User Guide

# General MCU RT_Thread User Guide

## Introduction

This document mainly introduces the use of RT_Thread system in Nsing general MCU, which is applicable to N32G45x, N32G4FR, N32WB452, N32G43x, N32L40x, N32L43x series chips. This document uses N32G45x as an example to introduce the related usage instructions for RT_Thread system.

# CONTENTS

# 1 RT_Thread

## 1.1 Overview

RT-Thread, short for Real Time-Thread, as the name suggests, is an embedded real-time multi-threaded operating system. One of its basic attributes is supporting multi-tasking. Allowing multiple tasks to run concurrently does not mean that the processor is actually executing multiple tasks at the same time. In fact, a processor core can only run one task at a time. However, because the short execution time of each task and the rapid switching between tasks through the task scheduler (the scheduler decides the task to be executed at any given moment according to the priority), it creates the illusion that multiple tasks are running at the same time. In the RT-Thread system, tasks are implemented by threads, and the thread scheduler in RT-Thread is the above-mentioned task scheduler.

RT-Thread is mainly written in C language, which is easy to understand and convenient for porting. It applies the object-oriented design method to the real-time system design, resulting in elegant code style, clear architecture, modular system, and excellent scalability. For resource-constrained MCU systems, the NANO version that only requires 3KB Flash and 1.2KB RAM memory resources can be tailored through easy-to-use tools (NANO is a minimalist version of the kernel officially released by RT-Thread in July 2017); For resource-rich IoT devices, RT-Thread can use online software package management tools, and cooperate with system configuration tools to achieve intuitive and fast modular tailoring. It seamlessly imports rich software function packages to realize complex functions such as Android-like graphical interface, touch and slide effects, and intelligent voice interaction effects.

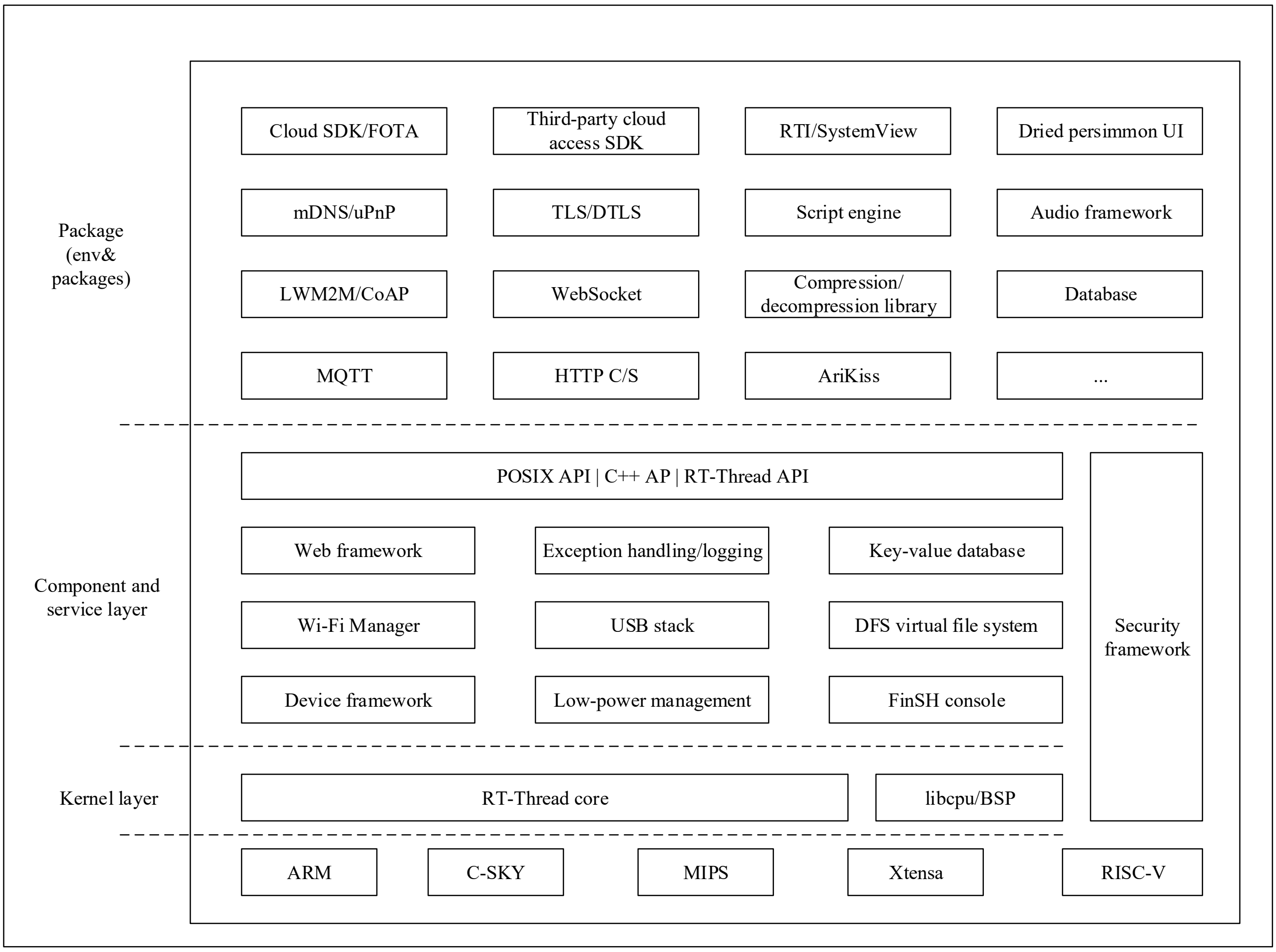
Compared with the Linux operating system, RT-Thread has the advantages of small size, low cost, low power consumption, and fast startup. In addition, RT-Thread also has the characteristics of high real-time performance and small resource consumption, which is very suitable for various resource constrainted scenarios ( such as cost, power consumption constraints, etc.). Although a 32-bit MCU is its main operating platform, in fact, many application processors with MMU, based on ARM9, ARM11 and even Cortex-A series-level CPU are also suitable for RT-Thread in specific applications.

## 1.2 RT-Thread Architecture

In recent years, the concept of Internet of Things (IoT) has been widely popularized, the IoT market has developed rapidly, and the networking of embedded devices has become the general trend. The terminal networking has greatly increased the software complexity, and the traditional RTOS kernel has become more and more difficult to meet the needs of the market. In this context, the concept of the Internet of Things Operating System (IoT OS) has emerged. IoT OS refers to a software platform based on an operating system kernel (which can be RTOS, Linux, etc.), including relatively complete middleware components such as file systems and graphics libraries. It has low power consumption and security features, supports for communication protocols and cloud connectivity. RT-Thread is an IoT OS.

One of the main differences between RT-Thread and many other RTOS such as FreeRTOS and uC/OS is that RT_Thread is not only a real-time kernel, but also has rich middle-layer components, as shown in Figure 1-1.

**Figure 1-1 RT_Thread Software Framework Diagram**



It specifically includes the following parts:

- Kernel layer: RT-Thread kernel is the core part of RT-Thread, including the realization of objects in the kernel system, such as multithreading and its scheduling, semaphores, mailboxes, message queues, memory management, timers, etc.; libcpu/BSP (chip porting related files/board support package) is closely related to hardware and consists of peripheral drivers and CPU porting.
- Component and service layer: components are upper-layer software based on RT-Thread kernel, such as virtual file system, FinSH command line interface, network framework, device framework, etc. Modular design is adopted to achieve high cohesion within components and low coupling between components.
- RT-Thread software package: it runs on the RT-Thread IoT operating system platform, and it is a general software component for different application fields, consisting of description information, source code or library files. RT-Thread provides an open software package platform, where official or developer provided software packages are stored. This platform provides developers with many choices of reusable software packages, which is also an important part of the RT-Thread ecosystem. The ecosystem of software packages is crucial to the choice of an operating system, because these software packages are highly reusable and highly modular, which

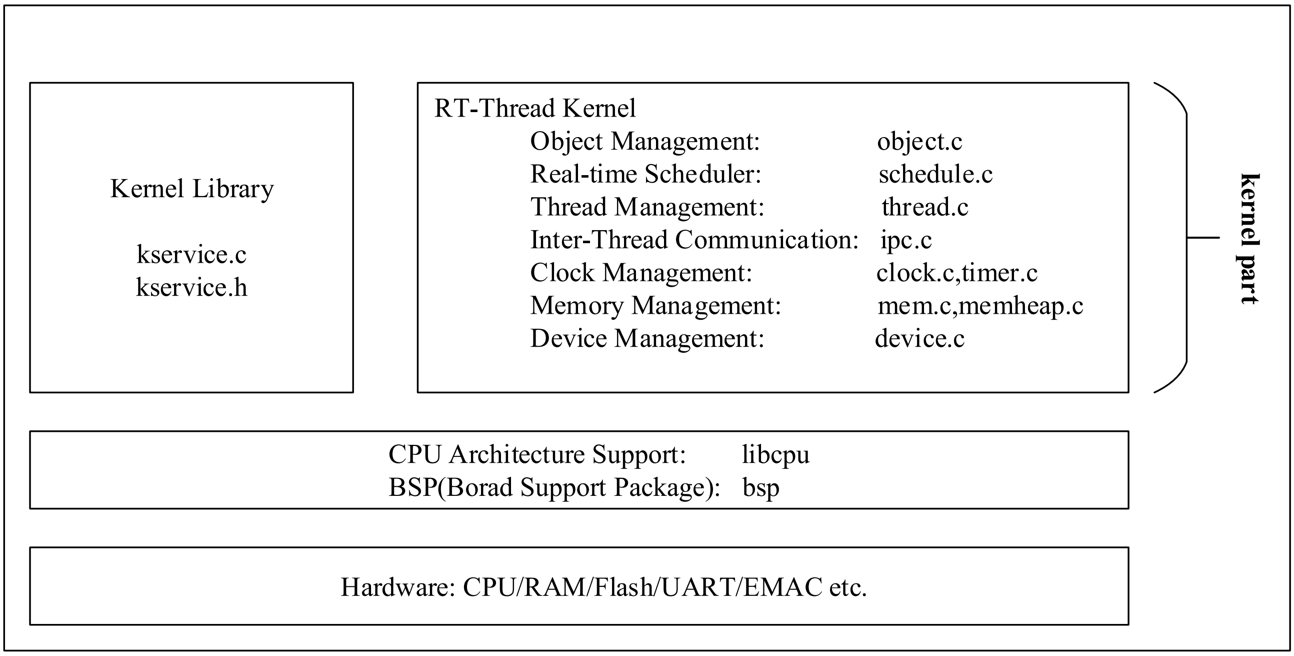
greatly facilitates application developers to create the system they want in the shortest time. RT-Thread has supported over 60 software packages, for example:

- IoT-related software packages: Paho MQTT, WebClient, mongoose, WebTerminal, etc.
- Scripting language related software packages: JerryScript, MicroPython
- Multimedia related software packages: Openmv, mupdf
- Tool software packages: CmBacktrace, EasyFlash, EasyLogger, SystemView
- System-related software packages: RTGUI, Persimmon UI, lwext4, partition, SQLite, etc.
- Peripheral library and driver software package: RealTek RTL8710BN SDK

## 1.3 RT_Thread Kernel

The kernel is the most basic and most important part of the operating system. Figure 1-2 is the RT-Thread kernel architecture diagram. The kernel is above the hardware layer, and the kernel part includes the kernel library and real-time kernel implementation.

**Figure 1-2 RT_Thread Kernel and Underlying Structure**



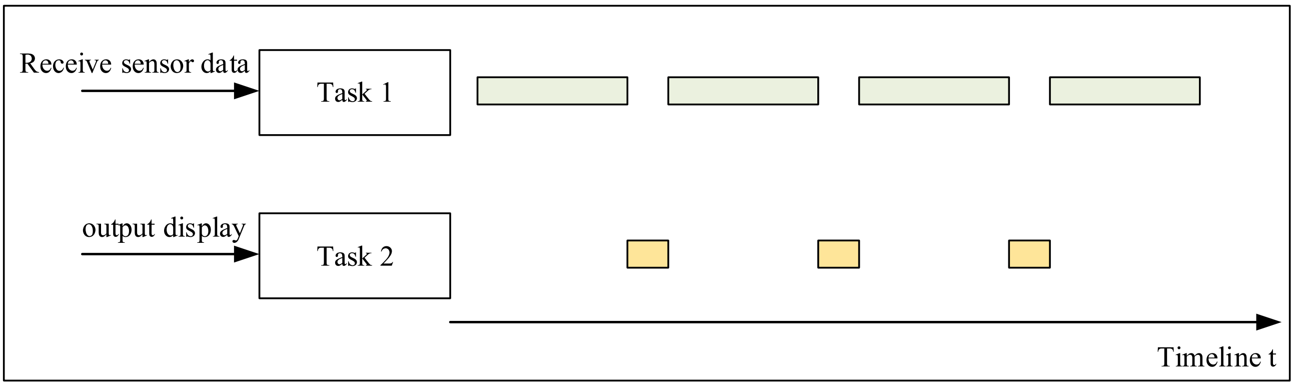
The kernel library is a small subset of functions similar C library implemented to ensure that the kernel can run independently. It provides implementations of functions like "strcpy", "memcpy", "printf", "scanf", etc. The RT-Thread kernel library only provides a small part of the C library of functions used by the kernel. In order to avoid the same name as the standard C library, the rt_ prefix will be added before these functions.

The implementation of real-time kernel includes: object management, thread management and scheduler, inter-thread communication management, clock management and memory management, etc. The minimum resource occupancy of the kernel is 3KB ROM and 1.2KB RAM.

## 1.4 RT_Thread Thread Management

In our daily life, when we want to complete a big task, we usually decompose it into many simple and easy-to-solve small tasks. The small tasks are completed one by one, and the big task is completed accordingly. In a multithreaded operating system, developers are also required to decompose a complex application into multiple small, schedulable, serialized program units. When the applicationis properly divided and executed correctly, this design enables the system to meet the performance and time requirements of a real-time system. For example, an embedded system perform a task that the system collects data through sensors and displays the data through the display screen. In a multi-threaded real-time system, this task can be decomposed into two sub-tasks, as shown in Figure 1-3, a subtask is continuously reading sensor data and writing the data to shared memory, another subtask is periodically reading data from shared memory and output sensor data to the display.

**Figure 1-3 Switching Execution of Sensor Data Receiving Task and Display Task**



In RT-Thread, the program entities corresponding to the sub-tasks described above are the threads. The thread is the carrier for completing the task, it is the most basic scheduling unit in RT-Thread. It describes the running environment of a task execution and the priority level of this task. Important tasks can be set to a relatively high priority, non-important tasks can be set to a lower priority, and different tasks can also be set to the same priority and run in turn.

When a thread is running, it perceives itself as running in a CPU-exclusive manner. The running environment of the thread is called context, which specifically includes all variables and data, such as register variables, stack, memory information, and so on.

## 1.5 RT_Thread Clock Management

The clock management of RT-Thread is based on the clock tick. The clock tick refers to the length of the interval between two interrupts of the periodic hardware timer. This periodic hardware timer is called the system clock. The clock tick (OS Tick) is the smallest clock unit in the RT-Thread operating system. The system tick is generally defined as a 32-bit unsigned integer, which is provided to the application for all time-related services, such as thread delay, thread time slice rotation scheduling and timer timeout, etc.. The number of clock ticks counted from the start of the system is called the system time. The clock tick is derived from the periodic interrupt of the timer, and an interrupt represents an OS Tick. The length of the OS Tick can be adjusted according to the definition of

RT_TICK_PER_SECOND, which is equal to 1/RT_TICK_PER_SECOND seconds. A clock with higher precision will cause the timer to be checked frequently in the system.

## 1.6 RT_Thread Interrupt Management

The interrupt management function of RT-Thread is mainly to manage interrupt devices, interrupt service routines, interrupt nesting, maintenance of interrupt stack, protection and recovery of contexts during thread switching, etc.

When the CPU is processing internal data, an emergency occurs externally, requiring the CPU to suspend the current work and turn to process this asynchronous event. After processing, the CPU returns to the original interrupted address and continues the original work. This process is called interrupt. The system that realizes this function is called the interrupt system, and the request source that applies for the CPU interrupt is called the interrupt source. When multiple interrupt sources request interrupts from the CPU at the same time, there is a problem of interrupt priority. Usually, according to the priority level of the interrupt source, the interrupt request source with the most urgent event will be processed first, meaning that the interrupt request with the highest level will be responded first.

When an interrupt occurs, the CPU will execute in the following order:

1) Saves the current processor state information
2) Loads exception or interrupt handler function into PC register
3) Transfers control to the handler function and start execution
4) Restores processor state information when handler function execution completes
5) Returns to the previous program execution point from an exception or interrupt

Interrupts allow the CPU to process events only when they occur, rather than requiring the CPU to continually query whether a corresponding event has occurred.

## 1.7 RT_Thread Memory Management

Static memory pool interface: memory pool is a memory allocation method used to allocate a large number of small memory blocks of the same size. It can greatly speed up the speed of memory allocation and release, and can try to avoid memory fragmentation. When the memory pool is empty, the allocated thread can be blocked (either return immediately, or wait for a period of time to return, which is determined by the timeout parameter). When other threads release memory blocks to this memory pool, the blocked thread will be woken up.

Dynamic memory heap interface: dynamic memory management is a real heap memory management module, which can allocate memory blocks of any size according to the needs of users when the current resources are enough. When the user no longer needs to use these memory blocks, they can be released back to the heap for use by other applications. In order to meet different needs, RT-Thread system provides two different sets of dynamic memory management algorithms, namely small heap memory management algorithm and SLAB memory management algorithm.

- The small heap memory management module is mainly used for systems with less system resources and is generally used for systems with less than 2MB memory space.
- The SLAB memory management module is primarily used in systems with relatively abundant resources, offering a fast algorithm that approximates a multi-memory pool management algorithm..

The two memory management modules can only choose one of them or not use the dynamic heap memory manager at all when the system is running. The API interfaces provided by these two management modules are exactly the same.

In addition to the above, RT-Thread also has a management mechanism for multiple memory heaps, namely memheap memory management. The memheap method is suitable for the situation where there are multiple memory heaps in the system. It can "paste" multiple memories together to form a large memory heap, which is very convenient for users to use.

# 2 RT_Thread Application

## 2.1 Thread Creation Example

Real-time applications using RTOS can construct a set of independent threads. Each thread executes in its own context without accidentally relying on other threads in the system or the RTOS scheduler. At any time, only one thread in the application can be executed, and the RTOS scheduler is responsible for determining which thread is executed.

Below is an example on thread creation.

led0_thread: this thread toggles LED0 every 500 ms

**Create Thread:**

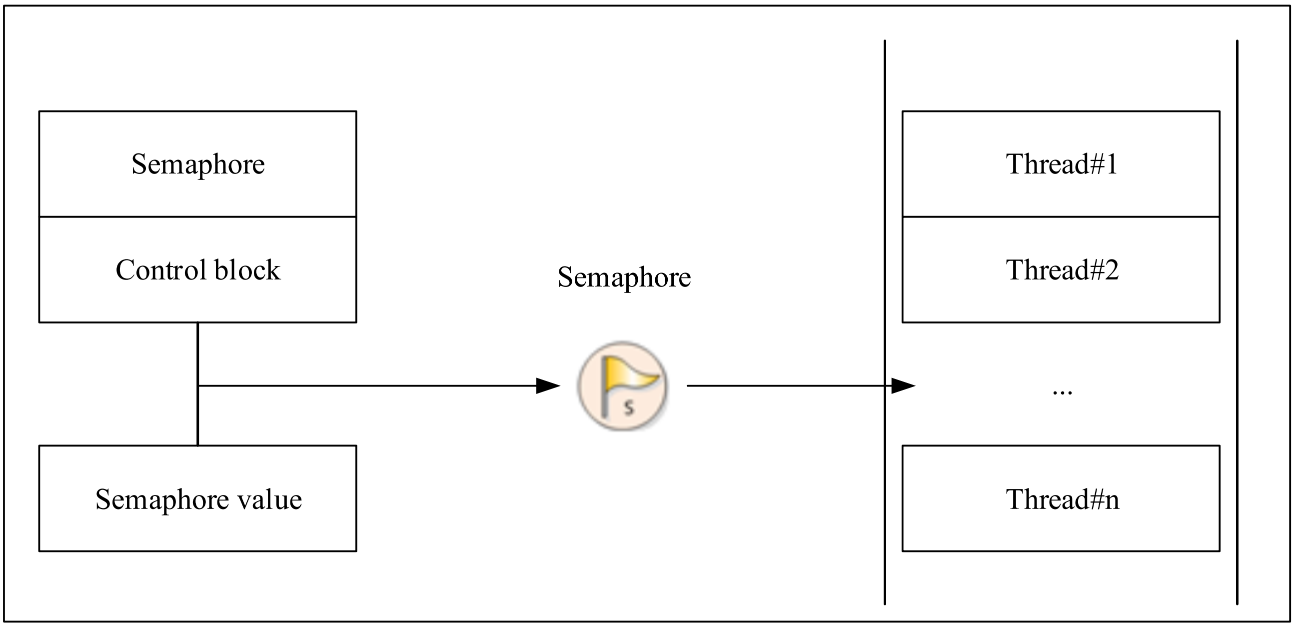
```
/* led0_thread definition */
rt_thread_init(&led0_thread,
                  "led0",
                  led0_thread_entry,
                  RT_NULL,
                  (rt_uint8_t*)&led0_stack[0],
                  sizeof(led0_stack),
                  3,
                  5);

/* Start led0_thread*/
rt_thread_startup(&led0_thread);
```

## 2.2 Semaphore Example

The semaphore is a lightweight kernel object used to complete the synchronization between threads. The thread can acquire or release it to achieve synchronization or mutual exclusion.

The schematic diagram of semaphore operation is shown in Figure 2-1. Each semaphore object has a semaphore value and a thread waiting queue, the semaphore value corresponds to the number of instances and resources of the semaphore object. If the semaphore value is 5, it means that there are 5 semaphore instances (resources) that can be used. When the number of semaphore instances is zero, the thread that applies for the semaphore will be suspended on the waiting queue of the semaphore, waiting for available semaphore instances (resources).

**Figure 2-1 Schematic Diagram of Semaphore Operation**



**Semaphore:**

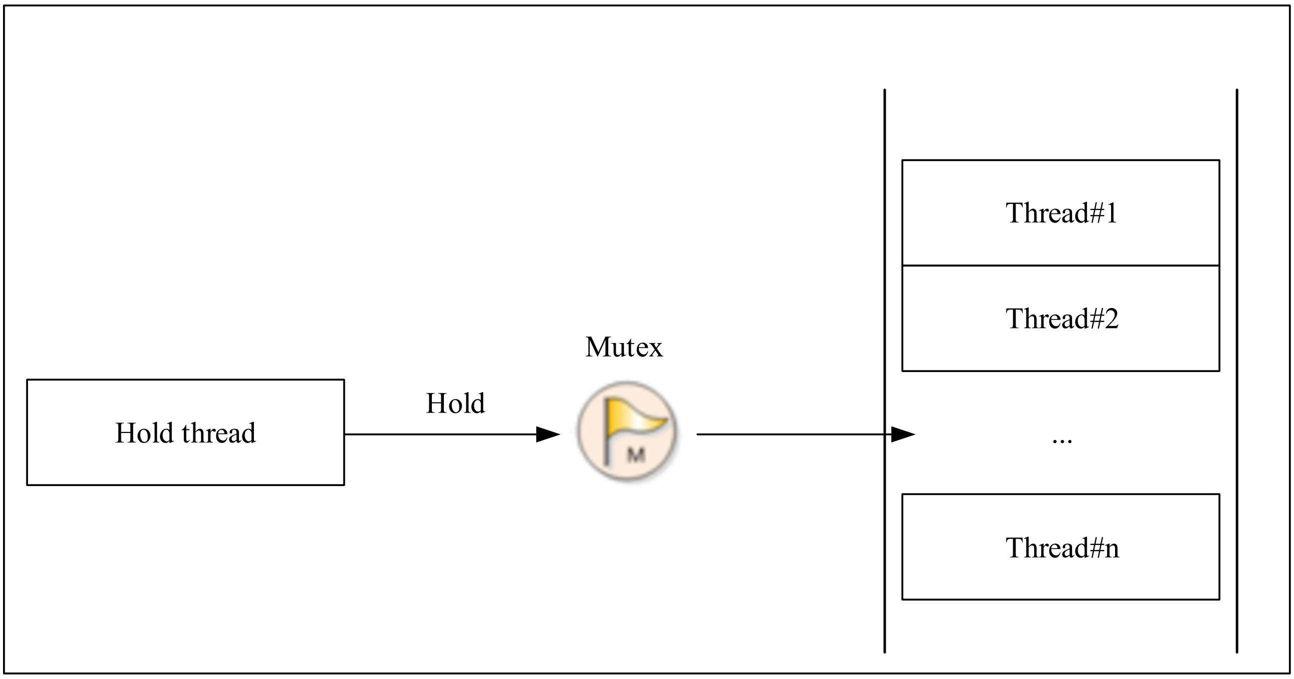
```
/* Create the binary semaphore */
rt_sem_init(&key_sem,
            "keysem",
            0,
            RT_IPC_FLAG_FIFO);

/* Get the semaphore*/
rt_sem_take(&key_sem,
            RT_WAITING_FOREVER);

/* Relase the semaphore*/
rt_sem_release(&key_sem);
```

## 2.3 Mutex Example

The difference between the mutex and the semaphore is that the mutex supports recursive access and can prevent thread priority reversion. The mutex can only be released by the holding thread, while the semaphore can be released by any thread.

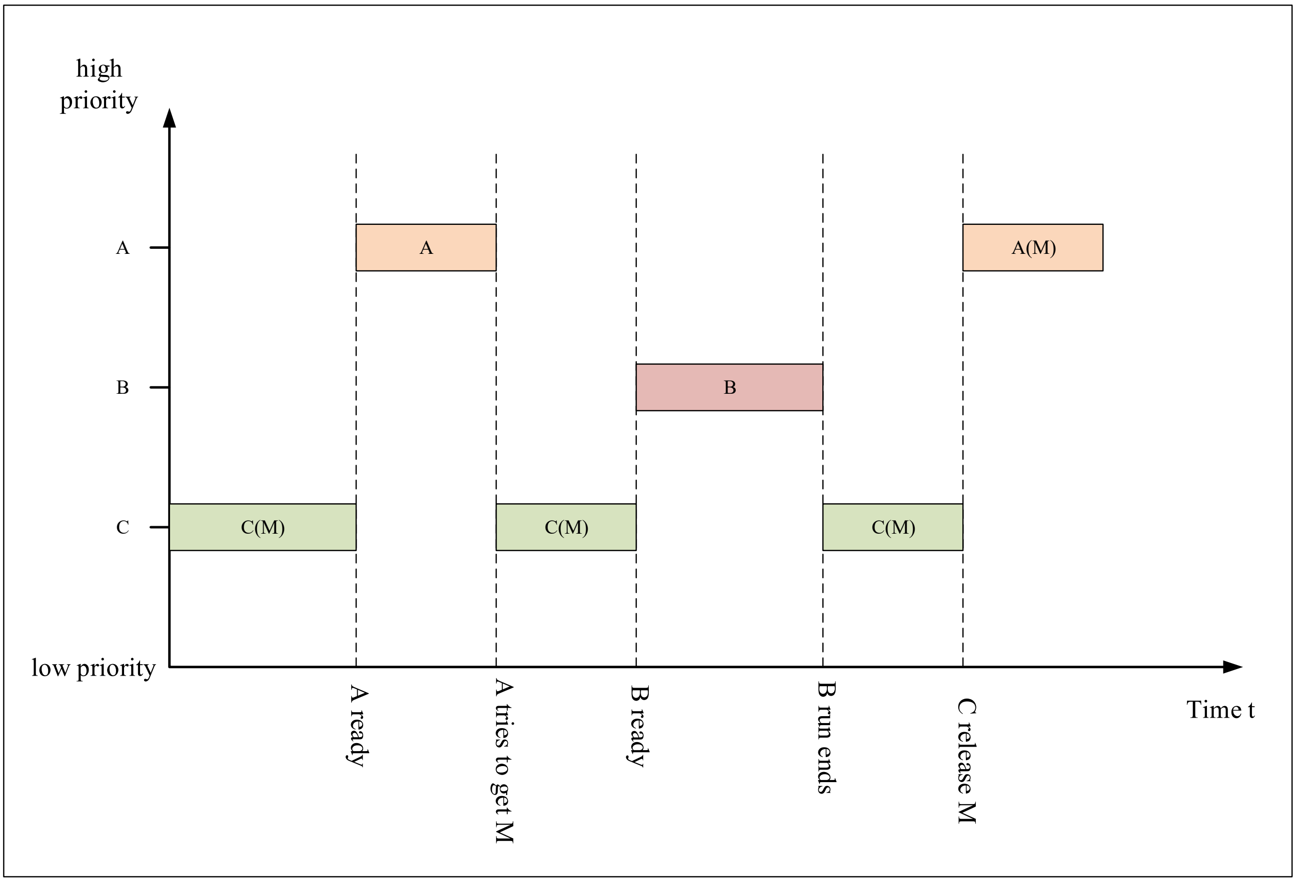
There are only two states of a mutex, unlocked or locked (two state values). When a thread holds it, the mutex is locked, and the thread takes ownership of it. Instead, when the thread releases it, the mutex is unlocked, losing ownership of it. When a thread holds a mutex, other threads will not be able to unlock it or hold it, and the thread holding the mutex can also acquire the lock again without being suspended, as shown in Figure 2-2. This feature is very different from the general binary semaphore: in the semaphore, because there is no instance, the thread recursively holds will actively suspend (eventually form a deadlock).

**Figure 2-2 Mutex Operating Diagram**



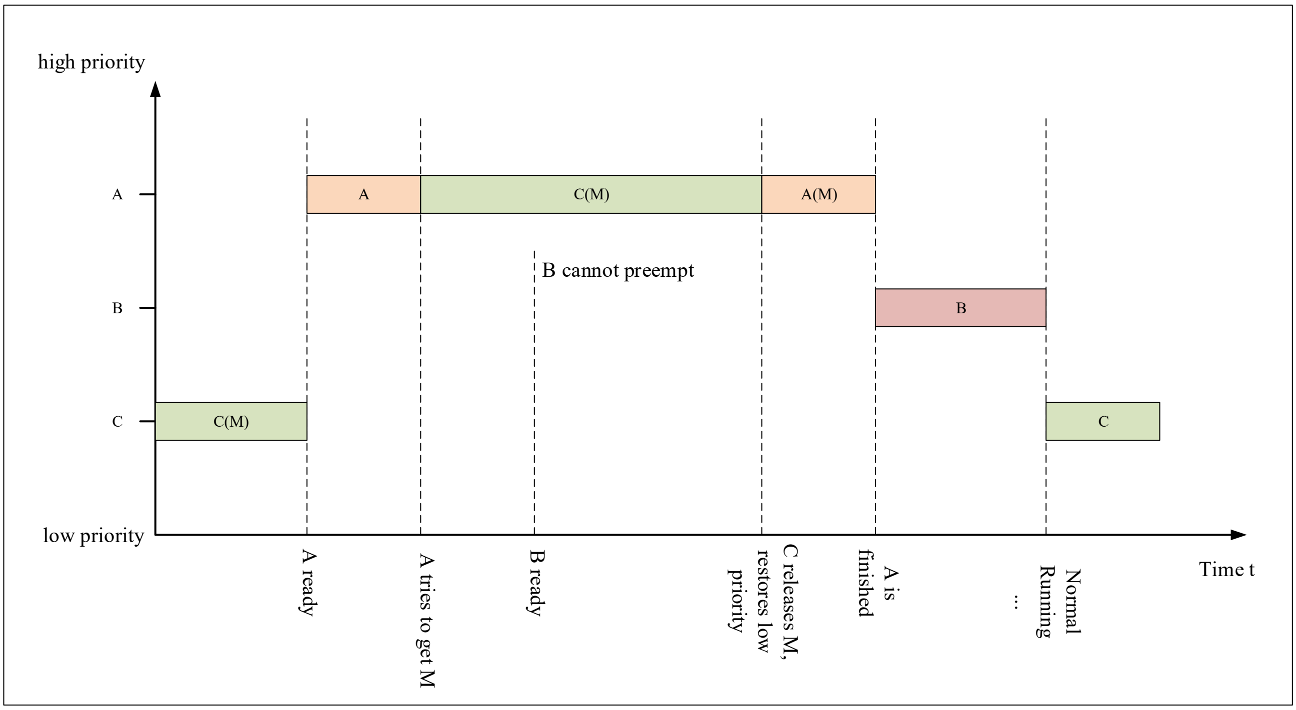
Another potential problem caused by using semaphore is thread priority inversion. The priority inversion occurs when a high-priority thread tries to access a shared resource through the semaphore mechanism, if the semaphore is already held by a low-priority thread, and this low-priority thread may be used by other medium-priority threads during the running process, causing high-priority threads to be blocked by many lower-priority threads, making it difficult to guarantee real-time performance. As shown in Figure 2-3, there are three threads with priority A, B and C, priority A > B > C. Threads A and B are in a suspended state, waiting for an event to be triggered, and thread C is running. At this time, thread C starts to use a shared resource M. During operation, the event that thread A is waiting for arrives, and thread A turns to the ready state, because it has a higher priority than thread C, so it is executed immediately. But when thread A wants to use shared resource M, because it is being used by thread C, thread A is suspended and switched to thread C to run. If the event that thread B is waiting for arrives at this time, thread B turns to the ready state. Since thread B has a higher priority than thread C, thread B starts running, and thread C does not start running until thread B finishes running. Thread A can execute only after thread C releases shared resource M. In this case, the priority inversion: thread B runs before thread A. This does not guarantee the response time of high-priority threads.

**Figure 2-3 Priority Inversion (M Is A Semaphore)**



In the RT-Thread operating system, the mutex can solve the priority inversion problem and implement the priority inheritance algorithm. Priority inheritance solves the priority inversion by raising the priority of thread C to the priority level of thread A when thread A is suspended. It prevents C (and indirectly A) from being preempted by B, as shown in Figure 2-4. Priority inheritance refers to raising the priority of a low-priority thread that occupies a resource to make it equal to the priority of the thread with the highest priority among all threads waiting for the resource, then execute, and when the low-priority thread releases the resource, the priority returns to the initial setting. Thus, threads with inherited priorities avoid preemption of system resources by any intermediate-priority thread.

**Figure 2-4 Priority Inheritance (M Is A Mutex)**



*Note: After obtaining the mutex, please release the mutex as soon as possible, and in the process of holding the mutex, you must not change the priority of the thread holding the mutex.*

**Mutex**

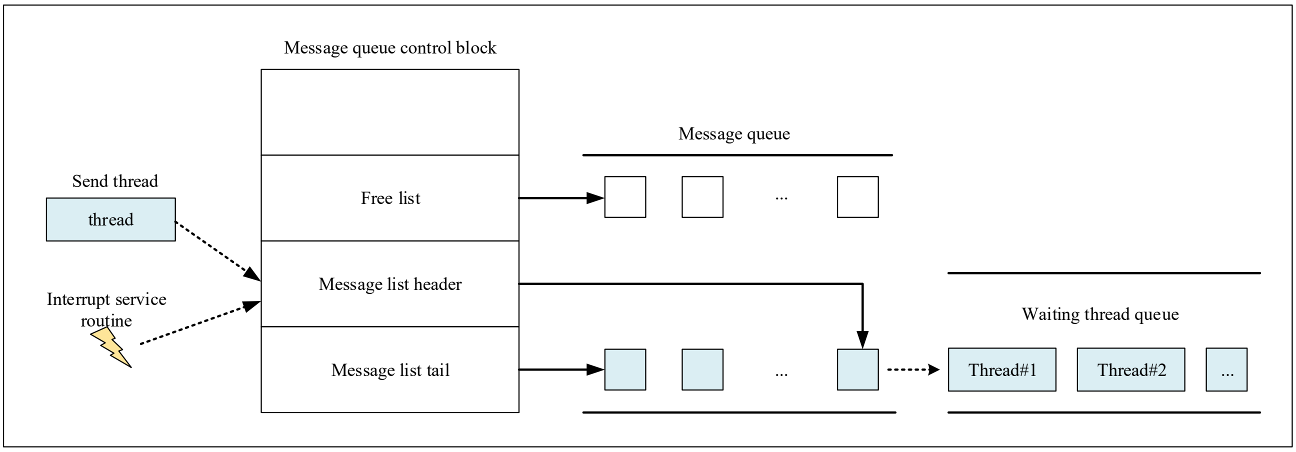
```
/* Create the mutex */
rt_mutex_init(&static_mutex,
              "smutex",
              RT_IPC_FLAG_FIFO);

/* Get the mutex */
rt_mutex_take(&static_mutex,
              10);

/* Relase the mutex */
rt_mutex_detach(&static_mutex);
```

## 2.4 Message Queue Example

The message queue can receive messages of variable length from threads or interrupt service routines, and buffer the messages in its own memory space. Other threads can also read the corresponding messages from the message queue, and when the message queue is empty, the reading thread can be suspended. When a new message arrives, the suspended thread will be woken up to receive and process the message. A message queue is an asynchronous communication method.

As shown in Figure 2-5, the thread or interrupt service routine can place one or more messages into a message queue. Likewise, one or more threads can get messages from the message queue. When multiple messages are transmitted to the message queue, the message that enters the message queue first is usually passed to the thread first. In other words, the thread receives the message that enters the message queue first, following the first-in-first-out principle (FIFO).

**Figure 2-5 Schematic Diagram of Message Queue Operation**



The message queue object of the RT-Thread operating system consists of multiple elements. When a message queue is created, it is assigned a message queue control block: message queue name, memory buffer, message size, and queue length. At the same time, each message queue object contains multiple message boxes, and each message box can store a message. The first and last message boxes in the message queue are called the message list header and the message list tail respectively, corresponding to msg_queue_head and msg_queue_tail in the message queue control block. Some message boxes may be empty, and they form a linked list of free message boxes through msg_queue_free. The total number of message boxes in all message queues is the length of the message queue, which can be specified when the message queue is created.

**Message Queue**

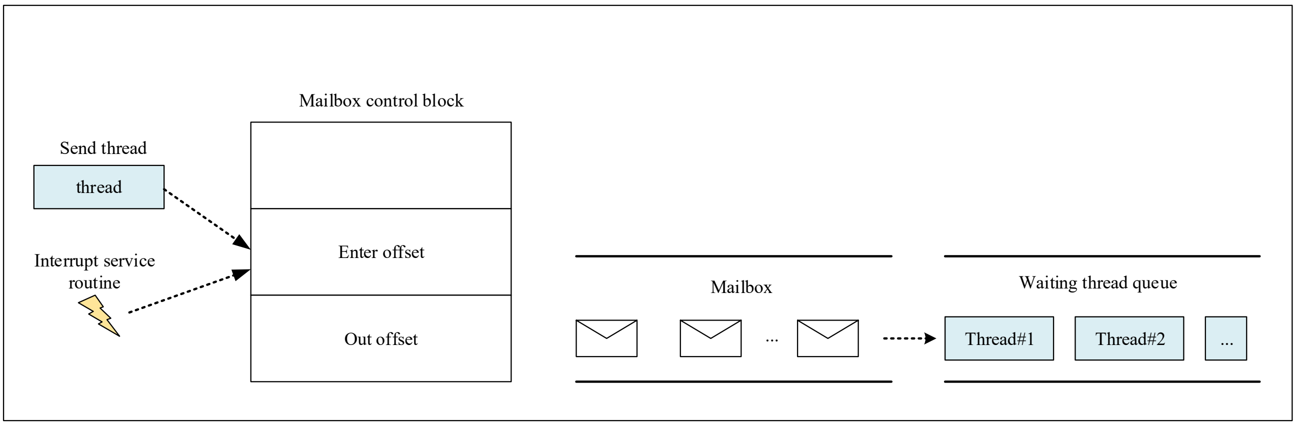
```
/* Create the message queue*/
rt_mq_init(&mq,
          "mqt",
           &msg_pool[0],
           128- sizeof(void*),
           sizeof(msg_pool),
           RT_IPC_FLAG_FIFO);

/* Send the message queue*/
rt_mq_send(&mq,
           &key_info[0],
           sizeof(key_info));

/* Receive the message queue*/
rt_mq_recv(&mq,
           &buf[0],
           sizeof(buf),
```

```
                RT_WAITING_FOREVER);
```

## 2.5 Mailbox Example

The mailbox of the RT-Thread operating system is used for inter-thread communication, which is characterized by low overhead and high efficiency. Each message in the mailbox can only hold a fixed 4-byte content (for a 32-bit processing system, the size of the pointer is 4 bytes, so a message can hold exactly one pointer). The typical mailbox is also called exchanging messages, as shown in Figure 2-6, the thread or interrupt service routine transmits a 4-byte message to mailbox,and one or more threads can receive these messages from the mailbox and process it.

**Figure 2-6 Schematic Diagram of Mailbox Operation**



**Mailbox**

```
/* Create the mailbox*/
rt_mb_init(&mb,
                "mbt",
                &mb_pool[0],
                sizeof(mb_pool)/4,
                RT_IPC_FLAG_FIFO);

/* Send the mailbox*/
rt_mb_send(&mb,
                (rt_uint32_t )&key_info[0]);

/* Receive the mailbox*/
rt_mb_recv(&mb,
                (rt_uint32_t*)&str,
                RT_WAITING_FOREVER);
```

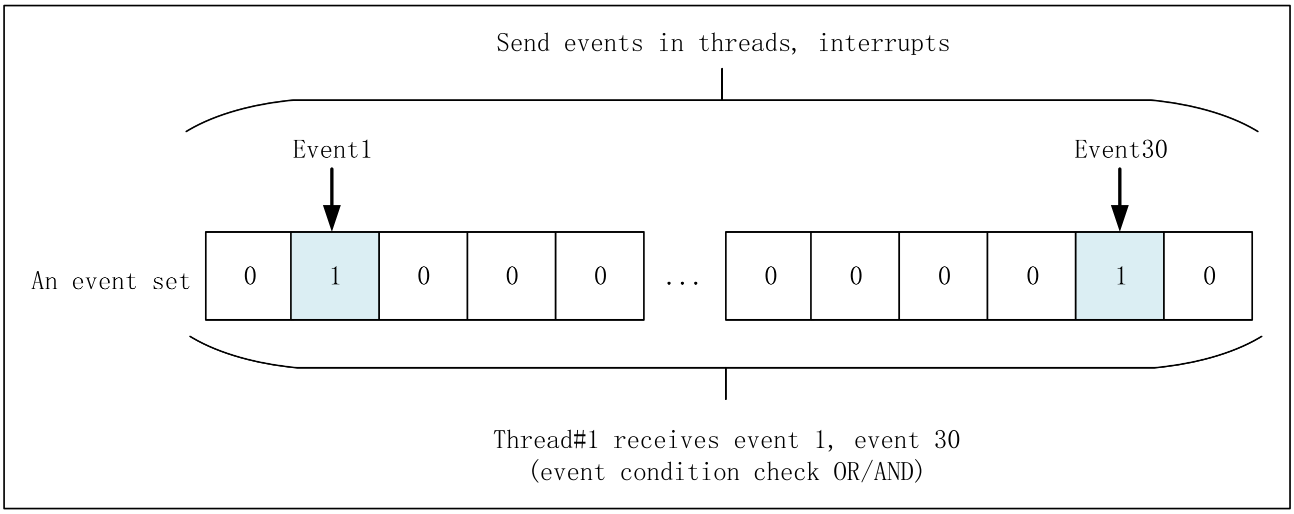
## 2.6 Event Example

The event set is mainly used for synchronization between threads. Unlike the semaphore, it is characterized by that it can achieve one-to-many and many-to-many synchronizations. The relationship between a thread and multiple events can be set as: any one event can wake up the thread, or the thread can be woken up for further processing only when several events have occurred. Similarly, multiple events can also be used to synchronize multiple threads.. This set of multiple events can be represented by a 32-bit unsigned integer variable, each bit of the variable represents an event, and the thread associates one or more events through "logical AND" or "logical OR" to form event combination. The "logical OR" of events is also called independent synchronization, which means that the thread is synchronized with any one of the events; the "logical AND" of events is also called associative synchronization, which means that the thread is synchronized with several events.

The event set defined by RT-Thread has the following characteristics:

- Events are only related to threads, and events are independent of each other: each thread can have 32 event flags, which are recorded by a 32-bit unsigned integer, and each bit represents an event
- Events are only used for synchronization and do not provide data transfer function
- Sending the same event to the thread multiple times (if the thread has not had time to read it away), the effect is equivalent to sending it only once. In RT-Thread, each thread has an event information flag, which has three attributes, namely RT_EVENT_FLAG_AND (logical AND), RT_EVENT_FLAG_OR (logical OR) and RT_EVENT_FLAG_CLEAR (clear flag). When a thread is waitting for event synchronization, it can judge whether the currently received event satisfies the synchronization condition through 32 event flags and this event information flag.

**Figure 2-7 Schematic Diagram of Event Operation**



As shown in Figure 2-7, 1st and 30th bits are set in the event flag of thread #1. If the event information flag bit is set to logical AND, it means that thread #1 will only be triggered to wake up after both event 1 and event 30 have occurred. If the event information flag bit is set to logical OR, the occurrence of either event 1 or event 30 will trigger the thread #1 to be woken up. If the clear flag bit of event information flag is set, then when thread #1 is woken up, it will actively clear event 1 and event 30 to zero. Otherwise the event flag will still exist (ie set to 1).

| **Event** |
|---|
| /* Create the event*/ |
| rt_event_init( &event, |

```
                "event",
                RT_IPC_FLAG_FIFO);

/* Send the event */
rt_event_send( &event,
                (1 << 0)) ;

/* Receive the event s*/
rt_event_recv( &event,
               ((1 << 0) | (1 << 1)),
                RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                10,
                &evt);
```

# 3 Supplementary Instructions

There are many different CPU architectures in the embedded system, such as Cortex-M, ARM920T, MIPS32, RISC-V, etc. In order to enable RT-Thread to run on chips with different CPU architectures, RT-Thread provides a libcpu abstraction layer to adapt to different CPU architectures. The libcpu layer provides a unified interface to the kernel, including global interrupt switches, thread stack initialization, context switching, etc.

The libcpu layer of RT-Thread also provides a unified CPU architecture porting interfaces. These interfaces include the global interrupt switch function, thread context switch function, clock tick configuration and interrupt function, Cache and so on. The following table shows the interfaces and variables that need to be implemented for CPU architecture porting.

**Table 3-1 Libcpu Porting Related API**

| Functions and Variables | Description |
|---|---|
| rt_base_t rt_hw_interrupt_disable(void); | Turn off global interrupt |
| void rt_hw_interrupt_enable(rt_base_t level); | Turn on global interrupt |
| rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter, rt_uint8_t *stack_addr, void *texit); | The initialization of the thread stack, the kernel will call this function in thread creation and thread initialization |
| void rt_hw_context_switch_to(rt_uint32 to); | Context switching without source thread, invoked when the scheduler starts the first thread, and in signal |
| void rt_hw_context_switch(rt_uint32 from,rt_uint32 to); | Switch from the “from” thread to the “to” thread for switching between threads |
| void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to); | Switch from the “from” thread to the “to” thread, which is used when switching in the interrupt |
| rt_uint32_t rt_thread_switch_interrupt_flag; | Indicates the flag that needs to be switched in the interrupt |
| rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread; | Used to save the “from” and “to” threads when the thread switches contexts |

# 4 Version History

| Version | Date | Changes |
|---|---|---|
| V1.0 | 2021.01.08 | Initial version |
| | | |

# 5 Disclaimer

This document is the exclusive property of NSING TECHNOLOGIES PTE. LTD.(Hereinafter referred to as NSING). This document, and the product of NSING described herein (Hereinafter referred to as the Product) are owned by NSING under the laws and treaties of Republic of Singapore and other applicable jurisdictions worldwide. The intellectual properties of the product belong to Nations Technologies Inc. and Nations Technologies Inc. does not grant any third party any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only. NSING reserves the right to make changes, corrections. enhancements, modifications, and improvements to this document at any time without notice. Please contact NSING and obtain the latest version of this document before placing orders. Although NATIONS has attempted to provide accurate and reliable information, NATIONS assumes no responsibility for the accuracy and reliability of this document. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NATIONS be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product.

NATIONS Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, Insecure Usage'. Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to supporter sustain life. All Insecure Usage shall be made at user's risk. User shall indemnify NATIONS and hold NATIONS harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage Any express or implied warranty with regard to this document or the Product, including, but not limited to. The warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law. Unless otherwise explicitly permitted by NATIONS, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.