Hi3861 V100 / Hi3861L V100 SDK

Development Guide

Issue	01
Date	2020-04-30

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document introduces how to develop the software development kit (SDK) of Hi3861 V100 and describes the SDK architecture, API implementation mechanism, and usage description (including the working principle, API usage and precautions by scenario).

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

The document is intended for:

- Technical support engineers
- Software development engineers

Symbol Conventions

The following table describes the symbols that may be found in this document.

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
🕮 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. ● In **1.1 Background**, the descriptions of the app layer and third party libraries are updated. **Figure 1-1** is updated. ● In **Application Scenario** of **2.2.2 Development Process**, the task priority range is updated. In **Table 2-2**, the descriptions of hi_task_register_idle_callback and hi_task_join are updated. ● In **2.2.3 Precautions**, the description of creating as few tasks as possible is added. ● In **2.4.3 Precautions**, the description that the **mutex**, **malloc**, **sleep**, and **delay** functions cannot be used is added. ● In **Table 2-8** of **2.5.2 Development Process**, the description of **hi_msg_queue_send_msg_to_front** is added. ● In **2.8.4 Programming Sample**, the code sample is updated. ● In **2.9.1 Overview**, the description of tick is updated. In **Table 2-16**, the descriptions of **hi_tick_register_callback** and **ms2systick** are added. ● In **2.10.1 Overview**, the description that the one-time trigger timer cannot be automatically deleted is removed. ● In **2.11.2 Development Process**, **Table 2-19** is updated. In **Table 2-20**, the descriptions of the following APIs are added: hi_get_hilink_partition_table, hi_get_hilink_pki_partition_table, hi_get_crash_partition_table, hi_get_fs_partition_table, hi_get_normal_nv_partition_table, hi_get_normal_nv_backup_partition_table, hi_get_usr_partition_table, and hi_get_factory_bin_partition_table. In **Flash Protection**, the description of the flash protection policy is updated. ● In **2.12.2 APIs**, the code sample of the function interface is deleted.
00B08	2020-04-20	In **2.12.4 Precautions**, the precaution that the **printf** and **diag** APIs share the same physical serial port is added.

Issue	Date	Change Description
00B07	2020-04-09	• In **2.2.1 Overview**, the recommended task priority range is updated. • In **Table 2-19** of **2.11.2 Development Process**, the size of the backup flash boot is updated. • **2.12 Maintainable and Testable APIs** is added.
00B06	2020-03-25	• **Table 2-19** is updated. • The description of ECC elements 1 and 2 when the kernel boot address changes during the modification of the flash partition table is updated.
00B05	2020-03-06	• In **2 System APIs**, the description of system interface functions, customized system resources, and common configuration items is added. • In **2.5.2 Development Process**, the reference solution for HI_ERR_MSG_Q_DELETE_FAIL in **Error Codes** is added. • In **2.5.4 Programming Sample**, the code sample for deleting a queue is deleted.
00B04	2020-02-12	• In **2.2.2 Development Process**, **Application Scenario** and **Error Codes** are updated. • In **2.3.2 Development Process**, **Error Codes** is updated. • In **2.4.2 Development Process**, **Error Codes** is updated. • In **2.5.2 Development Process**, **Error Codes** is updated. • In **2.11.2 Development Process**, **Modifying Flash Partitions** and the description of modifying the flash partition table if the flash memory does not comply with the default flash partition table are updated. • In **2.11.2 Development Process**, the description of feature macros in **Flash Protection** is added.
00B03	2020-01-15	**2.11 Flash Partitions and Protection** is added.

Issue	Date	Change Description
00B02	2019-12-19	• In **Table 2-3**, the reference solution description is updated. • In **Table 2-5**, the reference solution description of **HI_ERR_MEM_NOT_INIT** is updated. • In **Table 2-6**, the description of **hi_int_lock** is updated. • In **Table 2-7**, the reference solution description is updated. • In **2.4.4 Programming Sample**, the description of implemented functions is updated. • In **Table 2-9**, the reference solution description of **HI_ERR_MSG_Q_DELETE_FAIL** and **HI_ERR_MSG_WAIT_TIME_OUT** is updated. • In **2.5.4 Programming Sample**, the code sample is updated.
00B01	2019-11-15	This issue is the first draft release.

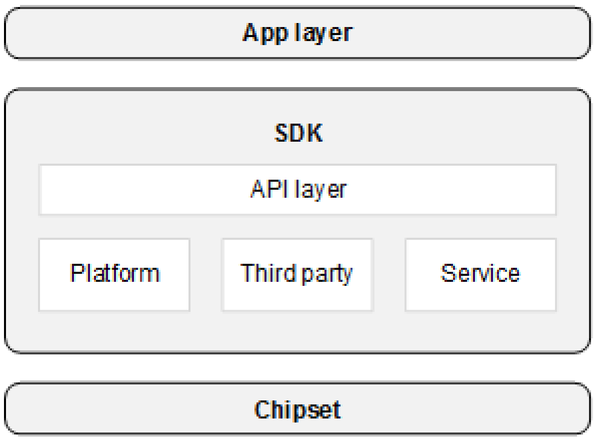
Contents

1 Introduction

1.1 Background

The Hi3861 platform software shields the underlying layer from the app layer, providing application programming interfaces (APIs) for the application software to implement required functions. **Figure 1-1** shows the typical system application framework.

Figure 1-1 System application framework



This framework can be divided into the following layers:

- App: indicates the application layer. The sample code provided by the SDK is stored in **app\demo\src**.
- API layer: provides common APIs developed based on the SDK.
- Platform layer: provides a board-level support package for the SoC, including the following functions:
  - Chip and peripheral driver

  - Operating system (OS)
  - System management
- Service layer: provides application protocol stacks such as Wi-Fi. It is used by upper-layer application software to send and receive data.
- Third party: provides third-party software libraries for the service layer or app layer.

1.2 Restrictions

- The UART driver, flash driver, watchdog, and non-volatile (NV) storage have been initialized during system boot. Therefore, do not initialize these modules again during development. Otherwise, system errors may occur.
- After the system is started, some system resources are occupied, including the interrupts, memory, tasks, message queues, events, semaphores, timers, and mutex. During application layer development, only the resources applied by the specified module can be released.
- System resources are configured in the **config/system_config.h** file. When new resources are added, the number of resources that are actually used must be also added.

2 System APIs

2.1 Overview

System APIs are used to perform operations on system resources such as tasks and events. The SDK allows you to customize system resources. To configure resources, you need to edit the **config/system_config.h** file. Properly configuring resource items as required can effectively reduce the waste of system resources, improve the running efficiency, and avoid resource insufficiency. **Table 2-1** describes the common configuration items.

Table 2-1 Common resource configuration items in system_config.h

Configuration Item	Description
LOSCFG_BASE_CORE_TSK_LIMIT_CONFIG	Maximum number of system tasks. If the ID of a task exceeds the value of this parameter, the task fails to be created.

Configuration Item	Description
LOSCFG_BASE_IPC_SEM_LIMIT_CONFIG	Maximum number of system semaphores. If resources are insufficient, a semaphore fails to be created.
LOSCFG_BASE_IPC_MUX_LIMIT_CONFIG	Maximum number of system mutexes. If resources are insufficient, a mutex fails to be created.
LOSCFG_BASE_IPC_QUEUE_LIMIT_CONFIG	Maximum number of message queues. If resources are insufficient, a message queue fails to be created.
LOSCFG_BASE_CORE_SWTMR_LIMIT_CONFIG	Maximum number of software timers. If resources are insufficient, a software timer fails to be created.
LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE_CONFIG	Stack size of an idle task
LOSCFG_BASE_CORE_TSK_SWTMR_STACK_SIZE_CONFIG	Stack size of a software timer task

Note: For the description of other configuration items, see the comments in **system_config.h**.

2.2 Task

2.2.1 Overview

A task is the minimum running unit that competes for system resources. A task can use or wait for system resources such as the CPU and memory space, and is independent of other tasks. The task module can provide multiple tasks for the user, implementing the switchover and communication between tasks, and helping the user to manage the service program flow.

- Multiple tasks are supported. One task represents one thread.
- Tasks are scheduled in preemption mode. In addition, tasks can be scheduled in turn based on time slices.
- A higher-priority task can interrupt a lower-priority task. A lower-priority task can be scheduled only after the higher-priority task is blocked or completed.
- System tasks need to be scheduled in time. Therefore, you are advised to set the task priority to a value ranging from 10 to 30. It is recommended that the priority of an application-level task be lower than that of a system-level task.

Important Concepts

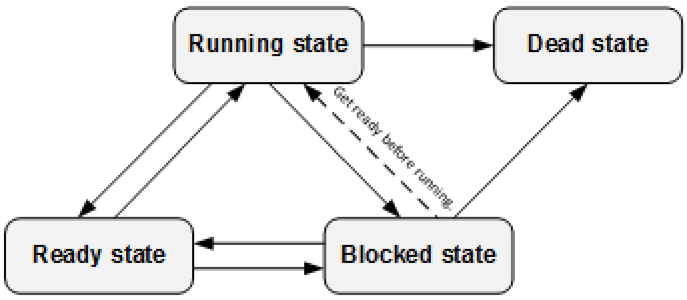
- **Task states**

Each task in the system has multiple runtime states. After the system is initialized, the created task can compete for certain resources in the system and be scheduled by the kernel.

There are four types of states:

- Ready state: The task is in the ready list and waits only for the CPU.
- Running state: The task is being executed.
- Blocked state: The task is not in the ready list. It may be suspended, delayed, waiting for semaphores, read/write queues, or read events.
- Dead state: The task is completed and waits for the system to reclaim resources.

Figure 2-1 Task states



Description of task state transition:

- Ready -> Running:

  After a task is created, it enters the ready state. When the task is switched, the highest-priority task in the ready list is executed and enters the running state. However, the task is still in the ready list.
- Running -> Blocked

  When a running task is blocked (for example, suspended, delayed, or reading semaphores), it is deleted from the ready list and changes from running state to blocked state. Then, the task is switched, and the highest-priority task remaining in the ready list is executed.
- Blocked -> Ready (Blocked -> Running):

  After a blocked task is restored (for example, the task is restored, the delay time expires, the semaphore reading times out, or the semaphore is successfully read), the restored task is added to the ready list and changes from blocked state to ready state. In this case, if the priority of the restored task is higher than that of the running task, the task is switched and changes from ready state to running state.
- Ready -> Blocked:

  A task may be blocked (suspended) when it is in the ready state. In this case, the task changes from ready state to blocked state and is deleted from the ready list. The task will not participate in task scheduling until it is restored.
- Running -> Ready:

  After a higher-priority task is created or restored, task scheduling is performed. The highest-priority task in the ready list changes to running state. Then, the original running task changes from running state to ready state and is still in the ready list.

- Running -> Dead

  After a running task is complete, it changes from running state to dead state. The dead state may refer to the normal exit or invalid state after a task is complete. For example, if the **LOS_TASK_STATUS_DETACHED** attribute is not set, the task is in the invalid state (dead state) after it is complete.

- Blocked -> Dead

  When a blocked task calls the deletion API, the task changes from blocked state to dead state.

- **Task ID**

  Task ID, which is returned to the user through parameters during task creation. The user can suspend, resume, or query a task by task ID.

- **Task Priority**

  The priority of a task determines the task to be executed after a task is switched. The highest-priority task in the ready list will be executed.

- **Task entry function**

  It is the function to be executed after a new task is scheduled. This function is implemented by the user. When a task is created, this function is specified by the task creation structure.

- **task control block (TCB)**

  Each task has a TCB. The TCB contains information such as the task context stack pointer, task status, task priority, task ID, task name, and task stack size. It reflects the running status of each task.

- **Task stack**

  Each task has an independent stack space, which is called a task stack. The information stored in the stack space includes local variables, registers, function parameters, and function return addresses. When a task is switched, context information of the switched task is stored in the task stack. In this way, after the task is restored, the task can be restored to the previous context and can continue to be executed at the switchover point.

- **Task context**

  Resources used during task running, such as registers, are called task context. When a task is suspended, other tasks continue to be executed. After the task is restored, if the task context is not saved, the register value may be changed during task switchover, which may lead to an unknown error. Therefore, when a task is suspended, task context information is stored in the task stack. In this way, the task can be restored to the previous context from the stack space, and code that is interrupted when the task is suspended continues to be executed.

- **Task switchover**

  Task switchover may involve obtaining the highest-priority task in the ready list, saving the context of the switched-out task, or restoring the context of switched-in task.

Operating Mechanism

The system task management module provides the following functions:

- Creating a task
- Delaying a task
- Suspending and restoring a task
- Scheduling a lock or unlock task
- Querying information about the TCB based on the ID

When a task is created, the system initializes the task stack and presets the context. In addition, the system places the "task entry function" in the corresponding address. In this way, when the task enters the running state for the first time, the "task entry function" is executed.

2.2.2 Development Process

Application Scenario

After a task is created, the kernel can perform operations such as lock task scheduling, unlock task scheduling, suspend, resume, and delay. In addition, the kernel can set and obtain the task priority. When a task is complete, if the task is in the **LOS_TASK_STATUS_DETACHED** state, the current task is automatically deleted.

The **app_main** API must be implemented in the user code. During system initialization, the **app_main** API is called. You can initialize the system in **app_main**. You can also create multiple tasks in **app_main**. It is recommended that the task priority range be [10, 30]. It is recommended that the priority of an application-level task be lower than that of a system-level task.

Function

Table 2-2 describes the APIs of the task management module.

Table 2-2 APIs of the task management module

API Name	Description
hi_task_create	Creates a task.
hi_task_delete	Deletes a task.
hi_task_suspend	Suspends a task.
hi_task_resume	Resumes a task.
hi_task_get_priority	Obtains the task priority.
hi_task_set_priority	Sets the task priority.
hi_task_get_info	Obtains the task information.

API Name	Description
hi_task_get_current_id	Obtains the current task ID.
hi_task_lock	Disables system task scheduling.
hi_task_unlock	Allows system task scheduling.
hi_sleep	Implements task sleep.
hi_task_register_idle_callback	Registers the callback function of an idle task.
hi_task_join	Waits until the task is complete.

Development Process

To create a task, perform the following steps:

Step 1 Configure the number of tasks in **system_config.h**.

Configure the maximum number of tasks supported by the system (**LOSCFG_BASE_CORE_TSK_LIMIT_CONFIG**) based on actual need.

Step 2 Lock a task by calling **hi_task_lock** to prevent high-priority task scheduling.

Step 3 Create a task by calling **hi_task_create**.

Step 4 Unlock the task by calling **hi_task_unlock**. The task is scheduled based on the priority.

Step 5 Suspend the task by calling **hi_task_suspend**. The task is suspended and waits for recovery.

Step 6 Resume the suspended task by calling **hi_task_resume**.

----End

Error Codes

Table 2-3 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_TASK_INVALID_PARAM	0x80000080	The input argument is invalid.	Check the task parameters.
2	HI_ERR_TASK_CREATE_FAIL	0x80000081	Failed to create a task.	Change the number of tasks because it has reached the upper limit.

No.	Definition	Actual Value	Description	Reference Solution
3	HI_ERR_TASK_DELETE_FAIL	0x80000082	Failed to delete the task.	Check to see that resources are released.
4	HI_ERR_TASK_SUPPEND_FAIL	0x80000083	Failed to suspend the task.	• Check to see that the task ID is valid. • Check to see that the task is not suspended.
5	HI_ERR_TASK_RESUME_FAIL	0x80000084	Failed to resume the task.	• Check to see that the task ID is valid. • Check to see that the task is not suspended.
6	HI_ERR_TASK_GET_PRI_FAIL	0x80000085	Failed to obtain the task priority.	• Check to see that the task ID is valid.
7	HI_ERR_TASK_SET_PRI_FAIL	0x80000086	Failed to set the task priority.	• Check to see that the priority is valid. • Check to see that the task is not idle. An idle task cannot be set. • Check to see that the task ID is valid.
8	HI_ERR_TASK_LOCK_FAIL	0x80000087	Failed to lock the task.	Check the parameters.
9	HI_ERR_TASK_UNLOCK_FAIL	0x80000088	Failed to unlock the task.	• Check to see that the task ID is valid. • Check to see that the task is not locked.
10	HI_ERR_TASK_DELAY_FAIL	0x80000089	Failed to delay the task.	Check the parameters.
11	HI_ERR_TASK_GET_INFO_FAIL	0x8000008A	Failed to obtain the task information.	• Check to see that the task ID is valid. • Check the parameters.
12	HI_ERR_TASK_REGISTER_SCHEDULE_FAIL	0x8000008B	Failed to register the task scheduling.	• Check to see that the task ID is valid. • Check the parameters.

No.	Definition	Actual Value	Description	Reference Solution
13	HI_ERR_TASK_NOT_CREATED	0x8000008C	Failed to create a task.	Check to see that the task ID is valid.

2.2.3 Precautions

- When a new task is created, the TCBs and task stack of the deleted task are reclaimed.
- The task name pointer is not allocated with space. When setting the task name, do not assign the address of the local variable to the task name pointer.
- If the size of a task stack is **0**, the default value is used.
- The size of the task stack should be 8-byte aligned. The principle for determining the task stack size is as follows: Do not use a too large or too small task stack size (to avoid waste or overflow).
- The current task is locked and cannot be suspended.
- Idle tasks and software timer tasks cannot be suspended or deleted.
- Interrupts are not disabled during lock task scheduling. Therefore, tasks can still be interrupted.
- Lock task scheduling must be used together with unlock task scheduling.
- Task scheduling may occur when you set the task priority.
- The number of task resources that can be configured in the system refers to the total number of task resources in the entire system instead of the number of task resources that can be used by users. For example, if the system software timer occupies one more task resource, the number of task resources that can be configured in the system decreases by one.
- You are advised not to use **hi_task_set_priority** to change the priority of a software timer task. Otherwise, the system may be faulty.
- **hi_task_set_priority** cannot be used in interrupts.
- If the task corresponding to the task ID passed by **hi_task_get_priority** is not created or the number of tasks exceeds the maximum, **0xFFFF** is returned.
- When deleting a task, ensure that the resources (such as mutexes and semaphores) applied by the task have been released.
- Create as few tasks as possible. Use the memory pool solution to avoid memory fragmentation.

2.2.4 Programming Sample

The following sample describes the basic methods of task operation, including how to create a task.

Sample code

```
#define TASK_PRI    25

static hi_void* example_task_entry(hi_void* data)
```

```
{
    dprintf("Example task is running!\n");
}
hi_void  example_task_init(hi_void)
{
    hi_u32 ret;
    hi_u32 taskid = 0;
    hi_task_attr attr = {0};

    attr.stack_size = 0x2000;
    attr.task_prio = TASK_PRI;
    attr.task_name = (hi_char*)"example_task";
    attr.task_policy = 1; /* SCHED_FIFO; */
    attr.task_cpuid = (hi_u32)NOT_BIND_CPU;

    ret = hi_task_create(&taskid, &attr, example_task_entry, HI_NULL);
    if (ret != HI_SUCCESS) {
        dprintf("Example_task create failed!\n");
    }
}
```

Verification

```
Example task is running!
```

2.3 Memory Management

2.3.1 Overview

The memory management module manages the memory resources of the system. It manages the memory usage of users and the OS by allocating and freeing the memory to optimize the memory usage and efficiency and handle memory fragments of the system to the greatest extent. The memory management of the OS is dynamic, and provides functions such as memory initialization, allocation, and release.

Dynamic memory refers to allocating memory blocks of specified sizes in the dynamic memory pool.

- Advantage: Resources are allocated on demand.
- Disadvantage: Fragments may occur in the memory pool.

2.3.2 Development Process

Application Scenario

Memory management aims to dynamically divide the memory allocated by the user and manage the memory range. Dynamic memory management is used when users need to use memory blocks of different sizes. To allocate memory, obtain memory blocks of specified sizes by using the dynamic memory allocation function of the OS. Once memory blocks are used up, the occupied memory is freed by using the dynamic memory release function, so that the memory blocks can be reused.

Function

Table 2-4 describes the APIs of the dynamic memory management module.

Table 2-4 APIs of the dynamic memory management module

API Name	Description
hi_malloc	Allocates memory with a specified size from a specified dynamic memory pool.
hi_free	Frees the memory that has been allocated.
hi_mem_get_sys_info	Obtains the memory information of a specified memory pool.
hi_mem_get_sys_info_crash	Obtains the memory information, which is used in the crash process.

Error Codes

Table 2-5 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_MEM_INVALID_PARAM	0x80000100	Input argument error	Check the input argument.
2	HI_ERR_MEM_CREAT_POOL_FAIL	0x80000101	Failed to create a buffer pool.	It is used internally in the system.
3	HI_ERR_MEM_CREATE_POOL_NOT_ENOUGH_HANDLE	0x80000102	Failed to create a buffer pool.	It is used internally in the system.
4	HI_ERR_MEM_FREE_FAIL	0x80000103	Failed to free the memory.	• Check the module ID. • Check the freed pointer.
5	HI_ERR_MEM_RE_INIT	0x80000104	The memory is initialized repeatedly.	It is used internally in the system.
6	HI_ERR_MEM_NOT_INIT	0x80000105	The memory is not initialized.	Check to see that the memory is initialized.
7	HI_ERR_MEM_CREAT_POOL_MALLOC_FAIL	0x80000106	Failed to create pool_malloc.	It is used internally in the system.

No.	Definition	Actual Value	Description	Reference Solution
8	HI_ERR_MEM_GET_INFO_FAIL	0x80000107	Failed to obtain memory usage information.	Check the input argument.
9	HI_ERR_MEM_GET_OS_INFO_NOK	0x80000108	Failed to obtain system information.	Check the memory usage.

2.3.3 Precautions

- If memory is successfully allocated by calling the **hi_malloc** function, the allocated space is returned. If the allocation fails, **NULL** is returned.
- When **hi_free** is called for multiple times in the system, a code indicating success is returned for the first time. However, repeated freeing of the same memory causes invalid pointer operations, and the result is unpredictable.
- You can check the memory usage of the module by calling **hi_mem_get_sys_info**.

2.3.4 Programming Sample

This sample shows how to allocate and free memory at the application layer.

Sample code

```
#define EXAMPLE_MEM_SIZE 100
hi_void example_mem(hi_void)
{
    hi_pvoid mem = hi_malloc(HI_MOD_ID_APP_COMMON,  EXAMPLE_MEM_SIZE);
    if (mem == HI_NULL) {
        dprintf("Malloc failed!\n");
    }

    dprintf("Using memory as expected!\n");

    hi_free(HI_MOD_ID_APP_COMMON, mem);
}
```

Verification

```
Using memory as expected!
```

2.4 Interrupt Mechanism

2.4.1 Overview

An interrupt is a process in which the CPU suspends execution of a current program and executes a new program. Interrupt-related hardware can be classified into the following types:

- Device: indicates the source that initiates an interrupt. When a device needs to request the CPU, an interrupt signal is generated. This signal is connected to the interrupt controller.
- Interrupt controller: receives interrupt inputs and reports them to the CPU. You can set the priority, trigger mode, and enable/disable status of an interrupt source.
- CPU: determines and executes interrupt tasks.

Terms and definitions:

- Interrupt ID: Interrupt request signals are numbered to facilitate interrupt source indexing.
- Interrupt request: An emergency event needs to apply to the CPU (by sending an electrical pulse signal) for interruption, and requires the CPU to suspend the current task and process the emergency event. This application process is called interrupt request.
- Interrupt priority: To enable the system to respond to and handle all interrupts in a timely manner, the system classifies interrupt sources into several levels based on the importance and urgency of interrupt events. All interrupt sources in the system have the same priority and do not support interrupt nesting or preemption.
- Interrupt handler: After a peripheral generates an interrupt request, the CPU suspends the current task and responds to the interrupt request.
- Interrupt triggering: The interrupt source sends a control signal to the CPU. When the interrupt trigger on the interface card is set to **1**, the interrupt source generates an interrupt, and the CPU is required to respond to the interrupt. The CPU suspends the current task and executes the corresponding interrupt service routine (ISR).
- Interrupt trigger type: An external interrupt request is sent to the CPU through a physical signal. The interrupt can be level-triggered or edge-triggered.
- Interrupt vector: indicates the entry address of the ISR.
- Interrupt vector table: stores interrupt vectors. Interrupt vectors map to interrupt IDs. The interrupt vectors are stored in the interrupt vector table in sequence based on the interrupt IDs.

2.4.2 Development Process

Application Scenario

When an interrupt request is generated, the CPU suspends the current task and responds to the peripheral request. You can register an interrupt handler by applying for an interrupt to specify the specific operation performed by the CPU when responding to the interrupt request.

Function

Table 2-6 describes the APIs of the interrupt mechanism.

Table 2-6 APIs of the interrupt mechanism

API Name	Description
hi_int_lock	Disables all interrupts. When interrupts are disabled, the functions that cause scheduling cannot be executed, such as **hi_sleep** or other blocking APIs. Disabling interrupts protects only predictable short-time operations. Otherwise, the interrupt response is affected, which may cause performance problems.
hi_int_restore	Restores the status before interrupts are disabled. The input argument must be the value of the current program status register (CPSR) saved before interrupts are disabled.
hi_is_int_context	Checks whether it is in the interrupt context.
hi_irq_enable	Enables a specified interrupt.
hi_irq_disable	Disables a specified interrupt.
hi_irq_request	Registers an interrupt.
hi_irq_free	Clears a registered interrupt.

Error Codes

Table 2-7 Definition of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_ISR_INVALID_PARAM	0x800000C0	Input argument error	Check the input argument.
2	HI_ERR_ISR_REQ_IRQ_FAIL	0x800000C1	Failed to register an interrupt.	• Check the input argument. • Check whether the number of interrupts reaches the upper limit. If yes, reduce the number of interrupt registrations.
3	HI_ERR_ISR_ADD_JOB_MALLOC_FAIL	0x800000C2	Failed to allocate memory for adding a task.	Check the input argument.

No.	Definition	Actual Value	Description	Reference Solution
4	HI_ERR_ISR_ADD_JOB_SYS_FAIL	0x800000C3	A system error occurred when adding a task.	Check the input argument.
5	HI_ERR_ISR_DEL_IRQ_FAIL	0x800000C4	Failed to delete the interrupt.	Check the input argument.
6	HI_ERR_ISR_ALREADY_CREATED	0x800000C5	Failed to register an interrupt because it has been registered before.	Check to see that no repeated registration interrupts exist.
7	HI_ERR_ISR_NOT_CREATED	0x800000C6	The interrupt is not registered.	Check to see that the interrupt has been registered.
8	HI_ERR_ISR_ENABLE_IRQ_FAIL	0x800000C7	Failed to enable the interrupt.	Check the input argument.
9	HI_ERR_ISR_IRQ_ADDR_NOK	0x800000C8	Interrupt address error	Check the registration interrupt flag.

2.4.3 Precautions

- You should configure the maximum number of supported interrupts and the register address for interrupt initialization based on the hardware.
- The interrupt handler should not take a long time. Otherwise, it may affect the timely response of the CPU to interrupts.
- The functions that cause task scheduling cannot be executed during interrupt response.
- The input argument of **hi_int_restore()** must be the CPSR value saved before interrupts are disabled by **hi_int_lock()**.
- The **mutex**, **malloc**, **sleep**, and **delay** functions cannot be used. The code must be as short as possible and run quickly. For complex operations, events must be thrown to the bottom half of the interrupt for processing.

2.4.4 Programming Sample

This sample implements the following functions:

- Disabling all interrupts
- Enabling an interrupt
- Disabling an interrupt
- Restoring the status before interrupts are disabled

Sample code

```
hi_void uart_irqhandle(hi_s32 irq,hi_pvoid dev)
{
    dprintf("\n int the func uart_irqhandle \n");
}
void example_irq()
{
    hi_u32 irq_idx = 1;
    hi_u32 uvIntSave;
    uvIntSave = hi_int_lock();
    hi_irq_enable(irq_idx);
    hi_irq_disable(irq_idx);
    hi_int_restore(uvIntSave);
}
```

2.5 Queue

2.5.1 Overview

A queue is also called a message queue. It is a data structure used for inter-task communication. It receives messages with variable lengths from tasks or interrupts. The receiver reads messages based on message IDs.

A task can read messages from the queue.

- When the message in the queue is empty, the read task is suspended.
- When a new message is in the queue, the suspended read task is woken up and the new message is processed.
- When processing services, the message queue provides an asynchronous processing mechanism. It allows a message to be put into a queue but does not process it immediately. In addition, the queue can buffer messages.

The system implements asynchronous task communication by using queues, which have the following features:

- Messages are queued in first in, first out (FIFO) mode, supporting asynchronous read/write.
- Both the read queue and write queue support the timeout mechanism.
- The type of the message to be sent is determined by both communication parties, and messages of different lengths (not exceeding the maximum value of the queue) may be allowed.
- A task can receive and send messages from any message queue.
- Multiple tasks can receive and send messages from the same message queue.
- After the queue is used, if the memory is dynamically allocated, the memory needs to be reclaimed by using the memory release function.

2.5.2 Development Process

Application Scenario

Multiple tasks can communicate with each other through message queues.

Function

Table 2-8 describes the APIs of the message queue.

Table 2-8 APIs of the message queue

API Name	Description
hi_msg_queue_create	Creates a message queue.
hi_msg_queue_delete	Deletes a message queue.
hi_msg_queue_send	Sends a message.
hi_msg_queue_wait	Receives a message.
hi_msg_queue_is_full	Checks whether the message queue is full.
hi_msg_queue_get_msg_num	Obtains the number of used message queues.
hi_msg_queue_get_msg_total	Obtains the number of message queues.
hi_msg_queue_send_msg_to_front	Sends a message to the queue header.

Development Process

To use the queue module, perform the following steps:

Step 1 Create a message queue by calling **hi_msg_queue_create**. After the creation is successful, you can obtain the ID of the message queue.

Step 2 Send a message by calling **hi_msg_queue_send**.

Step 3 Wait for the message to be received by calling **hi_msg_queue_wait**.

Step 4 Manage the queue status by calling **hi_msg_queue_is_full**, **hi_msg_queue_get_msg_num**, and **hi_msg_queue_get_msg_total**.

Step 5 Delete a queue by calling **hi_msg_queue_delete**.

----End

Error Codes

Queue failures may occur in creating and deleting a queue. You can locate the faults by checking the returned error codes.

Table 2-9 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_MSG_INVALID_PARAM	0x80000200	Input argument error	Check the input argument.
2	HI_ERR_MSG_CREATE_Q_FAIL	0x80000201	Failed to create a queue.	• Check the input argument. • Check the upper limit of the queue.
3	HI_ERR_MSG_DELETE_Q_FAIL	0x80000202	Failed to delete a queue.	• Check the input argument. • Check to see that the queue exists.
4	HI_ERR_MSG_WAIT_FAIL	0x80000203	Failed to receive a message.	Check the input argument.
5	HI_ERR_MSG_SEND_FAIL	0x80000204	Failed to send a message.	Check the input argument.
6	HI_ERR_MSG_GET_Q_INFO_FAIL	0x80000205	Failed to obtain the queue status.	Check the input argument.
7	HI_ERR_MSG_Q_DELETE_FAIL	0x80000206	Failed to delete a queue.	• Check the input argument. • Check to see that the queue exists. • Check whether the queue is occupied by a task.

No.	Definition	Actual Value	Description	Reference Solution
8	HI_ERR_MSG_WAIT_TIME_OUT	0x80000207	Message receiving timed out.	• Check to see that the sender sends messages and the RX queue is correctly configured. • Increase the timeout period.

2.5.3 Precautions

- Message wait: In the interrupt enable, interrupt disable, and lock task context, the message wait API must not be called, to avoid uncontrollable exception scheduling.
- Message TX: In the interrupt disable context, the message TX API must not be called, to avoid uncontrollable exception scheduling.
- Message TX (the timeout period is not 0): In the interrupt and lock task context, the message TX API must not be called, to avoid uncontrollable exception scheduling.
- The number of queue resources that can be configured in the system refers to the total number of queue resources in the entire system instead of the number of queue resources that can be used by users. For example, if the system software timer occupies one more queue resource, the number of queue resources that can be configured in the system decreases by one.
- The input argument **timeout** in the queue API function indicates the relative time.

2.5.4 Programming Sample

Create a queue and two tasks:

- Task 1: sending a message by using the send API
- Task 2: receiving a message by using the receive API

Perform the following steps:

Step 1 Create task 1 and task 2 by calling **hi_task_create**.

Step 2 Create a message queue by calling **hi_msg_queue_create**.

Step 3 Send the message in task 1 by calling **hi_msg_queue_send**.

Step 4 Receive the message in task 2 by calling **hi_msg_queue_wait**.

Step 5 Delete the queue by calling **hi_msg_queue_delete**.

----End

Sample code

```
static hi_u32 g_msg_queue;
hi_u8 abuf[] = "test is message x";
/*Task 1: Send data.*/
hi_void *example_send_task(hi_void *arg)
{
    hi_u32 i = 0,ret = 0;
    hi_u32 uwlen = sizeof(abuf);
    hi_unref_param(arg);
    while (i < 5) {
        abuf[uwlen - 2] = '0' + i;
        i++;
       /*Write the data in the ABUF to the queue.*/
        ret = hi_msg_queue_send(g_msg_queue, &abuf, 0, sizeof(abuf));
        if(ret != HI_ERR_SUCCESS) {
            dprintf("send message failure,error:%x\n",ret);
        }
        hi_sleep(5);
    }
    return HI_NULL;
}

/*Task 2: Receive data.*/
hi_void *example_recv_task(hi_void *arg)
{
    hi_u8 msg[50];
    hi_u32 ret = 0;
    hi_unref_param(arg);
    while (1) {
       /*Read data from the queue and save the data to the message.*/
        ret = hi_msg_queue_wait(g_msg_queue, &msg, HI_SYS_WAIT_FOREVER, sizeof(msg));
        if(ret != HI_ERR_SUCCESS) {
            dprintf("recv message failure,error:%x\n",ret);
            break;
        }
        dprintf("recv message:%s\n", (char *)msg);
        hi_sleep(5);
    }
/*Delete a queue as required. In most cases, you do not need to delete a queue. If a queue is
being used by a task, deleting the queue will fail. The following code is for API demonstration
only.*/
    if (hi_msg_queue_delete(g_msg_queue) != HI_ERR_SUCCESS) {
        dprintf("delete the queue failed!\n");
    } else {
        dprintf("delete the queue success!\n");
    }

    return HI_NULL;
}

int example_create_task(hi_void)
{
    hi_u32 ret = 0;
    hi_u32 task_send_msg, task_resv_msg;
    hi_task_attr task_param1;

   /*Create task 1.*/
    task_param1.task_prio = 25;
    task_param1.stack_size = 0x400;
    task_param1.task_name = "sendQueue";
    ret = hi_task_create(&task_send_msg, &task_param1, example_send_task, HI_NULL);
    if(ret != HI_ERR_SUCCESS) {
        dprintf("create task1 failed!,error:%x\n",ret);
        return ret;
    }
   /*Create task 2.*/
    ret = hi_task_create(&task_resv_msg, &task_param1, example_recv_task, HI_NULL);
    if(ret != HI_ERR_SUCCESS) {
        dprintf("create task2 failed!,error:%x\n",ret);
        return ret;
    }
  /*Create a queue.*/
    ret = hi_msg_queue_create(&g_msg_queue, 15, 50);
    if(ret != HI_ERR_SUCCESS) {
        dprintf("create queue failure!,error:%x\n",ret);
    }
    dprintf("create the queue success! queue_id = %d\n", g_msg_queue);
    return ret;
}
```

Verification

```
create the queue success! queue_id = 2
recv message:test is message 0
recv message:test is message 1
recv message:test is message 2
recv message:test is message 3
recv message:test is message 4
```

2.6 Event

2.6.1 Overview

An event is a mechanism for communication between tasks. It can be used to synchronize tasks. A task can wait for multiple events to occur.

- When one event occurs, the task is woken up to process the event.
- When multiple events occur, the task is woken up to process the events after all these events occur.

In a multi-task environment, tasks need to be synchronized. A waiting task is a sync task. Events can be synchronized in one-to-many or many-to-many mode.

- One-to-many sync mode: A task waits for the triggering of multiple events.
- Many-to-many sync mode: Multiple tasks wait for the triggering of multiple events.

A task can trigger and wait for an event by creating an event control block (ECB).

The event interface has the following features:

- An event is independent of a task. 32-bit unsigned integer variables are used to identify the event type in the task. The flags are as follows:
  - 0: The event does not occur.
  - 1: The event has occurred.
- An event is used only for synchronization between tasks and cannot be used to send data.

- Sending the same event type to a task for multiple times is equivalent to sending the event type only once.
- Multiple tasks can read and write the same event.
- The event read/write timeout mechanism is supported.

When reading an event, you can select the read mode as follows:

- All events (**HI_EVENT_WAITMODE_AND**): Read all event types in the mask. The read operation is successful only when all the read event types have occurred.
- Any event (**HI_EVENT_WAITMODE_OR**): Read any event type in the mask. The read operation is successful whenever any read event type has occurred.
- Clear event (**HI_EVENT_WAITMODE_CLR**): This is an additional read mode. It can be used together with **HI_EVENT_WAITMODE_AND** and **HI_EVENT_WAITMODE_OR**. After the read operation is successful, the corresponding event type is automatically cleared.

Operating mechanism:

When an event is read, one or more event types can be read based on the input argument **uwEventMask** (event mask type). After the event is read successfully, if **HI_EVENT_WAITMODE_CLR** is set, the read event type is cleared. Otherwise, the read event type is not cleared and needs to be cleared explicitly. You can select the read mode by using the input argument to read all events or any event of the event mask type.

- When an event is written, you can write one or multiple event types for the specified event. The write event will trigger task scheduling.
- When an event is cleared, the bit for the event is cleared based on the input argument event and the event type to be cleared.

2.6.2 Development Process

Application Scenario

Events can be used in multiple-task sync scenarios. In some sync scenarios, events can replace semaphores.

Functionality

Table 2-10 describes the APIs of the event module.

Table 2-10 APIs of the event module

API Name	Description
hi_event_init	Initializes an ECB.
hi_event_create	Obtains an event ID.
hi_event_wait	Reads the specified event type. The waiting timeout period is a relative time, in milliseconds (ms).

API Name	Description
hi_event_send	Writes a specified event type.
hi_event_clear	Clears a specified event type.
hi_event_delete	Destroys a specified ECB.

Development Process

To use the event module, perform the following steps:

Step 1 Initialize the event waiting queue by calling **hi_event_init**.

Step 2 Obtain an event ID by calling **hi_event_create**.

Step 3 Configure the event mask type by calling **hi_event_send**.

Step 4 Select the read mode by calling **hi_event_wait**.

Step 5 Clear the specified event type by calling **hi_event_clear**.

----End

Error Codes

Event failures may occur in event initialization, event destruction, event reading and writing, and event clearing.

Table 2-11 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_EVENT_INVALID_PARAM	0x80000240	The parameter is invalid.	Check the input argument.
2	HI_ERR_EVENT_CREATE_NO_HADNLE	0x80000241	No more handles can be allocated.	Increase the upper limit of the file handle count.
3	HI_ERR_EVENT_CREATE_SYS_FAIL	0x80000242	The ECB is a null pointer.	Check the initialization parameters of the system.
4	HI_ERR_EVENT_SEND_FAIL	0x80000243	The ECB is a null pointer.	Check the initialization parameters of the system.

No.	Definition	Actual Value	Description	Reference Solution
5	HI_ERR_EVENT_WAIT_FAIL	0x80000244	The ECB is a null pointer.	Check the initialization parameters of the system.
6	HI_ERR_EVENT_CLEAR_FAIL	0x80000245	Reserved for extension	None
7	HI_ERR_EVENT_RE_INIT	0x80000246	The event initialization is repeatedly called.	Avoid repeated calling of **hi_event_create**.
8	HI_ERR_EVENT_NOT_ENOUGH_MEMORY	0x80000247	The memory is insufficient.	Check the memory usage and free the memory resources.
9	HI_ERR_EVENT_NOT_INIT	0x80000248	The event is not initialized.	Initialize the ECB first.
10	HI_ERR_EVENT_DELETE_FAIL	0x80000249	Failed to destroy the event.	Check whether the event linked list is empty.
11	HI_ERR_EVENT_WAIT_TIME_OUT	0x8000024A	Read timeout	Set the timeout period to a proper value.

2.6.3 Precautions

- The read/write event API cannot be called before system initialization. If the API is called, the system running will become abnormal.
- In an interrupt, the event object can be written, but cannot be read.
- When the lock task is being scheduled, the task cannot be blocked or read.
- The input argument value of **hi_event_clear** is the ones' complement (**~event_bits**) for the specified event type to be cleared.
- The event mask supports bit[0] to bit[23], but does not support bit[24] to bit[31].

2.6.4 Programming Sample

In this example, the task **example_task_entry_event** creates the task **example_event** whose read is blocked. **example_task_entry_event** writes an event to the **example_event** task.

Step 1 Create the **example_event** task in the **example_task_entry_event** task. The priority of the **example_event** task is higher than that of the **example_task_entry_event** task.

Step 2 Read the event **0x00000001** in the **example_event** task. The task is blocked and switched to another task. Execute the **example_task_entry_event** task.

Step 3 Write the event **0x00000001** to the **example_event** task in the **example_task_entry_event** task. The task is switched. Execute the **example_event** task.

Step 4 Execute **example_event** until the task is complete.

Step 5 Execute **example_task_entry_event** until the task is complete.

----End

Sample code

```
#include "hi_event.h"
#include "hi_task.h"
#include <hi_mdm_types_base.h>
#define    TEST_EVENT     1
#define   UNUSED_PARAM(p)         p=p
static hi_u32   g_event_id=0;
extern int printf(const char *fmt, ...);
hi_void * example_event(hi_void* param)
{
    hi_u32 ret;
    hi_u32 uwEvent;
    UNUSED_PARAM(param);
/* Read the event in timeout wait mode. The timeout period is 200 ms. If the specified event is
not read after 200 ms, the read event times out and the task is woken up. */
    printf("example_event wait event 0x%x \n", TEST_EVENT);
    ret = hi_event_wait(g_event_id, TEST_EVENT,&uwEvent,200,
    HI_EVENT_WAITMODE_AND);
    if (ret == HI_ERR_SUCCESS) {
        printf("example_event read event :0x%x\n", uwEvent);
    } else {

        printf("example_event read event fail!\n");
    }
    return HI_NULL;
}
hi_u32 example_task_entry_event(hi_void)
{
    hi_u32 ret;
    hi_u32 taskid;
    hi_task_attr attr = {0};
/* Initialize the event.*/
hi_event_init(4, HI_NULL); /* 4: Set the maximum number of events to 4. */
    ret = hi_event_create(&g_event_id);
    if (ret != HI_ERR_SUCCESS) {
        printf("init event failed .\n");
        return HI_ERR_FAILURE;
    }
/*Create a task.*/
    attr.stack_size = HI_DEFAULT_STACKSIZE;
    attr.task_prio = 20;
    attr.task_name = "EventTsk1";
    ret = hi_task_create(&taskid, &attr, example_event, 0);
    if (ret!=HI_ERR_SUCCESS) {
        printf("create task failed .\n");
        return HI_ERR_FAILURE;
    }
/*Type of the event waiting for writing an example task*/
```

```
    printf("example_task_entry_event write event .\n");
    ret = hi_event_send(g_event_id,TEST_EVENT);
    if(ret != HI_ERR_SUCCESS){
        printf("event write failed .\n");
        return HI_ERR_FAILURE;
    }
    printf("example_task_entry_event event write success .\n");
/*Clear the event.*/
    ret = hi_event_clear(g_event_id,TEST_EVENT);
    if (ret != HI_ERR_SUCCESS) {
        printf("event clear failed .\n");
        return HI_ERR_FAILURE;
    }
    printf("example_task_entry_event event clear success.\n");
/*Delete a task.*/
    hi_task_delete(taskid);
    return HI_ERR_SUCCESS;
}
```

Verification

```
example_event wait event 0x1
example_task_entry_event write event .
example_event read event :0x1
example_task_entry_event event write success .
example_task_entry_event event clear success.
```

2.7 Mutex

2.7.1 Overview

A mutex is a locking mechanism that sometimes uses the same basic implementation as the binary semaphore. A mutex is used to exclusively process shared resources. A mutex can be in either of the following states:

- Lock: When a task is held, the mutex is locked. The task obtains the ownership of the mutex.
- Unlock: When a task releases the mutex, the mutex is unlocked and the task loses the ownership of the mutex.

When a task holds a mutex, other tasks cannot unlock or hold the mutex. In a multi-task environment, multiple tasks usually compete for the same shared resource. A mutex may be used to protect the shared resource, so as to implement exclusive access. In addition, a mutex can solve the priority inversion problem of the semaphore.

The mutex interface has the following feature:

- The priority inheritance algorithm is used to resolve the priority inversion problem.

2.7.2 Development Process

Application Scenario

A mutex provides a mutual exclusion mechanism to prevent two tasks from accessing the same shared resource at the same time.

Functionality

Table 2-12 describes the APIs of the mutex module.

Table 2-12 APIs of the mutex module

API Name	Description
hi_mux_create	Creates a mutex.
hi_mux_delete	Deletes a mutex.
hi_mux_pend	Applies for a mutex.
hi_mux_post	Releases a mutex.

Development Process

To use the mutex module, perform the following steps:

Step 1 Create a mutex by calling **hi_mux_create**.

Step 2 Apply for a mutex by calling **hi_mux_pend**.

There are three application modes:

- Non-blocking mode: A task needs to apply for a mutex. If the mutex is not held by any task or the task that holds the mutex is the task that applies for the mutex, the application is successful. Set the timeout period to **0**.
- Permanent blocking mode: A task needs to apply for a mutex. If the mutex is not occupied, the application is successful. Otherwise, the task enters the blocking state, and the system switches to a higher-priority task to continue execution. After a task enters the blocking state, the blocked task can be executed again only when another task releases the mutex. Set the timeout period to **HI_SYS_WAIT_FOREVER**.
- Scheduled blocking mode: A task needs to apply for a mutex. If the mutex is not occupied, the application is successful. Otherwise, the task enters the blocking state, and the system switches to a higher-priority task to continue execution. After a task enters the blocking state, the blocked task can be executed again only if another task releases the mutex before the specified timeout period expires, or if the specified timeout period times out. Set the timeout period to a proper value.

Step 3 Release a mutex by calling **hi_mux_post**.

- If a task is blocked by a specified mutex, the higher-priority task among the blocked tasks will be woken up. The task enters the ready state and is scheduled.
- If no task is blocked by the specified mutex, the mutex will be successfully released.

Step 4 Delete a mutex by calling **hi_mux_delete**.

----End

Error Codes

Mutex failures may occur in mutex creation, mutex deletion, mutex application, and mutex release.

Table 2-13 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_MUX_INVALID_PARAM	0x800001C0	The parameter is invalid.	Check the input argument of the function.
2	HI_ERR_MUX_CREATE_FAIL	0x800001C1	Failed to create a mutex.	Increase the maximum number of mutexes.
3	HI_ERR_MUX_DELETE_FAIL	0x800001C2	Failed to delete the mutex.	Some tasks are waiting for mutex application in the mutex linked list. Unlock all tasks before deleting them.
4	HI_ERR_MUX_PEND_FAIL	0x800001C3	The mutex pending timed out.	Increase the waiting time or set the always waiting mode.
5	HI_ERR_MUX_POST_FAIL	0x800001C4	Failed to release the mutex.	The current task does not hold the lock or the lock has been released.

2.7.3 Precautions

- Two tasks cannot lock the same mutex. If a task locks a mutex that has been held, the task will be suspended until that task unlocks the mutex.
- Mutexes cannot be used in the ISR.
- The real-time OS needs to ensure real-time scheduling of tasks and avoid long-time blocking of tasks. Therefore, after obtaining a mutex, the OS needs to release the mutex as soon as possible.
- When a mutex is being held, you cannot change the priority of the task that holds the mutex by calling **hi_task_set_priority**.

2.7.4 Programming Sample

This sample implements the following steps:

Step 1 The **example_task_entry_mux** task creates a mutex. The lock task is scheduled. Two tasks **example_mutex_task1** and **example_mutex_task2** are created. The priority of **example_mutex_task2** is higher than that of **example_mutex_task1**. Then the task scheduling is unlocked.

Step 2 **example_mutex_task2** is scheduled to permanently apply for a mutex, and then the task sleeps for 100 ms. **example_mutex_task2** is suspended, and **example_mutex_task1** is woken up.

Step 3 **example_mutex_task1** applies for a mutex, and the wait time is 10 ms. Because the mutex is still held by **example_mutex_task2**, if the mutex is not obtained 10 ms after **example_mutex_task1** is suspended, **example_mutex_task1** is woken up. When the system attempts to apply for a mutex in permanent waiting mode, **example_mutex_task1** is suspended

Step 4 After 100 ms, **example_mutex_task2** is woken up. After the mutex is released, **example_mutex_task1** is scheduled and executed, and then the mutex is released.

Step 5 After **example_mutex_task1** is executed, the **example_task_entry_mux** task is scheduled and executed 300 ms later, and the mutex is deleted.

----End

Sample code

```
#include "hi_mux.h"
#include "hi_task.h"
#include <hi_mdm_types_base.h>
#define   UNUSED_PARAM(p)        p=p
extern int printf(const char *fmt, ...);
static hi_u32 g_mux_id;
hi_void * example_mutex_task1(hi_void* param)
{
    hi_u32 ret;
    UNUSED_PARAM(param);
    printf("task1 try to get mutex,wait 10 ms.\n");
    ret = hi_mux_pend(g_mux_id, 10);
    if (ret == HI_ERR_SUCCESS) {
        printf("task1 get mutex g_mux_id.\n");
        hi_mux_post(g_mux_id);
    }else {
        printf("task1 timeout and try to get  mutex, wait forever.\n");
        ret= hi_mux_pend(g_mux_id, HI_SYS_WAIT_FOREVER);
        if (ret == HI_ERR_SUCCESS) {
            printf("task1 wait forever,get mutex g_mux_id.\n");
            hi_mux_post(g_mux_id);
        }
    }
    return HI_NULL;
}
hi_void * example_mutex_task2(hi_void* param)
{
    UNUSED_PARAM(param);
    printf("task2 try to get mutex, wait forever.\n");
    hi_mux_pend(g_mux_id, HI_SYS_WAIT_FOREVER);
    printf("task2 get mutex g_mux_id and suspend 100 ms.\n");
    hi_sleep(100);
    printf("task2 resumed and post the g_mux_id\n");
    hi_mux_post(g_mux_id);
    return HI_NULL;
}
hi_u32 example_task_entry_mux(hi_void)
{
    hi_u32 ret;
    hi_u32 taskid1;
    hi_u32 taskid2;
    hi_task_attr attr = {0};
    hi_mux_create(&g_mux_id);
    hi_task_lock();
    attr.stack_size = HI_DEFAULT_STACKSIZE;
    attr.task_prio =21;
    attr.task_name = "MutexTsk1";
    ret = hi_task_create(&taskid1, &attr, example_mutex_task1, 0);
    if (ret != HI_ERR_SUCCESS) {
        printf("create task failed .\n");
        return HI_ERR_FAILURE;
    }
    attr.stack_size = HI_DEFAULT_STACKSIZE;
    attr.task_prio = 20;
    attr.task_name = "MutexTsk2";
    ret = hi_task_create(&taskid2, &attr, example_mutex_task2, 0);
    if (ret != HI_ERR_SUCCESS) {
        printf("create task failed .\n");
        return HI_ERR_FAILURE;
    }
    hi_task_unlock();
    hi_sleep(300);
    hi_mux_delete(g_mux_id);
    hi_task_delete(taskid1);
    hi_task_delete(taskid2);
    return HI_ERR_SUCCESS;
}
```

Verification

```
task2 try to get mutex, wait forever.
task2 get mutx g_mux_id and suspend 100 ms.
task1 try to get mutex, wait 10 ms.
task1 timeout and try to get mutex, wait forever.
task2 resumed and post the g_mux_id.
task1 wait forever,get mutex g_mux_id.
```

2.8 Semaphore

2.8.1 Overview

A semaphore is a signaling mechanism for implementing communication between tasks. It synchronizes tasks and provides mutual exclusive access to critical resources, assisting a group of competing tasks in accessing critical resources.

In a multi-task system, tasks need to be synchronized or mutually exclusive to protect critical resources, which can be achieved by using the semaphore function. Generally, the counter value of a semaphore corresponds to the number of valid resources, indicating the number of remaining exclusive resources that can be occupied. There are two types of counter values:

- 0: No accumulated release operation (**hi_mux_post**) is performed, and there may be tasks blocked on the semaphore.
- Positive value: One or more release operations (**hi_mux_post**) are performed.

A semaphore for synchronization is different from that for mutex in usage:

- When used for synchronization, a semaphore is empty after being created. Task 1 obtains the semaphore and is blocked. Task 2 releases the semaphore after a certain condition occurs. Then, task 1 enters the ready or running state, achieving synchronization between two tasks.

- When used for mutex, a semaphore is full after being created. When critical resources are required, the semaphore is obtained first and then becomes empty. In this way, other tasks that need to use critical resources are blocked because the semaphore cannot be obtained, thereby ensuring the security of critical resources.

Semaphore operation principle:

- Semaphore initialization: Allocate memory for *N* semaphores (the value of N can be configured by users and is limited by the memory), initialize all semaphores to unused ones, and add them to the unused semaphore linked list for the system to use.
- Semaphore creation: Obtain a semaphore resource from the unused semaphore linked list and set an initial value.
- Semaphore application: If the counter value is greater than 0, the counter value decreases by 1 and a success message is returned. Otherwise, the task is blocked and waits for other tasks to release the semaphore. The waiting timeout period can be set. When a task is blocked by a semaphore, the task is placed at the end of the semaphore waiting task queue. When the semaphore is released, if no task is waiting for the semaphore, the counter value increases by 1 and a success message is returned. Otherwise, the semaphore is woken up and waits for the first task in the task queue.
- Semaphore deletion: Set the semaphore in use to an unused semaphore and mount the semaphore to the unused linked list.

A semaphore allows multiple tasks to simultaneously access the same resource, but restricts the number of large tasks that simultaneously access the resource. When the number of tasks accessing the same resource reaches the maximum, other tasks that attempt to obtain the resource are blocked until a task releases the semaphore.

2.8.2 Development Process

Application Scenario

A semaphore is a flexible sync mode. It can be used in multiple scenarios to implement functions such as lock, sync, and resource counting. It can also be used for sync between tasks and tasks or between interrupts and tasks.

Function

Table 2-14 describes the APIs of the semaphore module.

Table 2-14 APIs of the semaphore module

API Name	Description
hi_sem_create	Creates a semaphore.
hi_sem_bcreate	Creates a binary semaphore.

API Name	Description
hi_sem_delete	Deletes a semaphore.
hi_sem_wait	Applies for a semaphore.
hi_sem_signal	Releases a semaphore.

Development Process

To use the semaphore module, perform the following steps:

Step 1 Create a semaphore by calling **hi_sem_create**.

Step 2 Apply for a semaphore by calling **hi_sem_wait**.

There are three modes for applying for a semaphore:

- Non-blocking mode: Task processing requires a semaphore. If the number of tasks of the semaphore does not reach the upper limit, the application is successful. Otherwise, a message indicating an application failure is returned. The timeout period is set to **0**.
- Permanent blocking mode: Task processing requires a semaphore. If the number of tasks of the semaphore does not reach the upper limit, the application is successful. Otherwise, the task enters the blocking state, and the system switches to a higher-priority task to continue execution. After a task enters the blocking state, the blocked task cannot be executed again until another task releases the semaphore. Set the timeout period to **HI_SYS_WAIT_FOREVER**.
- Scheduled blocking mode: Task processing requires a semaphore. If the number of tasks of the semaphore does not reach the upper limit, the application is successful. Otherwise, the task enters the blocking state, and the system switches to a higher-priority task to continue execution. After a task enters the blocking state, the blocked task cannot be executed again until another task releases the semaphore before the specified time expires or until the specified time has expired. Set the timeout period to a proper value.

Step 3 Release the semaphore by calling **hi_sem_signal**.

- If a task is blocked by a specified semaphore, the first task in the queue blocked by the semaphore is woken up. The task enters the ready state and is scheduled.
- If no task is blocked in the specified semaphore, the semaphore is released successfully.

Step 4 Delete the semaphore by calling **hi_sem_delete**.

----End

Error Codes

Semaphore failures may occur in creating, applying for, releasing, and deleting a semaphore. You can locate the faults by checking the returned error codes.

Table 2-15 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_SEM_INVALID_PARAM	0x80000180	The parameter is invalid.	Check the input argument of the function.
2	HI_ERR_SEM_CREATE_FAIL	0x80000181	Failed to create a semaphore.	Release semaphore resources.
3	HI_ERR_SEM_DELETE_FAIL	0x80000182	Failed to delete the semaphore.	Wake up all tasks waiting for the semaphore and delete the semaphore.
4	HI_ERR_SEM_WAIT_FAIL	0x80000183	The scheduled time is invalid or the input semaphore ID is abnormal.	Check for code exceptions.
5	HI_ERR_SEM_SIG_FAIL	0x80000184	The number of semaphores exceeded the maximum.	No task is waiting for this semaphore.
6	HI_ERR_SEM_WAIT_TIME_OUT	0x80000185	Obtaining the semaphore timed out.	Set the waiting time to be within a proper range.

2.8.3 Precautions

Interrupts cannot be blocked. Therefore, the blocking mode cannot be used in interrupts during semaphore application.

2.8.4 Programming Sample

This sample implements the following functions:

Step 1 The test task **example_task_entry_sem** creates a semaphore. The lock task is scheduled. Two tasks **example_sem_task1** and **example_sem_task2** are created. The priority of **example_sem_task2** is higher than that of **example_sem_task1**. The two tasks apply for the same semaphore. After the unlock task is scheduled, the two tasks are blocked. The test task **example_task_entry_sem** releases the semaphore.

Step 2 **example_sem_task2** obtains the semaphore and is scheduled. Then, the task sleeps for 200 ms. **example_sem_task2** is delayed, and **example_sem_task1** is woken up.

Step 3 **example_sem_task1** applies for a semaphore in scheduled blocking mode. The wait time is 100 ms. Because the semaphore is still held by **example_sem_task2**, if the semaphore is not obtained 10 ms after **example_sem_task1** is suspended, **example_sem_task1** is woken up, attempted to apply for a semaphore in permanent blocking mode. Then, **example_sem_task1** is suspended.

Step 4 After 200 ms, **example_sem_task2** is woken up. After the semaphore is released, **example_sem_task1** is scheduled and runs, and then the semaphore is released.

Step 5 After **example_sem_task1** is executed, **example_task_entry_sem** is woken up 400 ms later. The semaphore and two tasks are deleted.

----End

Sample code

```
#include "hi_sem.h"
#include "hi_task.h"
#include <hi_mdm_types_base.h>
#define   UNUSED_PARAM(p)       p=p
/*Test the task priority.*/
#define TASK_PRIO_TEST? 21
/*Semaphore structure ID*/
static hi_u32 g_usSemID;
extern int printf(const char *fmt, ...);
hi_void * example_sem_task1(hi_void* param)
{
    hi_u32 ret;
    UNUSED_PARAM(param);
    printf("example_sem_task1 try get sem g_usSemID ,timeout 100 ms.\n");
/*Apply for a semaphore in scheduled blocking mode. The scheduled time is 100 ms.*/
    ret = hi_sem_wait(g_usSemID, 100);
/*A semaphore is successfully applied for.*/
    if(HI_ERR_SUCCESS == ret) {
        hi_sem_signal(g_usSemID);
    }
/*The scheduled time is reached, but no semaphore is applied for.*/
    if (HI_ERR_SEM_WAIT_TIME_OUT == ret) {
        printf("example_sem_task1 timeout and try get sem g_usSemID wait forever.\n");
/*Apply for a semaphore in permanent blocking mode.*/
        ret = hi_sem_wait(g_usSemID, HI_SYS_WAIT_FOREVER);
        printf("example_sem_task1 wait_forever and get sem g_usSemID .\n");
        if (HI_ERR_SUCCESS == ret) {
            hi_sem_signal(g_usSemID);
        }
    }
    return HI_NULL;
}
hi_void * example_sem_task2(hi_void* param)
{
    hi_u32 ret;
    UNUSED_PARAM(param);
    printf("example_sem_task2 try get sem g_usSemID wait forever.\n");
/*Apply for a semaphore in permanent blocking mode.*/
    ret = hi_sem_wait(g_usSemID, HI_SYS_WAIT_FOREVER);
    if (HI_ERR_SUCCESS == ret) {
        printf("example_sem_task2 get sem g_usSemID and then delay 200ms .\n");
    }
/*The task sleeps for 200 ms.*/
    hi_sleep(200);
    printf("example_sem_task2 post sem g_usSemID .\n");
/*Release the semaphore.*/
    hi_sem_signal(g_usSemID);
    return HI_NULL;
}
hi_u32 example_task_entry_sem(hi_void)
{
    hi_u32 ret;
    hi_u32 taskid1;
    hi_u32 taskid2;
    hi_task_attr attr = {0};
/*Create a semaphore.*/
    hi_sem_create(&g_usSemID,0);
/*Schedule a lock task.*/
    hi_task_lock();
/*Create a task.*/
/*Create task 1.*/
    attr.stack_size = HI_DEFAULT_STACKSIZE;
    attr.task_prio = TASK_PRIO_TEST;
    attr.task_name = "MutexTsk1";
    ret = hi_task_create(&taskid1, &attr, example_sem_task1, 0);
    if (ret!=HI_ERR_SUCCESS) {
        printf("create task failed .\n");
        return HI_ERR_FAILURE;
    }
/*Create task 2.*/
    attr.stack_size = HI_DEFAULT_STACKSIZE;
    attr.task_prio = TASK_PRIO_TEST-1;
    attr.task_name = "MutexTsk2";
    ret = hi_task_create(&taskid2, &attr, example_sem_task2, 0);
    if (ret!=HI_ERR_SUCCESS) {
        printf("create task failed .\n");
        return HI_ERR_FAILURE;
    }
/*Schedule an unlock task.*/
    hi_task_unlock();
    ret = hi_sem_signal(g_usSemID);
/*The task sleeps for 400 ms.*/
    hi_sleep(400);
/*Delete the semaphore.*/
    hi_sem_delete(g_usSemID);
/*Delete task 1.*/
    hi_task_delete(taskid1);
/*Delete task 2.*/
    hi_task_delete(taskid2);
    return HI_ERR_SUCCESS;
}
```

Verification

The compilation result is as follows:

```
example_sem_task2 try get sem g_usSemID wait forever.
example_sem_task1 try get sem g_usSemID ,timeout 100 ms.
example_sem_task2 get sem g_usSemID and then delay 200ms .
example_sem_task1 timeout and tty get sem ggusSemID wait forever.
example_sem_task2 post sem g_usSemID .
example_sem_task1wait_forever and get sem g_usSemID .
```

2.9 Time Management

2.9.1 Overview

Based on the system clock, the time management module provides time-related services for applications, including time conversion, statistics collection, and delay functions.

- Cycle: minimum count time unit. The cycle duration is determined by the system dominant frequency, that is, the number of cycles per second.
- Tick: basic time unit of the OS. The duration is determined by the CPU frequency and the number of ticks per second. You can configure **OS_SYS_CLOCK_CONFIG** and **LOSCFG_BASE_CORE_TICK_PER_SECOND_CONFIG** in **system_config.h**.

Table 2-16 describes the APIs of the time management module.

Table 2-16 APIs of the time management module

API Name	Description
hi_udelay	Specifies the wait time (unit: microsecond).
hi_get_tick	Obtains the number of ticks for the four lower bytes.
hi_get_tick64	Obtains the number of ticks.
hi_get_seconds	Converts tick to second.
hi_get_milli_seconds	Converts tick to millisecond (ms).
hi_get_us	Converts tick to microsecond (μs).
hi_get_real_time	Obtains the system time. The structure member contains seconds and nanoseconds (ns).
hi_set_real_time	Sets the system time (unit: second).
hi_tick_register_callback	Registers the callback function for tick interrupt response.
ms2systick	Converts milliseconds to ticks.

2.9.2 Development Process

Application Scenario

You need to know the current system running time and the conversion between ticks and seconds or milliseconds.

Development Process

To use the time management module, perform the following steps:

Step 1 Call the clock conversion API.

Step 2 Obtains statistics about the system tick count.

Obtain the current tick count of the system by calling **hi_get_tick**.

----End

Error Codes

None

2.9.3 Precautions

- The system tick count can be obtained only after the system clock is enabled.
- System ticks are not counted when interrupts are disabled. Therefore, the number of system ticks cannot be used as the accurate time for calculation.

2.9.4 Programming Sample

None

2.10 Software Timer

2.10.1 Overview

A software timer is based on tick clock interrupts of the system. After the configured number of ticks is reached, the user-defined callback function is triggered. The timer accuracy is related to the tick clock period of the system.

- The number of hardware timers is limited by hardware and cannot meet user requirements. There, software timers are provided.
- With the use of software timers, the number of timers is increased, allowing more timing services to be created.

The software timer supports the following functions:

- Creating a software timer
- Starting the software timer
- Stopping the software timer
- Deleting the software timer

Operation mechanism:

- A software timer uses a queue, a task resource, and FIFO of the system. A shorter timer is always closer to the queue head than a longer timer, and is therefore preferentially triggered.
- When a tick interrupt is received, scan the timing task of the software timer in the tick interrupt handling function to check whether the timer times out. If yes, record the timeout timer.
- After the tick interrupt handling function is complete, the software timer task (with a high priority) is woken up. In this task, the timeout callback function for the recorded timer is called.

The software timer provides three timer mechanisms:

- One-off triggering timer: After the timer is started, only one timer event is triggered, and then the timer is automatically deleted.
- Periodic triggering timer: A timer event is triggered periodically until the timer is manually stopped. Otherwise, the timer is permanently executed.

2.10.2 Development Process

Application Scenario

- Create a timer that is triggered only once. After the timer times out, the callback function is executed.
- Create a timer that is triggered periodically. After the timer times out, the user-defined callback function is executed.

Functionality

Table 2-17 describes the APIs of the software timer module.

Table 2-17 APIs of the software timer module

API Name	Description
hi_timer_create	Creates a timer.
hi_timer_delete	Deletes the timer.
hi_timer_start	Starts the timer.
hi_timer_stop	Stops the timer.

Development Process

To use the software timer module, perform the following steps:

Step 1 Create a timer by calling **hi_timer_create**.

Return the function execution result, which can be success or failure.

Step 2 Start the timer by calling **hi_timer_start**.

Step 3 Stop the timer by calling **hi_timer_stop**.

Step 4 Delete the timer by calling **hi_timer_delete**.

----End

Error Codes

Software timer failures may occur in creating, deleting, pausing, and restarting a timer. You can locate the faults by checking the returned error codes.

Table 2-18 Description of error codes

No.	Definition	Actual Value	Description	Reference Solution
1	HI_ERR_TIMER_FAILURE	0x80000140	The timer fails.	Common timer error

No.	Definition	Actual Value	Description	Reference Solution
2	HI_ERR_TIMER_INVALID_PARAM	0x80000141	The input argument is invalid.	Check the input argument of the function.
3	HI_ERR_TIMER_CREATE_HANDLE_FAIL	0x80000142	Reserved for extension	None
4	HI_ERR_TIMER_START_FAIL	0x80000143	Reserved for extension	None
5	HI_ERR_TIMER_HANDLE_NOT_CREATE	0x80000144	The timer handle is not created.	Create a software timer.
6	HI_ERR_TIMER_HANDLE_INVALID	0x80000145	The handle is invalid.	Check the input argument of the function.
7	HI_ERR_TIMER_STATUS_INVALID	0x80000146	The software timer status is incorrect.	Check the software timer status.
8	HI_ERR_TIMER_STATUS_START	0x80000147	Reserved for extension	None
9	HI_ERR_TIMER_INVALID_MODE	0x80000148	The timer startup mode is invalid.	Set a valid timer startup mode.
10	HI_ERR_TIMER_EXPIRE_INVALID	0x80000149	The interval of the software timer is 0.	Set a correct timer interval.
11	HI_ERR_TIMER_FUNCTION_NULL	0x8000014a	The address passed to the timer timeout callback function is a null pointer.	Pass a valid address to the callback function.
12	HI_ERR_TIMER_HANDLE_MAXSIZE	0x8000014b	The number of software timers exceeds the maximum value.	Redefine the maximum number of software timers or wait for a software timer to release resources.

No.	Definition	Actual Value	Description	Reference Solution
13	HI_ERR_TIMER_MALLOC_FAIL	0x8000014c	Reserved for extension	None
14	HI_ERR_TIMER_NOT_INIT	0x8000014d	Reserved for extension	None

2.10.3 Precautions

- Do not perform too many operations in the callback function of a software timer. Do not use APIs or operations that may cause task suspension or blocking.
- A software timer uses a queue and a task resource of the system. The priority of the software timer task is set to 0 and cannot be modified.
- The number of software timer resources that can be configured in the system refers to the total number of software timer resources that can be used in the entire system rather than that can be used by users. For example, if the system software timer occupies one more software timer resource, the software timer resource that can be used by the user decreases by 1.
- For creating a one-off software timer, when the callback function is executed after the timer times out, the system automatically deletes the software timer and reclaims resources.
- For creating a timer that does not automatically delete attributes of itself, the timer deletion API must be called to delete the timer and the timer resources must be reclaimed to prevent resource leakage.

2.10.4 Programming Sample

This sample implements the following functions:

- Creating, starting, deleting, and pausing a software timer.
- Using the one-off software timer and periodic software timer

Sample code

```
#include "hi_sem.h"
#include "hi_timer.h"
#include "hi_task.h"
#include <hi_mdm_types_rom.h>
#define   UNUSED_PARAM(p)        p=p
extern int printf(const char *fmt, ...);
hi_void timer1_callback(hi_u32 arg);// callback fuction
hi_void timer2_callback(hi_u32 arg);
static hi_u32 g_timercount1 = 0;
static hi_u32 g_timercount2 = 0;
hi_void timer1_callback(hi_u32 arg)// Callback function 1
{
    UNUSED_PARAM(arg);
    g_timercount1++;
    printf("g_timercount1=%d\n",g_timercount1);
}
```

```
hi_void timer2_callback(hi_u32 arg)// Callback function 2
{
    UNUSED_PARAM(arg);
    g_timercount2 ++;
    printf("g_timercount2=%d\n",g_timercount2);
}
hi_void example_task_entry_timer(hi_void)
{
    hi_u32 timer_id1;
    hi_u32 timer_id2;
/*Create a one-off software timer with the duration of 1000 ms. Callback function 1 is executed
when 1000 ms is reached. */
    hi_timer_create(&timer_id1);
/*Create a periodic software timer and execute callback function 2 every 100 ms. */
    hi_timer_create(&timer_id2);
    printf("create Timer1 success\n");
    hi_timer_start(timer_id1, HI_TIMER_TYPE_ONCE, 1000, timer1_callback, 0);
    printf("start Timer1 success\n");
hi_sleep(1200);// Delay 1200 ms.
hi_timer_delete(timer_id1);// Delete the software timer.
    printf("delete Timer1 success\n");
hi_timer_start(timer_id2, HI_TIMER_TYPE_PERIOD, 100, timer2_callback, 0);// Start the periodic
software timer.
    printf("start Timer2\n");
    hi_sleep(1000);
    hi_timer_stop(timer_id2);
    printf("stop Timer2 success\n");
    hi_timer_delete(timer_id2);
    printf("delete Timer2 success\n");
}
```

Verification

```
create Timer1 success
start Timer1 success
g_timercount1=1
delete Timer1 success
start Timer2
g_timercount2=1
g_timercount2=2
g_timercount2=3
g_timercount2=4
g_timercount2=5
g_timercount2=6
g_timercount2=7
g_timercount2=8
g_timercount2=9
g_timercount2=10
stop Timer2 success
delete Timer2 success
```

2.11 Flash Partitions and Protection

2.11.1 Overview

Flash partitions provide guidance on planning flash space based on the actual flash capacity in the early phase of product development. Ensure that the flash memory space is consistent with the protection range and protection policy supported by the flash memory. Flash partitions are aligned based on the sector size (minimum erasable unit) of the flash memory. Generally, the flash size is 4 KB. The addresses of all partitions are relative to the flash address, that is, the addresses start from 0. You can re-allocate the size of each flash partition based on the space of the flash memory.

The SPI flash is easy to operate and fast. However, if the signal communication is abnormal, the flash content may be incorrect, which may cause serious consequences. To prevent the flash data from being tampered with, you need to protect the largest area of the flash memory as much as possible. The SPI flash supports almost all flash protection policies. Flash protection is highly recommended. You can replan the flash protection area based on the specific flash protection policy.

2.11.2 Development Process

Modifying Flash Partitions

Table 2-19 lists the default flash partitions.

Table 2-19 Default flash partitions

No.	Partition Name	Size (KB)	Address Space	Description
1	Flash Boot	32	0x0_0000–0x0_7FFF	The start address is fixed.
2	NV factory partition	8	0x0_8000–0x0_9FFF	NV factory partition, which stores key factory parameters. The flash partition table is stored in the NV factory partition.
3	NV working partition	8	0x0_A000–0x0_BFFF	-
4	Original backup of the NV working partition.	4	0x0_C000–0x0_CFFF	This NV partition is the original backup of the NV working partition when the .bin file is generated during compilation and cannot be modified during system running.
5	Kernel A	912	0x0_D000–0x0xF_0FFF	• Dual-partition OTA support: Kernel A and kernel B store the corresponding kernel and NV files. • Compression OTA support: Kernel A and kernel B are combined into one partition. The header of the partition stores the kernel and NV files, while the tail of the

No.	Partition Name	Size (KB)	Address Space	Description
6	Kernel B	968	0xF_1000–0x1E_2FFF	partition stores the compressed upgrade files. ● The partition of the production test image overlaps with that of kernel B (the last 600 KB of kernel B, that is, 0x14_D000–0x1E_3000). Therefore, if the production test image is not erased before delivery, it may be damaged during subsequent OTA update. For details about the upgrade method, see the *Hi3861 V100/ Hi3861L V100 Upgrade Development Guide*.
7	HILINK	8	0x1E_3000–0x1E_4FFF	Dedicated for HiLink storage configuration information. If HiLink is not used or other clouds are used, use partitions as required.
8	File system	44	0x1E_5000–0x1E_FFFF	-
9	User reserved partition	20	0x1F_0000~0x1F_4FFF	User reserved partition for storing user-defined information
10	HILINK PKI	8	0x1F_5000~0x1F_6FFF	Dedicated for HiLink to store PKI certificates. If HiLink is not used or other clouds are used, use partitions as required.
11	Crash information	4	0x1F_7000–0x1F_7FFF	-
12	FlashBoot backup	32	0x1F_8000–0x1F_FFFF	If the header flash boot is damaged, the system attempts to boot from the tail flash boot.

The structure of the partition table is as follows:

```
typedef struct {
    hi_u32  addr    :24;   /**<Address of the flash partition. The maximum size is 16 MB.*/
   hi_u32  id      :7;    /**<Flash area ID*/
    hi_u32  dir     :1;    /**<Storage direction of the flash area. 0: normal sequence. 1: reverse
sequence (CNend) */
    hi_u32  size    :24;   /**<Size of the flash partition (unit: byte)*/
    hi_u32  reserve :8;    /**<Reserved area*/
    hi_u32  addition;      /**<Supplementary information about the flash partition*/
} hi_flash_partition_info;

/**
 *  @ingroup hct_flash_partiton
  *  Flash partition table
 */
typedef struct {
    hi_flash_partition_info table[HI_FLASH_PARTITON_MAX]; /**< Flash partition table items*/
} hi_flash_partition_table;
```

If the flash memory is inconsistent with the default flash partition table, modify the flash partition table.

- Modify the **tools/nvtool/xml_file/mss_nvi_db.xml** file based on the flash memory. In the flash partition table information, the NV factory partition ID is **2**. Replace the original partition table information with the new one.
- If the address of the NV factory partition changes, modify the **HI_FNV_DEFAULT_ADDR** macro (the macro is located in **hi_nv.h** for the kernel and **hi_flashboot.h** for flashboot).
- If the kernel boot address changes, you need to change the address of **signature** in the **SConstruct** file. **RSA** in **signature** indicates the file encrypted by RSA, and **ECC** indicates the file encrypted by ECC. Element 1 indicates that an address of kernel A is a start address of a partition table entry of kernel A plus 0x3C0, and element 2 indicates that an address of kernel B is a start address of a partition table entry of kernel B plus 0x3C0. Modify the partition table information in the **build/scripts/make_upg_file.py** script, for example, the flash boot address (**boot_st_addr**) and size (**boot_size**), NV factory partition start address (**ftm1_st_addr**), NV factory partition size (**ftm1_size**), NV working partition start address (**nv_file_st_addr**), NV working partition size (**nv_file_size**), and kernel start address (**kernel_st_addr**). Then rebuild the project to replace the flash partition table.
- You are advised to change the default partition address macro in **flash_partition_table.c** and **boot_partition_table.c** to be the same as that in the new partition table.

Table 2-20 describes the APIs of the software timer module.

Table 2-20 API description of the flash partition table

API Name	Description
hi_flash_partition_init	Initializes the flash partition table.
hi_get_partition_table	Obtains the flash partition table.
hi_get_hilink_partition_table	Obtains the address and size of the HiLink partition.
hi_get_hilink_pki_partition_table	Obtains the address and size of the HiLink PKI partition.
hi_get_crash_partition_table	Obtains the address and size of the crash information partition.

API Name	Description
hi_get_fs_partition_table	Obtains the address and size of the file system partition.
hi_get_normal_nv_partition_table	Obtains the address and size of a non-factory partition.
hi_get_normal_nv_backup_partition_table	Obtains the address and size of the non-factory backup partition.
hi_get_usr_partition_table	Obtains the address and size of the user reserved partition.
hi_get_factory_bin_partition_table	Obtaining the address and size of the production test image.

Flash Protection

The SDK provides a flash protection policy with the following key points:

- The flash boot partition is not protected.
- When the flash memory is erased or written in FlashBoot, the full-chip protection is disabled to be compatible with more flash components. Non-volatile instructions are used. If the flash protection function is not required, you can disable the feature macro **HI_FLASH_SUPPORT_FLASH_PROTECT**, which is located in **hi_flashboot_flash.c**.
- When data is erased or written in the kernel, the protection range should be modified based on the address. During running, the high address protection and low address protection are disabled, and the time can be set. After the timeout period expires, the protection can be restored (area beyond the flash boot). If the flash protection function is not required, you can disable the feature macro **HI_FLASH_SUPPORT_FLASH_PROTECT**, which is located in **flash_prv.h**.

The policy varies according to the flash model. Prioritize the **Write Enable for Volatile Status Register (50h)** command, which will help to improve the register operation efficiency and get rid of operation times allowed by the service life of the flash memory.

The protection area varies according to the flash component. When replacing a flash component, you must update the **g_flash_protect_size_upper** table in **flash_protect.c**. The following components support flash protection by default: W25Q16JL, W25Q16JW, GD25LE16, GD25WQ16, EN25S16, EN25QH16, and P25Q16LE. If the flash model is not in the default list, ensure that **g_flash_protect_size_upper** is within the range supported by the flash memory. You can use flash protection by using either of the following methods:

- In the **flash_init_cfg** function of **flash_ram.c**, you can replace the default flash memory table **g_flash_default_info_tbl** as follows:
- Replace the comparison part of **flash chip_name** in the **hi_flash_protect_init** function of **flash_protect.c** or change **g_flash_ctrl->basic_info.chip_name** to a non-UNKNOWN value before comparison.

2.11.3 Precautions

- When updating the flash partition, you must divide the flash partitions based on the specific component and service.
- When updating the flash memory, you must update the flash protection table based on the flash memory.

2.11.4 Programming Sample

None

2.12 Maintainable and Testable APIs

2.12.1 Overview

Hi3861 V100 and Hi3861L V100 support four debugging methods: **printf**, **hi_at_printf**, shell, and HSO (HiStudio).

- **printf** and **hi_at_printf** can be used to print debug logs in the tool over the serial port.
- The shell provides initialization and registration functions.
- HSO must be used with a dedicated debugging tool.

2.12.2 APIs

2.12.2.1 printf

printf is the simplest and most commonly used API for printing debugging information. It can transmit messages to the corresponding serial port.

printf is declared in **hi_early_debug.h**.

2.12.2.2 hi_at_printf

hi_at_printf is used in the same as **printf**. The current code provides an API for printing log information for the AT command set. After an AT command is typed, **hi_at_printf** is called to print information and output a response.

hi_at_printf is declared in **hi_at.h**.

2.12.2.3 Shell

Hi3861 V100 and Hi3861L V100 provide shell initialization and input and output function registration for you to develop shell commands.

The APIs related to the shell are declared in **hi_shell.h**.

2.12.2.4 HSO

Hi3861 V100 and Hi3861L V100 allow the HiStudio-RD tool to print log messages and provide multiple printing APIs for different printing levels in different scenarios.

The APIs related to HSO are declared in **hi_diag.h**.

2.12.3 Instructions

NOTE

The HiStudio-RD tool is dedicated for debugging HSO APIs. For details about how to use HSO, see the *Hi3861 V100/Hi3861L V100 HSO User Guide*.

Step 1 Add a header file and call the log printing API.

Step 2 Compile and generate a .bin file, and burn the .bin file to the demo board.

Step 3 Start the serial port debugging tool to view the printed log information.

----End

2.12.4 Precautions

- The maintainability and testability solution can be enabled only during debugging. You are advised to disable it in the release version.
- Currently, the HSO debugging tool is used only to print logs during SDK fault locating.
- The **printf** and **diag** APIs share the same physical serial port. Therefore, you are advised not to use **printf** to output debugging logs (especially in the interrupt scenario) when using the HSO tool for debugging. This prevents the data output by **printf** from being inserted into the data packet reported by **diag** to the HSO tool, which causes lost or incorrect data received by the HSO tool.