



Hi3861V100 / Hi3861LV100 SDK

开发指南

文档版本 04

发布日期 2020-08-10

版权所有 © 上海海思技术有限公司2020。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思和其他海思商标均为海思技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

上海海思技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://www.hisilicon.com/cn/>

客户服务邮箱： support@hisilicon.com



前言

概述

本文档主要介绍Hi3861V100的SDK开发相关内容，包括SDK架构、接口实现机制与使用说明（包括工作原理、按场景描述接口使用方法和注意事项）。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3861	V100
Hi3861L	V100



读者对象

本文档主要适用于以下工程师：



- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	表示如不可避免则将会导致死亡或严重伤害的具有高等级风险的危害。
 警告	表示如不可避免则可能导致死亡或严重伤害的具有中等级风险的危害。



符号	说明
 注意	表示如不可避免则可能导致轻微或中度伤害的具有低等级风险的危害。
须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
04	2020-08-10	在“ 2.9.1 概述 ”的 表2-16 中更新接口名称ms2systick为hi_ms2systick。
03	2020-08-04	在“ 2.11.2 开发流程 ”中更新SDK交付策略中提供的Flash保护策略要点；新增关于Flash保护的注意内容。
02	2020-07-24	在“ 2.11.2 开发流程 ”的 修改Flash分区 中更新对 表2-19 描述的内容。



文档版本	发布日期	修改说明
01	2020-04-30	<p>第一次正式版本发布。</p> <ul style="list-style-type: none"> 更新“1.1 背景介绍”中关于APP层、第三方库的说明；更新图1-1。 在“2.2.2 开发流程”的使用场景中更新使用任务优先级范围；在表2-2中新增hi_task_register_idle_callback、hi_task_join的接口描述。 在“2.2.3 注意事项”中新增关于尽量少创建task的注意说明。 在“2.4.3 注意事项”中新增关于不能使用mutex、malloc、sleep、delay函数的注意说明。 在“2.5.2 开发流程”的表2-8中新增hi_msg_queue_send_msg_to_front的接口描述。 更新“2.8.4 编程实例”的代码示例。 在“2.9.1 概述”中更新关于Tick的说明；在表2-16中新增hi_tick_register_callback、ms2systick的接口描述。 在“2.10.1 概述”中删除关于不能自动删除单次触发定时器的说明。 在“2.11.2 开发流程”中更新表2-19；表2-20中新增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、hi_get_factory_bin_partition_table的接口描述；更新Flash保护中关于Flash保护策略的描述。 删除“2.12.2 接口介绍”中关于函数接口的代码示例。
00B08	2020-04-20	<p>在“2.12.4 注意事项”中新增关于printf和diag接口共用同一个物理串口的注意说明。</p>
00B07	2020-04-09	<ul style="list-style-type: none"> 更新“2.2.1 概述”中建议用户使用的任务优先级范围。 更新“2.11.2 开发流程”中表2-19的备份Flash Boot大小。 新增“2.12 可维可测接口”小节。
00B06	2020-03-25	<ul style="list-style-type: none"> 更新“表2-19”。 更新关于修改Flash分区表时如果Kernel启动地址也发生变化的“ECC”元素1和元素2含义的描述。
00B05	2020-03-06	<ul style="list-style-type: none"> 在“2 系统接口”中新增关于系统接口功能、客制化系统资源、常用配置项的描述。 更新“2.5.2 开发流程”的错误码中HI_ERR_MSG_Q_DELETE_FAIL的参考解决方案。 更新“2.5.4 编程实例”中关于删除队列的代码示例。



文档版本	发布日期	修改说明
00B04	2020-02-12	<ul style="list-style-type: none">更新“2.2.2 开发流程”的使用场景、错误码。更新“2.3.2 开发流程”的错误码。更新“2.4.2 开发流程”的错误码。更新“2.5.2 开发流程”的错误码。更新“2.11.2 开发流程”的修改Flash分区，以及如果Flash器件与默认Flash分区表不符合时修改Flash分区表的说明。在“2.11.2 开发流程”的Flash保护中新增关于特性宏的说明。
00B03	2020-01-15	新增“ 2.11 Flash分区与Flash保护 ”小节。
00B02	2019-12-19	<ul style="list-style-type: none">更新“表2-3”的参考解决方案说明。更新“表2-5”中HI_ERR_MEM_NOT_INIT的参考解决方案说明。更新“表2-6”中hi_int_lock的描述。更新“表2-7”的参考解决方案说明。更新“2.4.4 编程实例”实现的功能说明。更新“表2-9”中HI_ERR_MSG_Q_DELETE_FAIL、HI_ERR_MSG_WAIT_TIME_OUT的参考解决方案说明。更新“2.5.4 编程实例”的代码示例。
00B01	2019-11-15	第一次临时版本发布。



目录

前言.....	i
1 概述.....	1
1.1 背景介绍.....	1
1.2 使用约束.....	2
2 系统接口.....	3
2.1 概述.....	3
2.2 任务.....	4
2.2.1 概述.....	4
2.2.2 开发流程.....	6
2.2.3 注意事项.....	9
2.2.4 编程实例.....	9
2.3 内存管理.....	10
2.3.1 概述.....	10
2.3.2 开发流程.....	10
2.3.3 注意事项.....	11
2.3.4 编程实例.....	11
2.4 中断机制.....	12
2.4.1 概述.....	12
2.4.2 开发流程.....	13
2.4.3 注意事项.....	14
2.4.4 编程实例.....	14
2.5 队列.....	15
2.5.1 概述.....	15
2.5.2 开发流程.....	15
2.5.3 注意事项.....	17
2.5.4 编程实例.....	17
2.6 事件.....	19
2.6.1 概述.....	19
2.6.2 开发流程.....	20
2.6.3 注意事项.....	22
2.6.4 编程实例.....	22
2.7 互斥锁.....	23



2.7.1 概述.....	24
2.7.2 开发流程.....	24
2.7.3 注意事项.....	25
2.7.4 编程实例.....	26
2.8 信号量.....	27
2.8.1 概述.....	27
2.8.2 开发流程.....	28
2.8.3 注意事项.....	30
2.8.4 编程实例.....	30
2.9 时间管理.....	32
2.9.1 概述.....	32
2.9.2 开发流程.....	32
2.9.3 注意事项.....	33
2.9.4 编程实例.....	33
2.10 软件定时器.....	33
2.10.1 概述.....	33
2.10.2 开发流程.....	34
2.10.3 注意事项.....	35
2.10.4 编程实例.....	36
2.11 Flash 分区与 Flash 保护.....	37
2.11.1 概述.....	37
2.11.2 开发流程.....	37
2.11.3 注意事项.....	41
2.11.4 编程实例.....	41
2.12 可维可测接口.....	41
2.12.1 概述.....	41
2.12.2 接口介绍.....	41
2.12.2.1 printf 接口.....	41
2.12.2.2 hi_at_printf 接口.....	41
2.12.2.3 shell 工具.....	41
2.12.2.4 HSO 调试工具.....	41
2.12.3 使用方法.....	42
2.12.4 注意事项.....	42



1 概述

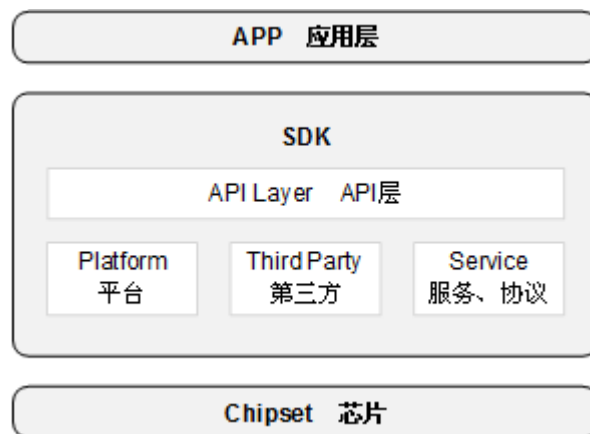
1.1 背景介绍

1.2 使用约束

1.1 背景介绍

海思Hi3861系列的平台软件对应用层实现了底层屏蔽，并对应用软件直接提供API(Application Programming Interface)接口完成相应功能。典型的系统应用架构如图1-1所示。

图 1-1 系统应用框架图



该框架可以分为以下几个层次：

- APP层：即应用层。SDK提供的代码示例在SDK的代码目录：app\demo\src。
- API层：提供基于SDK开发的通用接口。
- Platform平台层：提供SOC系统板级支持包，包括如下功能：
 - 芯片和外围器件驱动
 - 操作系统
 - 系统管理



- Service服务层：提供包含WiFi等应用协议栈。用于上层应用软件进行数据收发等操作。
- 第三方库：提供给Service服务层或提供给应用层使用的第三方软件库。

1.2 使用约束

- 系统在启动过程中已经初始化了UART驱动、Flash驱动、看门狗、NV等。用户在开发过程中，请勿重复初始化这些模块，否则引起系统错误。
- 系统启动运行后，会占用一些系统资源：中断、内存、任务、消息队列、事件、信号量、定时器、互斥锁等。用户在做应用层开发当中释放资源时，用户仅可释放该模块申请的资源。
- 系统资源配置在config/system_config.h文件中，用户新增资源使用时，需在原有基础上增加实际使用的资源个数。



2 系统接口

- 2.1 概述
- 2.2 任务
- 2.3 内存管理
- 2.4 中断机制
- 2.5 队列
- 2.6 事件
- 2.7 互斥锁
- 2.8 信号量
- 2.9 时间管理
- 2.10 软件定时器
- 2.11 Flash分区与Flash保护
- 2.12 可维可测接口

2.1 概述

系统接口是包括对任务、事件等系统资源进行所需操作的接口。SDK支持用户客制化系统资源，资源配置需编辑config/system_config.h文件，按需合理地配置资源项将有效降低系统资源浪费，提高运行效率，又可避免资源不足。常用配置项如表2-1所示。

表 2-1 system_config.h 中的常用资源配置项

配置项	描述
LOSCFG_BASE_CORE_TSK_LIMIT_CONFIG	系统任务数上限。创建任务时，ID超过此数值将创建失败。
LOSCFG_BASE_IPC_SEM_LIMIT_CONFIG	系统信号量个数上限。资源不足时，引起创建信号量失败。



配置项	描述
LOSCFG_BASE_IPC_MUX_LIMIT_CONFIG	系统互斥锁个数上限。资源不足时，引起创建互斥锁失败。
LOSCFG_BASE_IPC_QUEUE_LIMIT_CONFIG	消息队列个数上限。资源不足时，创建消息队列将失败。
LOSCFG_BASE_CORE_SWTMR_LIMIT_CONFIG	软件定时器个数上限。资源不足时，创建软件定时器将失败。
LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE_CONFIG	IDLE任务的栈大小。
LOSCFG_BASE_CORE_TSK_SWTMR_STACK_SIZE_CONFIG	软件定时器任务的栈大小。

注：其他配置项描述请参见system_config.h中注释。

2.2 任务

2.2.1 概述

任务是竞争系统资源的最小运行单元。任务可以使用或等待CPU、使用内存空间等系统资源，并独立于其它任务运行。任务模块可以给用户提供多个任务，实现了任务之间的切换和通信，帮助用户管理业务程序流程。

- 支持多任务，一个任务表示一个线程。
- 任务是抢占式调度机制，同时支持时间片轮转调度方式。
- 高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务阻塞或结束后才能得到调度。
- 由于系统自身任务需要及时调度，建议用户使用任务优先级范围是[10,30]。应用级任务建议使用低于系统级任务的优先级。

重要概念

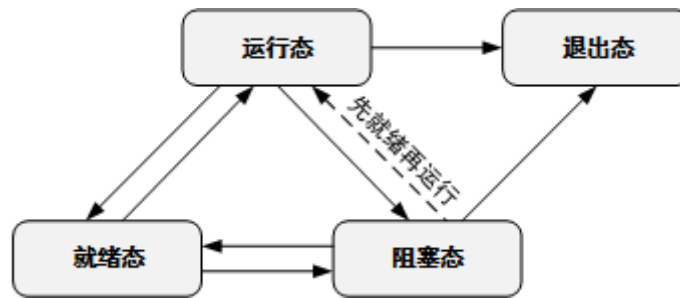
- **任务状态**

系统中的每一个任务都有多种运行状态。系统初始化完成后，创建的任务就可以在系统中竞争一定的资源，由内核进行调度。

任务状态通常分为以下4种：

- 就绪（Ready）：该任务在就绪列表中，只等待CPU。
- 运行（Running）：该任务正在执行。
- 阻塞（Blocked）：该任务不在就绪列表中。包含任务被挂起、任务被延时、任务正在等待信号量、读写队列或者等待读事件等。
- 退出态（Dead）：该任务运行结束，等待系统回收资源。

图 2-1 任务状态示意图



任务状态迁移说明：

- 就绪态→运行态：

任务创建后进入就绪态，发生任务切换时，就绪列表中最高优先级的任务被执行，从而进入运行态，但此刻该任务依旧在就绪列表中。

- 运行态→阻塞态：

正在运行的任务发生阻塞（挂起、延时、读信号量等）时，该任务会从就绪列表中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪列表中剩余最高优先级任务。

- 阻塞态→就绪态（阻塞态→运行态）：

阻塞的任务被恢复后（任务恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的任务会被加入就绪列表，从而由阻塞态变成就绪态；此时如果被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务由就绪态变成运行态。

- 就绪态→阻塞态：

任务也有可能就在就绪态时被阻塞（挂起），此时任务状态会有就绪态转变为阻塞态，该任务从就绪列表中删除，不会参与任务调度，直到该任务被恢复。

- 运行态→就绪态：

有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪列表中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就绪态，依然在就绪列表中。

- 运行态→退出态

运行中的任务运行结束，任务状态由运行态变为退出态。退出态包含任务运行结束的正常退出以及Invalid状态。例如，未设置分离属性（`LOS_TASK_STATUS_DETACHED`）的任务，运行结束后对外呈现的是Invalid状态，即退出态。

- 阻塞态→退出态

阻塞的任务调用删除接口，任务状态由阻塞态变为退出态。

• 任务ID

任务ID，在任务创建时通过参数返回给用户，作为任务的一个非常重要的标识。用户可以通过任务ID对指定任务进行任务挂起、任务恢复、查询任务名等操作。

• 任务优先级

优先级表示任务执行的优先顺序。任务的优先级决定了在发生任务切换时即将要执行的任务。在就绪列表中的最高优先级的任务将得到执行。

• 任务入口函数



每个新任务得到调度后将执行的函数。该函数由用户实现，在任务创建时，通过任务创建结构体指定。

- **任务控制块TCB**

每一个任务都含有一个任务控制块(TCB)。TCB包含了任务上下文栈指针 (stack pointer)、任务状态、任务优先级、任务ID、任务名、任务栈大小等信息。TCB可以反映出每个任务运行情况。

- **任务栈**

每一个任务都拥有一个独立的栈空间，我们称为任务栈。栈空间里保存的信息包含局部变量、寄存器、函数参数、函数返回地址等。任务在任务切换时会切出任务的上下文信息保存在自身的任务栈空间里面，以便任务恢复时还原现场，从而在任务恢复后在切出点继续开始执行。

- **任务上下文**

任务在运行过程中使用到的一些资源，如寄存器等，我们称为任务上下文。当这个任务挂起时，其他任务继续执行，在任务恢复后，如果没有把任务上下文保存下来，有可能任务切换会修改寄存器中的值，从而导致未知错误。因此，在任务挂起的时候会将本任务的任务上下文信息，保存在自己的任务栈里面，以便任务恢复后，从栈空间中恢复挂起时的上下文信息，从而继续执行被挂起时被打断的代码。

- **任务切换**

任务切换包含获取就绪列表中最高优先级任务、切出任务上下文保存、切入任务上下文恢复等动作。

运行机制

系统任务管理模块提供如下功能：

- 任务创建
- 任务延时
- 任务挂起和任务恢复
- 锁任务调度和解锁任务调度
- 根据ID查询任务控制块信息

用户创建任务时，系统会将任务栈进行初始化，预置上下文。此外，系统还会将“任务入口函数”地址放在相应位置。这样在任务第一次启动进入运行态时，将会执行“任务入口函数”。

2.2.2 开发流程

使用场景

任务创建后，内核可以执行锁任务调度，解锁任务调度，挂起，恢复，延时等操作，同时也可以设置任务优先级，获取任务优先级。任务结束的时候，如果任务的状态是自删除状态 (LOS_TASK_STATUS_DETACHED)，则进行当前任务自删除操作。

用户的代码需实现app_main接口。系统初始化阶段会调用app_main接口，用户的初始化操作可在app_main中完成。如果用户需要多个任务，可在app_main中创建新任务。建议用户使用任务优先级范围是[10,30]。应用级任务建议使用低于系统级任务的优先级。



功能

系统中的任务管理模块为用户提供的功能如表2-2所示。

表 2-2 系统任务管理模块接口描述

接口名称	描述
hi_task_create	创建任务。
hi_task_delete	删除指定的任务
hi_task_suspend	挂起指定任务。
hi_task_resume	恢复挂起指定任务。
hi_task_get_priority	获取任务优先级。
hi_task_set_priority	设置任务优先级。
hi_task_get_info	获取任务信息。
hi_task_get_current_id	获取当前任务ID。
hi_task_lock	禁止系统任务调度。
hi_task_unlock	允许系统任务调度。
hi_sleep	任务睡眠。
hi_task_register_idle_callback	注册Idle任务回调函数。
hi_task_join	等待任务执行结束。

开发流程

以创建任务为例，创建任务的开发流程：

- 步骤1 在system_config.h中配置任务数。

配置LOSCFG_BASE_CORE_TSK_LIMIT_CONFIG系统支持最大任务数需要根据用户需求配置。
- 步骤2 锁任务hi_task_lock，锁住任务，防止高优先级任务调度。
- 步骤3 创建任务hi_task_create。
- 步骤4 解锁任务hi_task_unlock，让任务按照优先级进行调度。
- 步骤5 挂起指定的任务hi_task_suspend，任务挂起等待恢复操作。
- 步骤6 恢复挂起的任务hi_task_resume。
- 结束



错误码

表 2-3 任务错误码说明

序号	定义	实际值	描述	参考解决方案
1	HI_ERR_TASK_INVALID_PARAM	0x80000080	无效入参	检查任务参数
2	HI_ERR_TASK_CREATE_FAIL	0x80000081	任务创建失败	任务个数达到上限，需修改任务个数
3	HI_ERR_TASK_DELETE_FAIL	0x80000082	任务删除失败	检查资源未释放的情况
4	HI_ERR_TASK_SUPPEND_FAIL	0x80000083	任务挂起失败	<ul style="list-style-type: none"> 检查任务ID是否有效 检查任务是否已经被挂起
5	HI_ERR_TASK_RESUME_FAIL	0x80000084	任务恢复失败	<ul style="list-style-type: none"> 检查任务ID是否有效 检查任务状态是否未被挂起
6	HI_ERR_TASK_GET_PRIORITY_FAIL	0x80000085	任务获取优先级失败	<ul style="list-style-type: none"> 检查任务ID是否无效
7	HI_ERR_TASK_SET_PRIORITY_FAIL	0x80000086	任务设置优先级失败	<ul style="list-style-type: none"> 检查优先级是否有效 检查任务是否是IDLE任务，IDLE任务不可设置 检查任务ID是否有效
8	HI_ERR_TASK_LOCK_FAIL	0x80000087	任务锁定失败	检查参数
9	HI_ERR_TASK_UNLOCK_FAIL	0x80000088	解锁任务失败	<ul style="list-style-type: none"> 检查任务ID是否有效 检查任务状态是否未被锁定
10	HI_ERR_TASK_DELAY_FAIL	0x80000089	任务延时失败	检查参数
11	HI_ERR_TASK_GET_INFO_FAIL	0x8000008A	获取任务信息失败	<ul style="list-style-type: none"> 检查任务ID是否有效 检查参数
12	HI_ERR_TASK_REGISTER_SCHEDULE_FAIL	0x8000008B	注册任务调度失败	<ul style="list-style-type: none"> 检查任务ID是否有效 检查参数
13	HI_ERR_TASK_NOT_CREATED	0x8000008C	任务未创建	检查任务ID是否有效



2.2.3 注意事项

- 创建新任务时，会对之前已删除任务的任务控制块和任务栈进行回收。
- 任务名指针没有分配空间，在设置任务名时，禁止将局部变量的地址赋值给任务名指针。
- 若指定的任务栈大小为0，则使用配置默认的任务栈大小。
- 任务栈的大小按8字节大小对齐。确定任务栈大小的原则为够用即可：多则浪费，少则任务栈溢出。
- 当前任务且已锁任务，不能被挂起。
- Idle任务及软件定时器任务不能被挂起或删除。
- 锁任务调度，并不关中断，因此任务仍可被中断打断。
- 锁任务调度必须和解锁任务调度配合使用。
- 设置任务优先级时可能会发生任务调度。
- 系统可配置的任务资源个数是指整个系统的任务资源总个数，而非用户能使用的任务资源个数。例如：系统软件定时器多占用一个任务资源数，则系统可配置的任务资源就会减少一个。
- 不建议使用hi_task_set_priority接口来修改软件定时器任务的优先级，否则可能会导致系统出现问题。
- hi_task_set_priority接口不能在中断中使用。
- hi_task_get_priority接口传入的task ID对应的任务未创建或者超过最大任务数，统一返回0xFFFF。
- 在删除任务时要保证任务申请的资源（如互斥锁、信号量等）已被释放。
- 尽量少创建task，可采用内存池方案避免内存碎片化。

2.2.4 编程实例

下面的示例介绍任务的基本操作方法包含：任务创建接口的应用。

代码示例

```
#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");
    }
}
```



```
}  
}
```

结果验证

Example task is running!

2.3 内存管理

2.3.1 概述

内存管理模块管理系统的内存资源，通过对内存的申请/释放操作来管理用户和OS对内存的使用，使内存的利用率和效率最优，最大限度地解决系统的内存碎片问题。其中，OS的内存管理为动态内存管理，提供内存初始化、分配、释放等功能。

动态内存是指在动态内存池中分配用户指定大小的内存块。

- 优点：按需分配。
- 缺点：内存池中可能出现碎片。

2.3.2 开发流程

使用场景

内存管理的主要工作是动态的划分并管理用户分配好的内存区间。动态内存管理主要是在用户需要使用大小不等的内存块的场景中使用。当用户需要分配内存时，可以通过操作系统的动态内存申请函数索取指定大小内存块，一旦使用完毕，通过动态内存释放函数归还所占用内存，使之可以重复使用。

功能

动态内存管理模块提供的接口如表2-4所示。

表 2-4 动态内存管理接口描述

接口名称	描述
hi_malloc	从指定动态内存池中申请size长度的内存。
hi_free	释放已申请的内存。
hi_mem_get_sys_info	获取指定内存池的内存信息。
hi_mem_get_sys_info_crash	获取内存信息，死机流程中使用。



错误码

表 2-5 内存管理错误码说明

序号	定义	实际数值	描述	参考解决方案
1	HI_ERR_MEM_INVALID_PARAM	0x80000100	入参错误	检查入参
2	HI_ERR_MEM_CREATE_POOL_FAIL	0x80000101	创建内存池失败	系统内部使用
3	HI_ERR_MEM_CREATE_POOL_NOT_ENOUGH_HANDLE	0x80000102	创建内存池失败	系统内部使用
4	HI_ERR_MEM_FREE_FAIL	0x80000103	内存释放失败	<ul style="list-style-type: none"> 检查模块ID 检查被释放的指针情况
5	HI_ERR_MEM_RE_INIT	0x80000104	内存重复初始化	系统内部使用
6	HI_ERR_MEM_NOT_INIT	0x80000105	内存未初始化	检查是否初始化了内存
7	HI_ERR_MEM_CREATE_POOL_MALLOC_FAIL	0x80000106	创建pool_malloc失败	系统内部使用
8	HI_ERR_MEM_GET_INFO_FAIL	0x80000107	内存使用信息获取失败	检查入参
9	HI_ERR_MEM_GET_OSS_INFO_NOK	0x80000108	获取系统信息失败	检查内存使用情况

2.3.3 注意事项

- 系统中hi_malloc函数如果分配成功，返回分配的空间。如果分配失败，则返回NULL。
- 系统中多次调用hi_free时，第一次会返回成功，但对同一块内存进行多次重复释放会导致非法指针操作，结果不可预知。
- 可通过hi_mem_get_sys_info，查看模块内存使用情况。

2.3.4 编程实例

本实例演示APP层内存申请以及释放操作。

代码示例

```
#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);
}
```

结果验证

Using memory as expected!

2.4 中断机制

2.4.1 概述

中断是指CPU暂停执行当前程序，转而执行新程序的过程。中断相关的硬件可以划分为3类：

- 设备：发起中断的源，当设备需要请求CPU时，产生一个中断信号，该信号连接至中断控制器。
- 中断控制器：接收中断输入并上报给CPU。可以设置中断源的优先级、触发方式、打开和关闭等操作。
- CPU：判断和执行中断任务。

中断相关的名词解释：

- 中断号：每个中断请求信号都会有特定的标志，使得计算机能够判断是哪个设备提出的中断请求，这个标志就是中断号。
- 中断请求：“紧急事件”需向CPU提出申请（发一个电脉冲信号），要求中断，及要求CPU暂停当前执行的任务，转而处理该“紧急事件”，这一申请过程称为中断申请。
- 中断优先级：为使系统能够及时响应并处理所有中断，系统根据中断事件的重要性和紧迫程度，将中断源分为若干个级别，称作中断优先级。系统中所有的中断源优先级相同，不支持中断嵌套或抢占。
- 中断处理程序：当外设产生中断请求后，CPU暂停当前的任务，转而响应中断申请，即执行中断处理程序。
- 中断触发：中断源发出并送给CPU控制信号，将接口卡上的中断触发器置“1”，表明该中断源产生了中断，要求CPU去响应该中断，CPU暂停当前任务，执行相应的中断处理程序。
- 中断触发类型：外部中断申请通过一个物理信号发送到CPU，可以是电平触发或边沿触发。
- 中断向量：中断服务程序的入口地址。
- 中断向量表：存储中断向量的存储区，中断向量与中断号对应，中断向量在中断向量表中按照中断号顺序存储。



2.4.2 开发流程

使用场景

当有中断请求产生时，CPU暂停当前的任务，转而去响应外设请求。根据需要，用户通过中断申请，注册中断处理程序，可以指定CPU响应中断请求时所执行的具体操作。

功能

系统支持的中断机制接口如表2-6所示。

表 2-6 中断机制接口描述

接口名称	描述
hi_int_lock	关闭全部中断。 关中断后不能执行引起调度的函数，如hi_sleep或其他阻塞接口。 关中断仅保护可预期的短时间的操作，否则影响中断响应，可能引起性能问题。
hi_int_restore	恢复关中断前的状态。 入参必须是与之对应的关中断时保存的关中断之前的CPSR的值。
hi_is_int_context	检查是否在中断上下文中。
hi_irq_enable	使能指定中断。
hi_irq_disable	去使能指定中断。
hi_irq_request	注册中断。
hi_irq_free	清除注册中断。

错误码

表 2-7 中断机制错误码说明

序号	定义	实际数值	描述	参考解决方案
1	HI_ERR_ISR_INVALID_PARAM	0x800000C0	入参错误	检查入参
2	HI_ERR_ISR_REQ_IRQ_FAIL	0x800000C1	注册中断失败	<ul style="list-style-type: none">检查入参检查中断数量是否已达已上限，减少中断注册数量



序号	定义	实际数值	描述	参考解决方案
3	HI_ERR_ISR_ADD_JOB_MALLOC_FAIL	0x800000C2	添加任务分配内存失败	检查入参
4	HI_ERR_ISR_ADD_JOB_SYS_FAIL	0x800000C3	添加任务时系统错误	检查入参
5	HI_ERR_ISR_DEL_IRQ_FAIL	0x800000C4	删除中断失败	检查入参
6	HI_ERR_ISR_ALREADY_CREATED	0x800000C5	注册中断失败，中断已注册	检查是否存在重复注册中断
7	HI_ERR_ISR_NOT_CREATED	0x800000C6	中断未注册	检查中断是否已注册
8	HI_ERR_ISR_ENABLE_IRQ_FAIL	0x800000C7	使能中断失败	检查入参
9	HI_ERR_ISR_IRQ_ADD_R_NOK	0x800000C8	中断地址错误	检查注册中断标记位

2.4.3 注意事项

- 根据具体硬件，配置支持的最大中断数及中断初始化操作的寄存器地址。
- 中断处理程序耗时不能过长，影响CPU对中断的及时响应。
- 中断响应过程中不能执行引起任务调度的函数。
- 中断恢复hi_int_restore()的入参必须是与之对应的hi_int_lock()保存的关中断之前的CPSR的值。
- 不能使用mutex、malloc、sleep、delay函数，代码须尽量短小、运行快速，对于较复杂的操作需通过抛事件给中断下半部处理。

2.4.4 编程实例

本实例实现如下功能：

- 关闭全部中断
- 中断使能
- 中断去使能
- 恢复关闭中断前的状态

代码示例

```
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 uvlntSave;  
uvlntSave = hi_int_lock();  
hi_irq_enable(irq_idx);  
hi_irq_disable(irq_idx);  
hi_int_restore(uvlntSave);  
}
```

2.5 队列

2.5.1 概述

队列又称消息队列，是一种常用于任务间通信的数据结构，实现了接收来自任务或中断的不固定长度的消息，接收方根据消息ID读取消息。

任务能够从队列里面读取消息：

- 当队列中的消息是空时，挂起读取任务。
- 当队列中有新消息时，挂起的读取任务被唤醒并处理新消息。
- 用户在处理业务时，消息队列提供了异步处理机制，允许将一个消息放入队列，但并不立即处理它，同时队列还能起到缓冲消息作用。

系统中使用队列数据结构实现任务异步通信工作，具有如下特性：

- 消息以先进先出方式排队，支持异步读写工作方式。
- 读队列和写队列都支持超时机制。
- 发送消息类型由通信双方约定，可以允许不同长度（不超过队列节点最大值）消息。
- 一个任务能够从任意一个消息队列接收和发送消息。
- 多个任务能够从同一个消息队列接收和发送消息。
- 当队列使用结束后，如果是动态申请的内存，需要通过释放内存函数回收。

2.5.2 开发流程

使用场景

多任务间通信，可通过消息队列完成。

功能

消息队列提供的接口如表2-8所示。

表 2-8 队列接口描述

接口名称	描述
hi_msg_queue_create	创建消息队列。
hi_msg_queue_delete	删除消息队列。
hi_msg_queue_send	发送消息。



接口名称	描述
hi_msg_queue_wait	接收消息。
hi_msg_queue_is_full	检查消息队列是否已满。
hi_msg_queue_get_msg_num	获取当前已经使用的消息队列个数。
hi_msg_queue_get_msg_total	获取消息队列总个数。
hi_msg_queue_send_msg_to_front	发送消息到队列头。

开发流程

使用队列模块的典型流程：

步骤1 创建消息队列hi_msg_queue_create。创建成功后，可以得到消息队列的ID值。

步骤2 发送消息（hi_msg_queue_send）。

步骤3 消息等待接收（hi_msg_queue_wait）。

步骤4 队列状态管理（hi_msg_queue_is_full、hi_msg_queue_get_msg_num、hi_msg_queue_get_msg_total）。

步骤5 删除队列hi_msg_queue_delete。

----结束

错误码

对队列存在失败可能性的操作包括创建队列、删除队列等，均需要返回对应的错误码，以便快速定位错误原因。

表 2-9 队列错误码说明

序号	定义	实际数值	描述	参考解决方案
1	HI_ERR_MSG_INVALID_PARAM	0x80000200	入参错误	检查入参
2	HI_ERR_MSG_CREATE_Q_FAIL	0x80000201	队列创建失败	<ul style="list-style-type: none"> 检查入参 检查队列上限
3	HI_ERR_MSG_DELETE_Q_FAIL	0x80000202	队列删除失败	<ul style="list-style-type: none"> 检查入参 检查队列是否存在
4	HI_ERR_MSG_WAIT_FAIL	0x80000203	接收消息失败	检查入参



序号	定义	实际数值	描述	参考解决方案
5	HI_ERR_MSG_SEND_FAIL	0x80000204	发送失败	检查入参
6	HI_ERR_MSG_GET_Q_INFO_FAIL	0x80000205	获取队列状态等信息失败	检查入参
7	HI_ERR_MSG_Q_DELETE_FAIL	0x80000206	队列删除失败	<ul style="list-style-type: none">• 检查入参• 检查队列是否存在• 检查队列是否被任务占用
8	HI_ERR_MSG_WAIT_TIME_OUT	0x80000207	接收消息超时	<ul style="list-style-type: none">• 检查发送方是否发出信息，接收队列配置是否正确• 适当增加超时时间

2.5.3 注意事项

- 等待消息：在中断、关中断、锁任务上下文禁止调用等待消息接口，进而产生不可控的异常调度。
- 发送消息：在关中断上下文禁止调用发送消息接口，进而产生不可控的异常调度。
- 发送消息（超时时间非0）：在中断、锁任务上下文禁止调用超时时间非0发送消息接口，进而产生不可控的异常调度。
- 系统可配置的队列资源个数是指整个系统的队列资源总个数，而非用户能使用的个数。例如：系统软件定时器多占用一个队列资源，那么系统可配置的队列资源就会减少一个。
- 队列接口函数中的入参timeout是指相对时间。

2.5.4 编程实例

创建一个队列，两个任务：

- 任务1调用发送接口发送消息
- 任务2通过接收接口接收消息

步骤如下：

- 步骤1** 通过hi_task_create创建任务1和任务2。
- 步骤2** 通过hi_msg_queue_create创建一个消息队列。
- 步骤3** 在任务1 hi_msg_queue_send中发送消息。
- 步骤4** 在任务2 hi_msg_queue_wait中接收消息。



步骤5 通过hi_msg_queue_delete删除队列。

----结束

代码示例

```
static hi_u32 g_msg_queue;
hi_u8 abuf[] = "test is message x";
/*任务1发送数据*/
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++;
        /*将abuf里的数据写入队列*/
        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;
}

/*任务2接收数据*/
hi_void *example_rcv_task(hi_void *arg)
{
    hi_u8 msg[50];
    hi_u32 ret = 0;
    hi_unref_param(arg);
    while (1) {
        /*读取队列里的数据存入msg里*/
        ret = hi_msg_queue_wait(g_msg_queue, &msg, HI_SYS_WAIT_FOREVER, sizeof(msg));
        if (ret != HI_ERR_SUCCESS) {
            dprintf("rcv message failure,error:%x\n", ret);
            break;
        }
        dprintf("rcv message:%s\n", (char *)msg);
        hi_sleep(5);
    }
    /*删除队列。需根据具体情况删除，大多数情况下无需删除队列，且在有任务占用等情况下删除队
    列会导致失败。以下代码仅供API展示*/
    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;

    /*创建任务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;
}
/*创建任务2*/
ret = hi_task_create(&task_resv_msg, &task_param1, example_rcv_task, HI_NULL);
if(ret != HI_ERR_SUCCESS) {
    dprintf("create task2 failed!,error:%x\n",ret);
    return ret;
}
/*创建队列*/
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;
}
```

结果验证

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

2.6 事件

2.6.1 概述

事件是一种任务间通信的机制，可用于实现任务间的同步。一个任务可以等待多个事件的发生：

- 任意一个事件发生时唤醒任务进行事件处理。
- 几个事件都发生后才唤醒任务进行事件处理。

多任务环境下，任务之间往往需要同步操作，一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。

- 一对多同步模型：一个任务等待多个事件的触发。
- 多对多同步模型：多个任务等待多个事件的触发。

任务可以通过创建事件控制块来实现对事件的触发和等待操作。

事件接口具有如下特点：

- 事件不与任务相关联，事件相互独立，内部实现为一个32位的无符号整型变量，用于标识该任务发生的事件类型，其中每一位表示一种事件类型：
 - 0：该事件类型未发生。
 - 1：该事件类型已经发生。
- 事件仅用于任务间的同步，不提供数据传输功能。



- 多次向任务发送同一事件类型等效于只发送一次。
- 多个任务可以对同一事件进行读写操作。
- 支持事件读写超时机制。

在读事件时，可以选择读取模式。读取模式如下：

- 所有事件（HI_EVENT_WAITMODE_AND）：读取掩码中所有事件类型，只有读取的所有事件类型都发生了，才能读取成功。
- 任一事件（HI_EVENT_WAITMODE_OR）：读取掩码中任一事件类型，读取的事件中任意一种事件类型发生了，就可以读取成功。
- 清除事件（HI_EVENT_WAITMODE_CLR）：这是一种附加读取模式，可以与HI_EVENT_WAITMODE_AND和HI_EVENT_WAITMODE_OR结合使用（HI_EVENT_WAITMODE_AND| HI_EVENT_WAITMODE_CLR或HI_EVENT_WAITMODE_OR | HI_EVENT_WAITMODE_CLR），设置该模式读取成功后，对应事件类型位会自动清除。

运行机制：

读事件时，可以根据入参事件掩码类型uwEventMask读取事件的单个或多个事件类型。事件读取成功后，如果设置HI_EVENT_WAITMODE_CLR会清除已读取到的事件类型，反之不会清除已读到的事件类型，需显式清除。可以通过入参选择读取模式，读取事件掩码类型中所有事件还是读取事件掩码类型中任意事件。

- 写事件时，对指定事件写入指定的事件类型，可以一次同时写多个事件类型。写事件会触发任务调度。
- 清除事件时，根据入参事件和待清除的事件类型，对事件对应位进行清0操作。

2.6.2 开发流程

使用场景

事件可应用于多种任务同步场景，在某些同步场景下可替代信号量。

功能

系统中的事件模块为用户提供的接口如表2-10所示。

表 2-10 事件接口描述

接口名称	描述
hi_event_init	初始化一个事件控制块。
hi_event_create	获取一个事件ID。
hi_event_wait	读取指定事件类型，等待超时时间为相对时间，单位：ms。
hi_event_send	写指定的事件类型。
hi_event_clear	清除指定的事件类型。
hi_event_delete	销毁指定的事件控制块。



开发流程

使用事件模块的典型流程：

- 步骤1 调用事件初始化hi_event_init接口，初始化事件等待队列。
- 步骤2 调用hi_event_create接口，获取事件ID。
- 步骤3 写事件hi_event_send，配置事件掩码类型。
- 步骤4 读事件hi_event_wait，选择读取模式。
- 步骤5 清除事件hi_event_clear，清除指定的事件类型。

----结束

错误码

对事件存在失败的可能性操作包括：事件初始化、事件销毁、事件读写、事件清除。

表 2-11 事件错误码说明

序号	定义	实际值	描述	参考解决方案
1	HI_ERR_EVENT_INVALID_PARAM	0x80000240	无效的参数	检查入参
2	HI_ERR_EVENT_CREATE_NO_HADNLE	0x80000241	无更多的句柄可分配	增加事件句柄数的上限
3	HI_ERR_EVENT_CREATE_SYS_FAIL	0x80000242	系统事件控制块为空指针	检查系统初始化参数
4	HI_ERR_EVENT_SEND_FAIL	0x80000243	系统事件控制块为空指针	检查系统初始化参数
5	HI_ERR_EVENT_WAIT_FAIL	0x80000244	系统事件控制块为空指针	检查系统初始化参数
6	HI_ERR_EVENT_CLEAR_FAIL	0x80000245	暂未使用，待扩展	无
7	HI_ERR_EVENT_RE_INIT	0x80000246	重复调用事件初始化	不要重复调用hi_event_create初始化
8	HI_ERR_EVENT_NOT_ENOUGH_MEMORY	0x80000247	内存不足	查看内存资源使用情况，并释放相应的内存资源
9	HI_ERR_EVENT_NOT_INIT	0x80000248	事件没有初始化	先初始化事件控制块
10	HI_ERR_EVENT_DELETE_FAIL	0x80000249	事件销毁失败	检查事件链表是否为空



序号	定义	实际值	描述	参考解决方案
11	HI_ERR_EVENT_WAIT_TIME_OUT	0x8000024A	读超时	将超时时间设置在合理的范围内

2.6.3 注意事项

- 在系统初始化之前不能调用读写事件接口。如果调用，则系统运行会不正常。
- 在中断中，可以对事件对象进行写操作，但不能进行读操作。
- 在锁任务调度状态下，禁止任务阻塞与读事件。
- hi_event_clear入参值是要清除的指定事件类型的反码（~event_bits）。
- 事件掩码支持bit[0] ~ bit[23]，bit[24] ~ bit[31]不支持。

2.6.4 编程实例

本示例中，任务example_task_entry_event 创建一个任务example_event，example_event读事件阻塞，example_task_entry_event 向该任务写事件。

- 步骤1** 在任务example_task_entry_event 创建任务example_event，其中任务example_event优先级高于example_task_entry_event 。
- 步骤2** 在任务example_event中读事件0x00000001，阻塞，发生任务切换，执行任务example_task_entry_event。
- 步骤3** 在任务example_task_entry_event 向任务Example_Event写事件0x00000001，发生任务切换，执行任务example_event。
- 步骤4** example_event得以执行，直到任务结束。
- 步骤5** example_task_entry_event 得以执行，直到任务结束。

----结束

代码示例

```
#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);
    /*超时 等待方式读事件,超时时间为200ms 若200ms 后未读取到指定事件，读事件超时，任务直接唤醒*/
    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};  
    /*事件初始化*/  
    hi_event_init(4, HI_NULL); /* 4:设置最大事件数为4 */  
    ret = hi_event_create(&g_event_id);  
    if (ret != HI_ERR_SUCCESS) {  
        printf("init event failed .\n");  
        return HI_ERR_FAILURE;  
    }  
    /*创建任务*/  
    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;  
    }  
    /*写用例任务等待的事件类型*/  
    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");  
    /*清标志位*/  
    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");  
    /*删除任务*/  
    hi_task_delete(taskid);  
    return HI_ERR_SUCCESS;  
}
```

结果验证

```
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 互斥锁



2.7.1 概述

互斥锁又称互斥型信号量，是一种特殊的二值性信号量，用于实现对共享资源的独占式处理。任意时刻互斥锁的状态只有两种：

- 闭锁：当有任务持有时，互斥锁处于闭锁状态，这个任务获得该互斥锁的所有权。
- 开锁：当该任务释放它时，该互斥锁被开锁，任务失去该互斥锁的所有权。

当一个任务持有互斥锁时，其他任务将不能再对该互斥锁进行开锁或持有。多任务环境下往往存在多个任务竞争同一共享资源的应用场景，互斥锁可被用于对共享资源的保护从而实现独占式访问。另外，互斥锁可以解决信号量存在的优先级翻转问题。

互斥锁接口具有如下特点：

- 通过优先级继承算法，解决优先级翻转问题。

2.7.2 开发流程

使用场景

互斥锁可以提供任务之间的互斥机制，用来防止两个任务在同一时刻访问相同的共享资源。

功能

系统中的互斥锁模块为用户提供的功能如表2-12所示。

表 2-12 互斥锁接口描述

接口名称	描述
hi_mux_create	创建互斥锁。
hi_mux_delete	删除指定的互斥锁。
hi_mux_pend	申请指定的互斥锁。
hi_mux_post	释放指定的互斥锁。

开发流程

互斥锁典型场景的开发流程：

步骤1 创建互斥锁hi_mux_create。

步骤2 申请互斥锁hi_mux_pend。

申请模式有3种：

- 无阻塞模式：任务需要申请互斥锁，若该互斥锁当前没有任务持有，或者持有该互斥锁的任务和申请该互斥锁的任务为同一个任务，则申请成功，超时时间设置为0。



- 永久阻塞模式：任务需要申请互斥锁，若该互斥锁当前没有被占用，则申请成功。否则，该任务进入阻塞态，系统切换到就绪任务中优先级高者继续执行。任务进入阻塞态后，直到有其他任务释放该互斥锁，阻塞任务才会重新得以执行，超时时间设置为HI_SYS_WAIT_FOREVER。
- 定时阻塞模式：任务需要申请互斥锁，若该互斥锁当前没有被占用，则申请成功。否则该任务进入阻塞态，系统切换到就绪任务中优先级高者继续执行。任务进入阻塞态后，指定时间超时前有其他任务释放该互斥锁，或者用户指定时间超时后，阻塞任务才会重新得以执行，超时时间设置为一个合理的超时值。

步骤3 释放互斥锁hi_mux_post。

- 如果有任务阻塞于指定互斥锁，则唤醒被阻塞任务中优先级高的，该任务进入就绪态，并进行任务调度。
- 如果没有任务阻塞于指定互斥锁，则互斥锁释放成功。

步骤4 删除互斥锁hi_mux_delete。

----结束

错误码

对互斥锁存在失败的可能性操作包括：互斥锁创建、互斥锁删除、互斥锁申请、互斥锁释放。

表 2-13 互斥锁错误码说明

序号	定义	实际值	描述	参考解决方案
1	HI_ERR_MUX_INVALID_PARAM	0x800001C0	无效的参数	检查函数入参
2	HI_ERR_MUX_CREATE_FAIL	0x800001C1	创建互斥锁失败	增加互斥锁限制数量的上限
3	HI_ERR_MUX_DELETE_FAIL	0x800001C2	互斥锁删除失败	还有任务在互斥锁链表中等待申请互斥锁，等所有任务解锁再删除
4	HI_ERR_MUX_PENDING_FAIL	0x800001C3	互斥锁PENDING超时	增加等待时间或者设置一直等待模式
5	HI_ERR_MUX_POST_FAIL	0x800001C4	互斥锁释放失败	当前任务不持有锁或锁已经被释放

2.7.3 注意事项

- 两个任务不能对同一把互斥锁加锁。如果某任务对已被持有的互斥锁加锁，则该任务会被挂起，直到持有该锁的任务对互斥锁解锁，才能执行对这把互斥锁的加锁操作。
- 互斥锁不能在中断服务程序中使用。
- 作为实时操作系统需要保证任务调度的实时性，尽量避免任务的长时间阻塞，因此在获得互斥锁之后，应该尽快释放互斥锁。



- 持有互斥锁的过程中，不得再调用hi_task_set_priority等接口更改持有互斥锁任务的优先级。

2.7.4 编程实例

本实例实现如下流程：

- 步骤1** 任务example_task_entry_mux创建一个互斥锁，锁任务调度，创建两个任务example_mutex_task1、example_mutex_task2，example_mutex_task2优先级高于example_mutex_task1，解锁任务调度。
- 步骤2** example_mutex_task2被调度，永久申请互斥锁，然后任务休眠100ms，example_mutex_task2挂起，example_mutex_task1被唤醒。
- 步骤3** example_mutex_task1申请互斥锁，等待时间为10ms，因互斥锁仍被example_mutex_task2持有，example_mutex_task1挂起，10ms后未拿到互斥锁，example_mutex_task1被唤醒，试图以永久等待申请互斥锁，example_mutex_task1挂起。
- 步骤4** 100ms后example_mutex_task2唤醒，释放互斥锁后，example_mutex_task1被调度运行，后释放互斥锁。
- 步骤5** example_mutex_task1执行完，300ms后任务example_task_entry_mux被调度运行，删除互斥锁。

----结束

代码示例

```
#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;
}
```

结果验证

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 信号量

2.8.1 概述

信号量（Semaphore）是一种实现任务间通信的机制，实现任务之间同步或临界资源的互斥访问。常用于协助一组相互竞争的任务来访问临界资源。

在多任务系统中，各任务之间需要同步或互斥实现临界资源的保护，信号量功能可以为用户提供这方面的支持。通常一个信号量的计数值用于对应有有效的资源数，表示剩下的可被占用的互斥资源数。其值的含义分2种情况：

- 0：没有积累下来的Post操作，且有可能有在此信号量上阻塞的任务。
- 正值：有一个或多个Post下来的释放操作。



以同步为目的的信号量和以互斥为目的的信号量在使用有如下不同：

- 用作同步时，信号量在创建后被置为空，任务1取信号量而阻塞，任务2在某种条件发生后，释放信号量，于是任务1得以进入READY或RUNNING态，从而达到了两个任务间的同步。
- 用作互斥时，信号量创建后记数是满的，在需要使用临界资源时，先取信号量，使其变空，这样其他任务需要使用临界资源时就会因为无法取到信号量而阻塞，从而保证了临界资源的安全。

信号量运作原理：

- 信号量初始化：为配置的N个信号量申请内存（N值可以由用户自行配置，受内存限制），并把所有的信号量初始化成未使用，并加入到未使用链表中供系统使用。
- 信号量创建：从未使用的信号量链表中获取一个信号量资源，并设定初值。
- 信号量申请：如果其计数器值 >0 ，则直接减1返回成功。否则任务阻塞，等待其它任务释放该信号量，等待的超时时间可设定。当任务被一个信号量阻塞时，将该任务挂到信号量等待任务队列的队尾。信号量释放，如果没有任务等待该信号量，则直接将计数器加1返回。否则唤醒该信号量等待任务队列上的第一个任务。
- 信号量删除：将正在使用的信号量置为未使用信号量，并挂回到未使用链表。

信号量允许多个任务在同一时刻访问同一资源，但会限制同一时刻访问此资源的大任务数目。访问同一资源的任务数达到该资源的最大数量时，会阻塞其他试图获取该资源的任务，直到有任务释放该信号量。

2.8.2 开发流程

使用场景

信号量是一种非常灵活的同步方式，可以运用在多种场合中，实现锁、同步、资源计数等功能，也能方便用于任务与任务、中断与任务的同步中。

功能

系统中的信号量模块为用户提供的功能如表2-14所示。

表 2-14 信号量接口描述

接口描述	描述
hi_sem_create	创建信号量。
hi_sem_bcreate	创建二进制信号量。
hi_sem_delete	删除指定的信号量。
hi_sem_wait	申请指定的信号量。
hi_sem_signal	释放指定的信号量。

开发流程

信号量的开发典型流程：



步骤1 创建信号量hi_sem_create。

步骤2 申请信号量hi_sem_wait。

信号量有3种申请模式：

- 无阻塞模式：任务需要申请信号量，若当前信号量的任务数没有到信号量设定的上限，则申请成功。否则，立即返回申请失败。超时时间设置为0。
- 永久阻塞模式：任务需要申请信号量，若当前信号量的任务数没有到信号量设定的上限，则申请成功。否则，该任务进入阻塞态，系统切换到就绪任务中优先级高者继续执行。任务进入阻塞态后，直到有其他任务释放该信号量，阻塞任务才会重新得以执行。超时时间设置为HI_SYS_WAIT_FOREVER。
- 定时阻塞模式：任务需要申请信号量，若当前信号量的任务数没有到信号量设定的上限，则申请成功。否则，该任务进入阻塞态，系统切换到就绪任务中优先级高者继续执行。任务进入阻塞态后，指定时间超时前有其他任务释放该信号量，或者用户指定时间超时后，阻塞任务才会重新得以执行。超时时间设置为合理的值。

步骤3 释放信号量hi_sem_signal。

- 如果有任务阻塞于指定信号量，则唤醒该信号量阻塞队列上的第一个任务。该任务进入就绪态，并进行调度。
- 如果没有任务阻塞于指定信号量，释放信号量成功。

步骤4 删除信号量hi_sem_delete。

----结束

错误码

对可能导致信号量操作失败的情况包括：创建信号量、申请信号量、释放信号量、删除信号量等，均需要返回对应的错误码，以便快速定位错误原因。

表 2-15 信号量错误码说明

序号	定义	实际值	描述	参考解决方案
1	HI_ERR_SEM_INVALID_PARAM	0x80000180	无效的参数	检查函数入参
2	HI_ERR_SEM_CREATE_FAIL	0x80000181	创建信号量失败	释放信号量资源
3	HI_ERR_SEM_DELETE_FAIL	0x80000182	删除信号量失败	唤醒所有等待该信号量的任务后删除该信号
4	HI_ERR_SEM_WAIT_FAIL	0x80000183	定时时间非法或传入的信号量ID异常	排查代码异常
5	HI_ERR_SEM_SIG_FAIL	0x80000184	超过最大的信号量计数	无任务等待此信号量
6	HI_ERR_SEM_WAIT_TIMEOUT	0x80000185	获取信号量时间超时	设置等待时间在合理范围内



2.8.3 注意事项

由于中断不能被阻塞，因此在申请信号量时，阻塞模式不能在中断中使用。

2.8.4 编程实例

本实例实现如下功能：

- 步骤1** 测试任务example_task_entry_sem创建一个信号量，锁任务调度，创建两个任务example_sem_task1、example_sem_task2，example_sem_task2优先级高于example_sem_task1，两个任务中申请同一信号量，解锁任务调度后两任务阻塞，测试任务example_task_entry_sem释放信号量。
- 步骤2** example_sem_task2得到信号量，被调度，然后任务休眠200ms，example_sem_task2延迟，example_sem_task1被唤醒。
- 步骤3** example_sem_task1定时阻塞模式申请信号量，等待时间为100ms，因信号量仍被Example_SemTask2持有，example_sem_task1挂起，100ms后仍未得到信号量，example_sem_task1被唤醒，试图以永久阻塞模式申请信号量，example_sem_task1挂起。
- 步骤4** 200ms后example_sem_task2唤醒，释放信号量后，example_sem_task1得到信号量被调度运行，后释放信号量。
- 步骤5** example_sem_task1执行完，400ms后任务example_task_entry_sem被唤醒，执行删除信号量，删除两个任务。

----结束

代码示例

```
#include "hi_sem.h"
#include "hi_task.h"
#include <hi_mdm_types_base.h>
#define UNUSED_PARAM(p) p=p
/*测试任务优先级*/
#define TASK_PRIO_TEST? 21
/*信号量结构体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");
    /*定时阻塞模式申请信号量，定时时间为100ms*/
    ret = hi_sem_wait(g_usSemID, 100);
    /*申请到信号量*/
    if(HI_ERR_SUCCESS == ret) {
        hi_sem_signal(g_usSemID);
    }
    /*定时时间到，未申请到信号量*/
    if (HI_ERR_SEM_WAIT_TIME_OUT == ret) {
        printf("example_sem_task1 timeout and try get sem g_usSemID wait forever.\n");
        /*永久阻塞模式申请信号量*/
        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");  
    /*永久阻塞模式申请信号量*/  
    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");  
    }  
    /*任务休眠200ms*/  
    hi_sleep(200);  
    printf("example_sem_task2 post sem g_usSemID .\n");  
    /*释放信号量*/  
    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};  
    /*创建信号量*/  
    hi_sem_create(&g_usSemID,0);  
    /*锁任务调度*/  
    hi_task_lock();  
    /*创建任务*/  
    /*创建任务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;  
    }  
    /*创建任务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;  
    }  
    /*解锁任务调度*/  
    hi_task_unlock();  
    ret = hi_sem_signal(g_usSemID);  
    /*任务休眠400ms*/  
    hi_sleep(400);  
    /*删除信号量*/  
    hi_sem_delete(g_usSemID);  
    /*删除任务1*/  
    hi_task_delete(taskid1);  
    /*删除任务2*/  
    hi_task_delete(taskid2);  
    return HI_ERR_SUCCESS;  
}
```




结果验证

编译运行得到的结果为：

```
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 时间管理

2.9.1 概述

时间管理以系统时钟为基础，提供给应用程序所有和时间有关的服务，系统中的时间管理模块提供时间转换、统计、延迟功能以满足用户对时间相关需求的实现。

- Cycle系统：最小的计时单位。Cycle的时长由系统主频决定，系统主频就是每秒钟的Cycle数。
- Tick：操作系统的基本时间单位，对应的时长由系统主频及每秒Tick数决定，由用户在 system_config.h中配置OS_SYS_CLOCK_CONFIG和LOSCFG_BASE_CORE_TICK_PER_SECOND_CONFIG。

时间管理接口主要提供的功能如表2-16所示。

表 2-16 时间管理接口描述

接口名称	描述
hi_udelay	等待时间（单位：微秒）。
hi_get_tick	获取当前低位4个字节的Tick数。
hi_get_tick64	获取当前的Tick数。
hi_get_seconds	Tick转换为秒。
hi_get_milli_seconds	Tick转换为毫秒。
hi_get_us	Tick转换为微秒。
hi_get_real_time	获取系统时间，结构体成员中包含秒和纳秒。
hi_set_real_time	设置系统时间（单位：秒）。
hi_tick_register_callback	注册Tick中断响应回调函数。
hi_ms2systick	将毫秒转换为ticks。

2.9.2 开发流程

使用场景

用户需要了解当前系统运行的时间以及Tick与秒、毫秒之间的转换关系等。



开发流程

时间管理的典型开发流程：

步骤1 调用时钟转换接口。

步骤2 获取系统Tick数完成时间统计。

通过hi_get_tick获取系统当前Tick值。

----结束

错误码

无

2.9.3 注意事项

- 获取系统Tick数需要在系统时钟使能之后。
- 系统的Tick数在关中断的情况下不进行计数，故系统Tick数不能作为准确时间计算。

2.9.4 编程实例

无

2.10 软件定时器

2.10.1 概述

软件定时器是基于系统Tick时钟中断且由软件来模拟的定时器，当经过设定的Tick时钟计数值后会触发用户定义的回调函数。定时精度与系统Tick时钟的周期有关。

- 硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，因此为了满足用户需求，提供更多的定时器，系统提供软件定时器功能。
- 软件定时器扩展了定时器的数量，允许创建更多的定时业务。

软件定时器功能支持：

- 软件定时器创建
- 软件定时器启动
- 软件定时器停止
- 软件定时器删除

运作机制：

- 软件定时器使用了系统的一个队列和一个任务资源，先进先出。定时时间短的定时器总是比定时时间长的靠近队列头，满足优先被触发的准则。
- 当Tick中断到来时，在Tick中断处理函数中扫描软件定时器的计时任务，查看是否有定时器超时，如果有，则将超时的定时器记录下来。
- Tick中断处理函数结束后，软件定时器任务（优先级为高）被唤醒，在该任务中调用之前记录下来的定时器的超时回调函数。



软件定时器提供2类定时器机制：

- 单次触发定时器：在启动后只会触发一次定时器事件，然后定时器自动删除。
- 周期触发定时器：会周期性地触发定时器事件，直到用户手动地停止定时器，否则将永远持续执行。

2.10.2 开发流程

使用场景

- 创建一个单次触发的定时器，超时后执行回调函数。
- 创建一个周期性触发的定时器，超时后执行用户自定义的回调函数。

功能

系统中的软件定时器模块为用户提供的功能如表2-17所示。

表 2-17 软件定时器接口描述

接口名称	描述
hi_timer_create	创建定时器。
hi_timer_delete	删除定时器。
hi_timer_start	启动定时器。
hi_timer_stop	停止定时器。

开发流程

软件定时器的典型开发流程：

- 步骤1** 创建定时器hi_timer_create。
- 返回函数运行结果，成功或失败。
- 步骤2** 启动定时器hi_timer_start。
- 步骤3** 停止定时器hi_timer_stop。
- 步骤4** 删除定时器hi_timer_delete。
- 结束

错误码

对软件定时器存在失败可能性的操作包括：创建、删除、暂停、重启定时器等，均需要返回对应的错误码，以便快速定位错误原因。



表 2-18 软件定时器错误码说明

序号	定义	实际值	描述	参考解决方案
1	HI_ERR_TIMER_FAILURE	0x80000140	定时器失败	定时器通用错误
2	HI_ERR_TIMER_INVALID_PARAM	0x80000141	无效的入参	检查函数入参
3	HI_ERR_TIMER_CREATE_HANDLE_FAIL	0x80000142	未使用，待扩展	无
4	HI_ERR_TIMER_START_FAIL	0x80000143	未使用，待扩展	无
5	HI_ERR_TIMER_HANDLE_NOT_CREATE	0x80000144	定时器句柄未创建	创建软件定时器
6	HI_ERR_TIMER_HANDLE_INVALID	0x80000145	无效的句柄	检查函数入参
7	HI_ERR_TIMER_STATUS_INVALID	0x80000146	不正确的软件定时器状态	检查确认软件定时器状态
8	HI_ERR_TIMER_STATUS_START	0x80000147	未使用，待扩展	无
9	HI_ERR_TIMER_INVALID_MODE	0x80000148	无效的定时器启动模式	设置有效的定时器启动模式
10	HI_ERR_TIMER_EXPIRE_INVALID	0x80000149	软件定时器的间隔时间为0	设置正确的定时器间隔时间
11	HI_ERR_TIMER_FUNCTION_NULL	0x8000014a	传入的定时器超时回调函数地址为空指针	传入有效的回调函数地址
12	HI_ERR_TIMER_HANDLE_MAXSIZE	0x8000014b	软件定时器个数超过最大值	重新定义软件定时器最大个数，或者等待一个软件定时器释放资源
13	HI_ERR_TIMER_MALLOC_FAIL	0x8000014c	未使用，待扩展	无
14	HI_ERR_TIMER_NOT_INIT	0x8000014d	未使用，待扩展	无

2.10.3 注意事项

- 软件定时器的回调函数中请勿做过多操作，请勿使用可能引起任务挂起或阻塞的接口或操作。



- 软件定时器使用了系统的一个队列和一个任务资源，软件定时器任务的优先级设定为0，且不允许修改。
- 系统可配置的软件定时器资源个数是指整个系统可使用的软件定时器资源总个数，而并非为用户可使用的软件定时器资源个数。例如：系统软件定时器多占用一个软件定时器资源数，那么用户能使用的软件定时器资源就会减少一个。
- 创建单次软件定时器，该定时器超时执行完回调函数后，系统会自动删除该软件定时器，并回收资源。
- 创建单次不自删除属性的定时器，用户需要调用定时器删除接口删除定时器，回收定时器资源，避免资源泄露。

2.10.4 编程实例

在下面的例子中，演示如下功能：

- 软件定时器创建、启动、删除、暂停操作。
- 单次软件定时器，周期软件定时器使用方法。

代码示例

```
#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)//回调函数1
{
    UNUSED_PARAM(arg);
    g_timercount1++;
    printf("g_timercount1=%d\n",g_timercount1);
}
hi_void timer2_callback(hi_u32 arg)//回调函数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;
    /*创建单次软件定时器，时间为1000ms，启动到1000ms数时执行回调函数1 */
    hi_timer_create(&timer_id1);
    /*创建周期性软件定时器，每100ms数执行回调函数2 */
    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);//延时1200ms数
    hi_timer_delete(timer_id1);//删除软件定时器
    printf("delete Timer1 success\n");
    hi_timer_start(timer_id2, HI_TIMER_TYPE_PERIOD, 100, timer2_callback, 0);//启动周期性软件定时器
    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");  
}
```

结果验证

```
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 分区与 Flash 保护

2.11.1 概述

Flash分区用于在产品开发初期，指导用户根据实际Flash容量等信息，合理规划Flash空间。规划Flash空间时，请与Flash器件支持的保护区间和保护策略合理匹配。Flash分区以Flash的可擦最小单元sector大小为单位对齐，一般Flash均为4KB，所有分区地址为相对Flash地址，即从0地址开始。用户可根据具体的Flash器件的空间，重新分配Flash各个分区的大小。

SPI Flash操作简单、速度快，但通信时如果信号异常，可能会造成Flash内容错误导致严重后果，为了防止Flash数据被篡改，需要尽可能地保护Flash最大的区域。一般SPI Flash芯片均支持Flash保护相关策略，因此建议用户尽可能采取Flash保护策略。用户可根据具体的Flash器件的保护策略，重新规划Flash保护区域。

2.11.2 开发流程

修改 Flash 分区

默认Flash分区如表2-19所示。

表 2-19 默认 Flash 分区表

序号	分区名称	大小 (KB)	地址空间	说明
1	Flash Boot	32	0x0_0000 ~ 0x0_7FFF	起始地址固定。



序号	分区名称	大小 (KB)	地址空间	说明
2	工厂NV	8	0x0_8000 ~ 0x0_9FFF	工厂NV区，存放出厂重要参数，Flash分区表存储在工厂NV中。
3	NV工作区	8	0x0_A000 ~ 0x0_BFFF	-
4	NV工作区原始备份	4	0x0_C000 ~ 0x0_CFFF	此NV区为编译生成Bin文件时的NV工作区原始备份，不支持系统运行中修改。
5	Kernel A	912	0x0_D000 ~ 0x0xF_0FFF	<ul style="list-style-type: none"> “dual-partition ota support”（双分区升级模式），KernelA和KernelB分别存放对应的Kernel+NV文件。 “compression ota support”（压缩升级模式），KernelA和KernelB合成一个区使用，分区头部存放Kernel+NV文件，分区尾部存放压缩升级文件。 产测镜像和Kernel B区域重叠（Kernel B的后600K，即：0x14_D000~0x1E_3000），因此，如果产测镜像在出厂时未被擦除，后续OTA升级也可能将其破坏。 <p>升级方式请参见《Hi3861V100 / Hi3861LV100 升级 开发指南》。</p>
6	Kernel B	968	0xF_1000 ~ 0x1E_2FFF	
7	HILINK	8	0x1E_3000 ~ 0x1E_4FFF	HILINK存储配置信息专用。不使用HILINK或使用其他云的场景，可以参考并规划相应分区。
8	文件系统	44	0x1E_5000 ~ 0x1E_FFFF	-
9	用户保留区	20	0x1F_0000~0x1F_4FFF	用户预留区域，用于存储自定义信息。
10	HILINK PKI	8	0x1F_5000~0x1F_6FFF	HILINK存储PKI证书专用。不使用HILINK或使用其他云的场景，可以参考并规划相应分区。
11	死机信息	4	0x1F_7000 ~ 0x1F_7FFF	-
12	备份Flash Boot	32	0x1F_8000 ~ 0x1F_FFFF	当首部Flash Boot被破坏时，系统将尝试从尾部Flash Boot启动。



分区表结构体如下：

```
typedef struct {  
    hi_u32 addr :24; /**< Flash分区地址，限制为16MB */  
    hi_u32 id :7; /**< Flash区ID */  
    hi_u32 dir :1; /**< Flash区存放方向。0：分区内容正序；1：倒序末地址 CEnd */  
    hi_u32 size :24; /**< Flash分区大小（单位：byte） */  
    hi_u32 reserve :8; /**<保留区 */  
    hi_u32 addition; /**< Flash分区补充信息等 */  
} hi_flash_partition_info;  
  
/**  
 * @ingroup hct_flash_partiton  
 * Flash分区表。  
 */  
typedef struct {  
    hi_flash_partition_info table[HI_FLASH_PARTITON_MAX]; /**< Flash分区表项描述 */  
} hi_flash_partition_table;
```

如果Flash器件与默认Flash分区表不符合，可修改Flash分区表：

- 根据Flash器件修改“tools/nvtool/xml_file/mss_nvi_db.xml”文件，其中工厂NV ID为2即是Flash分区表信息，使用新的分区表信息替换原有的分区表信息。
- 如果工厂NV地址发生改变，需修改“HI_FNV_DEFAULT_ADDR”宏（kernel下位于hi_nv.h中，flashboot下位于hi_flashboot.h中）。
- 如果Kernel启动地址也发生变化，需要同步修改“SConstruct”文件中“signature”的地址，“signature”中“RSA”代表RSA加密的文件，“ECC”代表ECC加密文件，元素1代表kernel A的地址为kernel A的分区表项起始地址加0x3C0，元素2代表kernel B的地址为kernel B的分区表项起始地址加0x3C0；修改“build/scripts/make_upg_file.py”脚本中的分区表信息（例如：Flashboot地址boot_st_addr与大小boot_size、工厂区NV起始地址ftm1_st_addr与大小ftm1_size、NV工作区起始地址nv_file_st_addr与大小nv_file_size、Kernel起始地址kernel_st_addr），并重新编译工程即可完成Flash分区表的替换。
- 建议修改“flash_partition_table.c”与“boot_partition_table.c”中的默认分区地址宏，与新分区表保持一致。

系统中的分区表模块为用户提供的功能如表2-20所示。

表 2-20 Flash 分区表接口描述

接口名称	描述
hi_flash_partition_init	初始化Flash分区表。
hi_get_partition_table	获取Flash分区表。
hi_get_hilink_partition_table	获取HiLink分区地址和大小。
hi_get_hilink_pki_partition_table	获取HiLink PKI分区地址和大小。
hi_get_crash_partition_table	获取死机信息分区地址和大小。
hi_get_fs_partition_table	获取文件系统分区地址和大小。
hi_get_normal_nv_partition_table	获取非工厂区分区地址和大小。



接口名称	描述
hi_get_normal_nv_backup_partition_table	获取非工厂备份区分区地址和大小。
hi_get_usr_partition_table	获取用户保留区分区地址和大小。
hi_get_factory_bin_partition_table	获取产测Bin地址和大小

Flash 保护


SDK交付策略中提供一种Flash保护策略，其要点如下：

- 默认全片保护。
- Flashboot中有Flash擦/写时，解除相应的保护区域（分为全片解保护和低32KB区域保护），便于兼容更多的Flash器件，采用非易失指令。如果无需Flash保护功能，可关闭特性宏“HI_FLASH_SUPPORT_FLASH_PROTECT”，该特性宏位于“hi_flashboot_flash.c”中。
- Kernel中擦/写时，根据地址修改保护范围，运行时，低地址保护、高地址放开保护，并支持设置时间，超时到期后恢复保护；如果无需Flash保护功能，可关闭特性宏“HI_FLASH_SUPPORT_FLASH_PROTECT”，该特性宏位于“flash_prv.h”中。

根据Flash型号不同，策略会有所不同，优先选用支持Write Enable for Volatile Status Register（50h）命令，可以充分提高操作寄存器效率且没有操作次数达到寿命的限制。

不同的Flash器件其保护区域有所差异，更换Flash器件时务必更新“g_flash_protect_size_upper”表格，表格位于“flash_protect.c”中。默认支持Flash保护的型号：W25Q16JL、W25Q16JW、GD25LE16、GD25WQ16、EN25S16、EN25QH16、P25Q16LE，如果Flash型号不在默认列表内，确保g_flash_protect_size_lower修改为器件支持的范围后，可通过如下任一种修改使用Flash保护功能：

- flash_ram.c的flash_init_cfg函数中，可以通过替换默认Flash器件表“g_flash_default_info_tbl”。
- flash_protect.c的hi_flash_protect_init函数中替换flash chip_name比较部分或将g_flash_ctrl->basic_info.chip_name在比较之前修改为非UNKNOWN。

 **注意**

- 建议同时打开flashboot解保护以及kernel下动态保护特性；能够有效避免芯片上下电、低电压等对flash的非法写入。
- 如果选取的flash型号非默认列表中的成员，指令与保护分区存在差异，一定要进行适配，否则可能导致flash误操作。
- 由flash保护版本升级到不支持flash保护版本，需要在升级文件中解决，防止新版本无法解保护。
- 由不支持保护版本升级到支持flash保护版本：先升级支持解保护的flashboot，在升级支持flash保护的kernel。



2.11.3 注意事项

- 更新Flash分区时务必根据具体的器件以及业务划分Flash分区域。
- 更新Flash器件时务必根据器件重新更新Flash保护表。

2.11.4 编程实例

无

2.12 可维可测接口

2.12.1 概述

Hi3861V100、Hi3861LV100芯片的调试手段有四种：printf、hi_at_printf、shell、HSO（HiStudio）。其中：

- printf、hi_at_printf直接在串口工具中打印调试日志。
- shell提供了相关初始化以及注册函数。
- HSO需要配合专用的调试工具使用。

2.12.2 接口介绍

2.12.2.1 printf 接口

printf是最简单、最常用的一种打印调试信息的接口。调用该接口将消息传输到对应的串口。

printf接口在<hi_early_debug.h>中声明。

2.12.2.2 hi_at_printf 接口

hi_at_printf的使用方式与printf一致，当前代码是为AT指令集提供日志信息打印的接口，输入AT指令后调用hi_at_printf接口进行信息打印输出应答。

hi_at_printf接口在<hi_at.h>中声明。

2.12.2.3 shell 工具

Hi3861V100、Hi3861LV100芯片提供shell初始化、注册输入输出函数，以便用户进行shell命令的开发。

shell工具相关接口在<hi_shell.h>中声明。

2.12.2.4 HSO 调试工具

Hi3861V100、Hi3861LV100芯片支持HiStudio-RD工具打印日志消息，并提供多种用于不同场景的不同打印等级的打印接口。

HSO调试工具相关接口在<hi_diag.h>中声明。



2.12.3 使用方法

说明

HSO接口调试使用专用的调试工具HiStudio-RD。详细的工具使用方法请参见《Hi3861V100 / Hi3861LV100 HSO工具 使用指南》。

步骤1 添加头文件，调用日志打印接口。

步骤2 编译生成bin文件，将bin文件烧录到开发板中。

步骤3 打开串口调试工具，查看打印的日志消息。

----结束

2.12.4 注意事项

- 可维可测方案仅在调试时打开，Release版本建议用户关闭。
- HSO调试工具目前仅用于SDK问题定位时的调测日志打印。
- 由于printf和diag接口共用同一个物理串口，建议在使用HSO工具调测时，不同时使用printf输出调试日志（尤其是在中断中使用），避免printf输出的数据插入到diag接口上报到HSO工具的数据包中，造成HSO工具接收到的数据丢失或错乱。