow flag, which will cause a timer interrupt.

```
#define OS_TFIRQ()  VICSoftInt = OS_TIM_;  /* Force Interrupt */
```

- **OS_TIACK()** is used to acknowledge an interrupt from the timer interrupt function to release the timer interrupt logic.

```
#define OS_TIACK()  T0IR = 1;            /* Interrupt Ack */  \
                    VICSoftIntClr = OS_TIM_;                  \
                    VICVectAddr   = 0;
```

- **OS_TINIT()** macro is used to initialize the peripheral timer/counter, setup a timer mode, and set a timer reload. Timer interrupts are also activated here by enabling a peripheral timer interrupt. This code is executed from the **os_sys_init()** function.

```
#define OS_TINIT()  T0MR0 = OS_TRV;     /* Initialization */  \
                    T0MCR = 3;                                \
                    T0TCR = 1;                                \
                    VICDefVectAddr = (U32)os_def_interrupt;   \
                    VICVectAddr15  = (U32)os_clock_interrupt; \
                    VICVectCntl15  = 0x20 | OS_TID_;
```

- **OS_LOCK()** macro disables timer tick interrupts. It is used to avoid interrupting the system task scheduler. This macro should disable both the periodic timer interrupts and the forced interrupts. This code is executed from the **tsk_lock()** function.

```
#define OS_LOCK()   VICIntEnClr = OS_TIM_;    /* Task Lock */
```

- **OS_UNLOCK()** macro enables the timer tick interrupts. The code sequence specified here should enable the periodic timer interrupts and the forced interrupts. This code is executed from **tsk_unlock()** function.

```
#define OS_UNLOCK() VICIntEnable = OS_TIM_;  /* Task Unlock */
```

Library Files

RL-RTX includes eleven library files:

- **RTX_ARM_L.LIB** for microcontrollers based on ARM7TDMI™ and ARM9™ - Little Endian.
- **RTX_ARM_B.LIB** for microcontrollers based on ARM7TDMI™ and ARM9™ - Big Endian.
- **RTX_CM1.LIB** for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Little Endian.
- **RTX_CM1_B.LIB** for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Big Endian.
- **RTX_CM3.LIB** for microcontrollers based on Cortex™-M3 - Little Endian.
- **RTX_CM3_B.LIB** for microcontrollers based on Cortex™-M3 - Big Endian.
- **RTX_CM3X.LIB** for microcontrollers based on Cortex™-M3 without exclusive access instructions LDREX/STREX/CLREX - Little Endian.
- **RTX_CM4.LIB** for microcontrollers based on Cortex™-M4 - Little Endian.
- **RTX_CM4_B.LIB** for microcontrollers based on Cortex™-M4 - Big Endian.
- **RTX_CR4.LIB** for microcontrollers based on Cortex™-R4 - Little Endian.
- **RTX_CR4_B.LIB** for microcontrollers based on Cortex™-R4 - Big Endian.

All RL-ARM libraries are located in the **\Keil\ARM\RV31\LIB** folder. Depending on the target device selected for your project, the appropriate library file is automatically included into the link process when the RTX kernel operating system is selected for your project.

The **RTX_Lib_ARM.uvproj** and **RTX_Lib_CM.uvproj** projects found in the **\Keil\ARM\RL\RTX** folder are used to build the RL-RTX libraries.

Note

- You should not explicitly include any of the RL-RTX libraries in your application. That is done automatically when you use the **µVision® IDE**.

Related Knowledgebase Articles

- RL-ARM: INCOMPLETE TASK LIST DISPLAY WITH BIG ENDIAN

Using RTX Kernel

To use RTX Kernel for ARM7™/ARM9™ or Cortex™-M based applications, you must be able to successfully create RTX applications, compile and link them.

Writing Programs

When you write programs for RL-RTX, you can define RTX tasks using the **__task** keyword. You can use the RTX kernel routines whose prototypes are declared in **RTL.h**.

Include Files

The RTX kernel requires the use of only the **RTL.h** include file. All **library routines** and constants are defined in this header file. You can include it in your source files as follows:

```
#include <rtl.h>
```

Defining Tasks

Real-Time or multitasking applications are composed of one or more tasks that perform specific operations. The RTX kernel supports a maximum of 255 tasks.

Tasks are simply C functions that have a **void** return type, have a **void** argument list, and are declared using the **__task** function attribute. For example:

```
__task void func (void);
```

Where

func is the name of the task.

The following example defines the function task1 as a task. This task increments a counter indefinitely.

```
__task void task1 (void) {
  while (1) {
    counter0++;
  }
}
```

Note

- All tasks must be implemented as endless loops. A task must never return.
- The **__task** function attribute **must prefix** the task function declaration in RTX kernel version 3.40 and newer.

Multiple Instances

The RTX kernel enables you to run multiple copies of the same task at the same time. These are called **multiple instances** of the same task. You can simply create and run the same task several times. This is often required when you design a protocol based stack (like ISDN D channel stack).

The following example shows you how the function task2 can run in multiple instances.

```
#include <rtl.h>

OS_TID tsk_1, tsk2_1, tsk2_2, tsk2_3;
int cnt;

__task void task2 (void) {
  for (;;) {
    os_dly_wait (2);
    cnt++;
  }
}

__task void task1 (void) {
  /* This task will create 3 instances of task2 */
  tsk2_1 = os_tsk_create (task2, 0);
  tsk2_2 = os_tsk_create (task2, 0);
  tsk2_3 = os_tsk_create (task2, 0);
  /* The job is done, delete 'task1' */
  os_tsk_delete_self ();
}

void main (void) {
  os_sys_init(task1);
  for (;;);
}
```

Note

- Each instance of the same task must have a unique task ID.

External References

The **semaphore** and **mailbox** objects are referenced by the RTX kernel as typeless object pointers and are typecast inside the specific RTX kernel module. For semaphores and task handles, this is not a problem. The problem is when referencing the mailbox, which is declared using the macro **os_mbx_declare()**. That is why the **OS_MBX** type is defined. You have to use the OS_MBX object type identifier to reference mailboxes in external modules.

Here is an example of how the external RTX kernel objects are referenced:

```
extern OS_TID tsk1;
extern OS_SEM semaphore1;
extern OS_MUT mutex1;
extern OS_MBX mailbox1;
```

The following example shows you how to make a reference to a **mailbox** from a different C-module.

- C-Module with a **mailbox1** declaration:

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
  void *msg;

  os_mbx_init (mailbox1, sizeof (mailbox1));
  msg = alloc();
  /* set message content here */
  os_mbx_send (mailbox1, msg);
   ..
}
```

- C-Module with a **mailbox1** reference:

```
#include <RTL.h>

extern OS_MBX mailbox1;

__task void task2 (void) {
  void *msg;
   ..
  os_mbx_wait (mailbox1, &msg, 0xffff);
  /* process message content here */
  free (msg);
   ..
}
```

Using a Mailbox

The RTX kernel message objects are simply pointers to a block of memory where the relevant information is stored. There is no restriction regarding the message size or content. The RTX kernel handles only the pointer to this message.

Sending 8-bit, 16-bit, and 32-bit values

Because the RTX kernel passes only the pointer from the sending task to the receiving task, we can use the pointer itself to carry simple information like passing a character from a serial receive interrupt routine. An example can be found in the **serial.c** interrupt driven serial interface module for the **Traffic example**. You must cast the char to a pointer like in the following example:

```
  os_mbx_send (send_mbx, (void *)c, 0xffff);
```

Sending fixed size messages

To send fixed size messages, you must allocate a block of memory from the dynamic memory pool, store the information in it, and pass its **pointer** to a mailbox. The receiving task receives the pointer and restores the original information from the memory block, and then releases the allocated memory block.

Fixed Memory block memory allocation functions

RTX has very powerful **fixed memory block** memory allocation routines. They are **thread safe** and fully **reentrant**. They can be used with the RTX kernel with no restriction. It is better to use the fixed memory block allocation routines for sending fixed size messages. The memory pool needs to be properly initialized to the size of message objects:

- **32-bit values**: initialize to 4-byte block size.

```
    _init_box (mpool, sizeof(mpool), 4);
```

- **any size messages**: initialize to the size of message object.

```
    _init_box (mpool, sizeof(mpool), sizeof(struct message));
```

For 8-bit and 16-bit messages, it is better to use a parameter casting and convert a message value directly to a pointer.

The following example shows you how to send fixed size messages to a mailbox (see the mailbox example for more information). The message size is 8 bytes (two unsigned ints).

```
#include <rtl.h>

os_mbx_declare (MsgBox, 16);                /* Declare an RTX mailbox */
U32 mpool[16*(2*sizeof(U32))/4 + 3];        /* Reserve a memory for 16 messages */

__task void rec_task (void);

__task void send_task (void) {
  /* This task will send a message. */
  U32 *mptr;

  os_tsk_create (rec_task, 0);
  os_mbx_init (MsgBox, sizeof(MsgBox));
  mptr = _alloc_box (mpool);                /* Allocate a memory for the message */
  mptr[0] = 0x3215fedc;                     /* Set the message content. */
  mptr[1] = 0x00000015;
  os_mbx_send (MsgBox, mptr, 0xffff);       /* Send a message to a 'MsgBox' */
  os_tsk_delete_self ();
```

```
}

__task void rec_task (void) {
  /* This task will receive a message. */
  U32 *rptr, rec_val[2];

  os_mbx_wait (MsgBox, &rptr, 0xffff);      /* Wait for the message to arrive. */
  rec_val[0] = rptr[0];                     /* Store the content to 'rec_val' */
  rec_val[1] = rptr[1];
  _free_box (mpool, rptr);                  /* Release the memory block */
  os_tsk_delete_self ();
}

void main (void) {
  _init_box (mpool, sizeof(mpool), sizeof(U32));
  os_sys_init(send_task);
}
```

Sending variable size messages

To send a message object of variable size, you must use the memory allocation functions for the variable size memory blocks. The RVCT library provides these functions in **stdlib.h**

Note

- The **fixed block** memory allocation functions are **fully reentrant**. The **variable length** memory allocation functions are **not reentrant**. Therefore the system timer interrupts need to be disabled during the execution of the malloc() or free() function. Function **tsk_lock()** disables timer interrupts and function **tsk_unlock()** enables timer interrupts.

SWI Functions

Software Interrupt (SWI) functions are functions that run in Supervisor Mode of **ARM7™** and **ARM9™** core and are **interrupt protected**. SWI functions can accept arguments and can return values. They are used in the same way as other functions.

The difference is hidden to the user and is handled by the RealView C-compiler. It generates different code instructions to call SWI functions. SWI functions are called by executing the SWI instruction. When executing the SWI instruction, the controller changes the running mode to a Supervisor Mode and blocks any further IRQ interrupt requests. Note that the FIQ interrupts are not disabled in this mode. When the ARM controller leaves this mode, interrupts are enabled again.

If you want to use SWI functions in your RTX kernel project, you need to:

1. Copy the **SWI_Table.s** file to your project folder and include it into your project.
   This file is located in the **\Keil\ARM\RL\RTX\SRC\ARM** folder.
2. Declare a function with a **__swi(*x*)** attribute. Use the first SWI number, starting from 8, that is free.

```
void __swi(8)  inc_5bit (U32 *cp);
```

3. Write a function implementation and convert the function name into a **__SWI_*x*** function name. This name is referenced later by the linker from **SWI_Table.s** module.

```
void __SWI_8            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  *cp = (*cp + 1) & 0x1F;
}
```

4. Add the function **__SWI_*x*** to the SWI function table in the **SWI_Table.s** module.
   First import it from other modules:

```
; Import user SWI functions here.
                IMPORT  __SWI_8
```

   then add a reference to it into the table:

```
; Insert user SWI functions here. SWI 0..7 are used by RTL Kernel.
                DCD     __SWI_8                 ; SWI 8  User Function
```

5. Your **SWI** function should now look like this:

```
void __swi(8)  inc_5bit (U32 *cp);
void __SWI_8            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  *cp = (*cp + 1) & 0x1F;
}
```

Note

- SWI functions **0..7** are **reserved** for the RTX kernel.
- Do not leave gaps when numbering SWI functions. They must occupy a **continuous** range of numbers starting from 8.
- SWI functions can still be interrupted by **FIQ** interrupts.

Related Knowledgebase Articles

- RL-ARM: RTX KERNEL MODE USED IN ARM CPU

SVC Functions

Software Interrupt (SVC) functions are functions that run in Privileged Handler Mode of **Cortex™-M** core. SVC functions can accept arguments and can return values. They are used in the same way as other functions.

The difference is hidden to the user and is handled by the RealView C-compiler. It generates different code instructions to call SVC functions. SVC functions are called by executing the SVC instruction. When executing the SVC instruction, the controller changes the running mode to a Privileged Handler Mode.

Interrupts are **not disabled** in this mode. In order to protect SVC function from interrupts, you need to include the disable/enable intrinsic functions **__disable_irq()** and **__enable_irq()** in your code.

You may use SVC functions to access **protected peripherals**, for example to configure NVIC and interrupts. This is required if you run tasks in **unprivileged** (protected) mode and you need to change interrupts from the task.

If you want to use SVC functions in your RTX kernel project, you need to:

1. Copy the **SVC_Table.s** file to your project folder and include it into your project.
   This file is located in the **\Keil\ARM\RL\RTX\SRC\CM** folder.
2. Declare a function with a **__svc(*x*)** attribute. Use the first SVC number, starting from 1, that is free.

```
void __svc(1)  inc_5bit (U32 *cp);
```

3. Write a function implementation and convert the function name into a **__SVC_*x*** function name. This name is referenced later by the linker from **SVC_Table.s** module. You need also to disable/enable interrupts.

```
void __SVC_1            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  __disable_irq();
  *cp = (*cp + 1) & 0x1F;
  __enable_irq();
}
```

4. Add the function **__SVC_*x*** to the SVC function table in the **SVC_Table.s** module.
   First import it from other modules:

```
; Import user SVC functions here.
                IMPORT  __SVC_1
```

   then add a reference to it into the table:

```
; Insert user SVC functions here. SVC 0 used by RTL Kernel.
                DCD     __SVC_1                 ; user SVC function
```

5. Your **SVC** function should now look like this:

```
void __svc(1)  inc_5bit (U32 *cp);
void __SVC_1            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  __disable_irq();
  *cp = (*cp + 1) & 0x1F;
  __enable_irq();
}
```

Note

- SVC function **0** is **reserved** for the RTX kernel.
- Do not leave gaps when numbering SVC functions. They must occupy a **continuous** range of numbers starting from 1.

- SVC functions can still be interrupted.
- RTX must be **initialized** before SVC functions are called.

Debugging

The µVision Simulator allows you to run and test your RTX kernel applications. RTX kernel applications load just like non-RTX programs. No special commands or options are required for debugging.

A **kernel-aware** dialog displays all aspects of the **RTX kernel** and the tasks in your program. The simulator can be used also with your target hardware, if you are using a ULINK JTAG interface on your target, to debug your application.

Note

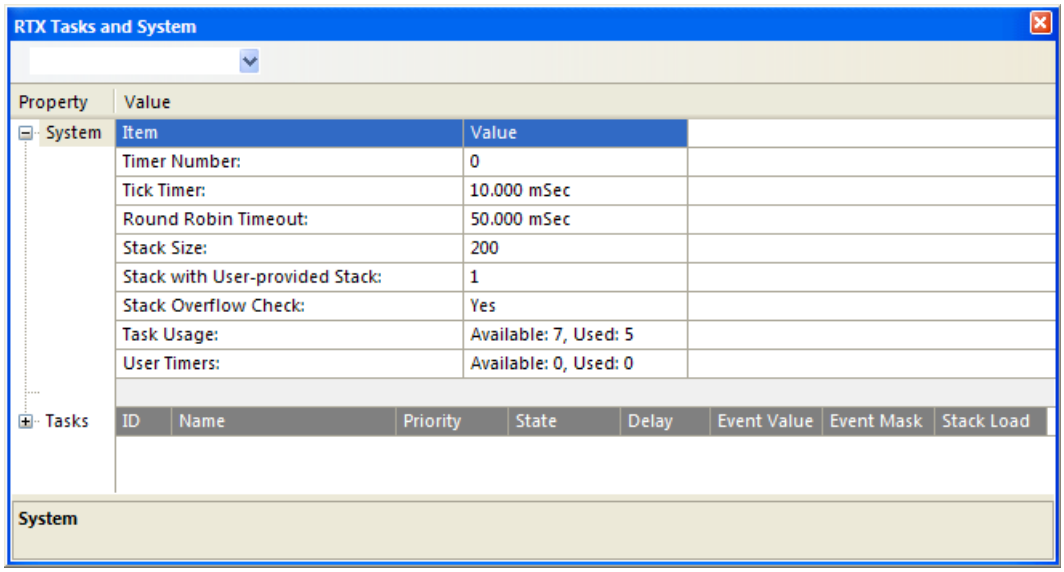
- You can have **source level debugging** if you enter the following SET variable into the debugger:

```
SET SRC=C:\Keil\ARM\RL\RTX\SRC
```

System Info

General information about the system resources and task usage is displayed by expanding the **System** property in the **RTX Tasks and System** dialog. You can use it to optimize your RTX application.

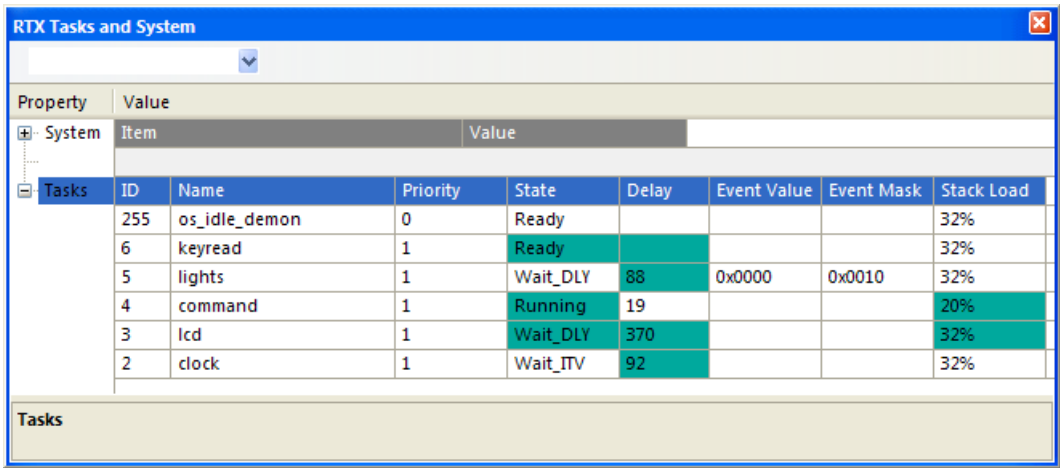
Select **RTX Tasks and System** from the **OS Support** item in the **Debug** menu to display this dialog.



Task Info

Detailed information about each running task is displayed when you expand the **Tasks** property in the **RTX Tasks and System** dialog. Note that one task can run in multiple instances. All active tasks are listed in this dialog.

Select **RTX Tasks and System** from the **OS Support** item in the **Debug** menu to display this dialog.



- **ID** is the Task Identification Value assigned when the task was started.
- **Name** is the **name** of the task function.
- **Priority** is the current task priority.
- **State** is the current **state of the task**.
- **Delay** is the delay timeout value for the task.
- **Event Value** specifies the event flags set for the task.
- **Event Mask** specifies the event flags mask for the events that the task is waiting for.
- **Stack Load** specifies the usage of the task's **stack**.

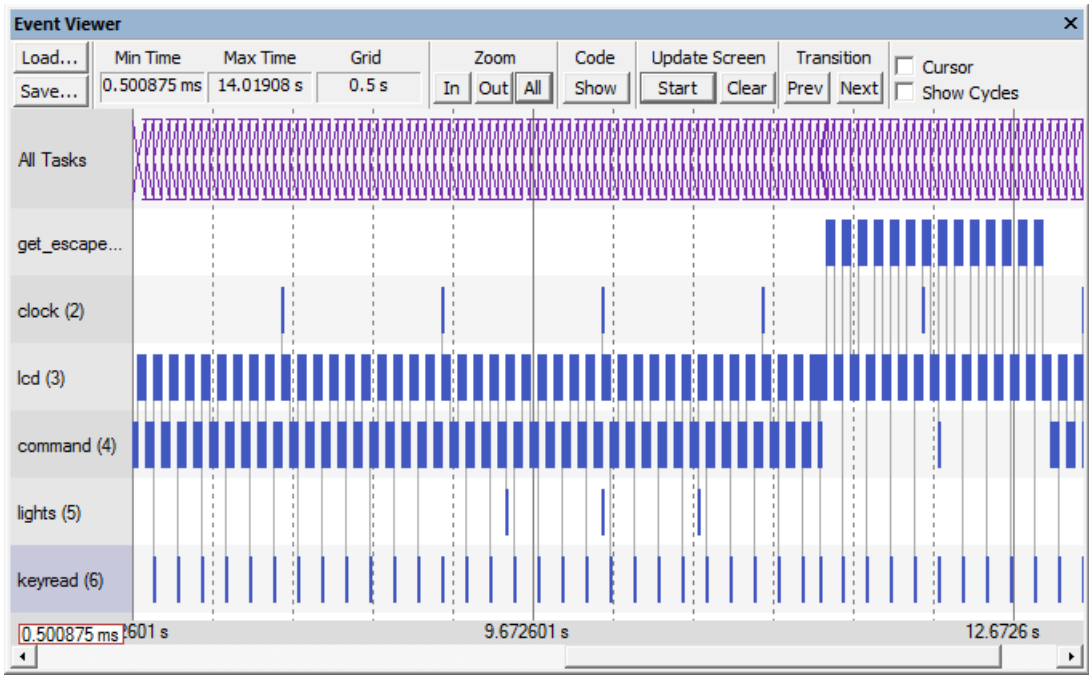
Related Knowledgebase Articles

- µVISION DEBUGGER: DISPLAYING RTX KERNEL VIEWER

Event Viewer

The Event Viewer displays a chronological history of task-switching events allowing to examine when tasks executed and for how long.

Select **Debug - OS Support - Event Viewer** to display this window.



Usage Hints

Here are a few hints to help you if you run into problems when using the RTX kernel.

Function usage

- Functions that begin with **os_** can be called from a task but not from an interrupt service routine.
- Functions that begin with **isr_** can be called from an **IRQ** interrupt service routine but not from a task.
- Never call **isr_** functions from **FIQ** (ARM7™, ARM9™) interrupt functions or from the task.
- Never call **tsk_lock()** or **tsk_unlock()** from an interrupt function.
- Before the kernel starts, never enable any **IRQ** interrupt that calls **isr_** functions.

Because of a two different implementations of RTX Kernel Library further hints depend on the Library version being used:

- **ARM7™/ARM9™ Version**
- **Cortex™-M Version**

ARM Version

Here are a few hints specific for ARM7™/ARM9™ library version.

Using IRQ interrupts

You can use **IRQ** interrupts with no limitation. RTX kernel uses only one timer interrupt to generate periodic **timer ticks** and activate the task scheduler.

- **IRQ** interrupts are disabled by RTX for a very short time (a few µseconds maximum).
- RTX kernel uses **Software Interrupts** to protect a critical code section from interrupts.
- Software interrupts **0-7** are used by RTX and cannot be used in your application.
- RTX uses its own **SWI Handler** which is automatically linked from the library. If you include another SWI handler (like that found in the **SWI.S** file) into your project, RTX could fail. Remove any user-created SWI handler from your project to resolve the Data Abort.
- Check the **IRQ stack size** configured from the startup file if you see sporadic crashes of your application. The IRQ stack usage depends on the complexity of your additional interrupt functions.

Using FIQ interrupts

ARM7™/ARM9™ Fast Interrupts are not used by the RTX kernel. You may freely use **FIQ** interrupts in you application in parallel with the kernel.

- **FIQ** interrupts are **never disabled** by RTX.
- You cannot call any kernel system function from the FIQ Interrupt Handler.

System Startup

RTX kernel uses a separate stack for each task it creates. The stack size is **configured** in the configuration file. However, before the kernel is started by the **os_sys_init()** function, the stack that is configured in the startup file **STARTUP.S** for the User Mode is used. When the RTX kernel is up and running, the User Mode stack is used for the task manager - an RTX task scheduler.

Minimum stack sizes for RTX kernel configured in **STARTUP.S** are:

- Supervisor Mode **32 bytes** (0x00000020)
- Interrupt Mode **64 bytes** (0x00000040)
- User Mode **80 bytes** (0x00000050)

Supervisor Mode stack is used when **SWI functions** are called. If you are using your own complex **__swi** functions, you might also need to increase the size of this stack.

Interrupt Mode stack is used on timer tick interrupts. This interrupt activates the system task scheduler. The scheduler uses the User/System Mode stack defined in **STARTUP.S** and runs in System Mode. If you are using interrupts in your application, you should increase the size of the Interrupt Mode stack. A stack size of 256 bytes is a good choice for a start. If the interrupt stack overflows, the application might crash.

User Mode stack is used until the kernel is started. It is better to initialize the user application from the first task which is created and started by the **os_sys_init()** function call.

You can initialize **simple IO**, like configure the port pins and enable AD converter, before the **os_sys_init()** function is called. The init_IO() function must be small and must not use many local variables because the User Mode stack can overflow otherwise.

```
void main (void) {
  /* Here a simple IO may be initialized. */
  init_IO ();
  os_sys_init (task1);
```

```
  /* The program execution shall never reach this point. */
  for (;;);
}
```

It is better to do a **complex initialization** from the first task that starts. In this case, the stack for this task is used, which is in general much bigger than User Mode stack.

```
__task void task1 (void) {
  /* Here the interrupts and more complex IO may be initialized. */
  Init_CAN ();
   ..
}
```

Cortex Version

Here are a few hints specific for Cortex™-M library version.

Using IRQ interrupts

You can use **IRQ** interrupts with no limitation. RTX kernel uses only one timer interrupt to generate periodic **timer ticks** and activate the task scheduler. Interrupt priority grouping can be used with some restrictions specified below.

- **IRQ** interrupts are never disabled by RTX Kernel.
- Software interrupt **0** is used by RTX and cannot be used in your application.
- RTX uses its own **SVC Handler** which is automatically linked from the library. If you include another SVC handler (like that found in the **SVC.S** file) into your project, RTX could fail. Remove any user-created SVC handler from your project to resolve the Hard Fault.
- When interrupt **priority grouping** is used, the PRIGROUP must be set before the **os_sys_init()** function is called. RTX kernel reads the value of PRIGROUP to correctly set internal interrupt pre-emption priorities.
- The **lowest two** pre-emption **priorities** are reserved for RTX kernel, all remaining pre-emption priorities are available to use in your application.
- Allowed values for **PRIGROUP** are from **0 to 6**. The PRIGROUP value 7 will cause RTX to fail.
- Check the **Main Stack size** configured from the startup file if you see sporadic crashes of your application. The RTX Kernel for Cortex™-M is implemented as a System Service Calls. All SVC calls use a Main Stack.

System Startup

RTX kernel uses a separate stack for each task it creates. The stack size is **configured** in the configuration file. However, before the kernel is started by the **os_sys_init()** function, the stack that is configured in the startup file **STARTUP.S** for the Main Stack is used.

Stack size used by RTX kernel is configured in **STARTUP.S**. Minimum size is **128 bytes**, however **256 bytes** is recommended when interrupts are used.

Main stack is also used when **SVC functions** are called. If you are using your own complex **__svc** functions, you might also need to increase the size of this stack.

You can initialize **simple IO**, like configure the port pins and enable AD converter, enable interrupts, before the **os_sys_init()** function is called. The init_IO() function is executed in **privileged** mode. It is recommended to configure peripherals in this function and use **unprivileged** mode for the tasks.

```
void main (void) {
  /* Here a simple IO may be initialized. */
  init_IO ();
   os_sys_init (task1);
  /* The code execution should never reach this point. */
  for (;;);
}
```

Create New RTX Application

This section describes how to create a new application that uses the RTX kernel.

1. First, **create a new project** in the µVision IDE by selecting **Project —> New Project**.
2. In the *Create New Project* window, select a new directory for your project and enter a name for your project.
3. In the *Select Device for Target* window, select your target ARM device and click OK. Allow µVision to copy and add the device startup file to your project. This creates a basic µVision project.
4. Now **setup the project** to use the RTX kernel. To do this, select **Project —> Options for Target**. Then select RTX Kernel for the **Operating system** and click OK.
5. Copy the RTX_Config.c configuration file for your target device from the **\Keil\ARM\Startup** directory:
   - for **ARM7™/ARM9™** devices, copy the configuration file for your specific device. If the file does not exist for your specific device, then copy the file from the Philips folder and modify it to suit your device.
   - for **Cortex-M™** devices, copy a generic **RTX_Conf_CM.c** configuration file.
6. Modify the device startup file for **ARM7™/ARM9™** devices to enable SWI_Handler function (no change required for **Cortex-M™** devices):
   - Comment out the following line from the startup file:

```
SWI_Handler    B    SWI_Handler
```

   - Add the following line to the startup file:

```
IMPORT    SWI_Handler
```

   This change prevents the code from sitting in a loop when a SWI interrupt occurs. The change allows the right function to run when a SWI interrupt occurs.
7. Copy the **retarget.c** file from **\Keil\ARM\Startup** to your project directory, and add it to your project. The main purpose of this file is to avoid the use of semihosting SWIs. Thus the file must contain the following:

```
#include <rt_misc.h>

#pragma import(__use_no_semihosting_swi)

void _ttywrch(int ch) {
  // Not used (No Output)
}

void _sys_exit(int return_code) {
label:  goto label;  /* endless loop */
}
```

   Depending on your application, you might have to retarget more functions. For example if you use the RL-FlashFS library, you can obtain retarget.c from the **\Keil\ARM\RL\FlashFS\SRC** directory. Now the project is setup to use the RTX kernel.

   Note

   - For **MicroLIB** runtime library you do not need a **retarget.c** in your project.
8. Now you must **configure the RTX kernel** for the needs of your application by making the required changes in the **RTX_Config.c** file.
9. **Create the application source files** if they do not already exist. Add these source files to the project. You can do this in the project workspace of µVision by right clicking on the Source Group and selecting **Add Files to Group**.
10. Build your application using **Project —> Build Target**.

11. If you project builds successfully, you can download it to your hardware or run it using the µVision Simulator. You can also debug the application using **Debug —> Start Debug Session**.

Related Knowledgebase Articles

- ARM: SETUP PROJECTS WITH RTX KERNEL

Function Reference

The RTX functions are grouped into the following categories:

Event Flag Management Routines

Event flag management routines enable tasks to send and wait for events from other tasks.

Routine	Attributes	Description
os_evt_clr		Clears one or more event flags of a task.
os_evt_get		Retrieves the event flags that caused **os_evt_wait_or** to complete.
os_evt_set		Sets one or more event flags of a task.
os_evt_wait_and		Waits for one or more event flags to be set.
os_evt_wait_or		Waits for any one event flag to be set.
isr_evt_set		Sets one or more event flags of a task.

Mailbox Management Routines

Routine	Attributes	Description
os_mbx_check		Determines the number of messages that can still be added to the mailbox.
os_mbx_declare		Creates a mailbox object.
os_mbx_init		Initializes a mailbox so that it can be used.
os_mbx_send		Sends a message to a mailbox.
os_mbx_wait		Gets the next message from a mailbox, or waits if the mailbox is empty.
isr_mbx_check		Determines the number of messages that can still be added to the mailbox.
isr_mbx_receive		Gets the next message from a mailbox.
isr_mbx_send		Sends a message to a mailbox.

Note

- The mailbox management routines enable you to send and receive messages between tasks using mailboxes.
- The **os_mbx_declare** routine is implemented as a macro.

Memory Allocation Routines

Routine	Attributes	Description
_declare_box		Creates a memory pool of fixed size blocks with 4-byte alignment.
_declare_box8		Creates a memory pool of fixed size blocks with 8-byte alignment.
_init_box		Initializes a memory pool with 4-byte aligned blocks.
_init_box8		Initializes a memory pool with 8-byte aligned blocks.
_alloc_box	Reentrant	Allocates a memory block from a memory pool.
_calloc_box	Reentrant	Allocates a memory block from a memory pool, and clears the contents of the block to 0.
_free_box	Reentrant	Returns a memory block back to its memory pool.

Note

- The memory allocation routines enable you to use the system memory dynamically by creating memory pools and using fixed size blocks from the memory pools.
- The **_init_box8**, **_declare_box** and **_declare_box8** routines are implemented as macros.

Mutex Management Routines

Routine	Attributes	Description
os_mut_init		Initializes a mutex object.
os_mut_release		Releases a mutex object.
os_mut_wait		Waits for a mutex object to become available.

Note

- The mutex management routines enable you to use mutexes to synchronize the activities of the various tasks and to protect shared variables from corruption.
- The **priority inheritance** method is used in mutex management routines to eliminate **priority inversion** problems.

Semaphore Management Routines

Routine	Attributes	Description
os_sem_init		Initializes a semaphore object.
os_sem_send		Sends a signal (token) to the semaphore.
os_sem_wait		Waits for a signal (token) from the semaphore.
isr_sem_send		Sends a signal (token) to the semaphore.

Note

- The semaphore management routines enable you to use semaphores to synchronize the activities of the various tasks and to protect shared variables from corruption.

Related Knowledgebase Articles

- RL-ARM: ACQUIRE/RELEASE OR BINARY SEMAPHORE IN RTX KERNEL

System Functions

Routine	Attributes	Description
tsk_lock		Disables task switching.
tsk_unlock		Enables task switching.

Note

- The system functions enable you to control the timer interrupt and task switching.

Task Management Routines

Routine	Attributes	Description
os_sys_init		Initializes and starts RL-RTX.
os_sys_init_prio		Initializes and starts RL-RTX assigning a priority to the starting task.
os_sys_init_user		Initializes and starts RL-RTX assigning a priority and custom stack to the starting task.
os_tsk_create		Creates and starts a new task.
os_tsk_create_ex		Creates, starts, and passes an argument pointer to a new task.
os_tsk_create_user		Creates, starts, and assigns a custom stack to a new task.
os_tsk_create_user_ex		Creates, starts, assigns a custom stack, and passes an argument pointer to a new task.
os_tsk_delete		Stops and deletes a task.
os_tsk_delete_self		Stops and deletes the currently running task.
os_tsk_pass		Passes control to the next task of the same priority.
os_tsk_prio		Changes a task's priority.
os_tsk_prio_self		Changes the currently running task's priority.
os_tsk_self		Obtains the task ID of the currently running task.
isr_tsk_get		Obtains the task ID of the interrupted task.

Note

- The task management routines enable you to start the RTX kernel, create and delete various types of tasks, and control their execution priorities.

Time Management Routines

Routine	Attributes	Description
os_dly_wait		Pauses the calling task for a specified interval.
os_itv_set		Enables the calling task for periodic wake up.
os_itv_wait		Pauses the calling task until the periodic wake up interval expires.

Note

- The time management routines enable you to pause and restart tasks using a timer.

User Timer Management Routines

Routine	Attributes	Description
os_tmr_create		Starts a countdown timer to call the **os_tmr_call** function.
os_tmr_kill		Aborts a user defined timer.
os_tmr_call		User customizable function that gets called when the user defined timer expires.

Note

- The **user timer** management routines enable you to use timers to control when a user customizable function runs.

Library Reference

The Real-Time Library provides more than 50 predefined functions and macros you may use in your ARM® real-time programs. The library makes embedded software development easier by providing routines that perform common real-time tasks.

Reference

The following pages describe the routines in the Real-Time Library. Routines are listed in alphabetical order and each is divided into several sections:

Summary	Briefly describes the routine's effect, lists include file(s) containing its declaration and prototype, illustrates the syntax, and describes any arguments.
Description	Provides a detailed description of the routine and how it is used.
Return Value	Describes the value returned by the routine.
See Also	Names related routines.
Example	Gives a function or program fragment demonstrating proper use of the function.

_alloc_box

Summary

```
#include <rtl.h>

void *_alloc_box (
    void* box_mem );    /* Start address of the memory pool */
```

Description

The **_alloc_box** function allocates a block of memory from the memory pool that begins at the address *box_mem*.

The **_alloc_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the memory pool using the **_init_box** function before performing any other operation on the memory pool.
- The **_alloc_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_alloc_box** function returns a pointer to the allocated block if a block was available. If there was no available block, it returns a NULL pointer.

See Also

_calloc_box, **_free_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_calloc_box

Summary

```
#include <rtl.h>

void *_calloc_box (
    void* box_mem );    /* Start address of the memory pool */
```

Description

The **_calloc_box** function allocates a block of memory from the memory pool that begins at the address *box_mem* and initializes the entire memory block to 0.

The **_calloc_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the memory pool using the **_init_box** function before performing any other operation on the memory pool.
- The **_calloc_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_calloc_box** function returns a pointer to the allocated block if a block was available. If there was no available block, it returns a NULL pointer.

See Also

_alloc_box, **_free_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_declare_box

Summary

```
#include <rtl.h>

#define _declare_box( \
    pool,       \    /* Name of the memory pool variable. */
    size,       \    /* Number of bytes in each block. */
    cnt )       \    /* Number of blocks in the memory pool. */
    U32 pool[((size+3)/4)*(cnt) + 3]
```

Description

The **_declare_box** macro declares an array of bytes that can be used as a memory pool for fixed block allocation.

The argument *pool* specifies the name of the memory pool variable, which can be used by the memory block allocation routines. The argument *size* specifies the size of the blocks, in bytes. The argument *cnt* specifies the number of blocks required in the memory pool.

The **_declare_box** macro is part of RL-RTX. The definition is in rtl.h.

Note

- The macro rounds up the value of *size* to the next multiple of 4 to give the blocks a 4-byte alignment.
- The macro also declares an additional 12 bytes at the start of the memory pool to store internal pointers and size information about the memory pool.

Return Value

The _declare_box macro does not return any value.

See Also

_alloc_box, **_calloc_box**, **_free_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
_declare_box(mpool,20,32);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_declare_box8

Summary

```
#include <rtl.h>

#define _declare_box8( \
    pool,       \    /* Name of the memory pool variable. */
    size,       \    /* Number of bytes in each block. */
    cnt )       \    /* Number of blocks in the memory pool. */
    U64 pool[((size+7)/8)*(cnt) + 2]
```

Description

The **_declare_box8** macro declares an array of bytes that can be used as a memory pool for allocation of fixed blocks with 8-byte alignment.

The argument *pool* specifies the name of the memory pool variable that is used by the memory block allocation routines. The argument *size* specifies the size of the blocks, in bytes. The argument *cnt* specifies the number of blocks required in the memory pool.

The **_declare_box8** macro is part of RL-RTX. The definition is in rtl.h.

Note

- The macro rounds up the value of *size* to the next multiple of 8 to give the blocks an 8-byte alignment.
- The macro also declares an additional 16 bytes at the start of the memory pool to store internal pointers and size information about the memory pool.

Return Value

The **_declare_box8** macro does not return any value.

See Also

_alloc_box, **_calloc_box**, **_free_box**, **_init_box8**

Example

```
#include <rtl.h>

/* Reserve a memory for 25 blocks of 30-bytes. */
_declare_box8(mpool,30,25);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box8 (mpool, sizeof (mpool), 30);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
  /* 'box' and 'cbox' are always 8-byte aligned. */
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_free_box

Summary

```
#include <rtl.h>

int _free_box (
    void* box_mem,    /* Start address of the memory pool */
    void* box );      /* Pointer to the block to free */
```

Description

The **_free_box** function returns a memory block, which was allocated using **_alloc_box** or **_calloc_box**, back to the memory pool where it was obtained from.

The *box* argument specifies the address of the memory block to be freed.

The *box_mem* argument specifies the start address of the memory pool where the block was obtained from.

The **_free_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- If you return the memory block to a memory pool that did not provide the memory block, serious memory errors might occur.
- The **_free_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_free_box** function returns 0 if the memory block was successfully returned to the memory pool. If there was an error while freeing the block, it returns 1.

See Also

_alloc_box, **_calloc_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_init_box

Summary

```
#include <rtl.h>

int _init_box  (
    void* box_mem,       /* Start address of the memory pool */
    U32   box_size,      /* Number of bytes in the memory pool */
    U32   blk_size );    /* Number of bytes in each block of the pool */
```

Description

The **_init_box** function initializes a fixed block size memory pool. When the memory pool is initialized, the RTX kernel handles memory requests by allocating a block of memory from the memory pool.

The *box_mem* specifies the start address of the memory pool, and this address must be 4-byte aligned.

The *box_size* argument specifies the size of the memory pool, in bytes.

The *blk_size* argument specifies the size, in bytes, of the blocks in the memory pool. You can set the block size to any value from 1 to *box_size*-12. However, the *blk_size* is rounded up to the next multiple of 4 to maintain 4-byte address alignment of the blocks. For example if you initialize a memory pool for 10-byte blocks, the **_init_box** function actually initializes the memory pool for 12-byte blocks.

The **_init_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The first 12 bytes from the memory pool are reserved for storing pointers and size information that can be used by the functions that handle the memory pool. The *box_size* must therefore be more than 12 bytes long.
- If the start address is not 4-byte aligned, the memory pool handling functions might fail.

Return Value

The **_init_box** function returns 0 if the memory pool was initialized without any problem. If there was an initialization error, it returns 1.

See Also

_alloc_box, **_calloc_box**, **_declare_box**, **_free_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
_declare_box(mpool,20,32);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_init_box8

Summary

```
#include <rtl.h>

int _init_box8 (
    void* box_mem,       /* Start address of the memory pool */
    U32   box_size,      /* Number of bytes in the memory pool */
    U32   blk_size );    /* Number of bytes in each block of the pool */
```

Description

The **_init_box8** function initializes a fixed block size memory pool with 8-byte alignment. When the memory pool is initialized, the RTX kernel handles memory requests by allocating a block of memory from the memory pool.

The *box_mem* specifies the start address of the memory pool, and this address must be 8-byte aligned.

The *box_size* argument specifies the size of the memory pool, in bytes.

The *blk_size* argument specifies the size, in bytes, of the blocks in the memory pool. You can set the block size to any value from 1 to *box_size*-16. However, the *blk_size* is rounded up to the next multiple of 8, to maintain 8-byte alignment of the blocks. For example if you initialize a memory pool for 10-byte blocks, the **_init_box8** function actually initializes the memory pool for 16-byte blocks.

The **_init_box8** function is implemented as a macro and is part of RL-RTX. The definition is in rtl.h.

Note

- The first 16 bytes from the memory pool are reserved for storing pointers and size information that can be used by the functions that handle the memory pool. The *box_size* must therefore be more than 16 bytes long.
- If the start address is not 8-byte aligned, the memory pool handling functions might fail.

Return Value

The **_init_box8** function returns 0 if the memory pool was initialized without any problem. If there was an initialization error, it returns 1.

See Also

_alloc_box, **_calloc_box**, **_declare_box8**, **_free_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 25 blocks of 30-bytes. */
_declare_box8(mpool,30,25);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box8 (mpool, sizeof (mpool), 30);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
  /* 'box' and 'cbox' are always 8-byte aligned. */
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

isr_evt_set

Summary

```
#include <rtl.h>

void isr_evt_set (
    U16    event_flags,    /* Bit pattern of event flags to set */
    OS_TID task );         /* The task that the events apply to */
```

Description

The **isr_evt_set** function sets the event flags for the *task* identified by the function argument. The function only sets the event flags whose corresponding bit is set to 1 in the *event_flags* argument.

The **isr_evt_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can call the **isr_evt_set** function only from the IRQ interrupt service routine. You cannot call it from the FIQ interrupt service routine.
- When the **isr_evt_se**t function is called too frequently, it forces too many tick timer interrupts and the task manager task scheduler is executed most of the time. It might happen that two **isr_evt_set** functions for the same task are called before the task gets a chance to run from one of the event waiting functions (os_evt_wait_). Of course one event is lost because event flags are not counting objects.

Return Value

The **isr_evt_set** function does not return any value.

See Also

os_evt_clr, **os_evt_set**, **os_evt_wait_and**, **os_evt_wait_or**

Example

```
#include <rtl.h>

void timer1 (void) __irq {
   ..
  isr_evt_set (0x0008, tsk1);
   ..
}
```

isr_mbx_check

Summary

```
#include <rtl.h>

OS_RESULT isr_mbx_check (
    OS_ID mailbox );    /* The mailbox to check for free space */
```

Description

The **isr_mbx_check** function determines the number of messages that can still be added into the *mailbox* identified by the function argument. You can avoid losing the message by calling the **isr_mbx_check** function to check for available space in the mailbox before calling the **isr_mbx_send** function to send a message.

The **isr_mbx_check** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can call the **isr_mbx_check** function only from the IRQ interrupt service routine. You cannot call it from the FIQ interrupt service routine.
- When sending more than one message from ISR, the mailbox might overflow, because the **isr_mbx_send** puts the messages in the fifo queue, not directly to the mailbox.

Return Value

The **isr_mbx_check** function returns the number of message entries in the mailbox that are free.

See Also

isr_mbx_send, **os_mbx_declare**, **os_mbx_init**

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

void timer1 (void) __irq {
   ..
  if (isr_mbx_check (mailbox1) != 0) {
    isr_mbx_send (mailbox1, msg);
  }
   ..
}
```

Example for sending more than one message from ISR:

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

void timer1 (void) __irq {
  int i,free;
   ..
  free = isr_mbx_check (mailbox1);
  for (i = 0; i < 16; i++) {
    if (free > 0) {
      free--;
      isr_mbx_send (mailbox1, msg);
    }
  }
   ..
}
```

isr_mbx_receive

Summary

```
#include <rtl.h>

OS_RESULT isr_mbx_receive (
    OS_ID  mailbox,      /* The mailbox to put the message in */
    void** message );    /* Location to store the message pointer */
```

Description

The **isr_mbx_receive** function gets a pointer to a message from the *mailbox* if the mailbox is not empty. The function puts the message pointer from the mailbox into the location pointed by the *message* argument. The **isr_mbx_receive** function does not cause the current task to sleep even if there is no message in the mailbox. Hence this function can be called from an interrupt function.

You can use the **isr_mbx_receive** function to receive a message or a protocol frame (for example TCP-IP, UDP, and ISDN) in an interrupt function.

The **isr_mbx_receive** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- You can call the **isr_mbx_receive** function only from IRQ interrupt functions. You cannot call it from the FIQ interrupt function.
- When you get the message from the mailbox, you must free the memory block containing the message to avoid running out of memory.
- When you get the message from the mailbox, space is created in the mailbox for a new message.

Return Value

The **isr_mbx_receive** function returns the completion value:

Return Value	Description
OS_R_MBX	A message was available and was read from the mailbox.
OS_R_OK	No message was available in the mailbox.

See Also

isr_mbx_send, **os_mbx_declare**, **os_mbx_init**, **os_mbx_wait**

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);
void *msg;

void EtherInt (void) __irq {
   ..
  if (isr_mbx_receive (mailbox1, &msg) == OS_R_MBX) {
    /* Transmit Ethernet frame. */
  }
  else {
    /* No message was available, stop transmitter */
  }
   ..
}
```

isr_mbx_send

Summary

```
#include <rtl.h>

void isr_mbx_send (
    OS_ID mailbox,          /* The mailbox to put the message in */
    void* message_ptr );    /* Pointer to the message */
```

Description

The **isr_mbx_send** function puts the pointer to the message *message_ptr* in the *mailbox* if the mailbox is not already full. The **isr_mbx_send** function does not cause the current task to sleep even if there is no space in the mailbox to put the message.

When an interrupt receives a protocol frame (for example TCP-IP, UDP, or ISDN), you can call the **isr_mbx_send** function from the interrupt to pass the protocol frame as a message to a task.

The **isr_mbx_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- You can call the **isr_mbx_send** function only from IRQ interrupt functions. You cannot call it from FIQ interrupt functions.
- If the *mailbox* is full, the RTX kernel ignores the message since it cannot be put into the mailbox, and calls the **error function**. Thus before sending a message using **isr_mbx_send**, you must use the **isr_mbx_check** function to check if the mailbox is full.

Return Value

The **isr_mbx_send** function does not return any value.

See Also

isr_mbx_check, **isr_mbx_receive**, **os_mbx_declare**, **os_mbx_init**

Example

```
#include <RTL.h>

os_mbx_declare (mailbox1, 20);

void timer1 (void) __irq {
   ..
  isr_mbx_send (mailbox1, msg);
   ..
}
```

isr_sem_send

Summary

```
#include <rtl.h>

void isr_sem_send (
    OS_ID semaphore );  /* The semaphore whose token count is incremented */
```

Description

The **isr_sem_send** function increments the number of tokens in the *semaphore* object.

The **isr_sem_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.
- You can call the **isr_sem_send** function only from IRQ interrupt functions and not from the FIQ interrupt function.

Return Value

The **isr_sem_send** function does not return any value.

See Also

os_sem_init, **os_sem_send**, **os_sem_wait**

Example

```
#include <rtl.h>

OS_SEM semaphore1;

void timer1 (void) __irq {
   ..
  isr_sem_send (semaphore1);
   ..
}
```

isr_tsk_get

Summary

```
#include <rtl.h>

OS_TID isr_tsk_get (void);
```

Description The **isr_tsk_get** function identifies the interrupted task by returning its task ID.

The **isr_tsk_get** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value The **isr_tsk_get** function returns the task identifier number (TID) of the interrupted task.

See Also **os_tsk_self**

Example

```
#include <rtl.h>

void os_error (U32 err_code) {
  /* This function is called when a runtime error is detected. */
  OS_TID err_task;

  switch (err_code) {
    case OS_ERR_STK_OVF:
      /* Identify the task with stack overflow. */
      err_task = isr_tsk_get();
      break;
    case OS_ERR_FIFO_OVF:
      break;
    case OS_ERR_MBX_OVF:
      break;
  }
  for (;;);
}
```

Related Knowledgebase Articles

- RL-ARM: TRANSITION FROM OS_STK_OVERFLOW() TO OS_ERROR()

os_dly_wait

Summary

```
#include <rtl.h>

void os_dly_wait (
    U16 delay_time );    /* Length of time to pause */
```

Description

The **os_dly_wait** function pauses the calling task. The argument *delay_time* specifies the length of the pause and is measured in number of system_ticks. You can set the *delay_time* to any value between 1 and 0xFFFE.

The **os_dly_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You cannot intermix with a single task the wait method **os_itv_wait ()** and **os_dly_wait ()**.

Return Value

The **os_dly_wait** function does not return any value.

See Also

os_itv_set, **os_itv_wait**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_dly_wait (20);
   ..
}
```

Related Knowledgebase Articles

- RL-ARM: WRONG TIMING WITH OS_ITV_SET()

os_evt_clr

Summary

```
#include <rtl.h>

void os_evt_clr (
    U16    clear_flags,    /* Bit pattern of event flags to clear */
    OS_TID task );         /* The task that the events apply to */
```

Description

The **os_evt_clr** function clears the event flags for the task identified by the function argument. The function only clears the event flags whose corresponding bit is set to 1 in the *clear_flags* argument.

The **os_evt_clr** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_evt_clr** function does not return any value.

See Also

isr_evt_set, **os_evt_set**, **os_evt_wait_and**, **os_evt_wait_or**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_evt_clr (0x0002, tsk2);
   ..
}
```

os_evt_get

Summary

```
#include <rtl.h>

U16 os_evt_get (void);
```

Description

You can use the **os_evt_get** function to identify the event that caused the **os_evt_wait_or** function to complete.

The **os_evt_get** function identifies this event by setting the corresponding flag in the returned value. If more than one event occurred simultaneously, all their flags are set in the returned value.

The **os_evt_get** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- When the **os_evt_wait_or** function has been waiting on more than one event, it is not immediately known which event caused the **os_evt_wait_or** function to return. This is why the **os_evt_get** function is useful.

Return Value

The **os_evt_get** function returns a bit pattern that identifies the events that caused the **os_evt_wait_or** function to complete.

See Also

os_evt_wait_or

Example

```
#include <RTL.h>

__task void task1 (void) {
  U16 ret_flags;

  if (os_evt_wait_or (0x0003, 500) == OS_R_EVT) {
    ret_flags = os_evt_get ();
    printf("Events %04x received.\n",ret_flags);
  }
   ..
}
```

os_evt_set

Summary

```
#include <rtl.h>

void os_evt_set (
    U16    event_flags,    /* Bit pattern of event flags to set */
    OS_TID task );         /* The task that the events apply to */
```

Description

The **os_evt_set** function sets the event flags for the *task* identified by the function argument. The function only sets the event flags whose corresponding bit is set to 1 in the *event_flags* argument.

The **os_evt_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_evt_set** function does not return any value.

See Also

isr_evt_set, **os_evt_clr**, **os_evt_wait_and**, **os_evt_wait_or**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_evt_set (0x0003, tsk2);
   ..
}
```

os_evt_wait_and

Summary

```
#include <rtl.h>

OS_RESULT os_evt_wait_and (
    U16 wait_flags,    /* Bit pattern of events to wait for */
    U16 timeout );     /* Length of time to wait for event */
```

Description

The **os_evt_wait_and** function waits for all the events specified in the *wait_flags* to occur. The function only waits on events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The function can wait on as many as 16 different events.

You can use the *timeout* argument to specific the length of time after which the function must return even if none of the events have occurred. You can use any value of timeout with the exception of 0xFFFF, which you can use to specify an indefinite timeout. The unit of measure of the *timeout* argument is the number of system intervals.

The **os_evt_wait_and** function returns when all of the events specified in the *wait_flags* have occurred or when the timeout expires. If all events specified in *wait_flags* have arrived, this function clears them before the function returns. The function actually clears the events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The other event flags are not changed.

The **os_evt_wait_and** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Each task has its own 16 bit wait flag.

Return Value

The **os_evt_wait_and** function returns a value to indicate whether an event occurred or the timeout expired.

Return Value	Description
OS_R_EVT	All the flags specified by **wait_flags** have been set.
OS_R_TMO	The timeout has expired.

See Also

os_evt_get, **os_evt_wait_or**

Example

```
#include <rtl.h>

__task void task1 (void) {
  OS_RESULT result;

  result = os_evt_wait_and (0x0003, 500);
  if (result == OS_R_TMO) {
    printf("Event wait timeout.\n");
  }
  else {
    printf("Event received.\n");
  }
   ..
}
```

os_evt_wait_or

Summary

```
#include <rtl.h>

OS_RESULT os_evt_wait_or (
    U16 wait_flags,    /* Bit pattern of events to wait for */
    U16 timeout );     /* Length of time to wait for event */
```

Description

The **os_evt_wait_or** function waits for one of the events specified in the *wait_flags* to occur. The function only waits on events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The function can wait on as many as 16 different events.

You can use the *timeout* argument to specific the length of time after which the function must return even if none of the events have occurred. You can use any value of timeout with the exception of 0xFFFF, which you can use to specify an indefinite timeout. The unit of measure of the *timeout* argument is the number of system intervals.

The **os_evt_wait_or** function returns when at least one of the events specified in the *wait_flags* has occurred or when the timeout expires. The event flag or flags that caused the **os_evt_wait_or** function to complete are cleared before the function returns. You can identify those event flags with **os_evt_get** function later.

The **os_evt_wait_or** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Each task has its own 16 bit wait flag.

Return Value

The **os_evt_wait_or** function returns a value to indicate whether an event occurred or the timeout expired.

Return Value	Description
OS_R_EVT	At least one of the flags specified by *wait_flags* has been set.
OS_R_TMO	The timeout has expired.

See Also

os_evt_get, **os_evt_wait_and**

Example

```
#include <rtl.h>

__task void task1 (void) {
  OS_RESULT result;

  result = os_evt_wait_or (0x0003, 500);
  if (result == OS_R_TMO) {
    printf("Event wait timeout.\n");
  }
  else {
    printf("Event received.\n");
  }
   ..
}
```

os_itv_set

Summary

```
#include <rtl.h>

void os_itv_set (
    U16 interval_time );    /* Time interval for periodic wake-up */
```

Description

The **os_itv_set** function sets up the calling task for periodic wake-up after a time interval specified by *interval_time*. You must use the **os_itv_wait** function to wait for the completion of the time interval. The time interval is measured in units of system ticks, and you can set it to any value between 1 and 0x7FFF.

The **os_itv_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_itv_set** function does not return any value.

See Also

os_dly_wait, **os_itv_wait**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_itv_set (50);
   ..
}
```

os_itv_wait

Summary

```
#include <rtl.h>

void os_itv_wait (void);
```

Description

The **os_itv_wait** function waits for a periodic time interval after which the RTX kernel wakes up the calling task. You must set the time interval using the **os_itv_set** function.

you can use the **os_itv_wait** function to perform a job at regular intervals independent of the execution time of the task.

The **os_itv_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You cannot intermix with a single task the wait method **os_itv_wait ()** and **os_dly_wait ()**.

Return Value

The **os_itv_wait** function does not return any value.

See Also

os_dly_wait, **os_itv_set**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_itv_set (20);
  for (;;) {
    os_itv_wait ();
    /* do some actions at regular time intervals */
  }
}
```

Related Knowledgebase Articles

- RL-ARM: WRONG TIMING WITH OS_ITV_SET()

os_mbx_check

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_check (
    OS_ID mailbox );    /* The mailbox to check for free space */
```

Description

The **os_mbx_check** function determines the number of messages that can still be added into the *mailbox* identified by the function argument. You can avoid blocking the current task by calling the **os_mbx_check** function to check for available space in the mailbox before calling the **os_mbx_send** function to send a message.

The **os_mbx_check** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_mbx_check** function returns the number of message entries in the mailbox that are free.

See Also

isr_mbx_check, **os_mbx_declare**, **os_mbx_send**

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  if (os_mbx_check (mailbox1) == 0) {
    printf("Mailbox is full.\n");
  }
   ..
}
```

os_mbx_declare

Summary

```
#include <rtl.h>

#define os_mbx_declare( \
    name,               \ /* Name of the mailbox */
    cnt )               \ /* Number of message entries */
U32 name [4 + cnt]
```

Description

The **os_mbx_declare** macro defines a mailbox object. The argument *name* is the name of the mailbox object. The argument *cnt* is the number of messages that can be entered into the mailbox object. A cnt value of 20 is sufficient in most cases.

The **os_mbx_declare** macro is part of RL-RTX. The definition is in rtl.h.

Return Value

The **os_mbx_declare** macro does not return any value.

See Also

os_mbx_init

Example

```
#include <rtl.h>

/* Declare a mailbox for 20 messages. */
os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  os_mbx_init (mailbox1, sizeof(mailbox1));
   ..
}
```

os_mbx_init

Summary

```
#include <rtl.h>

void os_mbx_init (
    OS_ID mailbox,       /* The mailbox to initialize */
    U16   mbx_size );    /* Number of bytes in the mailbox */
```

Description

The **os_mbx_init** function initializes the *mailbox* object identified by the function argument.

The argument *mbx_size* specifies the size of the mailbox, in bytes. However, the number of message entries in the mailbox is defined by the **os_mbx_declare** macro.

The **os_mbx_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox before you perform any operation on it.

Return Value

The **os_mbx_init** function does not return any value.

See Also

os_mbx_declare

Example

```
#include <rtl.h>

/* Declare a mailbox for 20 messages. */
os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  os_mbx_init (mailbox1, sizeof(mailbox1));
   ..
}
```

os_mbx_send

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_send (
    OS_ID mailbox,        /* The mailbox to put the message in */
    void* message_ptr,    /* Pointer to the message */
    U16   timeout );      /* Wait time for mailbox to be free */
```

Description

The **os_mbx_send** function puts the pointer to a message, *message_ptr*, in the *mailbox*, if the mailbox is not already full.

If the mailbox is full, the RTX kernel puts the calling task to sleep. The *timeout* specifies the length of time the task can wait for a space to become available in the mailbox. The kernel wakes up the task either when the timeout has expired or when a space becomes available in the mailbox.

You can set the timeout to any value between 0 and 0xFFFE. You can set the timeout to 0xFFFF for an indefinite timeout.

The **os_mbx_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- The unit of measure of the *timeout* argument is numbers of system intervals.
- The *message_ptr* points to a block of allocated memory holding a message of any type. The block of memory is allocated when the message is created and freed by the destination task when the message is received.

Return Value

The function returns the completion value:

Return Value	Description
OS_R_TMO	The timeout has expired.
OS_R_OK	The message has been put in the mailbox.

See Also

isr_mbx_send, **os_mbx_check**, **os_mbx_declare**, **os_mbx_init**, **os_mbx_wait**

Example

```
#include <RTL.h>

os_mbx_declare (mailbox1, 20);
OS_TID tsk1, tsk2;

__task void task1 (void);
__task void task2 (void);

__task void task1 (void) {
  void *msg;
   ..
  tsk2 = os_tsk_create (task2, 0);
  os_mbx_init (mailbox1, sizeof(mailbox1));
  msg = alloc();
  /* set message content here*/
  os_mbx_send (mailbox1, msg, 0xFFFF);
   ..
}

__task void task2 (void) {
  void *msg;
   ..
  os_mbx_wait (mailbox1, &msg, 0xffff);
  /* process message content here */
  free (msg);
   ..
```

```
}
```

os_mbx_wait

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_wait (
    OS_ID  mailbox,      /* The mailbox to get message from */
    void** message,      /* Location to store the message pointer */
    U16    timeout );    /* Wait time for message to become available */
```

Description

The **os_mbx_wait** function gets a pointer to a message from the *mailbox* if the mailbox is not empty. The function puts the message pointer from the mailbox into the location pointed by the *message* argument.

If the mailbox is empty, the RTX kernel puts the calling task to sleep. The *timeout* specifies the length of time the task can wait for a message. The kernel wakes up the task either when the timeout expires or when a message becomes available in the mailbox.

You can set the timeout to any value between 0 and 0xFFFE. You can set the timeout to 0xFFFF for an indefinite timeout. If you specify the *timeout* to 0, the calling task continues immediately even if there are higher priority tasks in the ready list, irrespective of whether a message is present in the mailbox or not.

The **os_mbx_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- The unit of measure of the *timeout* argument is numbers of system intervals.
- When you get the message from the mailbox, you must free the memory block containing the message to avoid running out of memory.
- When you get the message from the mailbox, space is created in the mailbox for a new message.

Return Value

The **os_mbx_wait** function returns a completion value:

Return Value	Description
OS_R_MBX	The task has waited until a message was put in the mailbox.
OS_R_TMO	The timeout specified by timeout has expired before a message was available in the mailbox.
OS_R_OK	A message was available in the mailbox, and the task continues without waiting.

See Also

isr_mbx_receive, **os_mbx_declare**, **os_mbx_init**, **os_mbx_send**

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void){
  void *msg;
   ..
  if (os_mbx_wait (mailbox1, &msg, 10) == OS_R_TMO) {
    printf ("Wait message timeout!\n");
    }
  else {
    /* process message here */
    free (msg);
  }
   ..
}
```

os_mut_init

Summary

```
#include <rtl.h>

void os_mut_init (
    OS_ID mutex );    /* The mutex to initialize */
```

Description

The **os_mut_init** function initializes the *mutex* object identified by the function argument.

The **os_mut_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must define the mutex object of type OS_MUT. You can use the name of the mutex object to identify it during operation.

Return Value

The **os_mut_init** function does not return any value.

See Also

os_mut_release, **os_mut_wait**

Example

```
#include <rtl.h>

OS_MUT mutex1;

__task void task1 (void) {
   ..
  os_mut_init (mutex1);
   ..
}
```

os_mut_release

Summary

```
#include <rtl.h>

OS_RESULT os_mut_release (
    OS_ID mutex );    /* The mutex to release */
```

Description

The **os_mut_release** function decrements the internal counter of the *mutex* identified by the function argument in order to release the mutex. Only when the internal counter of the mutex is zero, the mutex is really free to be acquired by another task.

The mutex object knows the task that currently owns it. Hence the owning task can acquire and lock the mutex as many times as needed using the **os_mut_wait** function. When a task that owns a mutex tries to acquire it again, the task does not get blocked, but the mutex's internal counter is incremented. The task that acquired the mutex must release the mutex as many times as it was acquired, so that the internal counter of the mutex is decremented to 0.

This function also restores the original task's priority if **priority inheritance** has been applied to the owning task of the mutex and his priority has been temporary raised.

The **os_mut_release** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the mutex object using the **os_mut_init** function before you can perform any operation on it.

Return Value

The **os_mut_release** function returns the completion value:

Return Value	Description
OS_R_OK	The mutex was successfully released
OS_R_NOK	An error occurred. This can be either because the internal counter of the mutex is not 0 or because the calling task is not the owner of the mutex.

See Also

os_mut_init, **os_mut_wait**

Example

```
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (mutex1);
}

void f2 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 2 */
   ..
  os_mut_release (mutex1);
}

__task void task1 (void) {
   ..
  os_mut_init (mutex1);
  f1 ();
```

```
   ..
}

__task void task2 (void) {
   ..
  f2 ();
   ..
}
```

os_mut_wait

Summary

```
#include <rtl.h>

OS_RESULT os_mut_wait (
    OS_ID mutex,        /* The mutex to acquire */
    U16   timeout );    /* Length of time to wait */
```

Description

The **os_mut_wait** function tries to acquire the *mutex* identified by the function argument. If the mutex has not be locked by another task, the calling task acquires and locks the mutex and might continue immediately. If the mutex has been locked by another task, then the RTX kernel puts the calling task to sleep until the mutex becomes unlocked or until the *timeout* expires.

You can specify any value between 0 and 0xFFFE for the *timeout* argument. You must set *timeout* to 0xFFFF for an indefinite timeout period.

If you specify a value of 0 for the *timeout*, the calling task continues immediately even if there is a higher priority task in the ready list.

This function also raises the priority of the owning task of the mutex, if it is lower than the priority of the calling task. This programming method is called **priority inheritance** and is used to eliminate priority inversion problems. When a mutex is released, the original priority of the owning task will be restored.

The **os_mut_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Timeout is measured in number of system intervals.
- You must initialize the mutex object using the **os_mut_init** function before you can perform any operation on it.

Return Value

The **os_mut_wait** function returns the completion value:

Return Value	Description
OS_R_MUT	The task waited until the mutex was released and has now acquired and locked the mutex.
OS_R_TMO	The timeout has expired.
OS_R_OK	The mutex was available and the **os_mut_wait** function returned to the calling task immediately.

See Also

os_mut_init, **os_mut_release**

Example

```
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (mutex1);
}

void f2 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 2 */
   ..
  os_mut_release (mutex1);
}
```

```
__task void task1 (void) {
   ..
  os_mut_init (mutex1);
  f1 ();
   ..
}

__task void task2 (void) {
   ..
  f2 ();
   ..
}
```

os_sem_init

Summary

```
#include <rtl.h>

void os_sem_init (
    OS_ID semaphore,        /* The semaphore object to initialize */
    U16   token_count );    /* Initial number of tokens */
```

Description

The **os_sem_init** function initializes the *semaphore* object identified by the function argument.

The argument *token_count* determines the number of tokens stored in the semaphore initially.

The **os_sem_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must define the semaphore object of type OS_SEM. You can use the name of the semaphore object to identify it during operation.

Return Value

The **os_sem_init** function does not return any value.

See Also

isr_sem_send, **os_sem_send**, **os_sem_wait**

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
   ..
  os_sem_init (semaphore1, 0);
  os_sem_send (semaphore1);
   ..
}
```

os_sem_send

Summary

```
#include <rtl.h>

OS_RESULT os_sem_send (
    OS_ID semaphore );  /* The semaphore whose token count is incremented */
```

Description

The **os_sem_send** function increments the number of tokens in the *semaphore* object identified by the function argument.

The **os_sem_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.

Return Value

The **os_sem_send** function always returns OS_R_OK.

See Also

isr_sem_send, **os_sem_init**, **os_sem_wait**

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
  ..
  os_sem_init (semaphore1, 0);
  os_sem_send (semaphore1);
  ..
}
```

os_sem_wait

Summary

```
#include <rtl.h>

OS_RESULT os_sem_wait (
    OS_ID semaphore,    /* The semaphore to get the token from */
    U16   timeout );    /* Length of time to wait for the token */
```

Description

The **os_sem_wait** function requests a token from the *semaphore* identified by the function argument. If the token count in the semaphore is more than zero, the function gives a token to the calling task and decrements the token count in the semaphore. The calling task might then continue immediately or is put in the ready list depending on the priorities of other tasks in the ready list and the value of *timeout.*

If the token count in the semaphore is zero, the calling task is put to sleep by the RTX kernel. When a token becomes available in the semaphore or when the timeout period expires, the RTX kernel wakes the task and puts it in the ready list .

You can specify any value between 0 and 0xFFFE for the *timeout* argument. You must set *timeout* to 0xFFFF for an indefinite timeout period.

If you specify a value of 0 for the *timeout*, the calling task continues immediately even if there is a higher priority task in the ready list.

The **os_sem_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Timeout is measured in number of system intervals.
- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.

Return Value

The **os_sem_wait** function returns a completion value:

Return Value	Description
OS_R_SEM	The calling task has waited until a semaphore became available.
OS_R_TMO	The timeout expired before the token became available.
OS_R_OK	A token was available and the function returned immediately.

See Also

isr_sem_send, **os_sem_init**, **os_sem_send**

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
   ..
  os_sem_wait (semaphore1, 0xffff);
   ..
}
```

os_sys_init

Summary

```
#include <rtl.h>

void os_sys_init (
    void (*task)(void) );  /* Task to start */
```

Description

The **os_sys_init** function initializes and starts the *Real-Time eXecutive* (RTX) kernel.

The *task* argument points to the task function to start after the kernel is initialized. The RTX kernel gives the task a default priority of 1.

The **os_sys_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init** function must be called from the **main** C function.
- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.

Return Value

The **os_sys_init** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init_prio, **os_sys_init_user**

Example

```
#include <rtl.h>

void main (void) {
  os_sys_init (task1);  /* start the kernel     */
  while(1);             /* will never come here */
}
```

os_sys_init_prio

Summary

```
#include <rtl.h>

void os_sys_init_prio (
    void (*task)(void),   /* Task to start */
    U8 priority);         /* Task priority (1-254) */
```

Description

The **os_sys_init_prio** function initializes and starts the *Real-Time eXecutive* (RTX) kernel.

The *task* argument points to the task to start after the kernel is initialized.

The *priority* argument specifies the priority for the *task*. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the *priority*, it is automatically replaced with a value of 1. Priority 255 is also reserved.

The **os_sys_init_prio** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init_prio** function must be called from the **main** C function.
- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.
- Priority value of 255 represents the most important task.

Return Value

The **os_sys_init_prio** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init, **os_sys_init_user**

Example

```
#include <rtl.h>

void main (void) {
  os_sys_init_prio (task1, 10);
  while(1);
}
```

os_sys_init_user

Summary

```
#include <rtl.h>

void os_sys_init_user (
    void (*task)(void),     /* Task to start */
    U8    priority,         /* Task priority (1-254) */
    void* stack,            /* Task stack */
    U16   size);            /* Stack size */
```

Description

The **os_sys_init_user** function initializes and starts the *Real-Time eXecutive* (RTX) kernel. Use this function when you must specify a large stack for the starting task.

The *task* argument points to the task function to start after the kernel is initialized.

The *priority* argument specifies the priority for the *task.* The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the *priority*, it is automatically replaced with a value of 1. Priority 255 is also reserved.

The *stack* argument points to a memory block reserved for the stack to use for the *task*. The *size* argument specifies the size of the stack in bytes.

The **os_sys_init_user** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init_user** function must be called from the **main** C function.
- The stack must be aligned at an 8-byte boundary and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.
- Priority value of 255 represents the most important task.

Return Value

The **os_sys_init_user** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init, **os_sys_init_prio**

Example

```
#include <rtl.h>

static U64 stk1[400/8];     /* 400-byte stack */

void main (void) {
  os_sys_init_user (task1, 10, &stk1, sizeof(stk1));
  while(1);
}
```

Related Knowledgebase Articles

- ARMCC: PRINTF OUTPUTS 0.000000 FOR FLOAT VARIABLES

os_tmr_call

Summary

```
void os_tmr_call (
    U16 info );    /* Identification of an expired timer. */
```

Description

The **os_tmr_call** function is a user defined function that gets called by the RTX kernel's **task manager** task scheduler, when the user defined timer expires. After the **os_tmr_call** function returns, the task manager deletes this user timer.

The *info* argument contains the value that was specified when the timer was created using **os_tmr_create**.

The **os_tmr_call** function is part of RL-RTX. The prototype is defined in rtl.h. You can customize the function in rtx_config.c.

Note

- You can call any of the **isr_** system functions from the **os_tmr_call** function, but you cannot call any of the **os_** system functions.
- Do not call **os_tmr_kill** for an expired user timer.

Return Value

The **os_tmr_call** function does not return any value.

See Also

os_tmr_create, **os_tmr_kill**

Example

```
void os_tmr_call (U16 info) {
  switch (info) {
    case 1:          /* The supervised task is locked, */
                     /* recovery actions required.     */
      break;
    case 2:          /* The second task is locked.     */

      break;
       ..
  }
}
```

os_tmr_create

Summary

```
#include <rtl.h>

OS_ID os_tmr_create (
    U16 tcnt,      /* Length of the timer. */
    U16 info );    /* Argument to the callback function. */
```

Description

The **os_tmr_create** function sets up and starts a timer. When the timer expires, the RTX kernel calls the user defined **os_tmr_call** callback function and passes *info* as an argument to the **os_tmr_call** function.

The *tcnt* argument specifies the length of timer, in number of system ticks. You can specify tcnt to any value between 1 and 0xFFFF.

The **os_tmr_create** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_tmr_create** function returns a timer ID if the timer was successfully created. Otherwise, it returns NULL.

See Also

os_tmr_call, **os_tmr_kill**

Example

```
#include <rtl.h>

OS_TID tsk1;
OS_ID  tmr1;

__task void task1 (void) {
   ..
  tmr1 = os_tmr_create (10, 1);
  if (tmr1 == NULL) {
    printf ("Failed to create user timer.\n");
  }
   ..
}
```

os_tmr_kill

Summary

```
#include <rtl.h>

OS_ID os_tmr_kill (
    OS_ID timer );    /* ID of the timer to kill */
```

Description

The **os_tmr_kill** function deletes the *timer* identified by the function argument. *timer* is a user timer that was created using the **os_tmr_create** function. If you delete the timer before it expires, the **os_tmr_call** callback function does not get called.

The **os_tmr_kill** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Do not call **os_tmr_kill** for an expired user timer. It has already been deleted by the system.

Return Value

The **os_tmr_kill** function returns NULL if the timer is killed successfully. Otherwise, it returns the *timer* value.

See Also

os_tmr_call, **os_tmr_create**

Example

```
#include <rtl.h>

OS_TID tsk1;
OS_ID  tmr1;

__task void task1 (void) {
   ..
  if (os_tmr_kill (tmr1) != NULL) {
    printf ("\nThis timer is not on the list.");
  }
  else {
    printf ("\nTimer killed.");
  }
   ..
}
```

os_tsk_create

Summary

```
#include <rtl.h>

OS_TID os_tsk_create (
    void (*task)(void),    /* Task to create */
    U8    priority );      /* Task priority (1-254) */
```

Description

The **os_tsk_create** function creates the task identified by the *task* function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The **os_tsk_create** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.
- Priority value of 255 represents the most important task.

Return Value

The **os_tsk_create** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create_ex, **os_tsk_create_user**, **os_tsk_create_user_ex**

Example

```
#include <rtl.h>

OS_TID tsk1, tsk2;

__task void task1 (void) {
   ..
  tsk2 = os_tsk_create (task2, 1);
   ..
}

__task void task2 (void) {
   ..
}
```

os_tsk_create_ex

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_ex (
    void (*task)(void *),    /* Task to create */
    U8    priority,          /* Task priority (1-254) */
    void* argv );            /* Argument to the task */
```

Description

The **os_tsk_create_ex** function creates the task identified by the *task* function pointer argument and adds the task to the ready queue. The function dynamically assigns a task identifier value (TID) to the new task. The **os_tsk_create_ex** function is an extension to the **os_tsk_create** function that enables you to pass an argument to the task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *argv* argument is passed directly to the task when it starts. An argument to a task can be useful to differentiate between multiple instances of the same task.

The **os_tsk_create_ex** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.

Return Value

The **os_tsk_create_ex** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, **os_tsk_create_user**, **os_tsk_create_user_ex**

Example

```
#include <rtl.h>

OS_TID tsk1, tsk2_0, tsk2_1;
int param[2] = {0, 1};

__task void task1 (void) {
   ..
  tsk2_0 = os_tsk_create_ex (task2, 1, &param[0]);
  tsk2_1 = os_tsk_create_ex (task2, 1, &param[1]);
   ..
}

__task void task2 (void *argv) {
   ..
  switch (*(int *)argv) {
    case 0:
      printf("This is a first instance of task2.\n");
      break;
    case 1:
      printf("This is a second instance of task2.\n");
      break;
  }
   ..
}
```

os_tsk_create_user

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_user(
    void (*task)(void),    /* Task to create */
    U8    priority,        /* Task priority (1-254) */
    void* stk,             /* Pointer to the task's stack */
    U16   size );          /* Number of bytes in the stack */
```

Description

The **os_tsk_create_user** function creates the task identified by the task function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task. This function enables you to provide a separate stack for the task. This is useful when a task needs a bigger stack for its local variables.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *stk* argument is a pointer to the memory block reserved for the stack of this task. The *size* argument specifies the number of bytes in the stack.

The **os_tsk_create_user** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The stack must be aligned at an 8-byte boundary, and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.

Return Value

The **os_tsk_create_user** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, **os_tsk_create_ex**, **os_tsk_create_user_ex**

Example

```
#include <rtl.h>

OS_TID tsk1,tsk2;
static U64 stk2[400/8];

__task void task1 (void) {
   ..
  /* Create task 2 with a bigger stack */
  tsk2 = os_tsk_create_user (task2, 1, &stk2, sizeof(stk2));
   ..
}

__task void task2 (void) {
  /* We need a bigger stack here. */
  U8 buf[200];
   ..
}
```

Related Knowledgebase Articles

- ARMCC: PRINTF OUTPUTS 0.000000 FOR FLOAT VARIABLES

os_tsk_create_user_ex

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_user_ex (
    void (*task)(void *),    /* Task to create */
    U8    priority,          /* Task priority (1-254) */
    void* stk,               /* Pointer to the task's stack */
    U16   size,              /* Size of stack in bytes */
    void* argv );            /* Argument to the task */
```

Description

The **os_tsk_create_user_ex** function creates the task identified by the *task* function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task. This function enables you to provide a separate stack for the task. This is useful when a task needs a bigger stack for its local variables. The **os_tsk_create_user_ex** function is an extension to the **os_tsk_create_user** function that enables you to pass an argument to the task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *stk* argument is a pointer to the memory block reserved for the stack of this task. The *size* argument specifies the number of bytes in the stack.

The *argv* argument is passed directly to the task when it starts. An argument to a task can be useful to differentiate between multiple instances of the same task. Multiple instances of the same task can behave differently based on the argument.

The **os_tsk_create_user_ex** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The stack *stk* must be aligned at an 8-byte boundary and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.

Return Value

The **os_tsk_create_user_ex** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, **os_tsk_create_ex**, **os_tsk_create_user**

Example

```
#include <rtl.h>

OS_TID tsk1,tsk2_0,tsk2_1;
static U64 stk2[2][400/8];

__task void task1 (void) {
  ..
  /* Create task 2 with a bigger stack */
  tsk2_0 = os_tsk_create_user_ex (task2, 1,
                                  &stk2[0], sizeof(stk2[0]),
                                  (void *)0);
  tsk2_1 = os_tsk_create_user_ex (task2, 1,
                                  &stk2[1], sizeof(stk2[1]),
                                  (void *)1);
  ..
}

__task void task2 (void *argv) {
  /* We need a bigger stack here. */
  U8 buf[200];
```

```
  ..
  switch ((int)argv) {
    case 0:
      printf("This is a first instance of task2.\n");
      break;
    case 1:
      printf("This is a second instance of task2.\n");
      break;
  }
   ..
}
```

Related Knowledgebase Articles

- ARMCC: PRINTF OUTPUTS 0.000000 FOR FLOAT VARIABLES

os_tsk_delete

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_delete (
    OS_TID task_id );    /* Id of the task to delete */
```

Description

If a task has finished all its work or is not needed anymore, you can terminate it using the **os_tsk_delete** function. The **os_tsk_delete** function stops and deletes the task identified by *task_id*.

The **os_tsk_delete** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- If *task_id* has a value of 0, the task that is currently running is stopped and deleted. The program execution continues with the task with the next highest priority in the ready queue.
- Deleting a task frees all dynamic memory resources allocated to that task.

Return Value

The **os_tsk_delete** function returns OS_R_OK if the task was successfully stopped and deleted. In all other cases, for example if the task with *task_id* does not exist or is not running, the function returns OS_R_NOK.

See Also

os_tsk_delete_self

Example

```
#include <rtl.h>
OS_TID tsk3;

__task void task2 (void) {
  tsk3 = os_tsk_create (task3, 0);
   ..
  if (os_tsk_delete (tsk3) == OS_R_OK) {
    printf("\n'task 3' deleted.");
  }
  else {
    printf ("\nFailed to delete 'task 3'.");
  }
}

__task void task3 (void) {
   ..
}
```

os_tsk_delete_self

Summary

```
#include <rtl.h>

void os_tsk_delete_self (void);
```

Description

The **os_tsk_delete_self** function stops and deletes the currently running task. The program execution continues with the task with the next highest priority in the ready queue.

The **os_tsk_delete_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Deleting a task frees all dynamic memory resources allocated to that task.

Return Value

The **os_tsk_delete_self** function does not return. The program execution continues with the task with the next highest priority in the ready queue.

See Also

os_tsk_delete

Example

```
#include <rtl.h>

__task void task2 (void) {
   ..
  os_tsk_delete_self();
}
```

os_tsk_pass

Summary

```
#include <rtl.h>

void os_tsk_pass (void);
```

Description

The **os_tsk_pass** function passes control to the next task of the same priority in the ready queue. If there is no task of the same priority in the ready queue, the current task continues and no task switching occurs.

The **os_tsk_pass** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can use this function to implement a task switching system between several tasks of the same priority.

Return Value

The **os_tsk_pass** function does not return any value.

See Also

os_tsk_prio, **os_tsk_prio_self**

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
   ..
  os_tsk_pass();
   ..
}
```

os_tsk_prio

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_prio (
    OS_TID task_id,       /* ID of the task */
    U8     new_prio );    /* New priority of the task (1-254) */
```

Description

The **os_tsk_prio** function changes the execution priority of the task identified by the argument *task_id*.

If the value of *new_prio* is higher than the priority of the currently executing task, a task switch occurs to enable the task identified by *task_id* to run. If the value of *new_prio* is lower than the priority of the currently executing task, then the currently executing task resumes its execution.

If the value of *task_id* is 0, the priority of the currently running task is changed to *new_prio*.

The **os_tsk_prio** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The value of *new_prio* can be anything from 1 to 254.
- The new priority stays in effect until you change it.
- Priority 0 is reserved for the idle task. If priority 0 is specified to the function, it is automatically replaced with the value of 1 by the RTX kernel. Priority 255 is also reserved.
- A higher value for *new_prio* indicates a higher priority.

Return Value

The **os_tsk_prio** function returns one of these values:

Return Value	Description
OS_R_OK	The priority of a task has been successfully changed.
OS_R_NOK	The task with *task_id* does not exist or has not been started.

See Also

os_tsk_pass, **os_tsk_prio_self**

Example

```
#include <RTL.h>

OS_TID tsk1,tsk2;

__task void task1 (void) {
   ..
  os_tsk_prio_self (5);

  /* Changing the priority of task2 will cause a task switch. */
  os_tsk_prio(tsk2, 10);
   ..
}

__task void task2 (void) {
   ..
  /* Change priority of this task will cause task switch. */
  os_tsk_prio_self (1);
   ..
}
```

os_tsk_prio_self

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_prio_self (
    U8 new_prio );    /* New priority of task (1-254) */
```

Description

The **os_tsk_prio_self** macro changes the priority of the currently running task tonew_prio.

The **os_tsk_prio_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The value of *new_prio* can be anything from 1 to 254.
- The new priority stays in effect until you change it.
- Priority 0 is reserved for the idle task. If priority 0 is specified to the function, it is automatically replaced with the value of 1 by the RTX kernel. Priority 255 is also reserved.
- A higher value for *new_prio* indicates a higher priority.

Return Value

The **os_tsk_prio_self** function always returns OS_R_OK.

See Also

os_tsk_pass, **os_tsk_prio**

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
   ..
  os_tsk_prio_self(10);  /* Increase its priority, for the critical section */
   ..                    /* This is a critical section */
   ..
  os_tsk_prio_self(2);   /* Decrease its priority at end of critical section
*/
   ..
}
```

os_tsk_self

Summary

```
#include <rtl.h>

OS_TID os_tsk_self (void);
```

Description

The **os_tsk_self** function identifies the currently running task by returning its task ID.

The **os_tsk_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_tsk_self** function returns the task identifier number (TID) of the currently running task.

See Also

isr_tsk_get, **os_tsk_create**, **os_tsk_create_user**

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
  tsk1 = os_tsk_self();
   ..
}
```

tsk_lock

Summary

```
#include <rtl.h>

void tsk_lock (void);
```

Description

The **tsk_lock** function disables the RTX kernel timer interrupts and thereby disables task switching.

The **task_lock** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Nested calls of **tsk_lock** function are not supported.
- Calling **tsk_lock** function from an interrupt handler is not allowed.
- For the duration when the timer interrupts are disabled, the RTX kernel task scheduler is blocked, timeouts do not work, and the Round Robin task scheduling is also blocked. Hence, it is highly recommended that disabling of the RTX kernel timer interrupts is kept to a very short time period.

Return Value

The **tsk_lock** function does not return any value.

See Also

tsk_unlock

Example

```
#include <rtl.h>

void free_mem (void *ptr) {
  /* 'free()' is not reentrant. */
  tsk_lock ();
  free (ptr);
  tsk_unlock ();
}
```

tsk_unlock

Summary

```
#include <rtl.h>

void tsk_unlock (void);
```

Description

The **tsk_unlock** function enables the RTX kernel timer interrupts and thereby enables task switching if they had been disabled by the **tsk_lock** function.

The **tsk_unlock** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Calling **tsk_lock** function from an interrupt handler is not allowed.

Return Value

The **tsk_unlock** function does not return any value.

See Also

tsk_lock

Example

```
#include <rtl.h>

void free_mem (void *ptr) {
  /* 'free()' is not reentrant. */
  tsk_lock ();
  free (ptr);
  tsk_unlock ();
}
```