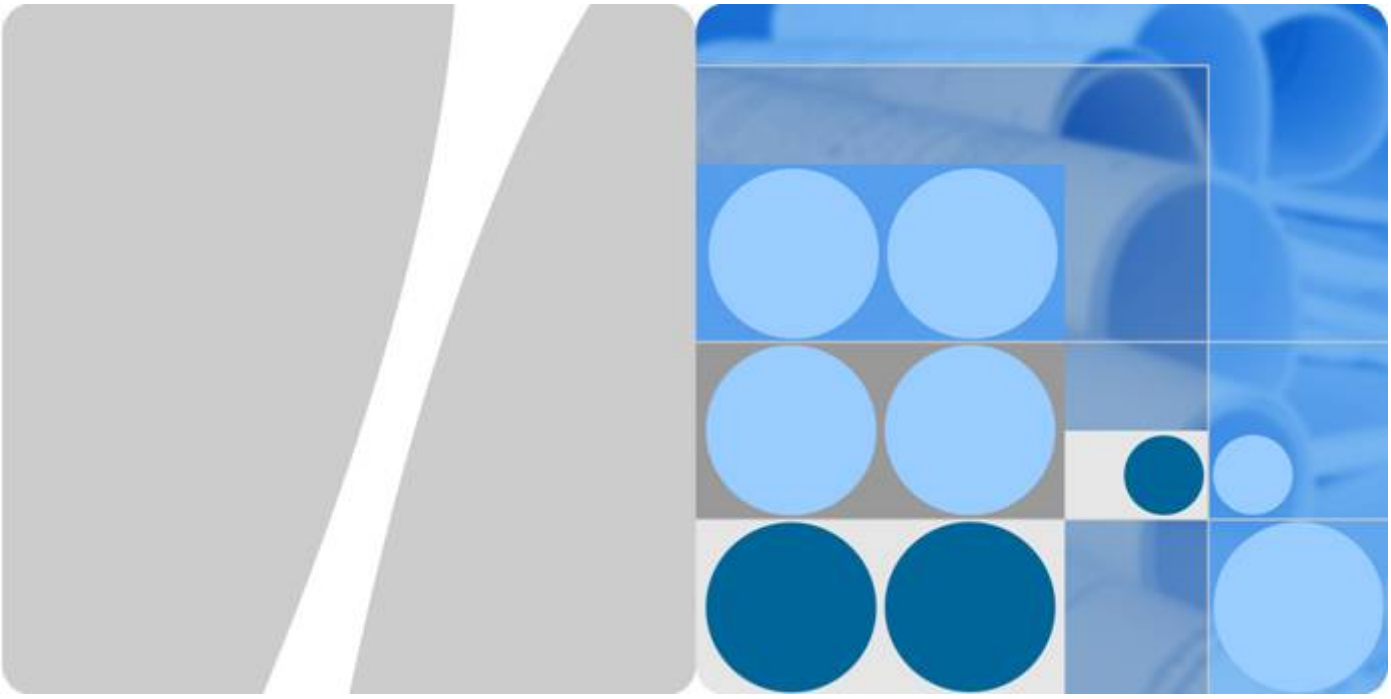


Huawei LiteOS Kernel 开发指南

文档版本　01

发布日期　2016-09-18

华为技术有限公司

HUAWEI

版权所有 © 华为技术有限公司 2016。 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


和其他华为商标均为华为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、服务或特性等应受华为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，华为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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： http://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

版权所有 © 华为技术有限公司

1 前言

目的

本文档介绍Huawei LiteOS的体系结构，并介绍如何进行内核相关的开发和调试。

读者对象

本文档主要适用于Huawei LiteOS Kernel的开发者。

本文档主要适用于以下对象：

- 物联网端侧软件开发工程师
- 物联网架构设计师

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
	用于警示紧急的危险情形，若不避免，将会导致人员死亡或严重的人身伤害
	用于警示潜在的危险情形，若不避免，可能会导致人员死亡或严重的人身伤害
	用于警示潜在的危险情形，若不避免，可能会导致中度或轻微的人身伤害
	用于传递设备或环境安全警示信息，若不避免，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 “注意”不涉及人身伤害
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

版权所有 © 华为技术有限公司

修订记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

日期	修订版本	描述
2015年10月28日	1.0	完成初稿
2016年03月26日	2.0	手册优化

版权所有 © 华为技术有限公司

2 概述

版权所有 © 华为技术有限公司

2.1 背景介绍

Huawei LiteOS Kernel是轻量级的实时操作系统，是华为IOT OS的内核。

版权所有 © 华为技术有限公司

图 2-1 Huawei LiteOS 的基本框架图

版权所有 © 华为技术有限公司

Huawei LiteOS基础内核是最精简的Huawei LiteOS操作系统代码，包括任务管理、内存管理、时间管理、通信机制、中断管理、队列管理、事件管理、定时器、异常管理等操作系统基础组件，可以单独运行。

Huawei LiteOS Kernel 的优势

- 高实时性，高稳定性。
- 超小内核，基础内核体积可以裁剪至不到10K。
- 低功耗。
- 支持动态加载、分散加载。
- 支持功能静态裁剪。

各模块简介

任务

提供任务的创建、删除、延迟、挂起、恢复等功能，以及锁定和解锁任务调度。支持任务按优先级高低的抢占调度及同优先级时间片轮转调度。

任务同步

- 信号量：支持信号量的创建、删除、申请和释放等功能。
- 互斥锁：支持互斥锁的创建、删除、申请和释放等功能。

硬件相关

提供中断、定时器等功能。

- 中断：提供中断的创建、删除、使能、禁止、请求位的清除等功能。
- 定时器：提供定时器的创建、删除、启动、停止等功能。

IPC通信

提供事件、消息队列功能。

- 事件：支持读事件和写事件功能。
- 消息队列：支持消息队列的创建、删除、发送和接收功能。

时间管理

- 系统时间：系统时间是由定时/计数器产生的输出脉冲触发中断而产生的。
- Tick时间：Tick是操作系统调度的基本时间单位，对应的时长由系统主频及每秒Tick数决定，由用户配置。
- 软件定时器：以Tick为单位的定时器功能，软件定时器的超时处理函数在系统创建的Tick软中断中被调用。

内存管理

- 提供静态内存和动态内存两种算法，支持内存申请、释放。目前支持的内存管理算法有固定大小的BOX算法、动态申请DLINK算法。
- 提供内存统计、内存越界检测功能。

异常接管

版权所有 © 华为技术有限公司

异常接管是指在系统运行过程中发生异常后，跳转到异常处理信息的钩子函数，打印当前发生异常函数调用栈信息，或者保存当前系统状态的一系列动作。

Huawei LiteOS的异常接管，会在异常后打印发生异常的任务ID号、栈大小，以及LR、PC等寄存器信息。

动态加载

动态加载是一种软件加载链接技术，不对组成程序的目标文件进行一次性链接加载，等到程序要运行时才进行链接加载。

Huawei LiteOS提供支持OBJ目标文件和SO共享目标文件的动态加载机制。

分散加载

分散加载是通过重排镜像等手段，把关键业务优先加载，从而缩短启动时间。

2.2 支持的核

表 2-1 Huawei LiteOS 支持的核

支持的核	芯片
Cortex-A7	Hi3516A
Cortex-M3	K3V3，K3V3+
Cortex-M4	STMF411，STMF429
Cortex-M7	K3V5
ARM9	Hi3911，Hi3518EV200

2.3 使用约束

- Huawei LiteOS提供一套Huawei LiteOS接口，同时支持POSIX接口，它们功能一致，但混用POSIX和Huawei LiteOS接口可能会导致不可预知的错误,例如用POSIX接口申请信号量，但用Huawei LiteOS接口释放信号量。
- 开发驱动程序只能用Huawei LiteOS的接口。上层APP建议用POSIX接口。

版权所有 © 华为技术有限公司

3 基础内核

版权所有 © 华为技术有限公司

3.1 任务

3.1.1 概述

基本概念

从系统的角度看，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。

Huawei LiteOS的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。这样用户可以将更多的精力投入到业务功能的实现中。

Huawei LiteOS是一个支持多任务的操作系统。在Huawei LiteOS中，一个任务表示一个线程。

Huawei LiteOS中的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。

Huawei LiteOS的任务一共有32个优先级(0-31)，最高优先级为0，最低优先级为31。

任务相关概念

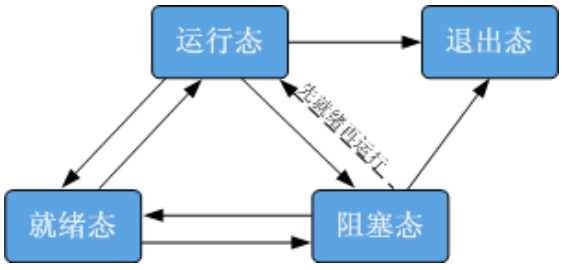
任务状态

Huawei LiteOS系统中的每一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四种：

- 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写事件等。
- 退出态（Dead）：该任务运行结束，等待系统回收资源。

图 3-1 任务状态示意图



任务状态迁移说明：

- 就绪态→运行态：

版权所有 © 华为技术有限公司

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、读信号量等待）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impossible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impossible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将切出任务的上

版权所有 © 华为技术有限公司

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，Huawei LiteOS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。

运作机制

Huawei LiteOS任务管理模块提供任务创建、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、根据任务控制块查询任务ID、根据ID查询任务控制块信息功能。

在用户创建任务之前，系统会先申请任务控制块需要的内存空间，如果系统可用的内存空间小于其所需要的内存空间，任务模块就会初始化失败。如果任务初始化成功，则系统对任务控制块内容进行初始化。

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”。

3.1.2 开发指导

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。

功能

Huawei LiteOS 系统中的任务管理模块为用户提供下面几种功能。

功能分类	接口名	描述
任务的创建和删除	LOS_TaskCreateOnly	创建任务，并使该任务进入suspend状态，并不调度
	LOS_TaskCreate	创建任务，并使该任务进入ready状态，并调度
	LOS_TaskDelete	删除指定的任务
任务状态控制	LOS_TaskResume	恢复挂起的任务

版权所有 © 华为技术有限公司

功能分类	接口名	描述
	LOS_TaskSuspend	挂起指定的任务
	LOS_TaskDelay	任务延时等待
	LOS_TaskYield	显式放权，调整指定优先级的任务调度顺序
任务调度的控制	LOS_TaskLock	锁任务调度
	LOS_TaskUnlock	解锁任务调度
任务优先级的控制	LOS_CurTaskPriSet	设置当前任务的优先级
	LOS_TaskPriSet	设置指定任务的优先级
	LOS_TaskPriGet	获取指定任务的优先级
任务信息获取	LOS_CurTaskIDGet	获取当前任务的ID
	LOS_TaskInfoGet	获取指定任务的信息

开发流程

以创建任务为例，讲解开发流程。

1. 在los_config.h中配置任务模块。

   配置LOSCFG_BASE_CORE_TSK_LIMIT系统支持最大任务数，这个可以根据需求自己配置。

   配置LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE IDLE任务栈大小，这个默认即可。

   配置LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE默认任务栈大小，用户根据自己的需求进行配置，在用户创建任务时，可以进行针对性设置。

   配置LOSCFG_BASE_CORE_TIMESLICE时间片开关为YES。

   配置LOSCFG_BASE_CORE_TIMESLICE_TIMEOUT时间片，根据实际情况自己配置。

   配置LOSCFG_BASE_CORE_TSK_MONITOR任务监测模块裁剪开关，可选择是否打开。
2. 锁任务LOS_TaskLock，锁住任务，防止高优先级任务调度。
3. 创建任务LOS_TaskCreate。
4. 解锁任务LOS_TaskUnlock，让任务按照优先级进行调度。
5. 延时任务LOS_TaskDelay，任务延时等待。
6. 挂起指定的任务LOS_TaskSuspend，任务挂起等待恢复操作。
7. 恢复挂起的任务LOS_TaskResume。

TASK 状态

Huawei LiteOS任务的大多数状态由内核维护，唯有自删除状态对用户可见，需要用户在创建任务时传入：

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述
1	LOS_TASK_STATUS_DETACHED	0x0080	任务是自删除的

用户在调用LOS_TaskCreate接口创建任务时，需要将创建任务的TSK_INIT_PARAM_S参数的uwResved域设置为LOS_TASK_STATUS_DETACHED，即自删除状态，设置成自删除状态的任务会在运行完成时进行自删除动作。

注意

在调用内核LOS_TaskCreate接口创建任务时，默认必须要将任务状态设置为LOS_TASK_STATUS_DETACHED。

TASK 错误码

对任务存在失败可能性的操作，包括创建任务、删除任务、挂起任务、恢复任务、延时任务等等，均需要返回对应的错误码，以便快速定位错误原因。

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_TSK_NO_MEMORY	0x03000200	内存空间不足	分配更大的内存分区
2	LOS_ERRNO_TSK_PTR_NULL	0x02000201	任务参数为空	检查任务参数
3	LOS_ERRNO_TSK_STKSZ_NOT_ALIGN	0x02000202	任务栈大小未对齐	对齐任务栈
4	LOS_ERRNO_TSK_PRIOR_ERROR	0x02000203	不正确的任务优先级	检查任务优先级
5	LOS_ERRNO_TSK_ENTRY_NULL	0x02000204	任务入口函数为空	定义任务入口函数
6	LOS_ERRNO_TSK_NAME_EMPTY	0x02000205	任务名为空	设置任务名
7	LOS_ERRNO_TSK_STKSZ_TOO_SMALL	0x02000206	任务栈太小	扩大任务栈
8	LOS_ERRNO_TSK_ID_INVALID	0x02000207	无效的任务ID	检查任务ID

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
9	LOS_ERRNO_TSK_ALREADY_SUSPENDED	0x02000208	任务已经被挂起	等待这个任务被恢复后，再去尝试挂起这个任务
10	LOS_ERRNO_TSK_NOT_SUSPENDED	0x02000209	任务未被挂起	挂起这个任务
11	LOS_ERRNO_TSK_NOT_CREATED	0x0200020a	任务未被创建	创建这个任务
12	LOS_ERRNO_TSK_DELETE_LOCKED	0x0300020b	删除任务时，任务处于被锁状态	等待解锁任务之后再进行删除操作
13	LOS_ERRNO_TSK_MSG_NONZERO	0x0200020c	任务信息非零	暂不使用该错误码
14	LOS_ERRNO_TSK_DELAY_IN_INT	0x0300020d	中断期间，进行任务延时	等待退出中断后再进行延时操作
15	LOS_ERRNO_TSK_DELAY_IN_LOCK	0x0200020e	任务被锁的状态下，进行延时	等待解锁任务之后再进行延时操作
16	LOS_ERRNO_TSK_YIELD_INVALID_TASK	0x0200020f	将被排入行程的任务是无效的	检查这个任务
17	LOS_ERRNO_TSK_YIELD_NOT_ENOUGH_TASK	0x02000210	没有或者仅有一个可用任务能进行行程安排	增加任务数
18	LOS_ERRNO_TSK_TCB_UNAVAILABLE	0x02000211	没有空闲的任务控制块可用	增加任务控制块数量
19	LOS_ERRNO_TSK_HOOK_NOT_MATCH	0x02000212	任务的钩子函数不匹配	暂不使用该错误码
20	LOS_ERRNO_TSK_HOOK_IS_FULL	0x02000213	任务的钩子函数数量超过界限	暂不使用该错误码
21	LOS_ERRNO_TSK_OPERATE_IDLE	0x02000214	这是个IDLE任务	检查任务ID，不要试图操作IDLE任务
22	LOS_ERRNO_TSK_SUSPEND_LOCKED	0x03000215	将被挂起的任务处于被锁状态	等待任务解锁后再尝试挂起任务
23	LOS_ERRNO_TSK_FREE_STACK_FAILED	0x02000217	任务栈free失败	该错误码暂不使用
24	LOS_ERRNO_TSK_STKAREA_TOO_SMALL	0x02000218	任务栈区域太小	该错误码暂不使用

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
25	LOS_ERRNO_TSK_ACTIVE_FAILED	0x03000219	任务触发失败	创建一个IDLE任务后执行任务转换
26	LOS_ERRNO_TSK_CONFIG_TOO_MANY	0x0200021a	过多的任务配置项	该错误码暂不使用
27	LOS_ERRNO_TSK_CP_SAVE_AREA_NOT_ALIGN	0x0200021b	暂无	该错误码暂不使用
28	LOS_ERRNO_TSK_MSG_Q_TOO_MANY	0x0200021d	暂无	该错误码暂不使用
29	LOS_ERRNO_TSK_CP_SAVE_AREA_NULL	0x0200021e	暂无	该错误码暂不使用
30	LOS_ERRNO_TSK_SELF_DELETE_ERR	0x0200021f	暂无	该错误码暂不使用
31	LOS_ERRNO_TSK_STKSZ_TOO_LARGE	0x02000220	任务栈大小设置过大	减小任务栈大小
32	LOS_ERRNO_TSK_SUSPEND_SWTMR_NOT_ALLOWED	0x02000221	不允许挂起软件定时器任务	检查任务ID, 不要试图挂起软件定时器任务

错误码定义：错误码是一个32位的存储单元，31~24位表示错误等级，23~16位表示错误码标志，15~8位代表错误码所属模块，7~0位表示错误码序号，如下

```
#define LOS_ERRNO_OS_NORMAL(MID,ERRNO)  \
(LOS_ERRTYPE_NORMAL | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | (ERRNO))
LOS_ERRTYPE_NORMAL : Define the error level as critical
LOS_ERRNO_OS_ID : OS error code flag.
MID：OS_MOUDLE_ID
ERRNO：error ID number
```

例如：

```
LOS_ERRNO_TSK_NO_MEMORY  LOS_ERRNO_OS_FATAL(LOS_MOD_TSK, 0x00)
```

注意

错误码序号 0x16、0x1c，未被定义，不可用。

平台差异性

无。

版权所有 © 华为技术有限公司

3.1.3 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。
- 任务名是指针没有分配空间，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，则使用配置项LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE指定默认的任务栈大小。
- 任务栈的大小按8字节大小对齐。确定任务栈大小的原则是，够用就行：多了浪费，少了任务栈溢出。
- 挂起任务的时候若为当前任务且已锁任务，则不能被挂起。
- Idle任务及软件定时器任务不能被挂起或者删除。
- 在中断处理函数中或者在锁任务的情况下，执行LOS_TaskDelay操作会失败。
- 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。
- 设置任务优先级的时候可能会发生任务调度。
- 除去Idle任务以外，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，那么系统可配置的任务资源就会减少一个。
- 不建议使用LOS_CurTaskPriSet或者LOS_TaskPriSet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。
- LOS_CurTaskPriSet和LOS_TaskPriSet接口不能在中断中使用。
- LOS_TaskPriGet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。

3.1.4 编程实例

实例描述

下面的示例介绍任务的基本操作方法，包含任务创建、任务延时、任务锁与解锁调度、挂起和恢复、查询当前任务PID、根据PID查询任务信息等操作，阐述任务优先级调度的机制以及各接口的应用。

1. 创建了2个任务:TaskHi和TaskLo。
2. TaskHi为高优先级任务。
3. TaskLo为低优先级任务。

编程示例

```
UINT32 g_uwTskLoID;
UINT32 g_uwTskHiID;
#define TSK_PRIOR_HI 4
#define TSK_PRIOR_LO 5

UINT32 Example_TaskHi()
{
    UINT32 uwRet;
    UINT32 uwCurrentID;
    TSK_INFO_S stTaskInfo;

     printf("Enter TaskHi Handler.\r\n");
```


版权所有 © 华为技术有限公司

```
    /*延时2个Tick，延时后该任务会挂起，执行剩余任务中最高优先级的任务(g_uwTskLoID任务)*/
    uwRet = LOS_TaskDelay(2);
    if (uwRet != LOS_OK)
    {
        printf("Delay Task Failed.\r\n");
        return LOS_NOK;
    }

    /*2个Tick时间到了后，该任务恢复，继续执行*/
    printf("TaskHi LOS_TaskDelay Done.\r\n");

    /*挂起自身任务*/
    uwRet = LOS_TaskSuspend(g_uwTskHiID);
    if (uwRet != LOS_OK)
    {
        printf("Suspend TaskHi Failed.\r\n");
        return LOS_NOK;
    }
    printf("TaskHi LOS_TaskResume Success.\r\n");
}

/*低优先级任务入口函数*/
UINT32 Example_TaskLo()
{
    UINT32 uwRet;
    UINT32 uwCurrentID;
    TSK_INFO_S stTaskInfo;

    printf("Enter TaskLo Handler.\r\n");

    /*延时2个Tick，延时后该任务会挂起，执行剩余任务中就高优先级的任务(背景任务)*/
    uwRet = LOS_TaskDelay(2);
    if (uwRet != LOS_OK)
    {
        printf("Delay TaskLo Failed.\r\n");
        return LOS_NOK;
    }

    printf("TaskHi LOS_TaskSuspend Success.\r\n");

    /*恢复被挂起的任务g_uwTskHiID*/
    uwRet = LOS_TaskResume(g_uwTskHiID);
    if (uwRet != LOS_OK)
    {
        printf("Resume TaskHi Failed.\r\n");
        return LOS_NOK;
    }

    printf("TaskHi LOS_TaskDelete Success.\r\n");
}

/*任务测试入口函数，在里面创建优先级不一样的两个任务*/
UINT32 Example_TskCaseEntry(VOID)
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S stInitParam;

    /*锁任务调度*/
    LOS_TaskLock();

    printf("LOS_TaskLock() Success!\r\n");

    stInitParam.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_TaskHi;
    stInitParam.usTaskPrio = TSK_PRIOR_HI;
    stInitParam.pcName = "HIGH_NAME";
    stInitParam.uwStackSize = 0x400;
    stInitParam.uwResved   = LOS_TASK_STATUS_DETACHED;
    /*创建高优先级任务，由于锁任务调度，任务创建成功后不会马上执行*/
```


版权所有 © 华为技术有限公司

```
    uwRet = LOS_TaskCreate(&g_uwTskHiID, &stInitParam);
    if (uwRet != LOS_OK)
    {
        LOS_TaskUnlock();

        printf("Example_TaskHi create Failed!\r\n");
        return LOS_NOK;
    }

    printf("Example_TaskHi create Success!\r\n");

    stInitParam.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_TaskLo;
    stInitParam.usTaskPrio = TSK_PRIOR_LO;
    stInitParam.pcName = "LOW_NAME";
    stInitParam.uwStackSize = 0x400;
    stInitParam.uwResved   = LOS_TASK_STATUS_DETACHED;
    /*创建低优先级任务，由于锁任务调度，任务创建成功后不会马上执行*/
    uwRet = LOS_TaskCreate(&g_uwTskLoID, &stInitParam);
    if (uwRet != LOS_OK)
    {
        LOS_TaskUnlock();

        printf("Example_TaskLo create Failed!\r\n");
        return LOS_NOK;
    }

    printf("Example_TaskLo create Success!\r\n");

    /*解锁任务调度，此时会发生任务调度，执行就绪列表中最高优先级任务*/
    LOS_TaskUnlock();

    while(1){};

    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
 --- Test start---
LOS_TaskLock() Success!
Example_TaskHi create Success!
Example_TaskLo create Success!
Enter TaskHi Handler.
Enter TaskLo Handler.
TaskHi LOS_TaskDelay Done.
TaskHi LOS_TaskSuspend Success.
TaskHi LOS_TaskResume Success.
TaskHi LOS_TaskDelete Success.
```

完整实例代码

sample_task.c

3.2 内存

版权所有 © 华为技术有限公司

3.2.1 概述

基本概念

内存管理模块管理系统的内存资源，它是操作系统的核心模块之一。主要包括内存的初始化、分配以及释放。

在系统运行过程中，内存管理模块通过对内存的申请/释放操作，来管理用户和OS对内存的使用，使内存的利用率和使用效率达到最优，同时最大限度地解决系统的内存碎片问题。

Huawei LiteOS的内存管理分为静态内存管理和动态内存管理，提供内存初始化、分配、释放等功能。

- 动态内存：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。
- 静态内存：在静态内存池中分配用户初始化时预设（固定）大小的内存块。
  - 优点：分配和释放效率高，静态内存池中无碎片。
  - 缺点：只能申请到初始化预设大小的内存块，不能按需申请。

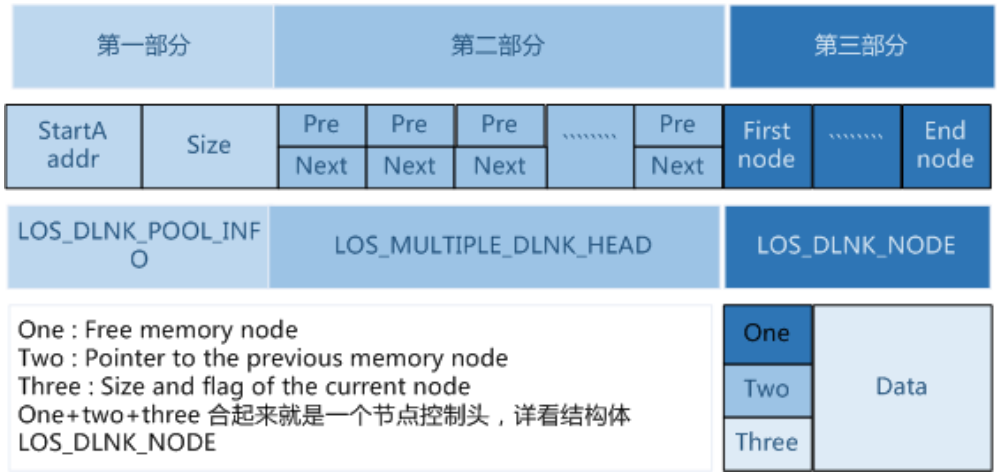
动态内存运作机制

动态内存管理，即在内存资源充足的情况下，从系统配置的一块比较大的连续内存（内存池），根据用户需求，分配任意大小的内存块。当用户不需要该内存块时，又可以释放回系统供下一次使用。

与静态内存相比，动态内存管理的好处是按需分配，缺点是内存池中容易出现碎片。

系统动态内存管理结构如图1所示：

图 3-2



第一部分：堆内存（也称内存池）的起始地址及堆区域总大小

第二部分：本身是一个数组，每个元素是一个双向链表，所有free节点的控制头都会被分类挂在这个数组的双向链表中。

版权所有 © 华为技术有限公司

假设内存允许的最小节点为2^{min}字节，则数组的第一个双向链表存储的是所有size为2^{min}<size< 2^{min+1}的free节点，第二个双向链表存储的是所有size为2^{min+1}<size< 2^{min+2}的free节点，依次类推第n个双向链表存储的是所有size为$2^{min+n-1}$<size< 2^{min+n}的free节点。每次申请内存的时候，会从这个数组检索最合适大小的free节点，进行分配内存。每次释放内存时，会将该片内存作为free节点存储至这个数组，以便下次再利用。

第三部分：占用内存池极大部分的空间，是用于存放各节点的实际区域。以下是

LOS_MEM_DYN_NODE节点结构体申明以及简单介绍：

typedef struct tagLOS_MEM_DYN_NODE

{

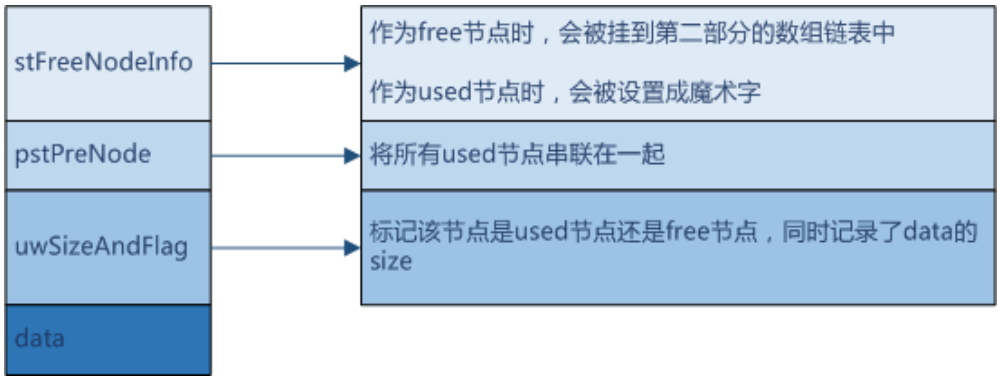
LOS_DL_LIST stFreeNodeInfo;

struct tagLOS_MEM_DYN_NODE *pstPreNode;

UINT32 uwSizeAndFlag;

}LOS_MEM_DYN_NODE;

图 3-3

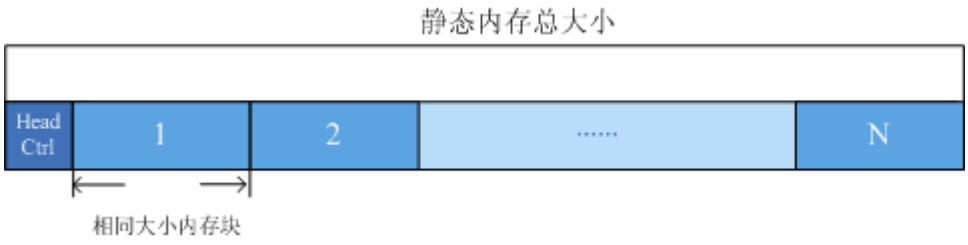


静态内存运作机制

静态内存实质上是一块静态数组，静态内存池内的块大小在初始化时设定，初始化后块大小不可变更。

静态内存池由一个控制块和若干相同大小的内存块构成。控制块位于内存池头部，用于内存块管理。内存块的申请和释放以块大小为粒度。

图 3-4 静态内存示意图



3.2.2 动态内存

版权所有 © 华为技术有限公司

3.2.2.1 开发指导

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

Huawei LiteOS系统中的动态内存管理模块为用户提供下面几种功能，具体的API详见接口手册。

功能分类	接口名	描述
内存初始化	LOS_MemInit	初始化一块指定的动态内存池，大小为size。
申请动态内存	LOS_MemAlloc	从指定动态内存池中申请size长度的内存。
释放动态内存	LOS_MemFree	释放已申请的内存。
重新申请内存	LOS_MemRealloc	按size大小重新分配内存块，并保留原内存块内容。
内存对齐分配	LOS_MemAllocAlign	从指定动态内存池中申请长度为size且地址按boundary字节对齐的内存。
获取内存大小	LOS_MemPoolSizeGet	获取指定的动态内存池总大小
获取内存大小	LOS_MemTotalUsedGet	获取指定动态内存池的总使用量大小
获取内存块数量	LOS_MemFreeBlksGet	获取指定内存池的空闲内存块数量
获取内存块数量	LOS_MemUsedBlksGet	获取指定内存池的已使用的内存块数量
获取分配指定内存区域的任务ID	LOS_MemTaskIdGet	获取分配了指定内存区域的任务ID
获取最后节点内存大小	LOS_MemLastNodeGet	获取除大小为零的终端节点之外的最后一个节点的内存大小
获取内存结构信息	LOS_MemInfoGet	获取指定内存池的内存结构信息
对指定内存池做完整性检查	LOS_MemIntegrityCheck	对指定内存池做完整性检查

版权所有 © 华为技术有限公司

功能分类	接口名	描述
获取节点大小	LOS_MemNodeSizeCheck	获取节点的总大小和可操作大小
设定内存检查级别	LOS_MemCheckLevelSet	设定内存检查级别
获取内存检查级别	LOS_MemCheckLevelGet	获取内存检查级别

开发流程

1. 配置：

   OS_SYS_MEM_ADDR：系统动态内存池起始地址，一般不需要修改

   OS_SYS_MEM_SIZE：系统动态内存池大小，以byte为单位，系统默认分配DDR后未使用的空间

   LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK：内存越界检测开关，默认关闭。打开后，每次申请动态内存时执行动态内存块越界检查；每次释放静态内存时执行静态内存块越界检查。

2. 初始化LOS_MemInit。

   初始一个内存池后如图，生成一个 EndNode，并且剩余的内存全部被标记为FreeNode节点。注：EndNode作为内存池末尾的节点，size为0。

3. 申请任意大小的动态内存LOS_MemAlloc。

   判断动态内存池中是否存在申请量大小的空间，若存在，则划出一块内存块，以指针形式返回，若不存在，返回NULL。

   调用三次LOS_MemAlloc函数可以创建三个节点,假设名称分别为UsedA，UsedB，UsedC，大小分别为sizeA，sizeB，sizeC。因为刚初始化内存池的时候只有一个大的FreeNode，所以这些内存块是从这个FreeNode中切割出来的。

   当内存池中存在多个FreeNode的时候进行malloc，将会适配最合适大小的FreeNode用来新建内存块，减少内存碎片。若新建的内存块不等于被使用的FreeNode的大小，则在新建内存块后，多余的内存又会被标记为一个新的FreeNode。

4. 释放动态内存LOS_MemFree。

   回收内存块，供下一次使用。

   假设调用LOS_MemFree释放内存块UsedB，则会回收内存块UsedB，并且将其标记为FreeNode

版权所有 © 华为技术有限公司

平台差异性

无。

3.2.2.2 注意事项

- 由于系统中动态内存管理需要消耗管理控制块结构的内存，故实际用户可使用空间总量小于在配置文件los_config.h中配置项OS_SYS_NOCACHEMEM_SIZE的大小。
- 系统中地址对齐申请内存分配LOS_MemAllocAlign可能会消耗部分对齐导致的空间，故存在一些内存碎片，当系统释放该对齐内存时，同时回收由于对齐导致的内存碎片。
- 系统中重新分配内存LOS_MemRealloc函数如果分配成功，系统会自己判定是否需要释放原来申请的空间，返回重新分配的空间。用户不需要手动释放原来的空间。
- 系统中多次调用LOS_MemFree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，导致结果不可预知。

3.2.2.3 编程实例

实例描述

Huawei LiteOS运行期间，用户需要频繁的使用内存资源，而内存资源有限，必须确保将有限的内存资源分配给急需的程序，同时释放不用的内存。

通过Huawei LiteOS内存管理模块可以保证高效、正确的申请、释放内存。

本实例执行以下步骤：

1. 初始化一个动态内存池。
2. 在动态内存池中申请一个内存块。
3. 使用这块内存块存放一个数据。
4. 打印出存放在内存块中的数据。
5. 释放掉这块内存。

编程实例

```
VOID los_memory_test() {
    UINT32 *p_num = NULL;
    UINT32 uwRet;
    uwRet = LOS_MemInit(m_aucSysMem0, 32);
    if (LOS_OK  == uwRet) {
        dprintf("内存池初始化成功!\n");
    }
    else {
        dprintf("内存池初始化失败!\n");
        return;
    }
    /*分配内存*/
    p_num = (int*)LOS_MemAlloc(m_aucSysMem0, 4);
    if (NULL == p_num) {
        dprintf("内存分配失败!\n");
        return;
    }
    dprintf("内存分配成功\n");
    /*赋值*/
    *p_num = 828;
```


版权所有 © 华为技术有限公司

```
    dprintf("*p_num = %d\n", *p_num);
    /*释放内存*/
    uwRet = LOS_MemFree(m_aucSysMem0, p_num);
    if (LOS_OK == uwRet) {
        dprintf("内存释放成功!\n");
    }
    else {
        dprintf("内存释放失败!\n");
    }
    return;
}
```

结果验证

图 3-5 结果显示

```
 --- Test start---
using new mem argorithm
内存池初始化成功!
内存分配成功
*p_num = 828
内存释放成功!
```

完整实例代码

sample_mem.c

3.2.3 静态内存

3.2.3.1 开发指导

使用场景

当用户需要使用固定长度的内存时，可以使用静态内存分配的方式获取内存，一旦使用完毕，通过静态内存释放函数归还所占用内存，使之可以重复使用。

功能

Huawei LiteOS的静态内存管理主要为用户提供以下功能。

功能分类	接口名	描述
初始化静态内存	LOS_MemboxInit	初始化一个静态内存池，设定其起始地址、总大小及每个块大小
清除静态内存内容	LOS_MemboxClr	清零静态内存块
申请一块静态内存	LOS_MemboxAlloc	申请一块静态内存块
释放内存	LOS_MemboxFree	释放一个静态内存块

开发流程

本节介绍使用静态内存的典型场景开发流程。

版权所有 © 华为技术有限公司

1. 规划一片内存区域作为静态内存池。
2. 调用LOS_MemboxInit接口。

   系统内部将会初始化静态内存池。将入参指定的内存区域分割为N块（N值取决于静态内存总大小和块大小），将所有内存块挂到空闲链表，在内存起始处放置控制头。
3. 调用LOS_MemboxAlloc接口。

   系统内部将会从空闲链表中获取第一个空闲块，并返回该块的用户空间地址。
4. 调用LOS_MemboxFree接口。

   将该块内存加入空闲块链表。
5. 调用LOS_MemboxClr接口。

   系统内部清零静态内存块，将入参地址对应的内存块清零。

平台差异性

无。

3.2.3.2 注意事项

- 静态内存池区域，可以通过定义全局数组或调用动态内存分配接口方式获取。如果使用动态内存分配方式，在不需要静态内存池时，注意要释放该段内存，避免内存泄露。

3.2.3.3 编程实例

实例描述

Huawei LiteOS运行期间，用户需要频繁的使用内存资源，而内存资源有限，必须确保将有限的内存资源分配给急需的程序，同时释放不用的内存。

通过内存管理模块可以保证正确且高效的申请释放内存。

本实例执行以下步骤：

1. 初始化一个静态内存池。
2. 从静态内存池中申请一块静态内存。
3. 使用这块内存块存放一个数据。
4. 打印出存放在内存块中的数据。
5. 清除内存块中的数据。
6. 释放掉这块内存。

编程实例

```
VOID los_membox_test(void) {
    UINT32 *p_num = NULL;
    UINT32 uwBlkSize = 10, uwBoxSize = 100;
    UINT32 uwRet;
    UINT32 pBoxMem[1000];
    uwRet = LOS_MemboxInit(&pBoxMem[0], uwBoxSize, uwBlkSize);
    if (uwRet != LOS_OK)
    {
        dprintf("内存池初始化失败!\n");
        return;
    }
```


版权所有 © 华为技术有限公司

```
    else {
        dprintf("内存池初始化成功!\n");
    }

    /*申请内存块*/
    p_num = (int*)LOS_MemboxAlloc(pBoxMem);
    if (NULL == p_num) {
        dprintf("内存分配失败!\n");
        return;
    }
    dprintf("内存分配成功\n");
    /*赋值*/
    *p_num = 828;
    dprintf("*p_num = %d\n", *p_num);
     /*清除内存内容*/
     LOS_MemboxClr(pBoxMem, p_num);
     dprintf("清除内存内容成功\n *p_num = %d\n", *p_num);
    /*释放内存*/
    uwRet = LOS_MemboxFree(pBoxMem, p_num);
    if (LOS_OK == uwRet) {
        dprintf("内存释放成功!\n");
    }
    else{
        dprintf("内存释放失败!\n");
    }
     return;
}
```

结果验证

图 3-6 结果显示

```
dist:1

 --- Test start---
内存池初始化成功!
内存分配成功
*p_num = 828
清除内存内容成功
*p_num = 0
内存释放成功!

 --- Test End ---
```

完整实例代码

sample_membox.c

3.3 中断机制

3.3.1 概述

基本概念

中断是指出现需要时，CPU暂停执行当前程序，转而执行新程序的过程。即在程序运行过程中，系统出现了一个必须由CPU立即处理的事务，此时，CPU暂时中止当前程序的执行转而处理这个事务，这个过程就叫做中断。

众多周知，CPU的处理速度比外设的运行速度快很多，外设可以在没有CPU介入的情况下完成一定的工作，但某些情况下需要CPU为其做一定的工作。

通过中断机制，在外设不需要CPU介入时，CPU可以执行其它任务，而当外设需要CPU时通过产生中断信号使CPU立即中断当前任务来响应中断请求。这样可以使CPU

版权所有 © 华为技术有限公司

避免把大量时间耗费在等待，查询外设状态的操作上，因此将大大提高系统实时性以及执行效率。

Huawei LiteOS的中断支持：

- 中断初始化。
- 中断创建。
- 开/关中断。
- 恢复中断。
- 中断使能。
- 中断屏蔽。

Huawei LiteOS的中断机制支持中断共享。

中断的介绍

与中断相关的硬件可以划分为三类：设备、中断控制器、CPU本身。

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：中断控制器是CPU众多外设中的一个，它一方面接收其它外设中断引脚的输入，另一方面，它会发出中断信号给CPU。可以通过对中断控制器编程实现对中断源的优先级、触发方式、打开和关闭源等设置操作。常用的中断控制器有VIC（Vector Interrupt Controller）和GIC（General Interrupt Controller），在ARM Cortex-A7中使用的中断控制器是GIC。

CPU：CPU会响应中断源的请求，中断当前正在执行的任务，转而执行中断处理程序。

和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，系统根据中断时间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。Huawei LiteOS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GIC，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。

版权所有 © 华为技术有限公司

中断共享：当外设较少时，可以实现一个外设对应一个中断号，但为了支持更多的硬件设备，可以让多个设备共享一个中断号，共享同一个中断的中断处理程序形成一个链表，当外部设备产生中断申请时，系统会遍历中断号对应的中断处理程序链表。

中断底半部：中断处理程序耗时应尽可能短，以满足中断的快速响应，为了平衡中断处理程序的性能与工作量，将中断处理程序分解为两部分：顶半部和底半部。

顶半部完成尽可能少的比较紧急的任务，它往往只是简单地读取寄存器中的中断状态并清除中断标志位即进行“登记工作”，将耗时的底半部处理程序挂到系统的底半部执行队列中去。

运作机制

Huawei LiteOS的中断机制支持中断共享：

中断共享的实现依赖于链表，对应每一个中断号创建一个链表，链表节点中包含注册的中断处理函数和函数入参。当对同一中断号多次创建中断时，将中断处理函数和函数入参添加到中断号对应的链表中，因此当硬件产生中断时，通过中断号查找到其对应的结构体链表，遍历执行链表中的中断处理函数。

Huawei LiteOS的中断机制支持中断底半部：

中断底半部的实现基于workqueue，在中断处理程序中将工作分为顶半部和底半部，底半部处理程序与work关联，并挂载到合法workqueue上。系统空闲时执行workqueue中的work上的底半部程序。

3.3.2 开发指导

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

Huawei LiteOS 系统中的中断模块为用户提供下面几种功能。

接口名	描述
LOS_HwiCreate	硬中断创建，注册硬中断处理程序
LOS_IntUnLock	开中断
LOS_IntRestore	恢复到关中断之前的状态
LOS_IntLock	关中断
hal_interrupt_mask	中断屏蔽（通过设置寄存器，禁止CPU响应该中断）
hal_interrupt_unmask	中断使能（通过设置寄存器，允许CPU响应该中断）

版权所有 © 华为技术有限公司

HWI 错误码

对创建中断存在失败可能性的操作返回对应的错误码，以便快速定位错误原因。

序号	定义	实际数值	描述	参考解决方案
1	OS_ERRNO_HWI_NUM_INVALID	0x02000900	无效中断号	检查中断号，给定有效中断号
2	OS_ERRNO_HWI_PROC_FUNC_NULL	0x02000901	中断程序指针为空	传入非空中断处理程序指针
3	OS_ERRNO_HWI_CB_UNAVAILABLE	0x02000902	无可用中断资源	通过配置，增大可用中断最大数量
4	OS_ERRNO_HWI_NO_MEMORY	0x02000903	内存不足	增大内存空间
5	OS_ERRNO_HWI_ALREADY_CREATED	0x02000904	中断处理程序已经创建	检查传入的中断号对应的中断处理程序是否已经被创建
6	OS_ERRNO_HWI_PRIO_INVALID	0x02000905	中断优先级无效	传入有效中断优先级[0,31]
7	OS_ERRNO_HWI_MODE_INVALID	0x02000906	中断模式无效	传入有效中断模式[0,1]
8	OS_ERRNO_HWI_FASTMODE_ALREADY_CREATED	0x02000907	快速模式中断已经创建	检查传入的中断号对应的中断处理程序是否已经被创建
9	OS_ERRNO_HWI_INTERR	0x02000908	接口在中断中调用	中断中禁止调用该接口

开发流程

1. 修改配置项
   - 打开硬中断裁剪开关：OS_INCLUDE_HWI定义为YES.
   - 配置硬中断使用最大数：OS_HWI_MAX_USED_NUM.
2. 调用中断初始化Los_HwiInit接口。
3. 调用中断创建接口LOS_HwiCreate创建中断
4. 调用hal_interrupt_unmask接口使能指定中断。
5. 调用hal_interrupt_mask接口屏蔽指定中断。

3.3.3 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。
- 中断共享机制，支持同一中断处理程序的重复挂载，但中断处理程序的入参dev必须唯一，即同一中断号，同一dev只能挂载一次；但同一中断号，同一中断处理程序，dev不同则可以重复挂载。

版权所有 © 华为技术有限公司

- 中断处理程序耗时不能过长，影响CPU对中断的及时响应。
- 关中断后不能执行引起调度的函数。
- 中断恢复LOS_IntRestore()的入参必须是与之对应的LOS_IntLock()保存的关中断之前的CPSR的值。
- Cortex-A7中0-31中断为内部使用，因此不建议用户去申请和创建。
- 一般不直接调用LOS_HwiCreate()创建中断；建议使用系统compat中的linux适配接口request_irq创建中断。

3.3.4 编程实例

实例描述

本实例实现如下功能。

1. 关中断
2. 中断创建
3. 中断使能
4. 中断恢复
5. 中断屏蔽

编程示例

前提条件：

- 在los_config.h中，将OS_INCLUDE_HWI定义为YES。
- 在los_config.h中，设置最大硬中断个数OS_HWI_MAX_USED_NUM。

代码实现如下：

```
#include "los_hwi.h"
#include "los_typedef.h"
#define HWI_NUM_INT50 50
void uart_irqhandle(int irq,void *dev)
{
    printf("\n int the func uart_irqhandle \n");
}
void hwi_test()
{
    int a = 1;
    UINTPTR uvIntSave;
    uvIntSave = LOS_IntLock();
    LOS_HwiCreate(HWI_NUM_INT50, 0,0,uart_irqhandle,NULL);//创建中断
    hal_interrupt_unmask(HWI_NUM_INT50);
    LOS_IntRestore(uvIntSave);
    hal_interrupt_mask(HWI_NUM_INT50);
}
```

完整实例

sample_hwi.c

3.4 队列

版权所有 © 华为技术有限公司

3.4.1 概述

基本概念

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，并根据不同的接口选择传递消息是否存放在自己空间。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。

Huawei LiteOS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。

运作机制

队列控制块

```
/**
  * @ingroup los_queue
  * Queue information block structure
  */
typedef struct tagQueueCB
{
    UINT8       *pucQueue;      /**< 队列指针 */
    UINT16      usQueueState;   /**< 队列状态 */
    UINT16      usQueueLen;     /**< 队列中消息个数 */
    UINT16      usQueueSize;    /**< 消息节点大小 */
    UINT16      usQueueHead;    /**< 消息头节点位置（数组下标）*/
    UINT16      usQueueTail;    /**< 消息尾节点位置（数组下标）*/
    UINT16      usWritableCnt;  /**< 队列中可写消息数*/
    UINT16      usReadableCnt;  /**< 队列中可读消息数*/
    UINT16      usReserved;     /**< 保留字段 */
    LOS_DL_LIST stWriteList;    /**< 写入消息任务等待链表*/
    LOS_DL_LIST stReadList;     /**< 读取消息任务等待链表*/
    LOS_DL_LIST stMemList;      /**< MailBox模块使用 */
} QUEUE_CB_S;
```

每个队列控制块中都含有队列状态，表示该队列的使用情况：

- OS_QUEUE_UNUSED：队列没有使用
- OS_QUEUE_INUSED：队列被使用

队列运作原理

创建队列时，根据用户传入队列长度和消息节点大小来开辟相应的内存空间以供该队列使用，返回队列ID。

在队列控制块中维护一个消息头节点位置Head和一个消息尾节点位置Tail来表示当前队列中消息存储情况。Head表示队列中被占用消息的起始位置。Tail表示队列中被空闲消息的起始位置。刚创建时Head和Tail均指向队列起始位置。

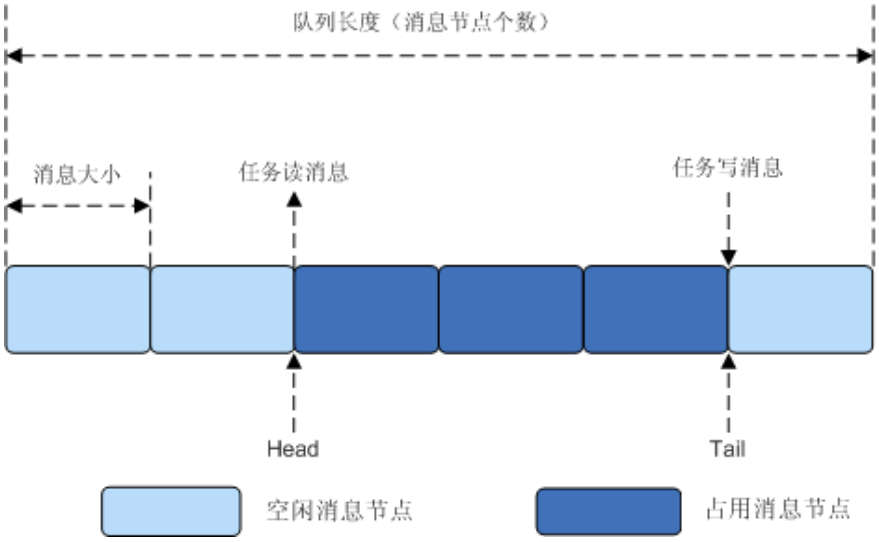
版权所有 © 华为技术有限公司

写队列时，根据Tail找到被占用消息节点末尾的空闲节点作为数据写入对象。如果Tail已经指向队列尾则采用回卷方式。根据usWritableCnt判断队列是否可以写入，不能对已满（usWritableCnt为0）队列进行写队列操作。

读队列时，根据Head找到最先写入队列中的消息节点进行读取。如果Head已经指向队列尾则采用回卷方式。根据usReadableCnt判断队列是否有消息读取，对全部空闲（usReadableCnt为0）队列进行读队列操作会引起任务挂起。

删除队列时，根据传入的队列ID寻找到对应的队列，把队列状态置为未使用，释放原队列所占的空间，对应的队列控制头置为初始状态。

图 3-7 队列读写数据操作示意图



3.4.2 开发指导

功能

Huawei LiteOS中Message消息处理模块提供了以下功能。

功能分类	接口名	描述
创建消息队列	LOS_QueueCreate	创建一个消息队列。
读队列（不带拷贝）	LOS_QueueRead	读取指定队列中的数据。（buff里存放的是队列节点的地址）。
写队列（不带拷贝）	LOS_QueueWrite	向指定队列写数据。（写入队列节点中的是buff的地址）。
读队列（带拷贝）	LOS_QueueReadCopy	读取指定队列中的数据。（buff里存放的是队列节点中的数据）。

版权所有 © 华为技术有限公司

功能分类	接口名	描述
写队列（带拷贝）	LOS_QueueWriteCopy	向指定队列写数据。（写入队列节点中的是buff中的数据）。
写队列（头部）	LOS_QueueWriteHead	向指定队列的头部写数据
删除队列	LOS_QueueDelete	删除一个指定的队列。
获取队列信息	LOS_QueueInfoGet	获取指定队列信息。

开发流程

使用队列模块的典型流程如下：

1. 创建消息队列LOS_QueueCreate。
   创建成功后，可以得到消息队列的ID值。
2. 写队列操作函数LOS_QueueWrite。
3. 读队列操作函数LOS_QueueRead。
4. 获取队列信息函数LOS_QueueInfoGet。
5. 删除队列LOS_QueueDelete。

QUEUE 错误码

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

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_QUEUE_MAXNUM_ZERO	0x02000600	队列资源的最大数目配置为0	配置要大于0的队列资源的最大数量。如果不使用队列模块，则将配置项设置为将队列资源的最大数量的剪裁设置为NO。
2	LOS_ERRNO_QUEUE_NO_MEMORY	0x02000601	队列块内存无法初始化	为队列块分配更大的内存分区，或减少队列资源的最大数量 。
3	LOS_ERRNO_QUEUE_CREATE_NO_MEMORY	0x02000602	队列创建的内存未能被请求	为队列分配更多的内存，或减少要创建的队列中的队列长度和节点的数目。
4	LOS_ERRNO_QUEUE_SIZE_TOO_BIG	0x02000603	队列创建时消息长度超过上限	更改创建队列中最大消息的大小至不超过上线

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
5	LOS_ERRNO_QUEUE_CB_UNAVAILABLE	0x02000604	已超过创建的队列的数量的上限	增加队列的配置资源数量
6	LOS_ERRNO_QUEUE_NOT_FOUND	0x02000605	无效的队列	确保队列ID是有效的
7	LOS_ERRNO_QUEUE_PEND_IN_LOCK	0x02000606	当任务被锁定时，禁止在队列中被阻塞	使用队列前解锁任务
8	LOS_ERRNO_QUEUE_TIMEOUT	0x02000607	等待处理队列的时间超时	检查设置的超时时间是否合适
9	LOS_ERRNO_QUEUE_IN_TSKUSE	0x02000608	阻塞任务的队列不能被删除	使任务能够获得资源而不是在队列中被阻塞
10	LOS_ERRNO_QUEUE_WRITE_IN_INTERRUPT	0x02000609	在中断处理程序中不能写队列	将写队列设为非阻塞模式
11	LOS_ERRNO_QUEUE_NOT_CREATE	0x0200060a	队列未创建	检查队列中传递的句柄是否有效
12	LOS_ERRNO_QUEUE_IN_TSKWRITE	0x0200060b	队列读写不同步	同步队列的读写
13	LOS_ERRNO_QUEUE_CREAT_PTR_NULL	0x0200060c	队列创建过程中传递的参数为空指针	确保传递的参数不为空指针
14	LOS_ERRNO_QUEUE_PARA_ISZERO	0x0200060d	队列创建过程中传递的队列长度或消息节点大小为0	传入正确的队列长度和消息节点大小
15	LOS_ERRNO_QUEUE_READ_INVALID	0x0200060e	读取的队列的handle无效	检查队列中传递的handle是否有效
16	LOS_ERRNO_QUEUE_READ_PTR_NULL	0x0200060f	队列读取过程中传递的指针为空	检查指针中传递的是否为空

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
17	LOS_ERRNO_QUEUE_READSIZE_ISZERO	0x02000610	队列读取过程中传递的缓冲区大小为0	通过一个正确的缓冲区大小
18	LOS_ERRNO_QUEUE_WRITE_INVALID	0x02000611	队列写入过程中传递的队列handle无效	检查队列中传递的handle是否有效
19	LOS_ERRNO_QUEUE_WRITE_PTR_NULL	0x02000612	队列写入过程中传递的指针为空	检查指针中传递的是否为空
20	LOS_ERRNO_QUEUE_WRITESIZE_ISZERO	0x02000613	队列写入过程中传递的缓冲区大小为0	通过一个正确的缓冲区大小
21	LOS_ERRNO_QUEUE_WRITE_NOT_CREATE	0x02000614	写入数据的队列未创建	传入有效队列ID
22	LOS_ERRNO_QUEUE_WRITE_SIZE_TOO_BIG	0x02000615	队列写入过程中传递的缓冲区大小比队列大小要大	减少缓冲区大小，或增大队列节点
23	LOS_ERRNO_QUEUE_ISFULL	0x02000616	在队列写入过程中没有可用的空闲节点	确保在队列写入之前，可以使用空闲的节点
24	LOS_ERRNO_QUEUE_PTR_NULL	0x02000617	正在获取队列信息时传递的指针为空	检查指针中传递的是否为空
25	LOS_ERRNO_QUEUE_READ_IN_INTERRUPT	0x02000618	在中断处理程序中不能读队列	将读队列设为非阻塞模式
26	LOS_ERRNO_QUEUE_MAIL_HANDLE_INVALID	0x02000619	正在释放队列的内存时传递的队列的handle无效	检查队列中传递的handle是否有效
27	LOS_ERRNO_QUEUE_MAIL_PTR_INVALID	0x0200061a	传入的消息内存池指针为空	检查指针是否为空

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
28	LOS_ERRNO_QUEUE_MAIL_FREE_ERROR	0x0200061b	membox内存释放失败	传入非空membox内存指针
29	LOS_ERRNO_QUEUE_READ_NOT_CREATE	0x0200061c	待读取的队列未创建	传入有效队列ID
30	LOS_ERRNO_QUEUE_ISEMPTY	0x0200061d	队列已空	确保在读取队列时包含消息
31	LOS_ERRNO_QUEUE_READ_SIZE_TOO_SMALL	0x0200061f	读缓冲区大小小于队列大小	增加缓冲区大小，或减小队列节点大小

平台差异性

在3516A平台上，写入队列的数据长度不需要四字节对齐，但在3518e平台上，写入队列的数据需要进行四字节对齐。

3.4.3 注意事项

- 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。
- 调用 LOS_QueueCreate 函数时所传入的队列名暂时未使用，作为以后的预留参数。
- 队列接口函数中的入参uwTimeOut是指相对时间。
- LOS_QueueReadCopy和LOS_QueueWriteCopy是一组接口，LOS_QueueRead和LOS_QueueWrite是一组接口，两组接口需要配套使用。
- 鉴于LOS_QueueWrite和LOS_QueueRead这组接口实际操作的是数据地址，用户必须保证调用LOS_QueueRead获取到的指针所指向内存区域在读队列期间没有被异常修改或释放，否则可能会导致不可预知的后果。

3.4.4 编程实例

实例描述

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。

1. 通过LOS_TaskCreate创建任务1和任务2。
2. 通过LOS_QueueCreate创建一个消息队列。
3. 在任务1 send_Entry中发送消息。

版权所有 © 华为技术有限公司

4. 在任务2 recv_Entry中接收消息。
5. 通过LOS_QueueDelete删除队列。

编程示例

```
#include "los_task.h"
#include "los_queue.h"
static UINT32 g_uwQueue;
CHAR abuf[] = "test is message x";

/*任务1发送数据*/
void *send_Entry(void *arg)
{
    UINT32 i = 0,uwRet = 0;
    UINT32 uwlen = sizeof(abuf);

    while (i <5)
    {
        abuf[uwlen -2] = '0' + i;
        i++;

        /*将abuf里的数据写入队列*/
        uwRet = LOS_QueueWrite(g_uwQueue, abuf, uwlen, 0);
        if(uwRet != LOS_OK)
        {
            dprintf("send message failure,error:%x\n",uwRet);
        }

        LOS_TaskDelay(5);
    }
}

/*任务2接收数据*/
void *recv_Entry(void *arg)
{
    UINT32 uwReadbuf;
    UINT32 uwRet = 0;

    while (1)
    {

        /*读取队列里的数据存入uwReadbuf里*/
        uwRet = LOS_QueueRead(g_uwQueue, &uwReadbuf, 50, 0);
        if(uwRet != LOS_OK)
        {
            dprintf("recv message failure,error:%x\n",uwRet);
            break;
        }

        dprintf("recv message:%s\n", (char *)uwReadbuf);
        LOS_TaskDelay(5);
    }
    /*删除队列*/
    while (LOS_OK != LOS_QueueDelete(g_uwQueue))
    {
        LOS_TaskDelay(1);
    }

    dprintf("delete the queue success!\n");
}

int Example_creat_task(void)
{
    UINT32 uwRet = 0;
    UINT32 uwTask1, uwTask2;
    TSK_INIT_PARAM_S stInitParam1;

    /*创建任务1*/
```


版权所有 © 华为技术有限公司

```
    stInitParam1.pfnTaskEntry = send_Entry;
    stInitParam1.usTaskPrio = 9;
    stInitParam1.uwStackSize = 0x400;
    stInitParam1.pcName = "sendQueue";
    stInitParam1.uwResved = LOS_TASK_STATUS_DETACHED;
    LOS_TaskLock();//锁住任务，防止新创建的任务比本任务高而发生调度
    uwRet = LOS_TaskCreate(&uwTask1, &stInitParam1);
    if(uwRet != LOS_OK)
    {
        dprintf("create task1 failed!,error:%x\n",uwRet);
        return uwRet;
    }

    /*创建任务2*/
    stInitParam1.pfnTaskEntry = recv_Entry;
    uwRet = LOS_TaskCreate(&uwTask2, &stInitParam1);
    if(uwRet != LOS_OK)
    {
        dprintf("create task2 failed!,error:%x\n",uwRet);
        return uwRet;
    }

    /*创建队列*/
    uwRet = LOS_QueueCreate("queue", 5, &g_uwQueue, 0, 50);
    if(uwRet != LOS_OK)
    {
        dprintf("create queue failure!,error:%x\n",uwRet);
    }

    dprintf("create the queue success!\n");
    LOS_TaskUnlock();//解锁任务，只有队列创建后才开始任务调度
}
```

结果验证

```
 --- Test start---
create the queue success!
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
recv message failure,error:200061d
delete the queue success!
```

完整实例代码

sample_queue.c

3.5 事件

3.5.1 概述

基本概念

事件是一种实现任务间通信的机制，可用于实现任务间的同步，但事件通信只能是事件类型的通信，无数据传输。一个任务可以等待多个事件的发生：可以是任意一个事件发生时唤醒任务进行事件处理;也可以是几个事件都发生后才唤醒任务进行事件处理。事件集合用32位无符号整型变量来表示，每一位代表一个事件。

版权所有 © 华为技术有限公司

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。Huawei LiteOS的事件仅用于任务间的同步，不提供数据传输功能。

Huawei LiteOS提供的事件具有如下特点：

- 事件不与任务相关联，事件相互独立，一个32位的变量，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表示该事件类型未发生、1表示该事件类型已经发生），一共31种事件类型（第25位保留）。
- 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。
- 允许多个任务对同一事件进行读写操作。
- 支持事件读写超时机制。

事件控制块

```
/**
 * @ingroup los_event
 * Event control structure
 */
typedef struct tagEvent
{
    UINT32 uwEventID;            /**标识发生的事件类型位*/
    LOS_DL_LIST    stEventList;  /**读取事件任务链表*/
} EVENT_CB_S, *PEVENT_CB_S;
```

uwEventID：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表示该事件类型未发生、1表示该事件类型已经发生），一共31种事件类型，第25位系统保留。

事件读取模式

在读事件时，可以选择读取模式。读取模式如下：

所有事件（LOS_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（LOS_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（LOS_WAITMODE_CLR）：LOS_WAITMODE_AND| LOS_WAITMODE_CLR或 LOS_WAITMODE_OR| LOS_WAITMODE_CLR 时表示读取成功后，对应事件类型位会自动清除。

运作机制

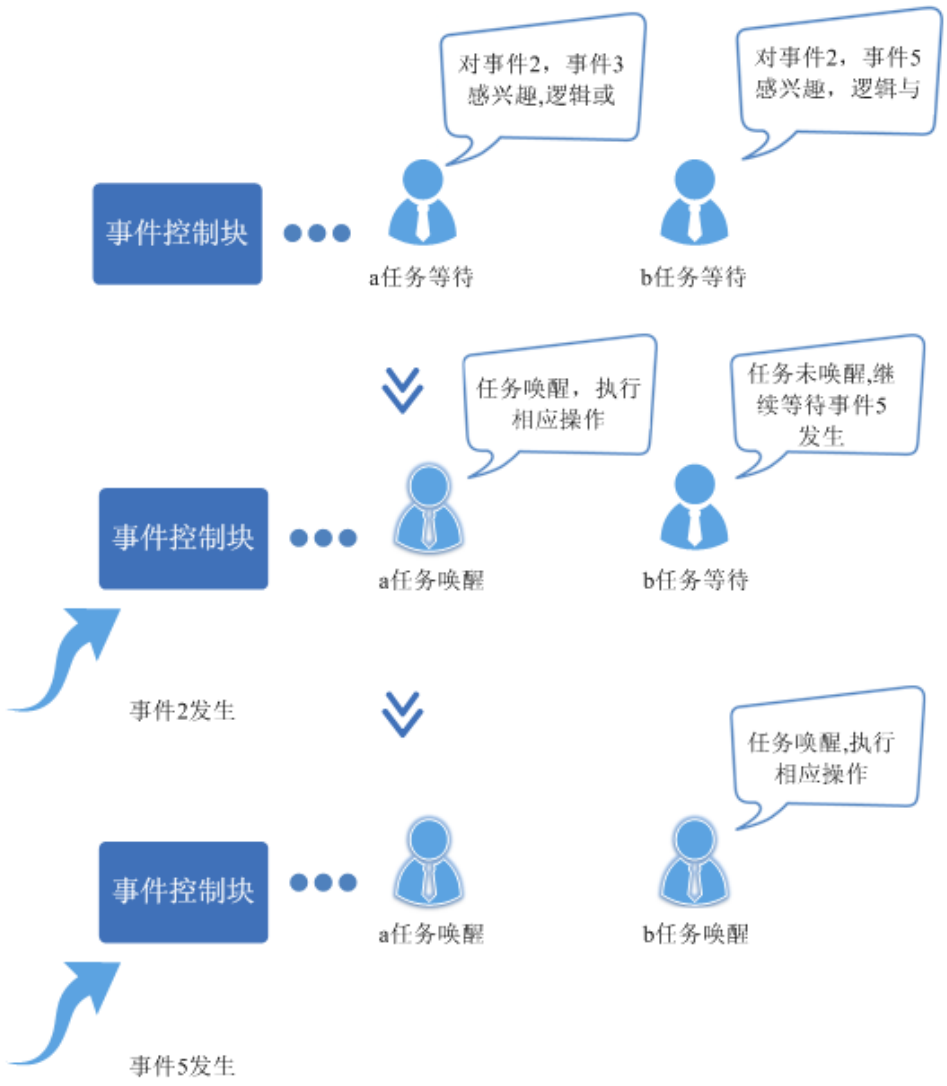
读事件时，可以根据入参事件掩码类型uwEventMask读取事件的单个或者多个事件类型。事件读取成功后，如果设置LOS_WAITMODE_CLR会清除已读取到的事件类型，反之不会清除已读到的事件类型，需显式清除。可以通过入参选择读取模式，读取事件掩码类型中所有事件还是读取事件掩码类型中任意事件。

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。

版权所有 © 华为技术有限公司

图 3-8 事件唤醒任务示意图



3.5.2 开发指导

使用场景

事件可应用于多种任务同步场合，能够一定程度替代信号量。

功能

Huawei LiteOS系统中的事件模块为用户提供下面几个接口。

功能分类	接口名	描述
事件初始化	LOS_EventInit	初始化一个事件控制块
读事件	LOS_EventRead	读取指定事件类型，超时时间为相对时间：单位为Tick

版权所有 © 华为技术有限公司

功能分类	接口名	描述
写事件	LOS_EventWrite	写指定的事件类型
清除事件	LOS_EventClear	清除指定的事件类型
校验事件掩码	LOS_EventPoll	根据用户传入的事件值、事件掩码及校验模式，返回用户传入的事件是否符合预期
销毁事件	LOS_EventDestroy	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如下：

1. 调用事件初始化LOS_EventInit接口，初始化事件等待队列。
2. 写事件LOS_EventWrite，配置事件掩码类型。
3. 读事件LOS_EventRead，可以选择读取模式。
4. 清除事件LOS_EventClear，清除指定的事件类型。

Event 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，时间清除

序号	定义	实际值	描述	参考解决方案
1	LOS_ERRNO_EVENT_SETBIT_INVALID	0x02001c00	事件ID的第25个bit不能设置为1，因为该位已经作为错误码使用	事件ID的第25bit置为0
2	LOS_ERRNO_EVENT_READ_TIMEOUT	0x02001c01	读超时	增加等待时间或者重新读取
3	LOS_ERRNO_EVENT_EVENTMASK_INVALID	0x02001c02	入参的事件ID是无效的	传入有效的事件ID参数
4	LOS_ERRNO_EVENT_READ_IN_INTERRUPT	0x02001c03	在中断中读取事件	启动新的任务来获取事件
5	LOS_ERRNO_EVENT_FLAGS_INVALID	0x02001c04	读取事件的mode无效	传入有效的mode参数

版权所有 © 华为技术有限公司

序号	定义	实际值	描述	参考解决方案
6	LOS_ERRNO_EVENT_READ_IN_LOCK	0x02001c05	任务锁住，不能读取事件	解锁任务，再读取事件
7	LOS_ERRNO_EVENT_PTR_NULL	0x02001c06	传入的参数为空指针	传入非空入参

错误码定义：错误码是一个32位的存储单元，31~24位表示错误等级，23~16位表示错误码标志，15~8位代表错误码所属模块，7~0位表示错误码序号，如下

#define LOS_ERRNO_OS_ERROR(MID, ERRNO) \

(LOS_ERRTYPE_ERROR | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | (ERRNO))

LOS_ERRTYPE_ERROR：Define critical OS errors

LOS_ERRNO_OS_ID：OS error code flag

MID：OS_MOUDLE_ID

LOS_MOD_EVENT：Event module ID

ERRNO：error ID number

例如：

#define LOS_ERRNO_EVENT_READ_IN_LOCK
LOS_ERRNO_OS_ERROR(LOS_MOD_EVENT, 0x05)

平台差异性

无。

3.5.3 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，但不能读操作。
- 在锁任务调度状态下，禁止任务阻塞与读事件。
- LOS_EventClear 入参值是：要清除的指定事件类型的反码（~uwEvents）。
- 事件掩码的第25位不能使用，原因是为了区别LOS_EventRead接口返回的是事件还是错误码。

3.5.4 编程实例

实例描述

示例中，任务Example_TaskEntry创建一个任务Example_Event，Example_Event读事件阻塞，Example_TaskEntry向该任务写事件。

1. 在任务Example_TaskEntry创建任务Example_Event，其中任务Example_Event优先级高于Example_TaskEntry。

版权所有 © 华为技术有限公司

2. 在任务Example_Event中读事件0x00000001，阻塞，发生任务切换，执行任务Example_TaskEntry。
3. 在任务Example_TaskEntry向任务Example_Event写事件0x00000001，发生任务切换，执行任务Example_Event。
4. Example_Event得以执行，直到任务结束。
5. Example_TaskEntry得以执行，直到任务结束。

编程示例

可以通过打印的先后顺序理解事件操作时伴随的任务切换。

代码实现如下：

```
#include "los_event.h"
#include "los_task.h"

/*任务PID*/
UINT32 g_TestTaskID01;

/*事件控制结构体*/
EVENT_CB_S  example_event;

/*等待的事件类型*/
#define event_wait 0x00000001

/*用例任务入口函数*/
VOID Example_Event()
{
    UINT32 uwRet;
    UINT32 uwEvent;

/*超时 等待方式读事件,超时时间为100 Tick
   若100 Tick 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",event_wait);

    uwEvent = LOS_EventRead(&example_event, event_wait, LOS_WAITMODE_AND, 100);
    if(uwEvent == event_wait)
    {
        printf("Example_Event,read event :0x%x\n",uwEvent);
    }
    else
        printf("Example_Event,read event timeout\n");
    return;
}

UINT32 Example_TaskEntry()
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S stTask1;

    /*事件初始化*/
    uwRet = LOS_EventInit(&example_event);
    if(uwRet != LOS_OK)
    {
        printf("init event failed .\n");
        return -1;
    }

    /*创建任务*/
    memset(&stTask1, 0, sizeof(TSK_INIT_PARAM_S));
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_Event;
    stTask1.pcName       = "EventTsk1";
    stTask1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = 5;
    uwRet = LOS_TaskCreate(&g_TestTaskID01, &stTask1);
    if(uwRet != LOS_OK)
```


版权所有 © 华为技术有限公司

```
    {
        printf("task create failed .\n");
        return LOS_NOK;
    }

    /*写用例任务等待的事件类型*/
    printf("Example_TaskEntry write event .\n");

    uwRet = LOS_EventWrite(&example_event, event_wait);
    if(uwRet != LOS_OK)
    {
        printf("event write failed .\n");
        return LOS_NOK;
    }

    /*清标志位*/
    printf("EventMask:%d\n",example_event.uwEventID);
    LOS_EventClear(&example_event, ~example_event.uwEventID);
    printf("EventMask:%d\n",example_event.uwEventID);

    /*删除任务*/
    uwRet = LOS_TaskDelete(g_TestTaskID01);
    if(uwRet != LOS_OK)
    {
        printf("task delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
Example_Event wait event 0x1
Example_TaskEntry write event .
Example_Event,read event :0x1
EventMask:1
EventMask:0
```

完整实例代码

sample_event.c

3.6 互斥锁

3.6.1 概述

基本概念

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，开锁或闭锁。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。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。

版权所有 © 华为技术有限公司

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。

Huawei LiteOS提供的互斥锁具有如下特点：

- 通过优先级继承算法，解决优先级翻转问题。

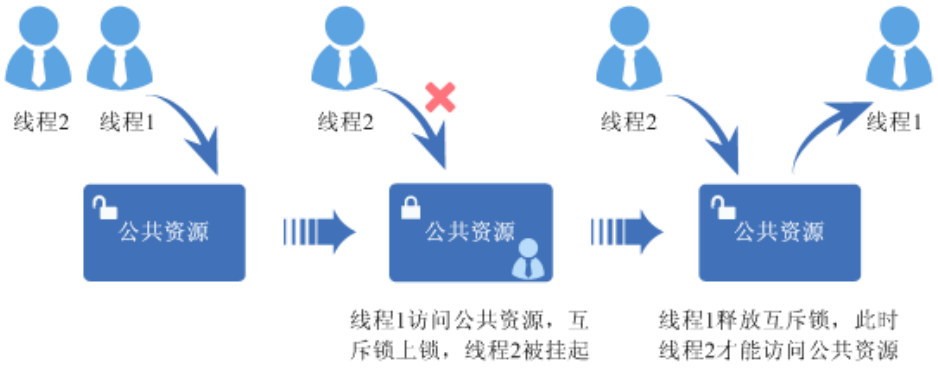
运作机制

互斥锁运作原理

多任务环境下会存在多个任务访问同一公共资源的场景，而有些公共资源是非共享的，需要任务进行独占式处理。互斥锁怎样来避免这种冲突呢？

用互斥锁处理非共享资源的同步访问时，如果有任务访问该资源，则互斥锁为加锁状态。此时其他任务如果想访问这个公共资源则会被阻塞，直到互斥锁被持有该锁的任务释放后，其他任务才能重新访问该公共资源，此时互斥锁再次上锁，如此确保同一时刻只有一个任务正在访问这个公共资源，保证了公共资源操作的完整性。

图 3-9 互斥锁运作示意图



3.6.2 开发指导

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

Huawei LiteOS 系统中的互斥锁模块为用户提供下面几种功能。

功能分类	接口名	描述
互斥锁的创建和删除	LOS_MuxCreate	创建互斥锁
	LOS_MuxDelete	删除指定的互斥锁
互斥锁的申请和释放	LOS_MuxPend	申请指定的互斥锁

版权所有 © 华为技术有限公司

功能分类	接口名	描述
	LOS_MuxPost	释放指定的互斥锁

开发流程

互斥锁典型场景的开发流程：

1. 创建互斥锁LOS_MuxCreate。
2. 申请互斥锁LOS_MuxPend。

   申请模式有三种：无阻塞模式、永久阻塞模式、定时阻塞模式。

   - 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
   - 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最高者继续执行。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
   - 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最高者继续执行。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
3. 释放互斥锁LOS_MuxPost。
   - 如果有任务阻塞于指定互斥锁，则唤醒最早被阻塞的任务，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。
4. 删除互斥锁LOS_MuxDelete。

互斥锁错误码

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

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_MUX_NO_MEMORY	0x02001d00	内存请求失败	减少互斥锁限制数量的上限
2	LOS_ERRNO_MUX_INVALID	0x02001d01	互斥锁不可用	传入有效的互斥锁的ID
3	LOS_ERRNO_MUX_PTR_NULL	0x02001d02	入参为空	确保入参可用
4	LOS_ERRNO_MUX_ALL_BUSY	0x02001d03	没有互斥锁可用	增加互斥锁限制数量的上限

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
5	LOS_ERRNO_MUX_UNAVAILABLE	0x02001d04	锁失败，因为锁被其他线程使用	等待其他线程解锁或者设置等待时间
6	LOS_ERRNO_MUX_PEND_INTERR	0x02001d05	在中断中使用互斥锁	在中断中禁止调用此接口
7	LOS_ERRNO_MUX_PEND_IN_LOCK	0x02001d06	任务调度没有使能，线程等待另一个线程释放锁	设置PEND为非阻塞模式或者使能任务调度
8	LOS_ERRNO_MUX_TIMEOUT	0x02001d07	互斥锁PEND超时	增加等待时间或者设置一直等待模式
9	LOS_ERRNO_MUX_OVERFLOW	0x02001d08	暂未使用，待扩展	无
10	LOS_ERRNO_MUX_PENDED	0x02001d09	删除正在使用的锁	等待解锁再删除锁
11	LOS_ERRNO_MUX_GET_COUNT_ERR	0x02001d0a	暂未使用，待扩展	无
12	LOS_ERRNO_MUX_REG_ERROR	0x02001d0b	暂未使用，待扩展	无

错误码定义：错误码是一个32位的存储单元，31~24位表示错误等级，23~16位表示错误码标志，15~8位代表错误码所属模块，7~0位表示错误码序号，如下

#define LOS_ERRNO_OS_ERROR(MID, ERRNO) \

(LOS_ERRTYPE_ERROR | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | (ERRNO))

LOS_ERRTYPE_ERROR：Define critical OS errors

LOS_ERRNO_OS_ID：OS error code flag

LOS_MOD_MUX：Mutex module ID

MID：OS_MOUDLE_ID

ERRNO：error ID number

例如：

LOS_ERRNO_MUX_TIMEOUT LOS_ERRNO_OS_ERROR(LOS_MOD_MUX, 0x07)

版权所有 © 华为技术有限公司

平台差异性

无。

3.6.3 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。
- Huawei LiteOS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LOS_TaskPriSet等接口更改持有互斥锁任务的优先级。

3.6.4 编程实例

实例描述

本实例实现如下流程。

1. 任务Example_TaskEntry创建一个互斥锁，锁任务调度，创建两个任务Example_MutexTask1、Example_MutexTask2,Example_MutexTask2优先级高于Example_MutexTask1，解锁任务调度。
2. Example_MutexTask2被调度，永久申请互斥锁，然后任务休眠100Tick，Example_MutexTask2挂起，Example_MutexTask1被唤醒。
3. Example_MutexTask1申请互斥锁，等待时间为10Tick，因互斥锁仍被Example_MutexTask2持有，Example_MutexTask1挂起，10Tick后未拿到互斥锁，Example_MutexTask1被唤醒，试图以永久等待申请互斥锁，Example_MutexTask1挂起。
4. 100Tick后Example_MutexTask2唤醒， 释放互斥锁后，Example_MutexTask1被调度运行，最后释放互斥锁。
5. Example_MutexTask1执行完，300Tick后任务Example_TaskEntry被调度运行，删除互斥锁。

编程示例

前提条件：

- 在los_config.h中，将OS_INCLUDE_MUX配置项打开。
- 配好OS_MUX_MAX_SUPPORT_NUM最大的互斥锁个数。

代码实现如下：

```
#include "los_mux.h"
#include "los_task.h"

/*互斥锁句柄ID*/
MUX_HANDLE_T  g_Testmux01;
/*任务PID*/
UINT32 g_TestTaskID01;
UINT32 g_TestTaskID02;

VOID Example_MutexTask1()
```


版权所有 © 华为技术有限公司

```
{
    UINT32 uwRet;

    printf("task1 try to get mutex, wait 10 Tick.\n");
    /*申请互斥锁*/
    uwRet=LOS_MuxPend(g_Testmux01, 10);

    if(uwRet == LOS_OK)
    {
        printf("task1 get mutex g_Testmux01.\n");
        /*释放互斥锁*/
        LOS_MuxPost(g_Testmux01);
        return;
    }
    else if(uwRet == LOS_ERRNO_MUX_TIMEOUT )
    {
            printf("task1 timeout and try to get  mutex, wait forever.\n");
            /*申请互斥锁*/
            uwRet = LOS_MuxPend(g_Testmux01, LOS_WAIT_FOREVER);
            if(uwRet == LOS_OK)
            {
                printf("task1 wait forever,get mutex g_Testmux01.\n");
                /*释放互斥锁*/
                LOS_MuxPost(g_Testmux01);
                return;
            }
    }
    return;
}

VOID Example_MutexTask2()
{
    UINT32 uwRet;

    printf("task2 try to get mutex, wait forever.\n");
    /*申请互斥锁*/
    uwRet=LOS_MuxPend(g_Testmux01, LOS_WAIT_FOREVER);

    printf("task2 get mutex g_Testmux01 and suspend 100 Tick.\n");

    /*任务休眠100 Tick*/
    LOS_TaskDelay(100);

    printf("task2 resumed and post the g_Testmux01\n");
    /*释放互斥锁*/
    LOS_MuxPost(g_Testmux01);
    return;

}

UINT32 Example_TaskEntry()
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S stTask1;
    TSK_INIT_PARAM_S stTask2;

    /*创建互斥锁*/
    LOS_MuxCreate(&g_Testmux01);

    /*锁任务调度*/
    LOS_TaskLock();

    /*创建任务1*/
    memset(&stTask1, 0, sizeof(TSK_INIT_PARAM_S));
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_MutexTask1;
    stTask1.pcName       = "MutexTsk1";
    stTask1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = 5;
    uwRet = LOS_TaskCreate(&g_TestTaskID01, &stTask1);
```


版权所有 © 华为技术有限公司

```
    if(uwRet != LOS_OK)
    {
        printf("task1 create failed .\n");
        return LOS_NOK;
    }

    /*创建任务2*/
    memset(&stTask2, 0, sizeof(TSK_INIT_PARAM_S));
    stTask2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_MutexTask2;
    stTask2.pcName       = "MutexTsk2";
    stTask2.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask2.usTaskPrio   = 4;
    uwRet = LOS_TaskCreate(&g_TestTaskID02, &stTask2);
    if(uwRet != LOS_OK)
    {
        printf("task2 create failed .\n");
        return LOS_NOK;
    }

    /*解锁任务调度*/
    LOS_TaskUnlock();
    /*任务休眠300 Tick*/
    LOS_TaskDelay(300);

    /*删除互斥锁*/
    LOS_MuxDelete(g_Testmux01);

    /*删除任务1*/
    uwRet = LOS_TaskDelete(g_TestTaskID01);
    if(uwRet != LOS_OK)
    {
        printf("task1 delete failed .\n");
        return LOS_NOK;
    }
    /*删除任务2*/
    uwRet = LOS_TaskDelete(g_TestTaskID02);
    if(uwRet != LOS_OK)
    {
        printf("task2 delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
task2 try to get  mutex, wait forever.
task2 get mutex g_Testmux01 and suspend 100 ticks.
task1 try to get  mutex, wait 10 ticks.
task1 timeout and try to get  mutex, wait forever.
task2 resumed and post the g_Testmux01
task1 wait forever,get mutex g_Testmux01.
```

完整实例代码

sample_mutex.c

3.7 信号量

版权所有 © 华为技术有限公司

3.7.1 概述

基本概念

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。其值的含义分两种情况：

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

运作机制

信号量控制块

```
/**
 * @ingroup los_sem
 * Semaphore control structure.
 */
typedef struct
{
    UINT8           usSemStat;          /**是否使用标志位*/
    UINT16          uwSemCount;         /**信号量索引号*/
    UINT32          usSemID;            /**信号量计数*/
    LOS_DL_LIST     stSemList;          /**挂接阻塞于该信号量的任务*/
}SEM_CB_S;
```

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，详见第十章 配置参考），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，若其计数器值大于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。

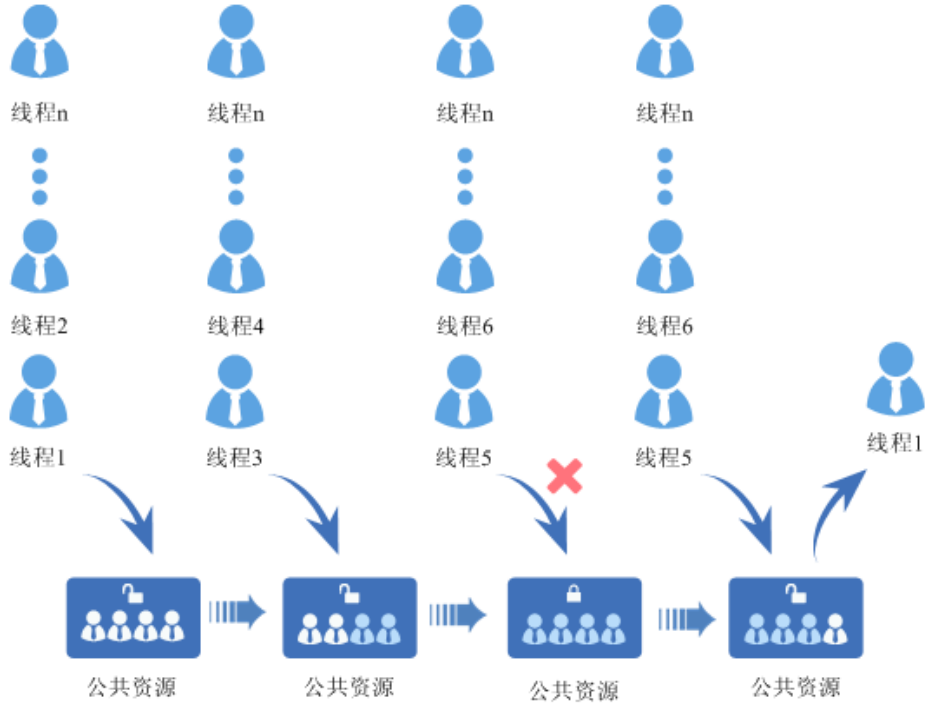
信号量释放，若没有任务等待该信号量，则直接将计数器加1返回。否则唤醒该信号量等待任务队列上的第一个任务。

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。

版权所有 © 华为技术有限公司

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最大任务数目。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。

图 3-10 信号量运作示意图



3.7.2 开发指导

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，也能方便的用于任务与任务，中断与任务的同步中。

功能

Huawei LiteOS 系统中的信号量模块为用户提供下面几种功能。

功能分类	接口名	描述
信号量的创建和删除	LOS_SemCreate	创建信号量
	LOS_SemDelete	删除指定的信号量
信号量的申请和释放	LOS_SemPend	申请指定的信号量
	LOS_SemPost	释放指定的信号量

开发流程

信号量的开发典型流程：

版权所有 © 华为技术有限公司

1. 创建信号量LOS_SemCreate。
2. 申请信号量LOS_SemPend。

   信号量有三种申请模式：无阻塞模式、永久阻塞模式、定时阻塞模式

   - 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
   - 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最高者继续执行。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
   - 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最高者继续执行。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
3. 释放信号量LOS_SemPost。
   - 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
4. 删除信号量LOS_SemDelete。

信号量错误码

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。

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_SEM_NO_MEMORY	0x02000700	内存空间不足	分配更大的内存分区
2	LOS_ERRNO_SEM_INVALID	0x02000701	非法传参	改变传数为合法值
3	LOS_ERRNO_SEM_PTR_NULL	0x02000702	传入空指针	传入合法指针
4	LOS_ERRNO_SEM_ALL_BUSY	0x02000703	信号量控制块不可用	释放资源信号量资源
5	LOS_ERRNO_SEM_UNAVAILABLE	0x02000704	定时时间非法	传入正确的定时时间
6	LOS_ERRNO_SEM_PEND_INTERR	0x02000705	中断期间非法调用LOS_SemPend	中断期间禁止调用LOS_SemPend
7	LOS_ERRNO_SEM_PEND_IN_LOCK	0x02000706	任务被锁，无法获得信号量	在任务被锁时，不能调用LOS_SemPend
8	LOS_ERRNO_SEM_TIMEOUT	0x02000707	获取信号量时间超时	将时间设置在合理范围内

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
9	LOS_ERRNO_SEM_OVERFLOW	0x02000708	信号量允许pend次数超过最大值	传入合法的值
10	LOS_ERRNO_SEM_PENDED	0x02000709	等待信号量的任务队列不为空	唤醒所有等待该型号量的任务后删除该信号量

错误码定义：错误码是一个32位的存储单元，31~24位表示错误等级，23~16位表示错误码标志，15~8位代表错误码所属模块，7~0位表示错误码序号，如下

```
#define LOS_ERRNO_OS_NORMAL(MID,ERRNO)  \
(LOS_ERRTYPE_NORMAL | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | (ERRNO))
LOS_ERRTYPE_NORMAL : Define the error level as critical
LOS_ERRNO_OS_ID : OS error code flag.
MID：OS_MOUDLE_ID
ERRNO：error ID number
```

例如：

```
LOS_ERRNO_SEM_NO_MEMORY         LOS_ERRNO_OS_ERROR(LOS_MOD_SEM, 0x00))
```

注意

错误码序号 0x16、0x1c，未被定义，不可用。

平台差异性

无。

3.7.3 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。

3.7.4 编程实例

实例描述

本实例实现如下功能;

1. 测试任务Example_TaskEntry创建一个信号量，锁任务调度，创建两个任务Example_SemTask1、Example_SemTask2,Example_SemTask2优先级高于Example_SemTask1，两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_TaskEntry释放信号量。
2. Example_SemTask2得到信号量，被调度，然后任务休眠20Tick，Example_SemTask2延迟，Example_SemTask1被唤醒。
3. Example_SemTask1定时阻塞模式申请信号量，等待时间为10Tick，因信号量仍被Example_SemTask2持有，Example_SemTask1挂起，10Tick后仍未得到信号量，

版权所有 © 华为技术有限公司

Example_SemTask1被唤醒，试图以永久阻塞模式申请信号量，Example_SemTask1挂起。

4. 20Tick后Example_SemTask2唤醒， 释放信号量后，Example_SemTask1得到信号量被调度运行，最后释放信号量。
5. Example_SemTask1执行完，40Tick后任务Example_TaskEntry被唤醒，执行删除信号量，删除两个任务。

编程示例

前提条件：

- 在los_config.h中，将LOSCFG_BASE_IPC_SEM配置为YES。
- 配置用户定义的LOSCFG_BASE_IPC_SEM_LIMIT最大的信号量数，如1024。

代码实现如下：

```
#include "los_sem.h"


/*任务PID*/
static UINT32 g_TestTaskID01,g_TestTaskID02;
/*测试任务优先级*/
#define TASK_PRIO_TEST  5
/*信号量结构体ID*/
static SEM_HANDLE_T g_usSemID;


VOID Example_SemTask1(void)
{
    UINT32 uwRet;

    printf("Example_SemTask1 try get sem g_usSemID ,timeout 10 ticks.\n");
    /*定时阻塞模式申请信号量，定时时间为10Tick*/
    uwRet = LOS_SemPend(g_usSemID, 10);

    /*申请到信号量*/
    if(LOS_OK == uwRet)
    {
         LOS_SemPost(g_usSemID);
         return;
    }
    /*定时时间到，未申请到信号量*/
    if(LOS_ERRNO_SEM_TIMEOUT == uwRet)
    {
        printf("Example_SemTask1 timeout and try get sem g_usSemID wait forever.\n");
        /*永久阻塞模式申请信号量*/
        uwRet = LOS_SemPend(g_usSemID, LOS_WAIT_FOREVER);
        printf("Example_SemTask1 wait_forever and get sem g_usSemID .\n");
        if(LOS_OK == uwRet)
        {
            LOS_SemPost(g_usSemID);
            return;
        }
    }
    return;

}
 VOID   Example_SemTask2(void)
{
    UINT32 uwRet;
    printf("Example_SemTask2 try get sem g_usSemID wait forever.\n");
    /*永久阻塞模式申请信号量*/
    uwRet = LOS_SemPend(g_usSemID, LOS_WAIT_FOREVER);

    if(LOS_OK == uwRet)
```


版权所有 © 华为技术有限公司

```
    printf("Example_SemTask2 get sem g_usSemID and then delay 20ticks .\n");

    /*任务休眠20 Tick*/
    LOS_TaskDelay(20);

    printf("Example_SemTask2 post sem g_usSemID .\n");
    /*释放信号量*/
    LOS_SemPost(g_usSemID);

    return;

}
UINT32 Example_TaskEntry()
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S stTask1;
    TSK_INIT_PARAM_S stTask2;

   /*创建信号量*/
    LOS_SemCreate(0,&g_usSemID);

    /*锁任务调度*/
    LOS_TaskLock();

    /*创建任务1*/
    memset(&stTask1, 0, sizeof(TSK_INIT_PARAM_S));
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_SemTask1;
    stTask1.pcName       = "MutexTsk1";
    stTask1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = TASK_PRIO_TEST;
    uwRet = LOS_TaskCreate(&g_TestTaskID01, &stTask1);
    if(uwRet != LOS_OK)
    {
        printf("task1 create failed .\n");
        return LOS_NOK;
    }

    /*创建任务2*/
    memset(&stTask2, 0, sizeof(TSK_INIT_PARAM_S));
    stTask2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_SemTask2;
    stTask2.pcName       = "MutexTsk2";
    stTask2.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask2.usTaskPrio   = (TASK_PRIO_TEST - 1);
    uwRet = LOS_TaskCreate(&g_TestTaskID02, &stTask2);
    if(uwRet != LOS_OK)
    {
        printf("task2 create failed .\n");
        return LOS_NOK;
    }

    /*解锁任务调度*/
    LOS_TaskUnlock();

    uwRet = LOS_SemPost(g_usSemID);

    /*任务休眠40 Tick*/
    LOS_TaskDelay(40);

    /*删除信号量*/
    LOS_SemDelete(g_usSemID);

    /*删除任务1*/
    uwRet = LOS_TaskDelete(g_TestTaskID01);
    if(uwRet != LOS_OK)
    {
        printf("task1 delete failed .\n");
        return LOS_NOK;
    }
    /*删除任务2*/
```


版权所有 © 华为技术有限公司

```
    uwRet = LOS_TaskDelete(g_TestTaskID02);
    if(uwRet != LOS_OK)
    {
        printf("task2 delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
Example_SemTask2 try get sem g_usSemID wait forever.
Example_SemTask1 try get sem g_usSemID ,timeout 10 ticks.
Example_SemTask2 get sem g_usSemID and then delay 20ticks .
Example_SemTask1 timeout and try get sem g_usSemID wait forever.
Example_SemTask2 post sem g_usSemID .
Example_SemTask1 wait_forever and get sem g_usSemID .
```

完整实例代码

sample_sem.c

3.8 时间管理

3.8.1 概述

基本概念

时间管理以系统时钟为基础。时间管理提供给应用程序所有和时间有关的服务。

系统时钟是由定时/计数器产生的输出脉冲触发中断而产生的，一般定义为整数或长整数。输出脉冲的周期叫做一个“时钟滴答”。系统时钟也称为时标或者Tick。一个Tick的时长可以静态配置。

用户是以秒、毫秒为单位计时，而芯片CPU的计时是以Tick为单位的，当用户需要对系统操作时，例如任务挂起、延时等，输入秒为单位的数值，此时需要时间管理模块对二者进行转换。

Tick与秒之间的对应关系可以配置。

Huawei LiteOS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。

相关概念

- 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

Tick是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，由用户配置。

版权所有 © 华为技术有限公司

3.8.2 开发指导

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。

功能

Huawei LiteOS系统中的时间管理主要提供以下两种功能：

- 时间转换：根据主频实现CPU Tick数到毫秒、微秒的转换。
- 时间统计：获取系统Tick数。

功能分类	接口名	描述
时间转换	LOS_MS2Tick	毫秒转换成Tick
	LOS_Tick2MS	Tick转化为毫秒
时间统计	LOS_CyclePerTickGet	每个Tick多少Cycle数
	LOS_TickCountGet	获取当前的Tick数

时间管理错误码

时间转换存在出错的可能性，需要返回对应的错误码，以便快速定位错误原因。

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_SYS_PTR_NULL	0x02000010	入参指针为空	检查入参，传入非空入参
2	LOS_ERRNO_SYS_CLOCK_INVAILD	0x02000011	无效的系统时钟配置	在los_config.h配置有效的时钟
3	LOS_ERRNO_SYS_MAXNUMOFCORES_IS_INVAILD	0x02000012	错误码暂时没有使用	错误码暂时没有使用
4	LOS_ERRNO_SYS_PERIERRCOREID	0x02000013	错误码暂时没有使用	错误码暂时没有使用
5	LOS_ERRNO_SYS_HOOK_IS_FULL	0x02000014	错误码暂时没有使用	错误码暂时没有使用

开发流程

时间管理的典型开发流程：

1. 确认配置项LOSCFG_BASE_CORE_TICK_HW_TIME为YES开启状态。

版权所有 © 华为技术有限公司

   - 在los_config.h中配置每秒的Tick数 LOSCFG_BASE_CORE_TICK_PER_SECOND；
2. 调用时钟转换接口。
3. 获取系统Tick数完成时间统计。
   - 通过LOS_TickCountGet获取全局g_ullTickCount。

3.8.3 注意事项

- 获取系统Tick数需要在系统时钟使能之后。
- 时间管理不是单独的功能模块，依赖于los_config.h中的OS_SYS_CLOCK和LOSCFG_BASE_CORE_TICK_PER_SECOND两个配置选项。
- 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。

3.8.4 编程实例

实例描述

在下面的例子中，介绍了时间管理的基本方法，包括：

1. 时间转换：将毫秒数转换为Tick数，或将Tick数转换为毫秒数。
2. 时间统计和时间延迟：统计每秒的Cycle数、Tick数和延迟后的Tick数。

编程示例

前提条件：

- 配好LOSCFG_BASE_CORE_TICK_PER_SECOND每秒的Tick数。
- 配好OS_SYS_CLOCK 系统时钟，单位: Hz。

时间转换：

```
VOID Example_TransformTime(VOID)
{
    UINT32 uwMs;
    UINT32 uwTick;

    uwTick = LOS_MS2Tick(10000);//10000 ms数转换为Tick数
     printf("uwTick = %d \n",uwTick);
    uwMs= LOS_Tick2MS(100);//100 Tick数转换为ms数
        printf("uwMs = %d \n",uwMs);
}
```

时间统计和时间延迟：

```
VOID Example_GetTime(VOID)
{
    UINT32 uwcyclePerTick;
    UINT64 uwTickCount;

    uwcyclePerTick = LOS_CyclePerTickGet();//每个Tick多少Cycle数
    if(0 != uwcyclePerTick)
    {
        dprintf("LOS_CyclePerTickGet = %d \n", uwcyclePerTick);
    }

    uwTickCount = LOS_TickCountGet();//获取Tick数
    if(0 != uwTickCount)
```


版权所有 © 华为技术有限公司

```
    {
        dprintf("LOS_TickCountGet = %d \n",  (UINT32) uwTickCount);
    }
    LOS_TaskDelay(200);//延迟200 Tick
    uwTickCount = LOS_TickCountGet();
    if(0 != uwTickCount)
    {
        dprintf("LOS_TickCountGet after delay = %d \n", (UINT32)uwTickCount);
    }
}
```

结果验证

编译运行得到的结果为：

时间转换：

```
tick = 1000
uwMs = 1000
```

时间统计和时间延迟：

```
LOS_CyclePerTickGet = 495000
LOS_TickCountGet = 1
LOS_TickCountGet after delay = 201
```

完整实例代码

sample_time.c

3.9 软件定时器

3.9.1 概述

基本概念

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，Huawei LiteOS操作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。

软件定时器功能上支持：

- 静态裁剪：能通过宏关闭软件定时器功能。
- 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。
- 软件定时器剩余Tick数获取

版权所有 © 华为技术有限公司

运作机制

软件定时器是系统资源，在模块初始化的时候已经分配了一块连续的内存，系统支持的最大定时器个数由los_config.h中的LOSCFG_BASE_CORE_SWTMR_LIMIT宏配置。

软件定时器使用了系统的一个队列和一个任务资源，软件定时器的触发遵循队列规则，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。

软件定时器以Tick为基本计时单位，当用户创建并启动一个软件定时器时，Huawei LiteOS会根据当前系统Tick时间及用户设置的定时间隔确定该定时器的到期Tick时间，并将该定时器控制结构挂入计时全局链表。

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全局链表，看是否有定时器超时，若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（优先级为最高）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。

定时器状态

- OS_SWTMR_STATUS_UNUSED（未使用）

系统在定时器模块初始化的时候将系统中所有定时器资源初始化成该状态。

- OS_SWTMR_STATUS_CREATED（创建未启动/停止）

在未使用状态下调用LOS_SwtmrCreate接口或者启动后调用LOS_SwtmrStop接口后，定时器将变成该状态。

- OS_SWTMR_STATUS_TICKING（计数）

在定时器创建后调用LOS_SwtmrStart接口，定时器将变成该状态，表示定时器运行时的状态。

定时器模式

Huawei LiteOS的软件定时器提供二类定时器机制：

- 第一类是单次触发定时器，这类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。
- 第二类是周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动地停止定时器，否则将永远持续执行下去。

3.9.2 开发指导

使用场景

- 创建一个单次触发的定时器，超时后执行用户自定义的回调函数。
- 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

Huawei LiteOS系统中的软件定时器模块为用户提供下面几种功能，下面具体的API详见软件定时器对外接口手册。

版权所有 © 华为技术有限公司

表 3-1

功能分类	接口名	描述
创建、删除定时器	LOS_SwtmrCreate	创建定时器
	LOS_SwtmrDelete	删除定时器
启动、停止定时器	LOS_SwtmrStart	启动定时器
	LOS_SwtmrStop	停止定时器
获得软件定时器剩余Tick数	LOS_SwtmrTimeGet	获得软件定时器剩余Tick数

开发流程

软件定时器的典型开发流程：

1. 配置软件定时器。
   - 确认配置项LOSCFG_BASE_CORE_SWTMR和LOSCFG_BASE_IPC_QUEUE为YES打开状态；
   - 配置LOSCFG_BASE_CORE_SWTMR_LIMIT最大支持的软件定时器数；
   - 配置OS_SWTMR_HANDLE_QUEUE_SIZE软件定时器队列最大长度；
2. 创建定时器LOS_SwtmrCreate。
   - 创建一个指定计时时长、指定超时处理函数、指定触发模式的软件定时器；
   - 返回函数运行结果，成功或失败；
3. 启动定时器LOS_SwtmrStart。
4. 获得软件定时器剩余Tick数LOS_SwtmrTimeGet。
5. 停止定时器LOS_SwtmrStop。
6. 删除定时器LOS_SwtmrDelete。

软件定时器错误码

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

序号	定义	实际数值	描述	参考解决方案
1	LOS_ERRNO_SWTMR_PTR_NULL	0x02000300	软件定时器回调函数为空	定义软件定时器回调函数
2	LOS_ERRNO_SWTMR_INTERVAL_NOT_SUITED	0x02000301	软件定时器间隔时间为0	重新定义间隔时间
3	LOS_ERRNO_SWTMR_MODE_INVALID	0x02000302	不正确的软件定时器模式	确认软件定时器模式，范围为[0,2]

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
4	LOS_ERRNO_SWTMR_RET_PTR_NULL	0x02000303	软件定时器ID指针入参为NULL	定义ID变量，传入指针
5	LOS_ERRNO_SWTMR_MAXSIZE	0x02000304	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
6	LOS_ERRNO_SWTMR_ID_INVALID	0x02000305	不正确的软件定时器ID入参	确保入参合法
7	LOS_ERRNO_SWTMR_NOT_CREATED	0x02000306	软件定时器未创建	创建软件定时器
8	LOS_ERRNO_SWTMR_NO_MEMORY	0x02000307	软件定时器链表创建内存不足	申请一块足够大的内存供软件定时器使用
9	LOS_ERRNO_SWTMR_MAXSIZE_INVALID	0x02000308	不正确的软件定时器个数最大值	重新定义该值
10	LOS_ERRNO_SWTMR_HWI_ACTIVE	0x02000309	在中断中使用定时器	修改源代码确保不在中断中使用
11	LOS_ERRNO_SWTMR_HANDLER_POOL_NO_MEM	0x0200030a	membox内存不足	扩大内存
12	LOS_ERRNO_SWTMR_QUEUE_CREATE_FAILED	0x0200030b	软件定时器队列创建失败	检查用以创建队列的内存是否足够
13	LOS_ERRNO_SWTMR_TASK_CREATE_FAILED	0x0200030c	软件定时器任务创建失败	检查用以创建软件定时器任务的内存是否足够并重新创建
14	LOS_ERRNO_SWTMR_NOT_STARTED	0x0200030d	未启动软件定时器	启动软件定时器
15	LOS_ERRNO_SWTMR_STATUS_INVALID	0x0200030e	不正确的软件定时器状态	检查确认软件定时器状态
16	LOS_ERRNO_SWTMR_SORTLIST_NULL	null	暂无	该错误码暂不使用

版权所有 © 华为技术有限公司

序号	定义	实际数值	描述	参考解决方案
17	LOS_ERRNO_SWTMR_TICK_PTR_NULL	0x02000310	用以获取软件定时器超时tick数的入参指针为NULL	创建一个有效的变量

错误码定义：错误码是一个32位的存储单元，31~24位表示错误等级，23~16位表示错误码标志，15~8位代表错误码所属模块，7~0位表示错误码序号，如下

```
#define LOS_ERRNO_OS_NORMAL(MID,ERRNO)  \
(LOS_ERRTYPE_NORMAL | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | (ERRNO))
LOS_ERRTYPE_NORMAL : Define the error level as critical
LOS_ERRNO_OS_ID : OS error code flag.
MID：OS_MOUDLE_ID
ERRNO：error ID number
```

例如：

```
#define LOS_ERRNO_SWTMR_PTR_NULL  \
LOS_ERRNO_OS_ERROR(LOS_MOD_SWTMR, 0x00)
```

3.9.3 注意事项

- 软件定时器的回调函数中不要做过多操作，不要使用可能引起任务挂起或者阻塞的接口或操作。
- 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，而并非是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。

3.9.4 编程实例

实例描述

在下面的例子中，演示如下功能：

1. 软件定时器创建、启动、删除、暂停、重启操作。
2. 单次软件定时器，周期软件定时器使用方法。
3.

编程示例

前提条件：

- 在los_config.h中，将LOSCFG_BASE_CORE_SWTMR配置项打开。
- 配置好LOSCFG_BASE_CORE_SWTMR_LIMIT最大支持的软件定时器数。

版权所有 © 华为技术有限公司

- 配置好OS_SWTMR_HANDLE_QUEUE_SIZE软件定时器队列最大长度。

代码实现如下：

```
void Timer1_Callback(uint32_t arg);  // callback fuction

void Timer2_Callback(uint32_t arg);

UINT32 g_timercount1 = 0;
UINT32 g_timercount2 = 0;

void Timer1_Callback(uint32_t arg)//回调函数1
{
  unsigned long tick_last1;
  g_timercount1++;
  tick_last1=(UINT32)LOS_TickCountGet();//获取当前Tick数
  dprintf("g_timercount1=%d\n",g_timercount1);
  dprintf("tick_last1=%d\n",tick_last1);

}

void Timer2_Callback(uint32_t arg)//回调函数2
{
  unsigned long tick_last2;
  tick_last2=(UINT32)LOS_TickCountGet();
  g_timercount2 ++;
  dprintf("g_timercount2=%d\n",g_timercount2);
  dprintf("tick_last2=%d\n",tick_last2);
}

void Timer_example (void)  {
  UINT16 id1;
  UINT16 id2;// timer id
  UINT32 uwTick;
 /*创建单次软件定时器，Tick数为1000，启动到1000Tick数时执行回调函数1 */
  LOS_SwtmrCreate (1000, LOS_SWTMR_MODE_ONCE,Timer1_Callback,&id1,1);
 /*创建周期性软件定时器，每100Tick数执行回调函数2 */
  LOS_SwtmrCreate(100,LOS_SWTMR_MODE_PERIOD,Timer2_Callback,&id2,1);
  dprintf("create Timer1 success\n");

  LOS_SwtmrStart (id1); //启动单次软件定时器
  dprintf("start Timer1 sucess\n");

  LOS_TaskDelay(200);//延时200Tick数
  LOS_SwtmrTimeGet(id1,&uwTick);//获得单次软件定时器剩余Tick数
  dprintf("uwTick =%d\n",uwTick);

  LOS_SwtmrStop(id1);//停止软件定时器
  dprintf("stop Timer1 sucess\n");

  LOS_SwtmrStart(id1);
  LOS_TaskDelay(1000);
  LOS_SwtmrDelete(id1);//删除软件定时器
  dprintf("delete Timer1 sucess\n");

  LOS_SwtmrStart(id2);//启动周期性软件定时器
  dprintf("start Timer2\n");

  LOS_TaskDelay(1000);
  LOS_SwtmrStop(id2);
  LOS_SwtmrDelete(id2);
 }
```

结果验证

得到的结果为：

版权所有 © 华为技术有限公司

```
version Huawei LiteOSTDV100BD01C00B038
build data : Jul 27 2015 17:00:59

**********************************
dist:1

 --- Test start---
create Timer1 success
start Timer1 sucess
uwTick =800
stop Timer1 sucess
g_timercount1=1
tick_last1=1201
delete Timer1 sucess
start Timer2
g_timercount2=1
tick_last2=1301
g_timercount2=2
tick_last2=1401
g_timercount2=3
tick_last2=1501
g_timercount2=4
tick_last2=1601
g_timercount2=5
tick_last2=1701
g_timercount2=6
tick_last2=1801
g_timercount2=7
tick_last2=1901
g_timercount2=8
tick_last2=2001
g_timercount2=9
tick_last2=2101
g_timercount2=10
tick_last2=2201

 --- Test End ---
```

完整实例代码

sample_Timer.c

3.10 错误处理

3.10.1 概述

基本概念

错误处理指用户代码发生错误时，系统调用错误处理模块的接口函数，完成上报错误信息，并调用用户自己的钩子函数，进行特定的处理。

错误处理模块实现OS内部错误码记录功能。OS内部错误码无法通过接口返回，通常会上报错误处理模块进行记录。用户可以通过挂接错误处理的钩子函数，进行特定的处理。如果OS上报的错误是致命错误，系统会进行异常流程接管，从而可以保存现场以便定位问题。

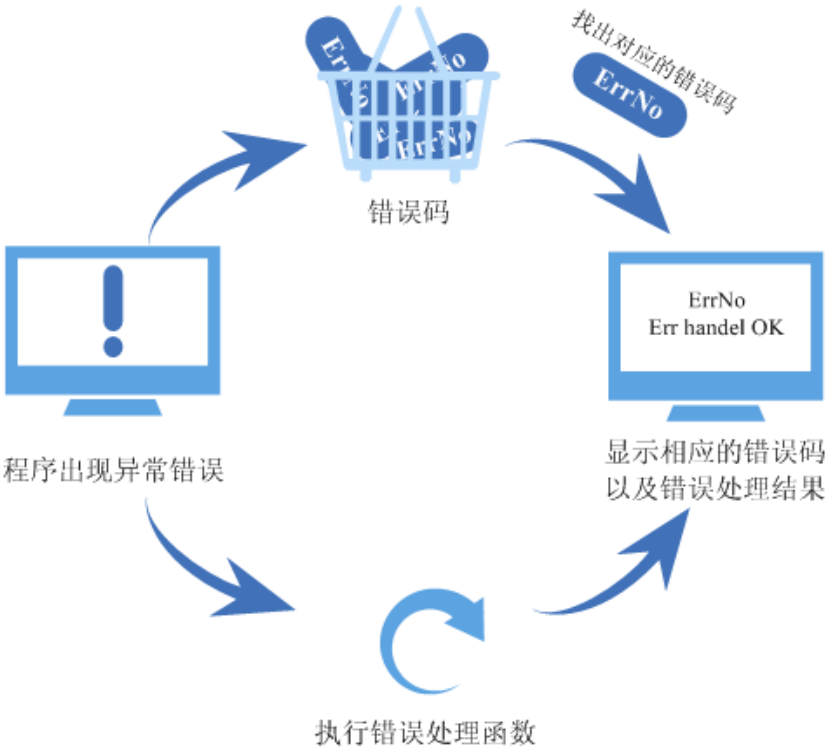
通过错误处理，我们可以把用户在程序中的非法输入进行控制和提示，以防程序崩溃。

运作机制

错误处理是一种机制，用于处理异常状况。通过错误处理，我们可以将用户在程序中的非法输入进行控制和提示，以防程序崩溃。当程序出现异常错误的时候，会显示相应的错误码用于提示用户。此外，如果有相应的错误处理程序，则会执行这个程序，以防程序崩溃。

版权所有 © 华为技术有限公司

图 3-11 错误处理示意图



3.10.2 开发指导

功能

错误处理模块为用户提供下面几种功能。

功能分类	接口名	描述
错误处理	LOS_ErrHandle	根据错误处理函数来对错误进行处理

3.10.3 注意事项

无。

3.10.4 编程实例

实例描述

在下面的例子中，演示如下功能：

1. 执行错误处理函数。

编程示例

代码实现如下：

版权所有 © 华为技术有限公司

```
extern USER_ERR_FUNC_S      g_stUserErrFunc;
void *err_handler(CHAR *pcFileName,UINT32 uwLineNo,
UINT32 uwErrorNo,UINT32 uwParaLen,VOID  *pPara)
{
    printf("err handel ok\n");
}
UINT32 Example_ErrCaseEntry(VOID)
{
    /*执行错误处理函数*/
    LOS_ErrHandle(NULL, 0,0,0, NULL);
    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
 --- Test start---
seterrno success
errno address:0x830d5768
err handel ok

 --- Test End ---
```

完整实例代码

sample_err.c

3.11 双向链表

3.11.1 概述

基本概念

双向链表是指含有往前和往后两个方向的链表，即每个结点中除存放下一个节点指针外，还增加一个指向其前一个节点的指针。其头指针head是唯一确定的。

从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点，这种数据结构形式使得双向链表在查找时更加方便，特别是大量数据的遍历。由于双向链表具有对称性，能方便地完成各种插入、删除等操作，但需要注意前后方向的操作。

3.11.2 开发指导

功能

Huawei LiteOS系统中的双向链表模块为用户提供下面几个接口。

功能分类	接口名	描述
初始化链表	LOS_ListInit	对链表进行初始化
增加节点	LOS_ListAdd	将新节点添加到链表中。

版权所有 © 华为技术有限公司

功能分类	接口名	描述
在链表尾端插入节点	LOS_ListTailInsert	将节点插入到双向链表尾端
删除节点	LOS_ListDelete	将指定的节点从链表中删除
判断双向链表是否为空	LOS_ListEmpty	判断链表是否为空
删除节点并初始化链表	LOS_ListDelInit	将指定的节点从链表中删除 使用该节点初始化链表

开发流程

双向链表的典型开发流程：

1. 调用LOS_ListInit初始双向链表。
2. 调用LOS_ListAdd向链表中增加节点。
3. 调用LOS_ListTailInsert向链表尾部插入节点。
4. 调用LOS_ListDelete删除指定节点。
5. 调用LOS_ListEmpty判断链表是否为空。
6. 调用LOS_ListDelInit删除指定节点并以此节点初始化链表。

3.11.3 注意事项

- 需要注意节点指针前后方向的操作。

3.11.4 编程实例

实例描述

使用双向链表，首先要申请内存，删除节点的时候要注意释放掉内存。

本实例实现如下功能：

1. 调用函数进行初始化双向链表。
2. 增加节点。
3. 删除节点。
4. 测试操作是否成功。

编程示例

代码实现如下：

```
#include "los_list.h"
#include<stdio.h>

VOID list_test(void)
{
 /*初始化，判断是否为空*/
 printf("initial.....\n");
```


版权所有 © 华为技术有限公司

```
LOS_DL_LIST* head;
head = (LOS_DL_LIST*)malloc(sizeof(LOS_DL_LIST));

LOS_ListInit(head);
if (!ListEmpty(head))
{
 printf("initial failed\n");
 return;
}

/*增加一个节点，在尾端插入一个节点*/
printf("node add and tail add......\n");
LOS_DL_LIST* node1 = (LOS_DL_LIST*)malloc(sizeof(LOS_DL_LIST));
LOS_DL_LIST* node2 = (LOS_DL_LIST*)malloc(sizeof(LOS_DL_LIST));
LOS_DL_LIST* tail = (LOS_DL_LIST*)malloc(sizeof(LOS_DL_LIST));
LOS_ListAdd(node1,head);
LOS_ListAdd(node2,node1);
if((node1->pstPrev == head) || (node2->pstPrev == node1)){
 printf("add node success\n");
}
LOS_ListTailInsert(tail,head);
if(tail->pstPrev == node2){
 printf("add tail success\n");
}


/*删除双向链表节点*/
printf("delete node......\n");
LOS_ListDelete(node1);
free(node1);
if(head->pstNext == node2){
 printf("delete node success\n");
}

}
```

结果验证

编译运行得到的结果为：

```
## Starting application at 0x83040000 ...
********hello Huawei LiteOS Cortex-A7********

version:Huawei LiteOS TDV100R001C00B038
build data : Jul 25 2015 17:33:02

**********************************
dist:1

 --- Test start---
initial.....
node add and tail add.....
add node success
add tail success
delete node.....
delete node success

 --- Test End ---
```


版权所有 © 华为技术有限公司

4 扩展内核

版权所有 © 华为技术有限公司

4.1 动态加载

4.1.1 概述

基本概念

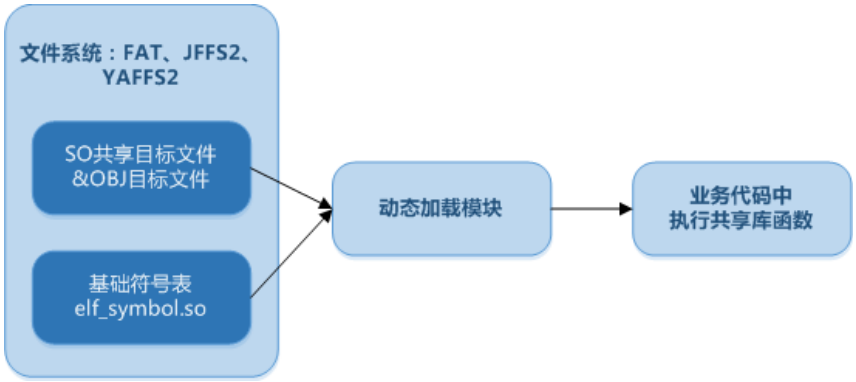
动态加载是一种程序加载技术。

静态链接是在链接阶段将程序各模块文件链接成一个完整的可执行文件，运行时作为整体一次性加载进内存。动态加载允许用户将程序各模块编译成独立的文件而不将它们链接起来，在需要使用到模块时再动态地将其加载到内存中。

静态链接将程序各模块文件链接成一个整体，运行时一次性加载入内存，具有代码装载速度快等优点。但当程序的规模较大，模块的变更升级较为频繁时，会存在内存和磁盘空间浪费、模块更新困难等问题。

动态加载技术可以较好地解决上述静态链接中存在的问题，在程序需要执行所依赖的模块中的代码时，动态地将外部模块加载链接到内存中，不需要该模块时可以卸载，可以提供公共代码的共享以及模块的平滑升级等功能。

Huawei LiteOS提供支持OBJ目标文件和SO共享目标文件的动态加载机制。



Huawei LiteOS的动态加载功能需要SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so、系统镜像bin文件配合使用。

动态加载相关概念

符号表

符号表在表现形式上是记录了符号名及其所在内存地址信息的数组，符号表在动态加载模块初始化时被载入到动态加载模块的符号管理结构中。在加载用户模块进行符号重定位时，动态加载模块通过查找符号管理结构得到相应符号所在地址，对相应重定位项进行重定位。

版权所有 © 华为技术有限公司

4.1.2 开发指导

功能

接口名	描述
LOS_LdInit	初始化动态加载模块
LOS_LdDestroy	销毁动态加载模块
LOS_SoLoad	动态加载一个so模块
LOS_ObjLoad	动态加载一个obj模块
LOS_FindSymByName	在模块或系统符号表中查找符号地址
LOS_ModuleUnload	卸载一个模块
LOS_PathAdd	添加一个相对路径

开发流程

动态加载主要有以下几个步骤：

1. 编译环境准备
2. 基础符号表elf_symbol.so的获取及镜像编译
3. 动态加载接口使用
4. 系统环境准备

编译环境准备

步骤1 添加.o和.so模块编译选项

- .o模块的编译选项中需要添加-mlong-calls -nostdlib选项
- .so模块的编译选项中需要添加-nostdlib -fPIC -shared选项

注意

IPC的动态加载需要用户保证所提供的模块文件中所有LD_SHT_PROGBITS、LD_SHT_NOBITS类型节区起始地址都4字节对齐，否则拒绝加载该模块。

.o和.so模块编译选项添加示例如下：

```
RM = -rm -rf

CC = arm-hisiv500-linux-gcc

SRCS = $(wildcard *.c)
OBJS = $(patsubst %.c,%.o,$(SRCS))
SOS = $(patsubst %.c,%.so,$(SRCS))

all: $(SOS)
```


版权所有 © 华为技术有限公司

```
$(OBJS): %.o : %.c
    @$(CC) -mlong-calls -nostdlib -c $< -o $@

$(SOS): %.so : %.c
    @$(CC) -mlong-calls -nostdlib $< -fPIC -shared -o $@

clean:
    @$(RM) $(SOS) $(OBJS)

.PHONY: all clean
```

步骤2 系统镜像编译

系统镜像bin文件编译makefile必须include根目录下config.mk文件，并使用其中的LITEOS_CFLAGS或LITEOS_CXXFLAGS编译选项，示例如下：

```
LITEOSTOPDIR ?= ../..

SAMPLE_OUT = .

include $(LITEOSTOPDIR)/config.mk
RM = -rm -rf

LITEOS_LIBDEPS := --start-group $(LITEOS_LIBDEP) --end-group

SRCS = $(wildcard sample.c)

OBJS = $(patsubst %.c,$(SAMPLE_OUT)/%.o,$(SRCS))

all: $(OBJS)

clean:
    @$(RM) *.o  sample *.bin *.map *.asm

$(OBJS): $(SAMPLE_OUT)/%.o : %.c
    $(CC) $(LITEOS_CFLAGS) -c $< -o $@
    $(LD) $(LITEOS_LDFLAGS) -uinit_jffspar_param --gc-sections -Map=$(SAMPLE_OUT)/sample.map -o $
(SAMPLE_OUT)/sample ./$@ $(LITEOS_LIBDEPS) $(LITEOS_TABLES_LDFLAGS)
    $(OBJCOPY) -O binary $(SAMPLE_OUT)/sample $(SAMPLE_OUT)/sample.bin
    $(OBJDUMP) -d $(SAMPLE_OUT)/sample >$(SAMPLE_OUT)/sample.asm
```

----结束

基础符号表 elf_symbol.so 的获取及镜像编译

请严格按如下步骤进行编译。

步骤1 编译.o和.so模块，并将系统运行所需的所有.o和.so文件拷贝到一个目录下，例如

```
/home/wmin/customer/out/so
```

说明

1. 如果a.so需要调用b.so中的函数，或者a.so引用到了b.so中的数据，则称a.so依赖b.so。
2. 当用户中a.so模块依赖b.so，且需要在加载a.so时自动将b.so也加载进来，则在编译a.so时需要将b.so作为编译参数。

步骤2 进入Huawei_LiteOS/tools/scripts/dynload_tools目录执行如下脚本命令：

```
$ ./ldsym.sh /home/wmin/customer/out/so
```


版权所有 © 华为技术有限公司

说明

1. “$”是linux shell提示符，下同
2. ldsym.sh脚本只需传入系统运行所需的所有.o和.so文件所在的那个目录绝对路径即可。
3. 目录路径必须是绝对路径。
4. 必须要在Huawei_LiteOS/tools/scripts/dynload_tools目录下执行该命令。

步骤3 编译系统镜像bin文件, 同时生成镜像文件，例如该目录在/home/wmin/customer/out/bin/，编译后在该目录下生成了sample镜像文件和用于烧写flash的sample.bin文件。

步骤4 进入Huawei_LiteOS/tools/scripts/dynload_tools目录执行sym.sh脚本得到基础符号表elf_symbol.so文件，示例如下：

```
$ ./sym.sh /home/wmin/customer/out/so arm-hisiv500-linux- /home/wmin/customer/out/bin/vs_server
```

说明

1. sym.sh的三个参数分别是：系统运行所需的所有.o和.so文件所在的那个目录绝对路径，编译器类型，系统镜像文件（不是烧写flash用的bin文件）。
2. 所有参数中路径都必须是绝对路径。
3. 第三个参数必须是系统镜像文件，不是烧写flash用的bin文件。
4. 基础符号表elf_symbol.so文件生成在系统镜像文件同一路径下。
5. 注意每次系统镜像的重新编译，都要将基础符号表elf_symbol.so重新生成并更新。
6. 必须要在Huawei_LiteOS/tools/scripts/dynload_tools目录下执行该命令。

步骤5 进入Huawei_LiteOS/tools/scripts/dynload_tools目录执行failed2reloc.py脚本得到用户模块中无法完成重定位的符号：

```
$ ./failed2reloc.py /home/wmin/customer/out/bin/vs_server /home/wmin/customer/out/so
```

说明

1. failed2reloc.py的两个参数分别是：系统镜像文件（不是烧写flash用的bin文件），系统运行所需的所有.o和.so文件所在的那个目录绝对路径。
2. 所有参数中路径都必须是绝对路径。
3. 第一个参数必须是系统镜像文件，不是烧写flash用的bin文件。
4. 该脚本输出用户模块中无法完成重定位的符号信息。

----结束

动态加载接口使用

步骤1 初始化动态加载模块

- 在使用动态加载特性前，需要调用LOS_LdInit接口初始化动态加载模块：

```
if (LOS_OK != LOS_LdInit("/yaffs/bin/dynload/elf_symbol.so", NULL, 0)) {
    printf("ld_init failed!!!!!!\n");
    return 1;
}
```

LOS_LdInit函数第一个参数是基础符号表文件路径，第二个参数是动态加载模块进行内存分配的堆起始地址，第三个参数是这块作为堆使用的内存的长度，在LOS_LdInit接口中会将这段内存初始化为堆；如果用户希望动态加载模块从系统堆上分配内存，第二个参数传入NULL，第三个参数被忽略。

说明

1. 动态加载模块的初始化只需要在业务启动时调用一次即可，重复初始化动态加载模块会返回失败。

上面这段代码演示使用系统堆，如下代码演示自定义堆：

版权所有 © 华为技术有限公司

```
#define HEAP_SIZE0x8000
INT8 usrHeap[HEAP_SIZE];
if (LOS_OK != LOS_LdInit("/yaffs/bin/dynload/elf_symbol.so, usrHeap,sizeof(usrHeap)")) {
    printf("ld_init failed!!!!!!\n")
    return 1;
}
```

说明

1. 用户不需要对自定义的这块内存进行初始化动作。
2. 动态加载所需分配的堆内存大小视要加载的模块而定，因此如果用户需要指定自定义堆时，需要保证堆长度足够大，否则建议使用系统堆。

步骤2 加载用户模块

- IPC的动态加载模块支持对.o以及.so模块的动态加载，对于obj文件的动态加载使用LOS_ObjLoad接口：

```
if ((handle = LOS_ObjLoad("/yaffs/bin/dynload/foo.o")) == NULL){
    printf("load module ERROR!!!!!!\n");
    return 1;
}
```

- 对于so文件的动态加载使用LOS_SoLoad接口：

```
if ((handle = LOS_SoLoad("/yaffs/bin/dynload/foo.so")) == NULL){
    printf("load module ERROR!!!!!!\n");
    return 1;
}
```

说明

对于so文件的动态加载，如果一个模块A需要另一个模块B，也就是存在模块A依赖于模块B的关系，则在加载A模块时会将模块B也加载进来。

步骤3 获取用户模块中的符号地址

- 在特定模块中查找符号

  需要在某个特定模块中查找用户模块的符号地址时，调用LOS_FindSymByName接口，并将LOS_FindSymByName的第一个参数置为需要查找的用户模块的句柄。

```
if ((ptr_magic = LOS_FindSymByName(handle, "os_symbol_table")) == NULL) {
    printf("symbol not found\n");
    return 1;
}
```

- 在全局符号表中查找符号

  需要在全局符号表（即OS模块，包括本模块和所有其他用户模块）中查找某个符号的地址时，调用LOS_FindSymByName接口，并将LOS_FindSymByName的第一个参数置NULL。

```
if ((pFunTestCase0 = LOS_FindSymByName(NULL, "printf")) == NULL) {
    printf("symbol not found\n");
    return 1;
}
```

步骤4 使用获取到的符号地址：LOS_FindSymByName返回一个符号的地址（VOID *指针），用户在拿到这个指针之后可以做相应的类型转换来使用该符号，下面举两个例子说明，一个针对数据类型符号，一个针对函数类型符号。

- 结构体数组类型符号（演示说明数据类型的符号使用）

  现有结构体KERNEL_SYMBOL定义如下：

```
typedef struct KERNEL_SYMBOL {
    UINT32 uwAddr;
    INT8 *pscName;
} KERNEL_SYMBOL;
```

  "/bin/dynload/elf_symbol.so"模块中定义了结构体数组：

```
KERNEL_SYMBOL los_elf_symbol_table[LENGTH_ARRAY] = {
    {0x83040000, "__exception_handlers"},
```


版权所有 © 华为技术有限公司

```
    {0x8313b864, "cmd_version"},
    ...
    {0, 0},
};
```

通过如下代码遍历los_elf_symbol_table中的各项：

```
const char *g_pscOsOSSymtblFilePath = "/yaffs/bin/dynload/elf_symbol.so";
const char *g_pscOsSymtblInnerArrayName = "los_elf_symbol_table";
typedef KERNEL_SYMBOL (*OS_SYMTBL_ARRAY_PTR)[LENGTH_ARRAY];/* 结构体数组指针类型声明 */
OS_SYMTBL_ARRAY_PTR pstSymtblPtr = (OS_SYMTBL_ARRAY_PTR)NULL;
VOID *pPtr = (VOID *)NULL;
UINT32 uwIdx = 0;
UINT32 uwAddr;
INT8 *pscName = (INT8 *)NULL;
if ((pOSSymtblHandler = LOS_SoLoad(g_pscOsOSSymtblFilePath)) == NULL) {
    return LOS_NOK;
}
if ((pPtr = LOS_FindSymByName(pOSSymtblHandler, g_pscOsSymtblInnerArrayName)) == NULL) {
    printf("os_symtbl not found\n");
    return LOS_NOK;
}
pstSymtblPtr = (OS_SYMTBL_ARRAY_PTR)pPtr;/* 强制类型转换成真实的指针类型 */
uwAddr = (*pstSymtblPtr)[0].uwAddr;
pscName = (*pstSymtblPtr)[0].pscName;
while (uwAddr != 0 && pscName != 0) {
    ++uwIdx;
    uwAddr= (*pstSymtblPtr)[uwIdx].uwAddr;
    pscName= (*pstSymtblPtr)[uwIdx].pscName;
}
```

- 函数类型符号

  foo.c中定义了一个无参的函数test_0和一个有两个参数的函数test_2，编译生成foo.o，代码演示在demo.c中获取foo.o模块中的函数并调用。

```
foo.c：
int test_0(void) { return 0; }
int test_2(int i, int j) { return 0; }
demo.c
typedef unsigned int (* TST_CASE_FUNC)();/* 无形参函数指针类型声明 */
typedef unsigned int (* TST_CASE_FUNC1)(UINT32); /* 单形参函数指针类型声明 */

typedef unsigned int (* TST_CASE_FUNC2)(UINT32, UINT32); /* 双形参函数指针类型声明 */
TST_CASE_FUNC pFunTestCase0 = NULL;/* 函数指针定义 */
TST_CASE_FUNC2 pFunTestCase2 = NULL;
handle = LOS_ObjLoad("/yaffs/bin/dynload/foo.o");
pFunTestCase0 = NULL;
pFunTestCase0 = LOS_FindSymByName(handle, "test_0");
if (pFunTestCase0 == NULL){
    printf("can not find the function name\n");
    return 1;
}
uwRet = pFunTestCase0();
pFunTestCase2 = NULL;
pFunTestCase2 = LOS_FindSymByName(NULL, "test_2");
if (pFunTestCase2 == NULL){
    printf("can not find the function name\n");
    return 1;
}
uwRet = pFunTestCase2(42, 57);
```

步骤5 卸载模块

当要卸载一个模块时，调用LOS_ModuleUnload接口，将需要卸载的模块句柄作为参数传入该接口。对于已被加载过的obj或so文件的句柄，卸载时统一使用LOS_ModuleUnload接口。

```
uwRet = LOS_ModuleUnload(handle);
if (uwRet != LOS_OK) {
    printf("unload module failed");
```


版权所有 © 华为技术有限公司

```
    return 1;
}
```

步骤6 销毁动态加载模块

不再需要动态加载功能时，调用LOS_LdDestroy接口，卸载动态加载模块。

注意

销毁动态加载模块时会自动卸载掉所有已被加载的模块。

```
uwRet = LOS_LdDestroy();
if (uwRet != LOS_OK) {
    printf("destroy dynamic loader failed");
    return 1;
}
```

说明

在业务不再需要动态加载模块时销毁动态加载模块，该接口是与LOS_LdInit配对的接口。在销毁动态加载模块后，如果业务后续再需要动态加载必须再调用LOS_LdInit重新初始化动态加载模块。

步骤7 使用相对路径

用户在使用动态加载接口时，如果想使用相对路径，也即使用类似环境变量的机制时，需要通过LOS_PathAdd接口添加相对路径：

```
uwRet = LOS_PathAdd("/yaffs/bin/dynload");
if (uwRet != LOS_OK) {
    printf("add relative path failed");
    return 1;
}
```

添加相对路径后，用户在调用LOS_LdInit、LOS_SoLoad、LOS_ObjLoad接口时传入文件名即可，而无需再传入完整的绝对路径，动态加载会在用户添加的相对路径下查找相应模块。

如果用户传入的多个路径下有相同文件名的模块，则动态加载在加载模块时按照添加的先后依次在所有相对路径中查找，且只加载第一个查找到的文件。

说明

1. 只有在调用LOS_PathAdd接口添加相对路径后，才能在调用动态加载接口时使用相对路径。
2. 用户可以通过多次调用LOS_PathAdd接口添加多个相对路径。

----结束

系统环境准备

SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so、系统镜像bin文件配合使用。

其中SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so必须放置在文件系统中，例如jffs2、yaffs、fat等文件系统。

建议操作顺序：

步骤1 烧写系统镜像bin文件到flash中，该镜像默认不启动动态加载功能

版权所有 © 华为技术有限公司

步骤2 如果SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so路径为可热拔插的SD卡设备，则可在电脑更新SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so到SD卡指定路径

步骤3 如果SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so路径为jffs2、yaffs文件系统，则可通过如下两种方式更新：

1. 烧写文件系统镜像
2. 系统启动后tftp命令下载SO共享目标文件（或OBJ目标文件）、基础符号表elf_symbol.so，示例命令如下：

```
tftp -g -l /yaffs0/foo.so -r foo.so 10.67.211.235
tftp -g -l / yaffs0/elf_symbol.so -r elf_symbol.so 10.67.211.235
```

步骤4 启动系统动态加载功能，进行验证

----结束

Shell 调试

在Shell里我们封装了一系列与动态加载有关的命令，方便用户进行调试。

具体的Shell命令详细说明参见**命令参考**。

- **初始化**动态加载模块

当用户需要在Shell中调试动态加载特性的时候，需要首先初始化动态加载模块。

Shell命令：ldinit

```
Huawei LiteOS# ldinit /yaffs/bin/dynload/elf_symbol.so


Huawei LiteOS#
```

注意

动态加载过程中发现符号重定义只作为一个warning而不作为error处理，所以仅反馈符号重定义而不返回其他错误信息表示动态加载模块初始化成功。

- **加载一个模块**

Shell命令：mopen

```
Huawei LiteOS# mopen /yaffs/bin/dynload/foo.o
module handle: 0x80391928

Huawei LiteOS#
```

注意

（1）模块路径必须要用绝对路径（2）必须要先初始化动态加载模块再加载模块。

- **查找一个符号**

Shell命令：findsym

版权所有 © 华为技术有限公司

```
Huawei LiteOS# findsym 0 printf
symbol address:0x8004500c

Huawei LiteOS#
Huawei LiteOS# findsym 0x80391928 test_0
symbol address:0x8030f241

Huawei LiteOS#
```

- **调用一个符号**

Shell 命令：call

```
Huawei LiteOS# call 0x8030f241
test_0

Huawei LiteOS#
```

- **卸载一个模块**

Shell命令：mclose

```
Huawei LiteOS# mclose 0x80391928

Huawei LiteOS#
```

- **销毁动态加载模块**

Shell命令：lddrop

```
Huawei LiteOS# lddrop

Huawei LiteOS#
```

注意

不反馈任何错误提示表示卸载动态加载模块成功。

4.1.3 注意事项

- .o模块的编译选项中需要添加-mlong-calls -nostdlib选项。
- .so模块的编译选项中需要添加-mlong-calls -nostdlib -fPIC -shared选项。
- IPC的动态加载需要用户保证所提供的模块文件中所有LD_SHT_PROGBITS、LD_SHT_NOBITS类型节区起始地址都4字节对齐，否则拒绝加载该模块。

4.1.4 编程实例

实例描述

示例中，sample_foo.c中定义了一个无参的函数test_0和一个有两个参数的函数test_2，编译生成foo.o，代码演示在sample_Dynamic_loading.c中获取foo.o模块中的函数并调用。

编程示例

代码实现如下：

```
foo.c:
int test_0(void) { printf("test_0\n"); return 0; }
```


版权所有 © 华为技术有限公司

```
int test_2(int i, int j) { printf("test_2: %d %d\n", i, j); return 0; }

demo.c:
typedef int (* TST_CASE_FUNC)(); /* 无形参函数指针类型声明 */
typedef int (* TST_CASE_FUNC1)(UINT32); /* 单形参函数指针类型声明 */
typedef int (* TST_CASE_FUNC2)(UINT32, UINT32); /* 双形参函数指针类型声明 */
if (LOS_OK != LOS_LdInit("/yaffs/bin/dynload/elf_symbol.so", NULL, 0)) {
    printf("ld_init failed!!!!!!\n");
    return 1;
}
unsigned int uwRet;
TST_CASE_FUNC pFunTestCase0 = NULL;/* 函数指针定义 */
TST_CASE_FUNC2 pFunTestCase2 = NULL;
handle = LOS_ObjLoad("/yaffs/bin/dynload/foo.o");

pFunTestCase0 = NULL;
pFunTestCase0 = LOS_FindSymByName(handle, "test_0");
if (pFunTestCase0 == NULL){
    printf("can not find the function name\n");
    return 1;
}
uwRet = pFunTestCase0(); /* 调用该函数指针 */

pFunTestCase2 = NULL;
pFunTestCase2 = LOS_FindSymByName(NULL, "test_2");
if (pFunTestCase2 == NULL){
    printf("can not find the function name\n");
    return 1;
}
uwRet = pFunTestCase2(42, 57);  /* 调用该函数指针 */

uwRet = LOS_ModuleUnload(handle);
if (uwRet != LOS_OK) {
    printf("unload module failed");
    return 1;
}

uwRet = LOS_LD_Destroy();
if (uwRet != LOS_OK) {
    printf("destroy dynamic loader failed");
    return 1;
}
```

结果验证

编译运行得到的结果为：

```
Huawei LiteOS#

*****************************
*****************************
test_0
test_2:42 57

*****************************
*****************************
```

完整实例代码

sample_foo.c

sample_Dynamic_loading.c

4.2 分散加载

版权所有 © 华为技术有限公司

4.2.1 概述

基本概念

分散加载是一种实现特定代码快速启动的技术，通过优先加载特定代码到内存，达到缩短从系统开机到特定代码执行的时间。可被应用来实现关键业务的快速启动。

嵌入式系统通过uboot加载flash上的镜像文件到内存并执行，而镜像文件本身可能较大，由于flash读取速度的限制，将镜像全部加载完再执行可能无法满足时间敏感的业务对启动速度的要求。

分散加载的思想是先加载部分镜像并执行，这部分镜像包含了时间敏感的关键业务，从而达到快速启动关键业务的效果。

Huawei LiteOS的分散加载

Huawei LiteOS的分散加载分为两个阶段，第一阶段通过uboot将关键业务部分镜像加载到内存并执行，待这部分业务得到执行后，第二阶段在代码中加载剩余部分镜像到内存继续执行。通过合理布局镜像，第一阶段加载部分镜像的速度会比加载完整镜像快，从而缩短系统启动到关键业务运行的时间。

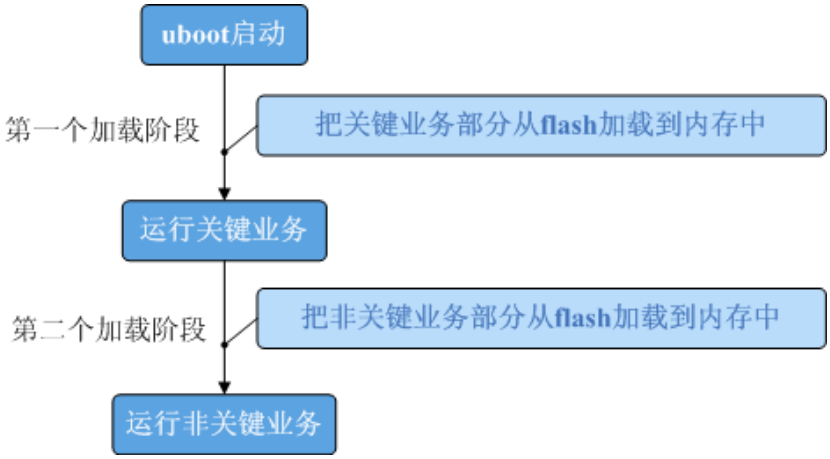
在IPC Huawei LiteOS版本上，通过应用分散加载技术，实现了1s内从开机启动到录制，超越Linux版本的3s-4.5s。

运作机制

分散加载的主体思想是将部分时间敏感的业务提前加载执行，具体手段是将与这些业务相关的数据、代码段布局到镜像文件的前端，第一阶段只加载前端这段镜像，达到最短时间内即可运行时间敏感业务的目的。

在这些业务得到执行之后，第一阶段的代码中调用分散加载接口加载剩余部分镜像，接着运行镜像剩余部分的业务。

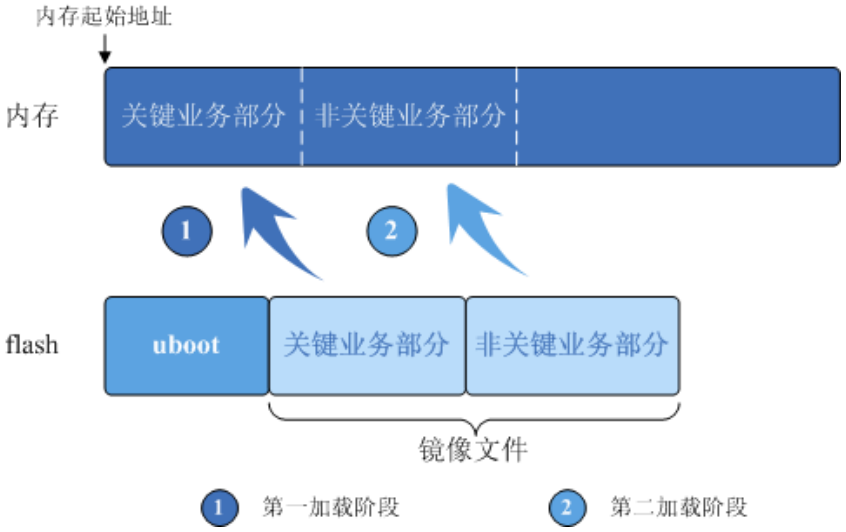
图 4-1 分散加载流程图



分散加载的内部原理图如图2所示，图中的运作顺序可参照图1的流程说明。

版权所有 © 华为技术有限公司

图 4-2 分散加载内部原理图



分散加载在关键业务第一时间被加载执行之后，再加载非关键业务。

4.2.2 开发指导

使用场景

分散加载技术应用的典型场景是快速启动对时间敏感的业务。

嵌入式系统中可能存在某些业务对启动时间要求比较高，譬如Huawei LiteOS IPC项目上对从开机到录制预览的时间要求较高，可以利用分散加载技术实现录制预览业务的快速启动。

功能

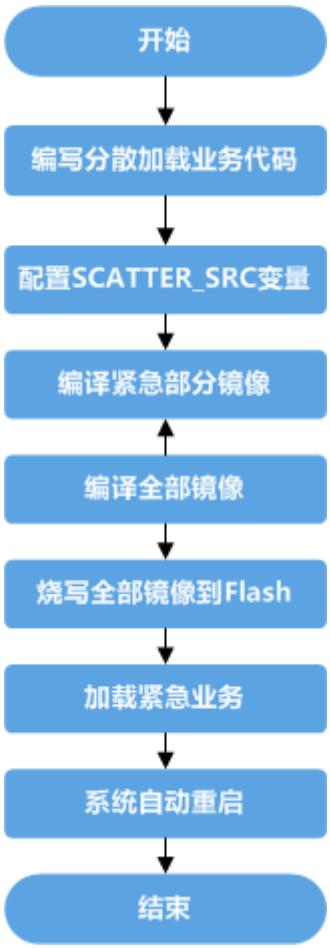
Huawei LiteOS系统中的分散加载模块为用户提供如下接口。

功能分类	接口名	描述
分散加载接口	LOS_ScatterLoad	在分散加载阶段的最后调用此接口，从镜像加载剩余非紧急业务

开发流程

分散加载流程图如下所示。

版权所有 © 华为技术有限公司



步骤1 调用接口**LOS_ScatterLoad**，编写分散加载业务代码

业务代码入口为函数app_init，该函数位于os_adapt.c。在紧急业务代码后调用LOS_ScatterLoad函数进行分散加载，并用#ifndef MAKE_SCATTER_IMAGE、#endif将该函数后的非紧急业务包围起来，用以编译紧急镜像和全部镜像时作区分，示例代码如下。

```
void app_init() {
    proc_fs_init();
    hi_uartdev_init();
    system_console_init("/dev/uartdev-0");
    LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
BEFORE_SCATTER);
    LOS_ScatterLoad(0x100000, flash_read, NAND_READ_ALIGN_SIZE);

#ifndef MAKE_SCATTER_IMAGE /* 以下为非紧急业务 */
    LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
AFTER_SCATTER);

    extern unsigned int osShellInit(void);
    osShellInit();
    rdk_fs_init();
    SDK_init();

    hi_product_driver_init();
    char *apszArgv[3]={"vs_server","./higv.bin","-i"};
    vs_server(3, apszArgv);
#endif /* MAKE_SCATTER_IMAGE */
}
```


版权所有 © 华为技术有限公司

说明

os_adapt.c位于Huawei_LiteOS代码包的platform/bsp/hi3516a/os_adapt路径下。

步骤2 配置**SCATTER_SRC**变量

在根目录下Makefile中配置**SCATTER_SRC**，将变量定义为调用分散加载函数的业务源文件路径，如下所示，其中LITEOSTOPDIR指代Huawei_LiteOS代码根目录。

```
SCATTER_SRC := $(LITEOSTOPDIR)/platform/bsp/$(LITEOS_PLATFORM)/os_adapt/os_adapt.c
```

步骤3 执行**make scatter**，编译紧急部分镜像

在根目录下执行如下命令，则不会编译#ifndef MAKE_SCATTER_IMAGE以下的业务代码。编译系统将自动调用工具链抽取分散加载最小镜像的符号表并根据该符号表提取分散加载最小镜像的.a库列表。

```
Huawei_LiteOS$ make scatter
```

步骤4 执行**make**，编译全部镜像

- 在根目录下执行如下命令，则编译全部业务代码。

  ```
  Huawei_LiteOS$ make
  ```

  编译后，命令行界面会返回紧急镜像大小信息，如下图所示。

  ```
  ###### ###### ##### ##### ##### ##### #####
  #######Calculate the size of scatter#######
  the size is 0x4E0000
  ####################end#####################
  ```

- 编译完成后，检查镜像段的排布，如果镜像中生成了分散加载相关的段则表明分散加载的镜像生成成功。进入系统镜像生成目录（hi3516a平台的镜像生成目录为out/hi3516a，其他类推），可以看到生成的系统镜像vs_server文件，执行命令**readelf -S vs_server**打开该文件，结果如下图所示。显示了与分散加载相关的段信息（包括段的名称、起始地址及偏移大小）。其中.fast_rodata为分散加载镜像的只读数据段，.fast_text为代码段，.fast_data为数据段。

  ```
  [15] .fast_rodata      PROGBITS        80159000 088868 0a6adc
  [16] .fast_text        PROGBITS        80200000 12f868 27ece0
  [17] .fast_data        PROGBITS        8047f000 3ae868 067b78
  [18] .got              PROGBITS        804e6b78 4163e0 000504
  [19] .text             PROGBITS        804e8000 417868 1b1b90
  [20] .rodata           PROGBITS        8069a000 5c9868 09f4d8
  [21] .data             PROGBITS        8073a000 669868 01cd7c
  [22] .bss              NOBITS          80757000 6865e4 6d74e0
  ```

  查看分散加载链接脚本.text段，新增了scatter.o(*.text*)，如下图所示，实现了将分散加载的快速启动部分代码相关符号归拢到一个同一个段中。

  ```
      .fast_text ALIGN (0x1000): { __fast_text_start = ABSOLUTE(.); . = .;
  scatter.o(*.text*);
      }

      . = (ABSOLUTE (.) + (0x1000 - 1)) & ~ (0x1000 - 1);
      __fast_text_end = ABSOLUTE (.);
  ```

说明

分散加载链接脚本路径：Huawei_LiteOS/tools/scripts/ld/scatter.ld

步骤5 执行**tftp** *0x82000000 vs_server.bin*;**nand erase** *0x100000 0x700000*;**nand write** *0x82000000 0x100000 0x700000*;，将全部镜像烧写到Flash

进入串口工具界面，输入如下命令，将全部镜像烧写到Flash的0x100000地址位。

版权所有 © 华为技术有限公司

```
tftp 0x82000000 vs_server.bin;nand erase 0x100000 0x700000;nand write 0x82000000 0x100000 0x700000;
```

其中，vs_server.bin为系统镜像文件名，先将其烧写到内存中一段高地址位0x82000000。然后烧写到Flash，起始地址为0x100000，烧写长度为0x700000，即烧写的镜像文件大小不能超过7M，跟据实际镜像大小调整数值。

步骤6 执行**nand read** *0x80008000 0x100000 0x4E0000*; **go** *0x80008000*;，加载紧急业务

执行如下命令，从Flash的0x100000地址处读取长度为0x4E0000的镜像，加载紧急业务到0x80008000。

```
nand read 0x80008000 0x100000 0x4E0000; go 0x80008000;
```

步骤7 系统自动重启

系统自动重启，在0x80008000地址处加载镜像。

----结束

4.2.3 注意事项

- 分散加载第一阶段拷贝过少或者拷贝偏移地址没有根据存储介质的差异进行对齐都会导致系统异常，因此使用时要按照编译最后给出的大小进行uboot加载镜像。
- 用户需保证提取的库文件列表是支持关键业务运行的超集，否则会导致分散加载第一阶段中的代码访问到第二阶段中的代码或数据，从而导致系统异常。
- 分散加载使用中可能存在这样一种场景：一个变量在第一阶段中运行后值被修改，但是在第二阶段加载运行之后，该变量值又成为一个未初始化的值。这种场景的原因是该变量在第一阶段中使用到，但是并没有被归拢到第一阶段中，所以在第一阶段修改之后，第二阶段加载进内存后该变量值又被覆盖成未初始化的值。解决的方法是将该变量归拢到第一阶段中，确保第一阶段使用到的数据都在快速启动段中。

4.2.4 常见问题汇总

本节介绍使用分散加载技术遇到的主要问题和解决方法。

- 缺少.O文件

  ```
  arm-hisiv300-linux-ld: cannot find libscatter.O
  make: *** [vs_server] Error 1
  ```

  这个问题出现的原因是修改了链接脚本后，没有对应生成.O文件，解决的方法是生成对应的.O文件并且放到目标目录下。

- 符号未定义

  ```
  /usr1/xxxxx/gerrit_code/modify-debug/liteos_ipc/out/lib/libar6003.a(ar6000_drv.o): In
  function `ar6000_avail_ev':
  /usr1/xxxxx/gerrit_code/modify-debug/liteos_ipc/vendor/ar6k3_wifi/AR6003/host/qca/source/
  ar6000_drv.c:1553: undefined reference to `wireless_init_event'
  /usr1/xxxxx/gerrit_code/modify-debug/liteos_ipc/out/lib/libar6003.a(drv_config.o): In
  function `ar6000_tkip_micerr_event':
  /usr1/xxxxx/gerrit_code/modify-debug/liteos_ipc/vendor/ar6k3_wifi/AR6003/host/qca/source/
  drv_config.c:1856: undefined reference to `wireless_send_event'
  make: *** [vs_server] Error 1
  ```

  这个问题的出现是比较常见的，可能是裁剪过程中在修改链接脚本的时候，将一些必要的.a文件也删除了，这时需要用grep指令在out/lib目录下搜索未定义的变量，找出都存在于哪些.a文件中，将未添加的.a文件添加到链接脚本中。

- 分散加载进指令异常。

  通过查看系统异常时pc的位置是否超出分散加载第一阶段的范围，如果是则应该是第一阶段库文件列表涵盖不全，导致有符号未被归拢到第一阶段的代码、数据

版权所有 © 华为技术有限公司

段中，需要结合系统镜像反汇编文件定位到异常pc所在函数名，找到该函数定义所在的库，将该库添加到库列表中。

4.3 异常接管

4.3.1 概述

基本概念

异常接管是操作系统对在运行期间发生异常的情况进行处理的一系列动作，譬如打印异常发生时当前函数调用栈信息、cpu现场信息、任务的堆栈情况等。

异常接管作为一种调测手段，可以在系统发生异常时提供给用户有用的异常信息，譬如异常的类型、发生异常时系统的状态等，方便用户定位分析问题。

Huawei LiteOS的异常接管，在系统发生异常时的处理动作是显示异常发生时正在运行的任务信息（包括任务名、任务号、堆栈大小等），以及cpu现场等信息。

运作机制

堆栈分析

- R11：可以用作通用寄存器，在开启特定编译选项时可以用作帧指针寄存器，可以用来实现栈回溯功能。

  gcc默认将R11作为存储变量的通用寄存器使用，因而默认情况下无法使用FP的栈回溯功能。为支持调用栈解析功能，需要在编译参数中添加-fno-omit-frame-pointer选项，提示编译器将R11作为FP使用。

- FP寄存器（Frame Point），帧指针寄存器，根据该寄存器可以实现追溯程序调用栈的功能，用来展示函数间的调用关系。

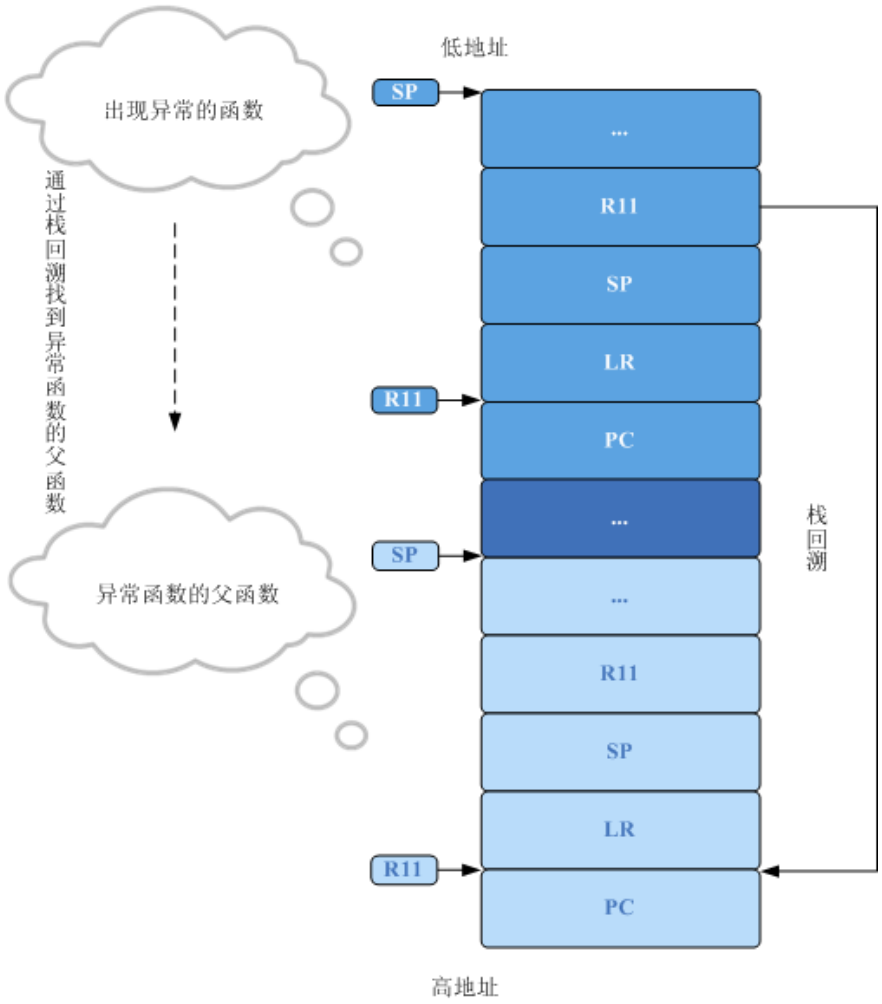
  FP寄存器指向当前执行函数的栈回溯结构。返回的FP值是指向由调用了这个当前函数的函数(父函数)建立的栈回溯结构的指针。而这个结构中的返回FP值是指向调用了父函数的函数的栈回溯结构的指针，以此类推。

  当运行发生异常时，系统打印FP寄存器内容，用户可以据此追溯函数间的调用关系，帮助定位异常的原因。

  堆栈分析原理如图1所示。

版权所有 © 华为技术有限公司

图 4-3 堆栈分析原理示意图



图中不同颜色的寄存器表示不同的函数，通过FP寄存器，栈回溯到了异常函数的父函数，从而使调用关系更清楚，方便用户定位问题。

调用栈关系

在上图1，可以看到函数调用过程中，栈帧寄存器的保存；因此可以按照规律对栈进行解析就可以推出函数调用执行关系，具体步骤如下：

1. 获取当前FP寄存器的值。
2. FP寄存器减去4字节得到当前PC值。通过PC值可以参考elf文件或者编译生成asm文件得到函数名称。
3. FP寄存器减去24字节，得到上次函数的调用栈帧的起始地址；FP寄存器减去16字节得到上次函数调用结束时SP的地址，那么FP到SP之间的栈就是一个函数调用的栈帧。
4. 通过每一层栈帧中的PC指针，就知道函数调用的关联。

4.3.2 开发指导

功能

异常接管为用户提供以下几种异常类型：

版权所有 © 华为技术有限公司

表 4-1

异常名称	描述	值
OS_EXCEPT_UNDEF_INSTR	未定义的指令异常	1
OS_EXCEPT_SWI	软中断异常	2
OS_EXCEPT_PREFETCH_ABORT	预指取指令异常	3
OS_EXCEPT_DATA_ABORT	数据中止异常	4
OS_EXCEPT_FIQ	FIQ异常	5

开发流程

异常接管一般的定位步骤如下：

1. 打开编译器编译后生成的.asm 文件。
2. 搜索PC指针在.asm中的位置。
3. 根据LR值查找对应的调用函数。
4. 重复步骤3，找到异常的任务函数。

具体的定位方法会在编程实例中举例说明。

4.3.3 注意事项

- 要查看调用栈信息，必须添加编译选项宏-fno-omit-frame-pointer支持stack frame，否则编译默认是关闭FP寄存器。

4.3.4 编程实例

实例描述

使用panic 命令手动触发了一个软中断异常，异常函数为LOS_Panic，下面两个代码 test_panic 为触发异常命令函数，另一个为异常调用栈打印信息。

uwExcType 2为软中断异常。

定位步骤如下：

1. 打开编译器编译后生成的 xxx.asm 文件。
2. 搜索PC指针 80121234 在asm文件位置（去掉0x）。
3. 根据 LR值查找对应的调用函数。
4. g_RunningTask 为当前异常时任务函数。

```
UINT32 test_panic(UINT32 argc, CHAR **args)
{
    LOS_Panic("*****Trigger an exception\n");
    return;
}
```

```
Huawei LiteOS# panic
*****Trigger an exception
uwExcType = 2
```


版权所有 © 华为技术有限公司

```
puwExcBuffAddr pc = 0x80121234
puwExcBuffAddr lr = 0x80121234
puwExcBuffAddr sp = 0x80e63400
puwExcBuffAddr fp = 0x80e6340c
*******backtrace begin*******
traceback 0 -- lr = 0x80138d04
traceback 0 -- fp = 0x80e635d4
traceback 1 -- lr = 0x80138d88
traceback 1 -- fp = 0x80e635e4
traceback 2 -- lr = 0x801247d4
traceback 2 -- fp = 0x80e635f4
traceback 3 -- lr = 0x801217c4
traceback 3 -- fp = 0x11111111
R0          = 0x1c
R1          = 0x800dba3a
R2          = 0x1b
R3          = 0xfe
R4          = 0x80e634a0
R5          = 0x0
R6          = 0x800cc7f8
R7          = 0x7070707
R8          = 0x8080808
R9          = 0x9090909
R10         = 0x10101010
R11         = 0x80e6340c
R12         = 0x1b
SP          = 0x80e63400
LR          = 0x80121234
PC          = 0x80121234
CPSR        = 0x60000013
g_pRunningTask->pcTaskName = shellTask
g_pRunningTask->uwTaskPID = 6
g_pRunningTask->uwStackSize = 12288
```

4.4 CPU 占用率

4.4.1 概述

基本概念

CPU（中央处理器，Central Processing Unit）占用率可以分为系统CPU占用率和任务CPU占用率两种。

系统CPU占用率（CPU Percent）是指周期时间内系统的CPU占用率，用于表示系统一段时间内的闲忙程度，也表示CPU的负载情况。系统CPU占用率的有效表示范围为0～100，其精度（可通过配置调整）为百分比。100表示系统满负荷运转。

任务CPU占用率指单个任务的CPU占用率，用于表示单个任务在一段时间内的闲忙程度。任务CPU占用率的有效表示范围为0～100，其精度（可通过配置调整）为百分比。100表示在一段时间内系统一直在运行该任务。

用户通过系统级的CPU占用率，判断当前系统负载是否超出设计规格。

通过系统中各个任务的占用情况，判断查看当前的各个任务的CPU占用率是否符合设计的预期。

运作机制

Huawei LiteOS的CPUP（CPU Percent，系统CPU占用率）采用任务级记录的方式，在任务切换中，记录任务启动时间，和任务切出或者退出时间，每次任务退出，系统会累加整个任务的占用时间。

版权所有 © 华为技术有限公司

在los_config.h中可以对CPU占用率模块进行选配，对于CPU占用模块，可通过LOSCFG_KERNEL_CPUP配置打开(YES)和关闭(NO)。

Huawei LiteOS提供以下两种CPU占用率的信息查询：

- 系统CPU占用率。
- 任务CPU占用率。

CPU占用率的计算方法：

系统CPU占用率=系统中除idle任务外其他任务运行总时间/系统运行总时间

任务CPU占用率=任务运行总时间/系统运行总时间

4.4.2 开发指导

使用场景

通过系统级的CPU占用率，判断当前系统负载是否超出设计规格。

通过系统中各个任务的占用情况，判断查看当前的各个任务的CPU占用率是否符合设计的预期。

功能

Huawei LiteOS系统中的CPU占用率模块为用户提供下面几种功能。

表 4-2 功能列表

功能分类	接口名	描述
获取系统CPU占用率	LOS_SysCpuUsage	获取当前系统CPU占用率
	LOS_HistorySysCpuUsage	获取系统历史CPU占用率
获取任务CPU占用率	LOS_TaskCpuUsage	获取指定任务CPU占用率
	LOS_HistoryTaskCpuUsage	获取指定任务历史CPU占用率
	LOS_AllTaskCpuUsage	获取所有任务CPU占用率

开发流程

CPU占用率的典型开发流程：

1. 调用获取系统CPU使用率函数LOS_SysCpuUsage。
2. 调用获取系统历史CPU使用率函数LOS_HistorySysCpuUsage。
   - 系统根据不同模式进入任务获取不同时间段的系统计数值，恢复中断；
3. 调用获取指定任务CPU使用率函数LOS_TaskCpuUsage。
   - 若任务已创建并且可用，则关中断，正常获取，恢复中断；
   - 若任务未创建或不可用，则返回错误码；

版权所有 © 华为技术有限公司

4. 调用获取指定任务历史CPU使用率函数LOS_HistoryTaskCpuUsage。
   - 若任务已创建并且可用，则关中断，根据不同模式正常获取，恢复中断；
   - 若任务未创建或不可用，则返回错误码；
5. 调用获取所有任务CPU使用率函数LOS_AllTaskCpuUsage。
   - 若CPUP已初始化，则关中断，根据不同模式正常获取，恢复中断；
   - 若CPUP未初始化或有非法入参，则返回错误码；

平台差异性

无。

4.4.3 注意事项

- 由于CPU占用率对性能有一定的影响，同时只有在产品开发时需要了解各个任务的占用率，因此建议在产品发布时，关掉CPUP模块的裁剪开关LOSCFG_KERNEL_CPUP。
- 通过上述接口获取到的返回值是千分值。该值可以通过与LOS_CPUP_PRECISION_MULT相除获得相应的百分值。

4.4.4 编程实例

实例描述

本实例实现如下功能：

1. 创建一个用于CPUP测试的任务。
2. 获取当前系统CPUP。
3. 以不同模式获取历史系统CPUP。
4. 获取创建的CPUP测试任务的CPUP。
5. 以不同模式获取创建的CPUP测试任务的CPUP。

编程示例

前提条件：

- 在los_config.h中，将OS_INCLUDE_CPUP配置项打开。

代码实现如下：

```
#include "los_task.h"
#include "los_cpup.h"

#define  MODE  4

UINT32 cpupUse;
OS_CPUP_TASK_S pstCpup;
UINT16 pusMaxNum = 0;
UINT32 g_CpuTestTaskID;

VOID Example_cpup()
{

    printf("entry cpup test example\n");
    while(1) {
        usleep(100);
```


版权所有 © 华为技术有限公司

```
    }
}

UINT32 it_cpup_test()
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S CpupTestTask;

    /*创建用于cpup测试的任务*/
    memset(&CpupTestTask, 0, sizeof(TSK_INIT_PARAM_S));
    CpupTestTask.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_cpup;
    CpupTestTask.pcName       = "TestCpupTsk";    /*测试任务名称*/
    CpupTestTask.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    CpupTestTask.usTaskPrio   = 5;
    CpupTestTask.uwResved   = LOS_TASK_STATUS_DETACHED;
    uwRet = LOS_TaskCreate(&g_CpuTestTaskID, &CpupTestTask);
    if(uwRet != LOS_OK)
    {
        printf("CpupTestTask create failed .\n");
        return LOS_NOK;
    }
         usleep(100);

    /*获取当前系统cpu占用率*/
    cpupUse = LOS_SysCpuUsage();
    printf("the current system cpu usage is: %d\n",cpupUse);

    /*获取历史系统cpu 1s内的占用率，历史cpu占用率的获取分三种模式，MODE1表示10s 内占用率,MODE2表示
前一个1s内占用率,MODE3表示小于1s内*/
    //cpupUse = LOS_HistorySysCpuUsage(MODE1);
    //printf("the history system cpu usage in 10s: %d\n",cpupUse);

    //cpupUse = LOS_HistorySysCpuUsage(MODE2);
    //printf("the history system cpu usage in 1s:  %d\n",cpupUse);

    cpupUse = LOS_HistorySysCpuUsage(MODE);
    printf("the history system cpu usage in <1s:    %d\n",cpupUse);

    /*获取指定任务的cpu占用率，该测试例程中指定的任务为以上创建的cpup测试任务*/
    cpupUse = LOS_TaskCpuUsage(g_CpuTestTaskID);
    printf("cpu usage of the CpupTestTask:\n TaskID: %d\n usage: %d\n",g_CpuTestTaskID,cpupUse);

  /*获取指定历史任务在<1s内的cpu占用率，该测试例程中指定的任务为以上创建的cpup测试任务*/
    cpupUse = LOS_HistoryTaskCpuUsage(g_CpuTestTaskID, MODE);
    printf("cpu usage of the CpupTestTask in <1s:\n TaskID: %d\n usage:%d
\n",g_CpuTestTaskID,cpupUse);
    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
 --- Test start---
entry cpup test example

Huawei LiteOS# the current system cpu usage is :       49
the history system cpu usage in <1s:     50
cpu usage of the CpupTestTask:
TaskID:4
usage:17
cpu usage of the CpupTestTask in <1s:
TaskID:4
usage:12

 ---Test End ---
```


版权所有 © 华为技术有限公司

完整实例代码

sample_cpup.c

4.5 linux 适配

4.5.1 完成量

4.5.1.1 概述

基本概念

完成量（Completion）是一种轻量级的机制，用于任务同步。完成量是对信号量的一种补充，之前Linux中信号量机制，up()和down()实现还允许这两个函数在同一个信号量上并发，在多CPU运行场景下，up()可能试图访问一个不存在的信号量数据结构,为了防止这种错误，专门设计了完成量这种机制。

当一个任务需要等待另一个任务完成某操作后才能执行时，通过完成量（Completion）允许被等待任务在完成指定操作后通知等待任务，从而唤醒等待任务,实现任务同步。

多任务环境下，任务之间往往需要同步操作。通过完成量可以实现任务之间的同步。

Huawei LiteOS中完成量机制实现类似于系统中信号量机制，通过调用内核相关函数实现完成量机制的功能，Huawei LiteOS中完成量机制具有类似信号量机制的特性。

4.5.1.2 开发指导

使用场景

完成量应用于多种任务同步场合，是一种任务间同步机制。

完成量是通过等待和唤醒完成量实现任务间的同步。

功能

功能分类	接口名	描述
完成量初始化	init_completion	初始化一个完成量
等待完成量	wait_for_completion	永久等待完成量
超时等待完成量	wait_for_completion_timeout	超时等待完成量，超时时间为Tick
唤醒完成量	complete	唤醒等待该完成量的任务队列中的首个任务
唤醒完成量	complete_all	唤醒所有等待该完成量的任务

版权所有 © 华为技术有限公司

开发流程

完成量的典型开发流程：

1. 完成量初始化init_completion
   - 创建一个完成量
2. 超时等待完成量wait_for_completion_timeout
3. 永久等待完成量wait_for_completion
4. 唤醒完成量complete/complete_all
   - complete是唤醒一个等待该完成量的任务；
   - complete_all是唤醒所有等待该完成量的任务；

4.5.1.3 注意事项

- 完成量类似于信号量机制，参考信号量机制**注意事项**。即申请阻塞模式操作（永久阻塞和定时阻塞）时不能在中断中使用，原因：中断不能被阻塞。
- 完成量接口的完成量指针入参的合法性需要用户保证，即入参必须为合法完成量指针。

4.5.1.4 编程实例

实例描述

示例中，任务Example_TaskEntry创建一个任务Example_Completion，Example_Completion等待完成量阻塞，Example_TaskEntry唤醒该完成量。通过打印的先后顺序可以理解完成量操作时伴随的任务切换。

1. 在任务Example_TaskEntry创建任务Example_Completion，其中任务Example_Completion优先级高于Example_TaskEntry。
2. 在任务Example_Completion中等待完成量，阻塞，发生任务切换，执行任务Example_TaskEntry。
3. 在任务Example_TaskEntry唤醒完成量，发生任务切换，执行任务Example_Completion。
4. Example_Completion得以执行，直到该任务结束。
5. Example_TaskEntry得以执行，直到任务结束。

编程示例

代码实现如下：

```
##include "linux/completion.h"
#include "los_task.h"
//#include "osTest.h"

/*任务PID*/
UINT32 g_TestTaskID01;

/*完成量*/
struct completion example_completion;

/*用例任务入口函数*/
VOID Example_Completion()
{
    UINT32 uwRet;
```


版权所有 © 华为技术有限公司

```
    /*超时 等待方式等待完成量,超时时间为100 ticks*/
    printf("Example_Completion wait completion\n");
    uwRet = wait_for_completion_timeout(&example_completion,100);

    if(uwRet == 0)
    {
        printf("Example_Completion,wait completion timeout\n");
    }
    else
        printf("Example_Completion,wait completion success\n");
    return;
}

UINT32 Example_TaskEntry()
{
    UINT32 uwRet;
    TSK_INIT_PARAM_S stTask1;

    /*完成量初始化*/
    init_completion(&example_completion);


    /*创建任务*/
    memset(&stTask1, 0, sizeof(TSK_INIT_PARAM_S));
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_Completion;
    stTask1.pcName       = "EventTsk1";
    stTask1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = 8;
    uwRet = LOS_TaskCreate(&g_TestTaskID01, &stTask1);
    if(uwRet != LOS_OK)
    {
        printf("task create failed \n");
        return LOS_NOK;
    }

    /*唤醒完成量*/
    printf("Example_TaskEntry complete\n");

    complete(&example_completion);

    printf("Delete Task.\n");
    /*删除任务*/
    uwRet = LOS_TaskDelete(g_TestTaskID01);
    if(uwRet != LOS_OK)
    {
        printf("task delete failed \n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

结果验证

编译运行得到的结果为：

```
Example_Completion wait completion
Example_TaskEntry complete
Example_Completion,wait completion success
Delete Task.
```

完整实例代码

sample_completion.c

版权所有 © 华为技术有限公司

4.5.2 工作队列

4.5.2.1 概述

基本概念

工作队列提供一种机制：任何系统模块都可以将工作挂载到工作队列，由工作队列机制专门负责处理挂载在上面的任务。工作队列中的工作没有优先级，按照FIFO的方式进行处理。工作队列处理任务允许被重新调度甚至睡眠。

工作队列机制简化了内核任务的创建，负责处理挂载到工作队列的工作。支持工作队列处理任务的调度和睡眠，从而节省系统资源。

Huawei LiteOS的工作队列机制简化了内核任务的创建，提供了丰富的对外接口，以便对工作队列进行管理。当有工作挂载到工作队列中时，工作队列处理任务将被唤醒去处理工作队列中的工作，直到所有工作都被处理完，工作队列处理任务将重新进入睡眠状态。

4.5.2.2 开发指导

使用场景

工作队列运用在对时间要求不高的场景，原因是工作队列里的工作只有在工作队列处理任务得到运行的时候才能被逐一处理，而工作队列处理任务是否能及时唤醒调度取决于其他内核任务，另外在中断里也不能调度，只有在退出中断后才能得到调度。

功能

功能分类	接口名	描述
工作队列的创建	create_workqueue	创建一个工作队列
工作队列的销毁	destroy_workqueue	销毁一个工作队列
初始化工作	INIT_WORK	将工作和处理函数绑定
初始化延迟工作	INIT_DELAYED_WORK	将延迟工作和处理函数绑定
工作挂载	queue_work	向指定工作队列挂载一个工作
延迟工作挂载	queue_delayed_work	向指定工作队列挂载一个延迟工作
工作挂载	schedule_work	向系统默认工作队列挂载一个工作
延迟工作挂载	schedule_delayed_work	向系统默认工作队列挂载一个延迟工作
工作状态查询	work_busy	查询一个工作当前的状态
执行延迟工作	flush_delayed_work	取消延迟，立即执行工作

版权所有 © 华为技术有限公司

功能分类	接口名	描述
取消延迟工作	cancel_delayed_work	取消未执行的延迟工作
执行工作	flush_work	立即执行工作
取消工作	cancel_work_sync	取消工作并等待该工作执行完毕

开发流程

工作队列的典型流程：

1. 调用create_workqueue接口创建工作队列。
   - 初始化信号量，用于任务的睡眠和唤醒，以及初始化重要结构体搭建工作队列链表；
2. 调用queue_work/queue_delayed_work接口挂载工作。
   - 挂载工作分为工作和延迟工作两种，工作可以得到立即的挂载，而延迟工作根据延时参数延时后才被挂载到工作队列；
3. 调用cancel_delayed_work/cancel_work_sync接口取消工作。
4. 调用flush_work/flush_delayed_work接口立即执行工作。
5. 调用destroy_workqueue接口删除工作队列。
   - 系统先对资源进行加锁操作，再删除工作队列任务，释放信号量资源，释放内存空间等，最后解锁

4.5.2.3 注意事项

- 挂载到工作队列的工作是由工作队列处理任务进行处理，而任务的调度运行在时间上具有不确定性，所以对时间要求高的工作不建议使用工作队列进行处理。
- Huawei LiteOS利用工作队列名对工作队列进行标识，所以不能存在相同名称的工作队列。
- 可以利用schedule_work/schedule_delayed_work接口向系统默认的工作队列挂载工作，而不用再创建工作队列。

4.5.2.4 编程实例

实例描述

本实例实现如下功能：

1. 创建名称为wq_test的工作队列。
2. 分配工作内存并且初始化。
3. 向工作队列中挂载工作。
4. 立即执行工作。
5. 销毁工作队列。

编程示例

代码实现如下：

版权所有 © 华为技术有限公司

```
#include "los_config.h"
#include  "linux/workqueue.h"

static void work_func(struct work_struct *work)
{
  int i;
  for (i = 0; i < 2; i++)
  {
    printk("workqueue function is been called!..%d..%d..\n",i,work->work_status);
  }
}

UINT32 workqueue_test()
{
  struct workqueue_struct *wq;
  struct work_struct *work;
  UINT32 uwRet = LOS_OK;

  wq = create_workqueue("wq_test");
  dprintf("create the workqueue successfully!\n\n");

  work =(struct work_struct *)malloc(sizeof(struct work_struct));
  if (!work)
  {
     uwRet = LOS_FAIL;
  }
  dprintf("create work ok!\n\n");

  INIT_WORK(work, work_func);
  dprintf("init the work ok!\n\n");

  uwRet = queue_work(wq, work);
  dprintf("mount the work into workqueue successfully!\n\n");

  uwRet = flush_work(work);
  dprintf("flush the work ok!\n\n");

  destroy_workqueue(wq);
  dprintf("destroy the work ok!\n\n");

  return uwRet;
}
```

结果验证

编译运行得到的结果为：

```
 --- Test start---
create the workqueue successfully!

create work ok!

init the work ok!

mount the work into workqueue successfully!

workqueue function is been called!..0..3..
workqueue function is been called!..1..3..
flush the work ok!

destroy the work ok!

  [Passed] It_workqueue_1008

 --- Test End ---
```


版权所有 © 华为技术有限公司

完整实例代码

sample_workqueue.c

4.5.3 中断

4.5.3.1 概述

基本概念

Linux内核中有专门用于中断的函数，基于LiteOS 中断机制，适配Linux中相关中断接口，提高系统使用的友好度。

Huawei LiteOS的中断支持：

- 中断申请
- 中断删除
- 中断使能
- 中断屏蔽
- 中断底半部（基于workqueue）

4.5.3.2 开发指导

功能

Huawei LiteOS 系统中中断模块适配的linux接口如下表

接口名	描述
request_irq	中断申请
free_irq	中断删除
enable_irq	使能指定中断
disable_irq	屏蔽指定中断
irq_bottom_half	中断底半部程序挂载到workqueue

开发流程

1. 调用中断申请request_irq接口

   将中断处理程序注册到指定中断号对应的链表中，支持一个中断号注册多个中断处理程序。
2. 调用中断底半部irq_bottom_half接口

   系统内部将动态申请work，将入参底半部处理程序与work关联，并挂载到给定的workqueue，系统在空闲时会执行workqueue上的处理程序。
3. 调用enable_irq接口使能指定中断

版权所有 © 华为技术有限公司

4. 调用disable_irq接口屏蔽指定中断
5. 调用free_irq接口删除中断

4.5.3.3 注意事项

- 调用request_irq()申请中断时，中断处理程序入参要符合(int, void*)标准格式。直接调用LOS_HwiCreate()创建中断时，中断处理程序入参可以为NULL。
- 中断处理程序中禁止调用request_irq()和free_irq()接口。
- 当中断ID为共享中断时，不能通过LOS_HwiCreate()创建中断,且request_irq()入参dev指针不能为NULL，应与处理程序一一对应。直接调用LOS_HwiCreate()创建的中断，不能通过free_irq()删除。
- irq_bottom_half()函数传入的work_queue指针用户需保证合法，不能为无效指针。
- 中断底半部程序入参为work指针，该work为irq_bottom_half()中动态申请，在中断底半部程序中必须free，否则会引起内存泄露。

4.5.3.4 编程实例

实例描述

本实例实现如下功能。

1. 申请中断。
2. 删除中断。

编程示例

前提条件：

- 在los_config.h中，将OS_INCLUDE_HWI定义为YES。
- 在los_config.h中，设置最大硬中断个数OS_HWI_MAX_USED_NUM。

代码实现如下：

```
#include "los_hwi.h"
#define HWI_NUM_INT50 50
void uart_irqhandle_1(int irq,void *dev)
{
    printf("\nuart0:the function1 \n");
}
void hwi_test()
{
    int a = 1;
    void *dev = &a;
    unsigned long flags = 0;
    const char * name = "hwiTest";
    request_irq(HWI_NUM_INT50,uart_irqhandle_1,flags,name,dev);//创建中断
    free_irq(HWI_NUM_INT50,dev);//删除中断
}
```

完整实例

sample_irq.c

4.5.4 linux 接口

版权所有 © 华为技术有限公司

4.5.4.1 linux 适配接口

Huawei LiteOS提供的linux适配接口如下表所示。

说明

完全兼容表示功能与linux一致，错误码返回值要以实际代码为准；部分兼容表示部分功能与linux一致。

头文件	名称	类型	说明
timer.h	add_timer	添加定时器	完全兼容
atomic.h	atomic_add	原子变量做加法	完全兼容
atomic.h	atomic_add_return	原子变量做加法，并返回最新值	完全兼容
atomic.h	atomic_dec	原子变量减1	完全兼容
atomic.h	atomic_dec_and_test	递减原子变量，并测试是否为0	完全兼容
atomic.h	atomic_dec_return	原子变量减1，并返回最新值	完全兼容
atomic.h	atomic_inc	原子变量加1	完全兼容
atomic.h	atomic_inc_return	原子变量加1，并返回最新值	完全兼容
atomic.h	atomic_read	读取原子变量的值	完全兼容
atomic.h	atomic_sub	原子变量做减法	完全兼容
string.h	bzero	置字节字符串前n个字节为零且包括‘\0’	完全兼容
workqueue.h	cancel_delayed_work	取消挂到执行队列上的delayed work	完全兼容
workqueue.h	cancel_delayed_work_sync	同步取消挂到执行队列上的delayed work	完全兼容
workqueue.h	cancel_work_sync	同步取消一个work并等待其执行完毕	完全兼容
completion.h	complete	唤醒一个等待completion的线程	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
completion.h	complete_all	唤醒所有等待completion的线程	完全兼容
crc32.h	crc32	计算crc	完全兼容
crc32.h	crc32_accumulate	计算crc	完全兼容
workqueue.h	create_singlethread_workqueue	创建工作队列	完全兼容
workqueue.h	create_workqueue	创建工作队列	完全兼容
timer.h	del_timer	删除定时器	完全兼容
timer.h	del_timer_sync	删除定时器	完全兼容
workqueue.h	destroy_workqueue	销毁工作队列	完全兼容
interrupt.h	disable_irq	屏蔽中断	完全兼容
kernel.h	div_s64	除法运算	完全兼容
kernel.h	div_s64_rem	除法运算	完全兼容
dlfcn.h	dlclose	关闭指定句柄的动态链接库	完全兼容
dlfcn.h	dlopen	打开一个动态链接库，并返回动态链接库的句柄	完全兼容
dlfcn.h	dlsym	根据动态链接库操作句柄与符号，返回符号对应的地址。	完全兼容
kernel.h	do_div_imp	除法运算	完全兼容
kernel.h	do_div_s64_imp	除法运算	完全兼容
rwsem.h	down_read	读者加锁操作	完全兼容
rwsem.h	down_read_trylock	读者加锁操作	完全兼容
rwsem.h	down_write	写者加锁操作	完全兼容
rwsem.h	down_write_trylock	写者加锁操作	完全兼容
interrupt.h	enable_irq	使能中断	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
kernel.h	ERR_PTR	错误码操作	完全兼容
fs.h	fb_alloc_cmap	分配颜色表内存空间	完全兼容
fs.h	fb_cmap_to_user	复制颜色表	完全兼容
fs.h	fb_copy_cmap	复制颜色表	完全兼容
fs.h	fb_dealloc_cmap	释放颜色表内存空间	完全兼容
fs.h	fb_default_cmap	设置默认颜色表	完全兼容
fs.h	fb_pan_display	刷新操作屏	完全兼容
fs.h	fb_set_cmap	设定颜色表	完全兼容
fs.h	fb_set_user_cmap	设定颜色表	完全兼容
fs.h	fb_set_var	设置fbinfo显示模式和可变参数	完全兼容
workqueue.h	flush_delayed_work	等待一个delayed work执行完毕	完全兼容
workqueue.h	flush_work	等待一个work执行完毕	完全兼容
fs.h	framebuffer_alloc	向内核申请一块空间	完全兼容
fs.h	framebuffer_release	向内核释放一块空间	完全兼容
interrupt.h	free_irq	释放中断服务	完全兼容
kernel.h	hi_sched_clock	获取时间	完全兼容
compiler.h	init_completion	初始化completion结构	完全兼容
workqueue.h	INIT_DELAYED_WORK	初始化delay_work	完全兼容
list.h	INIT_LIST_HEAD	链表初始化	完全兼容
timer.h	init_timer	初始化定时器	完全兼容
wait.h	init_waitqueue_head	初始化一个已经存在的等待队列头	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
workqueue.h	INIT_WORK	初始化work	完全兼容
interrupt.h	irq_bottom_half	中断底半部接口	该接口为Huawei LiteOS特有，linux中没有
kernel.h	IS_ERR	错误码判断	完全兼容
rtc.h	is_leap_year	是否是闰年	完全兼容
jiffies.h	jiffies_to_msecs	将机器启动时起的时间滴答数转换成毫秒	完全兼容
kernel.h	jiffies_to_tick	时间jiffies转换为tick	完全兼容
slab.h	kfree	释放内存空间	完全兼容
slab.h	kmalloc	分配内存空间	完全兼容
slab.h	kzalloc	分配内存空间并初始化	完全兼容
list.h	list_add	添加链表	完全兼容
list.h	list_add_tail	添加链表	完全兼容
list.h	list_del	删除链表	完全兼容
list.h	list_entry	返回包含结构的指针	完全兼容
list.h	list_first_entry	获取链表第一个元素	完全兼容
list.h	list_for_each	遍历链表	完全兼容
list.h	list_for_each_entry	遍历链表	完全兼容
list.h	list_for_each_entry_reverse	遍历链表	完全兼容
list.h	list_for_each_entry_safe	遍历链表	完全兼容
list.h	list_for_each_safe	遍历链表	完全兼容
list.h	LIST_HEAD	链表初始化	完全兼容
list.h	LIST_HEAD_INIT	链表初始化	完全兼容
list.h	list_is_last	判断是否为链表最后元素	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
list.h	list_move	从一个链表移动节点到链表	完全兼容
string.h	memchr	查找指定字符，在一段内存区域中	完全兼容
string.h	memcmp	比较两字符串前n个字节	完全兼容
string.h	memcpy	复制一段内存内容，源和目标区域重叠不保证正确	完全兼容
string.h	memmove	复制一段内存内容，源和目标区域重叠保证正确	完全兼容
string.h	memset	设置内存中一段字节内容为指定值	完全兼容
timer.h	mod_timer	添加定时器	完全兼容
mount.h	mount	挂载指定系统分区到一个文件夹下	完全兼容
delay.h	msleep	延时，程序暂停若干时间（单位ms）	完全兼容
prctl.h	prctl	进程操作函数Linux中支持多种参数，Huawei LiteOS中只支持PR_SET_NAME参数设置线程名，在使用POSIX接口创建线程的时候，建议进入线程的时候设置线程名	部分兼容
kernel.h	PTR_ERR	错误码操作	完全兼容
workqueue.h	queue_delayed_work	往指定工作队列中插入一个delayed work	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
workqueue.h	queue_work	往指定工作队列中插入一个work	完全兼容
rbtree.h	rb_erase	删除节点	完全兼容
rbtree.h	rb_first	求第一个节点	完全兼容
rbtree.h	rb_insert_color	新插入节点着色	完全兼容
rbtree.h	rb_next	求后继节点	完全兼容
rbtree.h	rb_prev	求前驱节点	完全兼容
rbtree.h	rb_replace_node	替换节点	完全兼容
io.h	readb	从I/O读取1字节	完全兼容
io.h	readl	从I/O读取4字节	完全兼容
uio.h	readv	将读入的数据按同样顺序散布读到缓冲区中	完全兼容
io.h	readw	从I/O读取2字节	完全兼容
fs.h	register_framebuffer	将fbinfo结构注册到内核	完全兼容
interrupt.h	request_irq	注册中断服务	完全兼容
rtc.h	rtc_time_to_tm	绝对时间转换为年月日时间分秒	完全兼容
rtc.h	rtc_tm_to_time	年月日时间分秒转换为绝对时间	完全兼容
workqueue.h	schedule_delayed_work	往系统默认工作队列中插入一个delayed work	完全兼容
kernel.h	schedule_timeout	任务调度	完全兼容
kernel.h	schedule_timeout_interruptible	任务调度	完全兼容
kernel.h	schedule_timeout_uninterruptible	任务调度	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
workqueue.h	schedule_work	往系统默认工作队列中插入一个work	完全兼容
seq_file.h	seq_lseek	偏移seq流指针	完全兼容
seq_file.h	seq_open	打开seq流	完全兼容
seq_file.h	seq_printf	向seq流方式写格式化信息	完全兼容
seq_file.h	seq_read	读seq流	完全兼容
seq_file.h	seq_release	释放seq流所分配的动态内存空间	完全兼容
scatterlist.h	sg_init_one	初始化集散序列	完全兼容
scatterlist.h	sg_init_table	初始化集散序列	完全兼容
scatterlist.h	sg_mark_end	标记集散序列结束	完全兼容
scatterlist.h	sg_set_buf	设置集散序列	完全兼容
seq_file.h	single_open	打开seq流	完全兼容
seq_file.h	single_release	释放seq流所分配的动态内存空间	完全兼容
string.h	strcasecmp	比较两字符串，忽略大小写	完全兼容
string.h	strcasestr	查找字符串中首次出现某字符串的位置，不区分大小写	完全兼容
string.h	strcat	将字符串A连接到字符串B尾	完全兼容
string.h	strchr	查找字符串中首次出现某字符的位置	完全兼容
string.h	strcmp	比较两字符串	完全兼容
string.h	strcoll	比较两字符串，特定语言环境下	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
string.h	strcpy	复制字符串	完全兼容
string.h	strcspn	查找字符串开头连续不含指定字符的字符数目并返回	完全兼容
string.h	strdup	拷贝字符串到新建位置处	完全兼容
string.h	strerror	获取系统错误信息或打印用户程序错误信息	完全兼容
string.h	strlcpy	复制指定长度字符串	完全兼容
string.h	strlcpy	拷贝字符串	完全兼容
string.h	strlen	计算给定字符串长度	完全兼容
string.h	strncasecmp	比较两字符串前几个字节，忽略大小写	完全兼容
string.h	strncat	将字符串A指定字符个数连接到字符串B尾	完全兼容
string.h	strncmp	比较两字符串前n个字符	完全兼容
string.h	strncpy	复制指定字节字符串	完全兼容
string.h	strpbrk	查找字符串中最先含有特定字符串中任一字符的位置并返回	完全兼容
string.h	strrchr	查找字符串中从末尾开始，首次出现某字符串的位置	完全兼容
string.h	strsep	分解字符串为一组字符串	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
string.h	strspn	查找字符串中第一个不在指定字符串中出现的字符下标并返回	完全兼容
string.h	strstr	查找字符串中首次出现某字符串的位置	完全兼容
string.h	strtok	分割字符串	完全兼容
string.h	strtok_r	分割字符串	完全兼容
string.h	strtoul	将字符串转换成无符号长整型数	完全兼容
string.h	strxfrm	字符串转换	完全兼容
mount.h	umount	卸载文件系统	完全兼容
fs.h	unregister_framebuffer	注销fbinfo	完全兼容
rwsem.h	up_read	读者解锁操作	完全兼容
rwsem.h	up_write	写者解锁操作	完全兼容
slab.h	vfree	释放内存空间	完全兼容
slab.h	vmalloc	分配内存空间	完全兼容
wait.h	wait_event	等待事件	完全兼容
wait.h	wait_event_interruptible	等待事件	完全兼容
wait.h	wait_event_interruptible_timeout	超时等待事件	完全兼容
completion.h	wait_for_completion	等待completion	完全兼容
completion.h	wait_for_completion_timeout	等待completion，若超时则结束等待	完全兼容
wait.h	waitqueue_active	检查等待队列是否为空	完全兼容
wait.h	wake_up	唤醒任务	完全兼容

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
wait.h	wake_up_interruptible	负责唤醒状态为TASK_INTERRUPTIBLE的任务	完全兼容
workqueue.h	work_busy	检测work状态	完全兼容
io.h	writeb	向I/O写入1字节	完全兼容
io.h	writel	向I/O写入4字节	完全兼容
uio.h	writev	将数据存储在多个不连接的缓冲区，同时写出去	完全兼容
io.h	writew	向I/O写入2字节	完全兼容
zlib.h	zlib_deflate	压缩数据	完全兼容
zlib.h	zlib_deflateEnd	释放为当前流分配的动态数据结构	完全兼容
zlib.h	zlib_deflateInit	初始化zlib状态	完全兼容
zlib.h	zlib_inflate	解压数据	完全兼容
zlib.h	zlib_inflateEnd	释放为当前流分配的动态数据结构	完全兼容
zlib.h	zlib_inflateInit	为解压初始化内部流状态	完全兼容
zlib.h	zlib_inflateInit2	解压数据	完全兼容

4.5.4.2 linux 不支持接口

Huawei LiteOS的linux适配接口中，有一些未支持，具体信息如下表所示。

文件名	函数接口名	说明	是否支持
adp.c	__assert	断言，用于判断程序运行的正确性	不支持
adp.c	__cxa_atexit	进程退出	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
adp.c	__tls_get_addr	TLS模块查找变量地址	不支持
wait.h	add_wait_queue	将进程插入队列尾部	不支持
adp.c	alarm	alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds 为0, 则之前设置的闹钟会被取消, 并将剩下的时间返回。	不支持
atomic.h	atomic_set	设置原子变量的值	不支持
bug.h	BUG	提供断言并输出信息	不支持
adp.c	chroot	chroot()用来改变根目录为参数path 所指定的目录。只有超级用户才允许改变根目录, 子进程将继承新的根目录。	不支持
adp.c	closelog	关闭已打开的system log的连接	不支持
sched.h	cond_resched	调度一个新程序投入运行	不支持
kernel.h	copy_from_user	将用户空间的数据传送到内核空间	不支持
kernel.h	copy_to_user	从内核区中读取数据到用户区	不支持
adp.c	daemon	创建守护进程	不支持
wait.h	DECLARE_WAITQUEUE	定义并初始化一个等待队列	不支持
semaphore.h	down	获取信号量，否则进入睡眠状态，且不可唤醒	不支持
semaphore.h	down_interruptible	获取信号量，否则进入睡眠状态，等待信号量被释放后，激活该程	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
semaphore.h	down_trylock	试图获取信号量，否则立刻返回非零值，调用者不会睡眠	不支持
adp.c	execve	execve()用来执行参数filename字符串所代表的文件路径, 第二个参数系利用数组指针来传递给执行文件, 最后一个参数则为传递给执行文件的新环境变量数组。	不支持
adp.c	fchown	fchown()会将参数fd指定文件的所有者变更为参数owner代表的用户, 而将该文件的组变更为参数group组。如果参数owner或group为-1, 对映的所有者或组有所改变。参数fd 为已打开的文件描述词。当root用fchown()改变文件所有者或组时, 该文件若具S_ISUID或S_ISGID权限, 则会清除此权限位。	不支持
adp.c	fork	创建一个与原来进程几乎完全相同的进程	不支持
adp.c	fs_fssync	文件同步	不支持
adp.c	getdtablesize	获取进程所能打开的最大文件数	不支持
adp.c	gethostname	获得本主机名称	不支持
adp.c	getpwnam	getpwnam()用来逐一搜索参数name 指定的账号名称, 找到时便将该用户的数据以passwd结构返回。passwd结构请参考getpwent()。	不支持
adp.c	getrlimit	获取系统资源上限	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
semaphore.h	init_MUTEX	初始化信号量值为1	不支持
semaphore.h	init_MUTEX_LOCKED	初始化信号量值为0	不支持
adp.c	initgroups	initgroups()用来从组文件(/etc/group)中读取一项组数据, 若该组数据的成员中有参数user时, 便将参数group组识别码加入到此数据中。	不支持
kernel.h	ioremap_cached	内核虚拟地址映射	不支持
kernel.h	ioremap_nocache	内核虚拟地址映射	不支持
kernel.h	iounmap	取消ioremap函数所做的映射	不支持
compiler.h	likely	判断语句，值为真可能性更大	不支持
adp.c	linux_module_init	模块加载	不支持
list.h	list_empty	链表是否为空判断	不支持
kernel.h	misc_deregister	移除一个混杂设备	不支持
kernel.h	misc_register	注册一个混杂设备	不支持
adp.c	mmap	mmap()用来将某个文件内容映射到内存中, 对该内存区域的存取即是直接对该文件内容的读写。参数start指向欲对应的内存起始地址, 通常设为NULL, 代表让系统自动选定地址, 对应成功后该地址会返回。参数length代表将文件中多大的部分对应到内存。	不支持
module.h	module_put	减少模块使用计数	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
adp.c	munmap	munmap()用来取消参数start所指的映射内存起始地址, 参数length则是欲取消的内存大小。当进程结束或利用exec相关函数来执行其他程序时, 映射内存会自动解除, 但关闭对应的文件描述词时不会解除映射。	不支持
adp.c	nice	nice()用来改变进程的进程执行优先顺序。参数inc数值越大则优先顺序排在越后面, 即表示进程执行会越慢。只有超级用户才能使用负的inc值, 代表优先顺序排在前面, 进程执行会较快。	不支持
adp.c	pipe	pipe()会建立管道, 并将文件描述词由参数filedes数组返回。filedes[0]为管道里的读取端, filedes[1]则为管道的写入端。	不支持
kernel.h	printtime	打印系统时间	不支持
adp.c	readlink	readlink()会将参数path的符号连接内容存到参数buf所指的内存空间, 返回的内容不是以NULL作字符串结尾, 但会将字符串的字符数返回。若参数bufsiz小于符号连接的内容长度, 过长的内容会被截断。	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
adp.c	recvmsg	recvmsg()用来接收远程主机经指定的socket传来的数据。参数s为已建立好连线的socket, 如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据结构内容, 参数flags一般设0, 详细描述请参考send()。关于结构msghdr的定义请参考sendmsg()。	不支持
wait.h	remove_wait_queue	将等待队列wait从附属的等待队列头q指向的等待队列链表中移除	不支持
adp.c	setgroups	setgroups()用来将list 数组中所标明的组加入到目前进程的组设置中。参数size为list()的gid_t数目, 最大值为NGROUP(32)。	不支持
sched.h	signal_pending	检查当前进程是否有信号处理	不支持
adp.c	sigset	将sig指定的信号部署改为disp指定的部署	不支持
string.h	simple_strtol	将一个字符串转换成unsigend long long型数据	不支持
adp.c	syscall	系统调用	不支持
timer.h	timer_pending	判断一个处在定时器管理队列中的定时器对象是否已经被调度执行	不支持
module.h	try_module_get	增加模块使用计数	不支持
rwsem.h	try_module_get	把写者降级为读者	不支持
mount.h	umount2	文件卸载	不支持
compiler.h	unlikely	判断语句，值为假可能性更大	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	说明	是否支持
semaphore.h	up	释放信号量，并唤醒等待该资源进程队列的第一个进程	不支持
adp.c	waitpid	waitpid()会暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束, 则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status返回, 而子进程的进程识别码也会一快返回。如果不在意结束状态值, 则参数status可以设成NULL。	不支持
wakelock.h	wake_lock	激活锁	不支持
wakelock.h	wake_lock_active	判断锁当前是否有效	不支持
wakelock.h	wake_lock_init	初始化一个锁	不支持
wakelock.h	wake_unlock	解锁使之成为无效锁	不支持
watchdog.h	watchdog_init	看门狗初始化	不支持

4.6 C++支持

4.6.1 概述

基本概念

C++作为目前使用最广泛的编程语言之一，支持类、封装、重载等特性，是在C语言基础上开发的一种面向对象的编程语言。

运作机制

STL（Standard Template Library）标准模板库，是一些“容器”的集合，也是算法和其他一些组件的集合。其目的是标准化组件，使用标准化组件后就可以不用重新开发，直接使用现成的组件。

版权所有 © 华为技术有限公司

4.6.2 开发指导

功能

功能分类	接口名	描述
使用C++特性的前置条件	LOS_CppSystemInit	C++构造函数初始化

开发流程

使用C++特性之前，需要调用函数LOS_CppSystemInit，实现C++构造函数初始化，

其中被初始化的构造函数存在init_array这个段中，段区间通过变量__init_array_start__、__init_array_end__传递。

由于在分散加载应用场景下，C++初始化中涉及到的相关代码、数据段加载的时机会有所不同，所以在用户开启和不开启分散加载特性这两种情况下，C++初始化函数LOS_CppSystemInit的调用有所不同。

不开启分散加载特性

在不开启分散加载特性的情况下，用户需要在调用相关C++代码之前，以NO_SCATTER为参数调用LOS_CppSystemInit：

```
LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
NO_SCATTER);
```

表 4-3 参数说明

参数	参数说明
__init_array_start__	起始段
__init_array_end__	结束段
NO_SCATTER	表示用户并未开启分散加载特性

开启分散加载特性

- 如果用户在分散加载的快速启动阶段需要调用相关C++代码，则用户需要在该阶段调用C++代码之前以BEFORE_SCATTER参数如下调用LOS_CppSystemInit：
  ```
  LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
  BEFORE_SCATTER);
  ```

  表 4-4 参数说明

参数	参数说明
__init_array_start__	起始段
__init_array_end__	结束段

版权所有 © 华为技术有限公司

参数	参数说明
BEFORE_SCATTER	表示用户是在分散加载快速启动阶段调用的LOS_CppSystemInit

在分散加载的非快速启动阶段再以AFTER_SCATTER参数如下调用LOS_CppSystemInit：

```
LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
AFTER_SCATTER);
```

表 4-5 参数说明

参数	参数说明
__init_array_start__	起始段
__init_array_end__	结束段
AFTER_SCATTER	表示用户是在分散加载非快速启动阶段调用的LOS_CppSystemInit

- 如果用户在分散加载的快速启动阶段无须调用相关C++代码，除了使用以上的方式在快速启动阶段前和非快速启动阶段前分别调用LOS_CppSystemInit外，用户还可以在分散加载的非快速启动阶段，分别以BEFORE_SCATTER和AFTER_SCATTER参数连续两次调用LOS_CppSystemInit：

  ```
  LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
  BEFORE_SCATTER);

  LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
  AFTER_SCATTER);
  ```

  或者以NO_SCATTER为参数调用一次LOS_CppSystemInit：

  ```
  LOS_CppSystemInit((unsigned long)&__init_array_start__, (unsigned long)&__init_array_end__,
  NO_SCATTER);
  ```

调用C库函数

在C++中调用C程序的函数，注意在声明该函数的时候增加如下语句：

extern "C"。

4.6.3 注意事项

- Huawei LiteOS暂不支持C++异常机制、RTTI。
- Huawei LiteOS中，C++暂不支持I/O字符流、I/O文件流等相关操作。

4.6.4 编程实例

实例描述

在代码初始化中，进行C++构造函数初始化，进而让程序可以使用C++特性。由于此处使用分散加载特性，所以需两次调用LOS_CppSystemInit

版权所有 © 华为技术有限公司

编程实例

```
void app_init(void)
{
......
/* 分散加载快速启动阶段C++初始化 */
LOS_CppSystemInit((UINT32)&__init_array_start__, (UINT32)&__init_array_end__,
BEFORE_SCATTER);
/* 分散加载 */
LOS_ScatterLoad(0x100000, flash_read, NAND_READ_ALIGN_SIZE);
/* 分散加载非快速启动阶段C++初始化 */
LOS_CppSystemInit((UINT32)&__init_array_start__, (UINT32)&__init_array_end__,
AFTER_SCATTER);
......
}
```

4.7 MMU

4.7.1 概述

基本概念

MMU全称“Memory Management Unit”，顾名思义就是“内存管理单元”。

运作机制

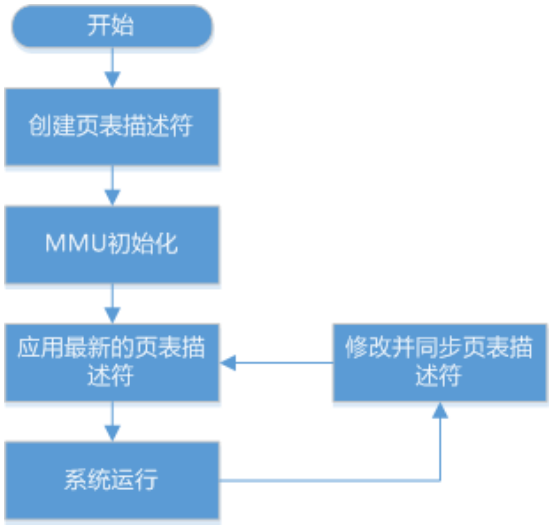
建立页表描述符号表，将物理地址映射成虚拟地址，以虚拟地址为媒介来操作和管理实际的物理内存。

页表描述符号表，是由用户根据所使用的主芯片的页表描述格式，去创建、修改和管理的，而内存则依据页表描述符号表进行映射、权限控制等。页表描述符号表在创建或修改后，需要将它写入协处理器CP15才能生效，而协处理CP15正是内存管理的实际执行者。

综合来说，对MMU操作就是通过修改页表描述符和控制CP15协处理器来实现的，具体运作流程如下图1所示。

版权所有 © 华为技术有限公司

图 4-4



Huawei LiteOS的MMU有两个方面的作用：

- 提供硬件机制的内存cache/nocache属性的控制接口。
- 提供硬件机制的内存访问权限控制接口。

4.7.2 开发指导

使用场景

系统内部有些内存不希望被修改，否则会造成不可预测的后果，此时可以用MMU修改该段内存的访问权限。访问该段内存时会检查访问权限，若权限不正确则会触发异常，起到保护该段内存数据的作用。

通过cache+buffer可控制内存缓存状态，包括：关闭cache、写通、写回。

功能

Huawei LiteOS中MMU模块为用户提供以下接口：

表 4-6

功能分类	接口名	描述
内存访问权限控制	LOS_MMUParamSet	修改指定地址段的内存cache状态、buffer状态、读写权限状态

参数：

BUFFER_ENABLE/BUFFER_DISABLE ---- 使能/关闭buffer

CACHE_ENABLE/CACHE_DISABLE ---- 使能/关闭cache

ACCESS_PERM_RW_RW/ACCESS_PERM_RO_RO ---- 可读写/只读

举例：

版权所有 © 华为技术有限公司

LOS_MMUParamSet(&__text_start, &__text_end, BUFFER_ENABLE|CACHE_ENABLE|ACCESS_PERM_RO_RO);

描述：

将__text_start，__text_end这两个地址之间的内存设置为（写回）启用cache、buffer、只读。

说明

LOS_MMUParamSet的入参1和入参2需要4KB对齐，入参3最好是显示的选择以上列举的3种类型的宏。

4.7.3 注意事项

- 目前MMU二级页表可操作最小内存单位是4KB，所以要设置访问权限的内存区域的起始地址和结束地址都要4KB对齐。一级页表修改未做对外接口，无需关注。
- 通过该接口可控制的内存大小，取决于分配用于存放二级页表的内存大小，例如Hi3516a：预留了0x80000000~0x80008000的内存用于存放页表，其中0x8000000~0x80004000用于存放一级页表，0x80004000~0x80008000（16K）用于存放二级页表（每条表项占用4byte，每条表项对应4K内存），所以最多可以控制16M的可配置内存。

4.7.4 编程实例

实例描述

调用接口LOS_MMUParamSet，修改内存区域的读写权限，再通过在该内存区域进行写操作，查看是否读写权限正确修改。

步骤1 修改一段区间的内存读写权限为只读。

步骤2 在该段区间的内存中进行简单的写操作。

系统进入异常，说明将该内存设置为“只读”成功。

步骤3 注释掉2中的写操作，而是直接调用接口将读写权限重新修改为可读可写。

系统不进入异常，说明将该内存设置为“可读可写”成功。

----结束

编程实例

```
UINT32 MMU_Sample()
{
    UINT32 *pAlignaddr;
    PRINTK("---- TEST START ----\n");
    pAlignaddr = (UINT32 *)(&__text_start);
    PRINTK(">>1\n");
    LOS_MMUParamSet(&__text_start, &__text_end, BUFFER_ENABLE|CACHE_ENABLE|ACCESS_PERM_RO_RO);
    *pAlignaddr = 0xa; //if done, be exc
    PRINTK(">>2\n");
    LOS_MMUParamSet(&__text_start, &__text_end, BUFFER_ENABLE|CACHE_ENABLE|ACCESS_PERM_RW_RW);
    *pAlignaddr = 0xb;
    PRINTK(">>3\n");
    PRINTK("---- TEST END ----\n");
    return LOS_OK;
}
```


版权所有 © 华为技术有限公司

运行结果

不注释代码“*pAlignaddr = 0xa;”

```
---- TEST START ----
>>1
uwExcType = 0x4
puwExcBuffAddr pc = 0x8000adc8
puwExcBuffAddr lr = 0x8000aa78
puwExcBuffAddr sp = 0x80093e30
puwExcBuffAddr fp = 0x80093e44
*******backtrace begin*******
traceback 0 -- lr = 0x8000b1e4
traceback 0 -- fp = 0x80093e84
traceback 1 -- lr = 0x8000b2c4
traceback 1 -- fp = 0x80093e9c
traceback 2 -- lr = 0x8000ae8c
traceback 2 -- fp = 0x80093eac
traceback 3 -- lr = 0x8000ac4c
traceback 3 -- fp = 0x80093ebc
traceback 4 -- lr = 0x800104fc
traceback 4 -- fp = 0x80093ecc
traceback 5 -- lr = 0x80026e1c
traceback 5 -- fp = 0x11111111

Name                        TID    Priority   Status
----                        ---    --------   --------
Swt_Task                    0x0    0          QueuePend
IdleCore000                 0x1    31         Ready
IT_TST_INI                  0x2    25         Running
R0          = 0x800040c0
R1          = 0x800040a8
R2          = 0x800040a4
R3          = 0x0
R4          = 0x8000a000
R5          = 0x0
R6          = 0x8002a000
R7          = 0x800302e4
R8          = 0x80051df4
R9          = 0x8002c020
R10         = 0x80051df4
R11         = 0x80093e44
R12         = 0x2a
SP          = 0x80093e30
LR          = 0x8000aa78
PC          = 0x8000adc8
CPSR        = 0x200f0013
pcTaskName = IT_TST_INI
TaskID = 2
Task StackSize = 86016
system mem addr:0x800561c0

stack name           stack addr     total size   used size
----------           ----------     ----------   ---------
undef_stack_addr     0x8002c820     0x20         0x0
abt_stack_addr       0x8002c840     0x20         0xb
irq_stack_addr       0x8002c880     0x40         0x10
fiq_stack_addr       0x8002c8c0     0x40         0x0
svc_stack_addr       0x8002d8c0     0x1000       0x14f
startup_stack_addr   0x800563a0     0x200        0x138
```

注释掉代码“*pAlignaddr = 0xa;”

```
---- TEST START ----
>>1
>>2
>>3
---- TEST END ----
```


版权所有 © 华为技术有限公司

完整实例代码

MMU_Sample1.c

4.8 原子操作

4.8.1 概述

基本概念

在支持多任务的操作系统中，修改一块内存区域的数据需要“读取-修改-写入”三个步骤。然而同一内存区域的数据可能同时被多个任务访问，如果在修改数据的过程中被其他任务打断，就会造成该操作的执行结果无法预知。

使用开关中断的方法固然可以保证多任务执行结果符合预期，显然这种方法会影响系统性能。

ARMv6架构引入了LDREX和STREX指令，以支持对共享存储器更缜密的非阻塞同步。由此实现的原子操作能确保对同一数据的“读取-修改-写入”操作在它的执行期间不会被打断，即操作的原子性。

运作机制

Huawei LiteOS通过对ARMv6架构中的LDREX和STREX进行封装，向用户提供了一套原子性的操作接口。

- **LDREX Rx, [Ry]**

  读取内存中的值，并标记对该段内存的独占访问：
  - 读取寄存器Ry指向的4字节内存数据，保存到Rx寄存器中。
  - 对Ry指向的内存区域添加独占访问标记。
- **STREX Rf, Rx, [Ry]**

  检查内存是否有独占访问标记，如果有则更新内存值并清空标记，否则不更新内存：
  - 有独占访问标记
    i. 将寄存器Rx中的值更新到寄存器Ry指向的内存。
    ii. 标志寄存器Rf置为0。
  - 没有独占访问标记
    i. 不更新内存。
    ii. 标志寄存器Rf置为1。
- 判断标志寄存器
  - 标志寄存器为0时，退出循环，原子操作结束。
  - 标志寄存器为1时，继续循环，重新进行原子操作。

版权所有 © 华为技术有限公司

4.8.2 开发指导

使用场景

有多个任务对同一个内存数据进行加减或交换操作时，使用原子操作保证结果的可预知性。

功能

Huawei LiteOS系统中的原子操作模块为用户提供下面几种功能。

表 4-7 功能列表

功能分类	接口名	描述
加	LOS_AtomicAdd	对内存数据做加减
	LOS_AtomicInc	对内存数据加1
	LOS_AtomicIncRet	对内存数据加1并返回
减	LOS_AtomicDec	对内存数据减1
	LOS_AtomicDecRet	对内存数据减1并返回
交换	LOS_AtomicXchgByte	交换8位内存数据
	LOS_AtomicXchg16bits	交换16位内存数据
	LOS_AtomicXchg32bits	交换32位内存数据
比较后交换	LOS_AtomicCmpXchgByte	比较并交换8位内存数据
	LOS_AtomicCmpXchg16bits	比较并交换16位内存数据
	LOS_AtomicCmpXchg32bits	比较并交换32位内存数据

注意

原子操作中，操作数以及其结果不能超过函数所支持位数的最大值。

平台差异性

a7、a17核与arm926的原子操作实现方式不同：a7与a17通过ldrex和strex指令实现，但由于arm926不支持ldrex和strex指令，arm926的原子操作仍使用开关中断实现。

4.8.3 注意事项

- 目前原子操作接口只支持整型数据。

版权所有 © 华为技术有限公司

4.8.4 编程实例

实例描述

调用原子操作相关接口，观察结果：

步骤1 创建两个任务

1. 任务一用LOS_AtomicInc对全局变量加100次。
2. 任务二用LOS_AtomicDec对全局变量减100次。

步骤2 子任务结束后在主任务中打印全局变量的值。

----结束

编程示例

```
#include "los_atomic.h"
UINT32 g_TestTaskID01;
UINT32 g_TestTaskID02;
UINT32 g_sum;
UINT32 g_count;
UINT32 It_atomic_001_f01()
{
    int i = 0;
    for(i = 0; i < 100; ++i)
    {
        LOS_AtomicInc(&g_sum);
    }

    ++g_count;
    return LOS_OK;
}
UINT32 It_atomic_001_f02()
{
    int i = 0;
    for(i = 0; i < 100; ++i)
    {
        LOS_AtomicDec(&g_sum);
    }

    ++g_count;
    return LOS_OK;
}
UINT32 it_atomic_test()
{
    UINT32 uwRet,uwCpupUse;
    INTPTR uvIntSave;
    TSK_INIT_PARAM_S stTask1={0};
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)It_atomic_001_f01;
    stTask1.pcName       = "TestAtomicTsk1";
    stTask1.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = 4;
    stTask1.uwResved     = LOS_TASK_STATUS_DETACHED;

    TSK_INIT_PARAM_S stTask2={0};
    stTask2.pfnTaskEntry = (TSK_ENTRY_FUNC)It_atomic_001_f02;
    stTask2.pcName       = "TestAtomicTsk2";
    stTask2.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    stTask2.usTaskPrio   = 4;
    stTask2.uwResved     = LOS_TASK_STATUS_DETACHED;

    LOS_TaskLock();
    LOS_TaskCreate(&g_TestTaskID01, &stTask1);
    LOS_TaskCreate(&g_TestTaskID02, &stTask2);
    LOS_TaskUnlock();
```


版权所有 © 华为技术有限公司

```
    while(g_count != 2);
    PRINTK("g_sum = %d\n", g_sum);

    return LOS_OK;
}
```

结果验证

```
g_sum = 0
```

完整实例代码

sample_atomic.c

4.9 休眠唤醒

4.9.1 概述

基本概念

休眠唤醒是Huawei LiteOS提供的一套保存系统现场镜像以及从系统现场镜像恢复运行的机制。

用户在系统运行的某一个时刻调用RunStop的接口，可以将此时系统的状态（CPU现场及内存）快照保存到flash上，系统重新启动之后，可以从flash上的快照恢复系统，从快照前的状态继续运行。

休眠唤醒机制可以被用来实现wifi模块的下电唤醒业务：在系统的wifi模块初始化并保持稳定后，调用休眠唤醒接口保存系统快照，这份快照包含了wifi模块所需要的所有CPU线程及内存状态，系统在运行一段时间后，wifi模块没有业务需要处理时，主核下电进入低功耗模式。当wifi模块有业务处理时，mcu发送特定数据包唤醒主核上电，恢复wifi硬件状态后，系统从flash快照处恢复运行，使得系统重新上电后wifi模块无需重新初始化，可以实现wifi连接的保活不断连。

4.9.2 开发指导

使用场景

如果用户希望系统在运行一段时间后，将此时系统的状态快照保存到介质上，在某个时刻可以从该快照重新运行系统，重新运行的瞬间系统状态与快照时的状态一致。在这种应用场景下用户可以选择使用休眠唤醒特性来实现。

在Huawei LiteOS IPC项目上，休眠唤醒特性被用来实现wifi芯片的休眠唤醒：在系统的wifi业务运行稳定后，将此时系统状态快照保存。当系统进入空闲状态时主核下电进入低功耗模式，有wifi报文到来时mmu对主核上电，系统从快照处恢复执行。可以实现主核在低功耗模式下wifi状态不断连，以及wifi状态的快速唤醒。

功能

Huawei LiteOS系统中的休眠唤醒模块为用户提供下面几种功能。

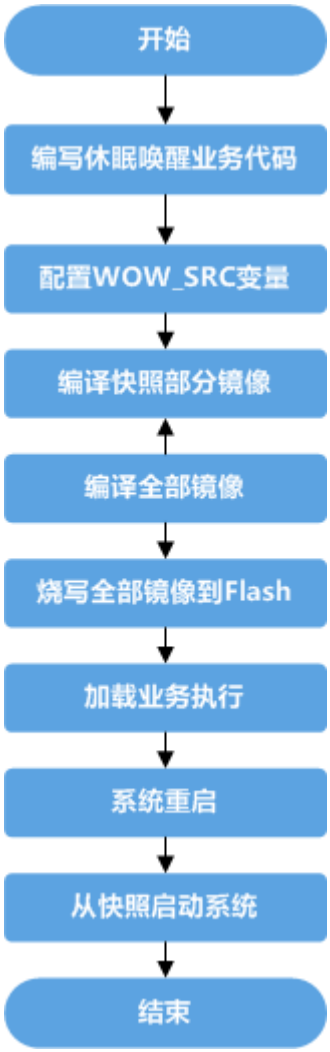
版权所有 © 华为技术有限公司

表 4-8 功能列表

接口名	描述
LOS_MakeImage	将系统状态快照到用户指定的介质上

开发流程

图 4-5 休眠唤醒流程图



开发流程

步骤1 调用LOS_MakeImage，编写快照业务代码

业务代码入口为函数app_init，该函数位于os_adapt.c。

说明

os_adapt.c位于Huawei_LiteOS代码包的platform/bsp/hi3516a/os_adapt路径下。

版权所有 © 华为技术有限公司

在快照业务代码后调用LOS_MakeImage将系统此时的状态快照到指定介质上，并用#ifndef MAKE_WOW_IMAGE、#endif将该函数后的非快照业务包围起来，用以编译快照镜像和全部镜像时作区分，示例代码如下。

```
void wakeup_callback(void)
{
hal_interrupt_unmask(83);
if(!nand_init())
{
PRINT_ERR("nand init failed\n");
}
}

#define NAND_ERASE_ALIGN_SIZE(128 * 1024)
#define NAND_READ_ALIGN_SIZE(2 * 1024)
#define NAND_WRITE_ALIGN_SIZE(2 * 1024)

int flash_read(void *memaddr, unsigned long start, unsigned long size)
{
extern int hinand_read(void *memaddr, unsigned long start, unsigned long size);
return hinand_read(memaddr, start, size);
}

int flash_write(void *memaddr, unsigned long start, unsigned long size)
{
extern int hinand_erase(unsigned long start, unsigned long size);
extern int hinand_write(void *memaddr, unsigned long start, unsigned long size);
(void)hinand_erase(start, size);
return hinand_write(memaddr, start, size);
}
void app_init(void)
{
RUNSTOP_PARAM_S stRunstopParam;

proc_fs_init();

if (hi_uartdev_init() != 0)
{
PRINT_ERR("hi_uartdev_init failed");
}
if (system_console_init(TTY_DEVICE) != 0)
{
PRINT_ERR("system_console_init failed\n");
}

if(nand_init() != 0)
{
PRINT_ERR("nand_init falied\n");
}

memset(&stRunstopParam, 0, sizeof(RUNSTOP_PARAM_S));

/* 参数配置 */
stRunstopParam.pfFlashReadFunc= flash_read;
stRunstopParam.pfFlashWriteFunc= flash_write;
stRunstopParam.pfImageDoneCallback= NULL;
stRunstopParam.pfWakeupCallback= wakeup_callback;
stRunstopParam.uwFlashEraseAlignSize= NAND_ERASE_ALIGN_SIZE;
stRunstopParam.uwFlashWriteAlignSize= NAND_WRITE_ALIGN_SIZE;
stRunstopParam.uwFlashReadAlignSize= NAND_READ_ALIGN_SIZE;
stRunstopParam.uwImageFlashAddr= 0x100000;
stRunstopParam.uwWowFlashAddr= 0x3000000;

LOS_MakeImage(&stRunstopParam);

#ifndef MAKE_WOW_IMAGE
extern UINT32 g_uwWowImgSize;
PRINTK("Image length 0x%x\n", g_uwWowImgSize);
```


版权所有 © 华为技术有限公司

```
extern unsigned int osShellInit(const char *);
if (osShellInit(TTY_DEVICE) != 0)
{
PRINT_ERR("osShellInit\n");
}

rdk_fs_init();
SDK_init();

vs_server(3, apszArgv);
#endif /* MAKE_WOW_IMAGE */
}
```

步骤2 配置WOW_SRC变量

在根目录下Makefile中配置WOW_SRC变量，将变量定义为调用休眠唤醒函数的业务源文件路径，如下所示，其中LITEOSTOPDIR指代Huawei_LiteOS代码根目录。

```
WOW_SRC := $(LITEOSTOPDIR)/platform/bsp/$(LITEOS_PLATFORM)/os_adapt.c
```

步骤3 执行make wow，编译快照部分镜像

在根目录下执行如下命令，则不会编译#ifdef MAKE_WOW_IMAGE以下的业务代码。编译系统将自动调用工具链抽取快照镜像的符号表并根据该符号表提取快照镜像的.a库列表。

```
Huawei_LiteOS$ make wow
```

步骤4 执行make，编译全部镜像

1. 在根目录下执行如下命令，则编译全部业务代码。
   ```
   Huawei_LiteOS$ make
   ```
2. 编译完成后，可以通过查看镜像段排布验证快照镜像是否生成成功。进入系统镜像生成目录（hi3516a平台的镜像生成目录为out/hi3516a，其他类推），可以看到生成的系统镜像vs_server文件，执行命令**readelf －S vs_server**打开该文件，结果如下图所示。

图 4-6 镜像文件 vs_server.bin

```
[12] .wow_rodata       PROGBITS        80008020 0000d8 00b550 00   A  0   0  8
[13] .wow_text         PROGBITS        80013570 00b628 04bb44 00  AX  0   0  8
[14] .wow_data         PROGBITS        8005f0b4 05716c 001c3c 00  WA  0   0  8
[15] .wow_bss          NOBITS          80060cf0 058da8 008e1c 00  WA  0   0  8
```

显示了与休眠唤醒有关的段信息（包括段的名称、起始地址及偏移大小）。其中.wow_rodata、.wow_constdata为休眠唤醒镜像的只读数据段，.wow_text为代码段，.wow_data为数据段，.wow_bss为bss段。

3. 查看休眠唤醒链接脚本.text段，新增了wow.O(*.text*)，如下图所示，实现了将休眠唤醒的快照部分代码相关符号归拢到同一个段中。

图 4-7 链接脚本.text 段

```
.wow_text ALIGN (0x4): {
wow.O(.text*);
}

.wow_data ALIGN (0x4): {
 . = ALIGN (4); KEEP(wow.O( SORT (.liteos.table.*.wow.*))) ; . = ALIGN (4); wow.O(.data*);
}

__wow_bss_start = ABSOLUTE (.);
```


版权所有 © 华为技术有限公司

说明

休眠唤醒链接脚本路径：Huawei_LiteOS/tools/scripts/ld/wow.ld

步骤5 执行如下命令，将全部镜像烧写到Flash

```
tftp 0x82000000 vs_server.bin;nand erase 0x100000 0x700000;nand write 0x82000000 0x100000 0x700000;
```

进入串口工具界面，输入如下命令，将全部镜像烧写到Flash的0x100000地址位。

```
tftp 0x82000000 vs_server.bin;nand erase 0x100000 0x700000;nand write 0x82000000 0x100000 0x700000;
```

其中，vs_server.bin为系统镜像文件名，先将其烧写到内存中一段高地址位0x82000000。然后烧写到Flash，起始地址为0x100000，烧写长度为0x700000，即烧写的镜像文件大小不能超过7M，跟据实际镜像大小调整数值。

步骤6 执行如下命令，加载全部镜像

```
nand read 0x80008000 0x100000 0x700000; go 0x80008000;
```

执行如下命令，启动系统

```
nand read 0x80008000 0x100000 0x700000; go 0x80008000;
```

步骤7 根据系统反馈的快照镜像大小，从快照处恢复运行

用户可以获取休眠唤醒快照镜像大小：

图 4-8 打印信息

```
TFTP from server 192.168.1.2; our IP address is 192.168.1.10
Download Filename 'vs_server.bin'.
Download to address: 0x80008000
Downloading: *#################################################
done
Bytes transferred = 825104 (c9710 hex)
## Starting application at 0x80008000 ...
********hello Huawei LiteOS Cortex-A7********

version : Huawei LiteOS V100R002C00B302
open-version : Huawei LiteOS 1.1.4T6
build data : Aug 11 2016 16:37:58

**********************************
osAppInit
Mount procfs finished.
Nand ID:0x01 0xDA 0x90 0x95 0x44 0x01 0xDA 0x90
Nand:"NAND 256MiB 3,3V 8-bit"
Size:256MB Block:128KB Page:2KB Oob:64B Ecc:8bit/1K
Image length 0xc0000
```

根据上图信息，本sample中快照大小为0xc0000，快照被写到介质的0x3000000处。

重启系统，在uboot中执行**nand read 0x80008000 0x3000000 0xc0000; go 0x80008000;**从快照启动系统，效果如下图：

图 4-9 uboot 执行打印信息

```
hisilicon # nand read 0x80008000 0x3000000 0xc0000; go 0x80008000;

NAND read: device 0 offset 0x3000000, size 0xc0000
 786432 bytes read: OK
## Starting application at 0x80008000 ...
Nand ID:0x01 0xDA 0x90 0x95 0x44 0x01 0xDA 0x90
Nand:"NAND 256MiB 3,3V 8-bit"
Size:256MB Block:128KB Page:2KB Oob:64B Ecc:8bit/1K
register nand err -17
nand_init(673): Error:nand node register fail!
Image length 0xc0000
```


版权所有 © 华为技术有限公司

系统成功从快照启动。

----结束

4.9.3 注意事项

- 用户在编写休眠唤醒业务时，需要保证#ifndef MAKE_WOW_IMAGE前涵盖了所有快照部分的代码和数据，否则从快照恢复后的系统中不会包含所有休眠唤醒的业务。

4.9.4 LOS_MakeImage 参数配置说明

RUNSTOP_PARAM_S stRunstopParam; // 定义一个LOS_MakeImage的参数变量

- 指定介质读函数：
  stRunstopParam.pfFlashReadFunc= flash_read;
- 指定介质写函数：
  stRunstopParam.pfFlashWriteFunc= flash_write;
- 指定快照镜像生成后的回调函数，此sample为NULL：
  stRunstopParam.pfImageDoneCallback= NULL;
- 指定从快照恢复后需要执行的回调函数：
  stRunstopParam.pfWakeupCallback= wakeup_callback;
- 介质的擦除对齐参数：
  stRunstopParam.uwFlashEraseAlignSize= NAND_ERASE_ALIGN_SIZE;
- 介质的写对齐参数：
  stRunstopParam.uwFlashWriteAlignSize= NAND_WRITE_ALIGN_SIZE;
- 介质的读对齐参数：
  stRunstopParam.uwFlashReadAlignSize= NAND_READ_ALIGN_SIZE;
- 系统全镜像在介质上的地址（此地址为系统全镜像在介质上被烧写的地址）：
  stRunstopParam.uwImageFlashAddr= 0x100000;
- 快照将要被保存到的介质上的起始地址：
  stRunstopParam.uwWowFlashAddr= 0x3000000;

版权所有 © 华为技术有限公司

5 文件系统

版权所有 © 华为技术有限公司

5.1 功能概述

Huawei LiteOS目前支持的文件系统有：VFS、NFS、JFFS2、FAT、YAFFS2、RAMFS、PROC。

各个文件系统功能概述

表 5-1 文件系统功能概述

文件系统	功能特点概述
VFS	VFS是Virtual File System（虚拟文件系统）的缩写。它的作用就是采用标准的Unix系统调用读写位于不同物理介质上的不同文件系统,即为各类文件系统提供了一个统一的操作方式。
NFS	NFS是Network File System（网络文件系统）的缩写。它最大的功能是可以通过网络，让不同的机器、不同的操作系统彼此分享其他用户的文件
JFFS2	JFFS2是Journalling Flash File System Version 2(日志文件系统)的缩写。它的功能是管理在设备上实现的日志型文件系统。主要应用于对NOR_FLASH闪存的文件管理。Huawei LiteOS的JFFS2支持多分区
FAT	FAT文件系统是File Allocation Table（文件配置表）的简称，有FAT12、FAT16、FAT32这几种类型。在可移动存储介质(U盘、SD卡、移动硬盘等)上多使用FAT文件系统，使设备与Windows、Linux等桌面系统之间保持很好的兼容性
YAFFS2	YAFFS是Yet Another Flash File System的简称，是一种开源的、针对Nand flash的嵌入式文件系统。适用于大容量的存储设备，同时也使得Nand flash具有高效性和健壮性 YAFFS2为文件系统提供了损耗平衡和掉电保护，保证数据在系统对文件系统修改的过程中发生意外而不损坏。Huawei LiteOS的YAFFS2支持多分区
RAMFS	RAMFS是一种基于RAM的文件系统。RAMFS文件系统把所有的文件都放在 RAM 中，所以读/写 操作发生在RAM中，避免了对存储器的读写损耗，也提高了数据读写速度。RAMFS是基于RAM的动态文件系统的一种存储缓冲机制
PROC	PROC文件系统是一种伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

5.2 VFS

版权所有 © 华为技术有限公司

5.2.1 概述

基本概念

VFS是Virtual File System（虚拟文件系统）的缩写，是一个异构文件系统之上的软件粘合层，为用户提供统一的Unix文件操作接口。

由于不同类型的文件系统接口不统一，若系统中有多个文件系统类型，访问不同的文件系统就需要使用不同的非标准接口。而通过在系统中添加VFS层，提供统一的抽象接口，屏蔽了底层异构类型的文件系统的差异，使得访问文件系统的系统调用不用关心底层的存储介质和文件系统类型,提高开发效率。

Huawei LiteOS中，VFS框架是通过在内存中的树结构来实现的，树的每个结点都是一个inode结构体。设备注册和文件系统挂载后会根据路径在树中生成相应的结点。VFS最主要是两个功能：

1.查找节点。

2.统一调用（标准）

运作机制

通过VFS层，可以使用标准的Unix文件操作函数（如open、read、write等）来实现对不同介质上不同文件系统的访问。

VFS框架内存中的inode树结点有三种类型

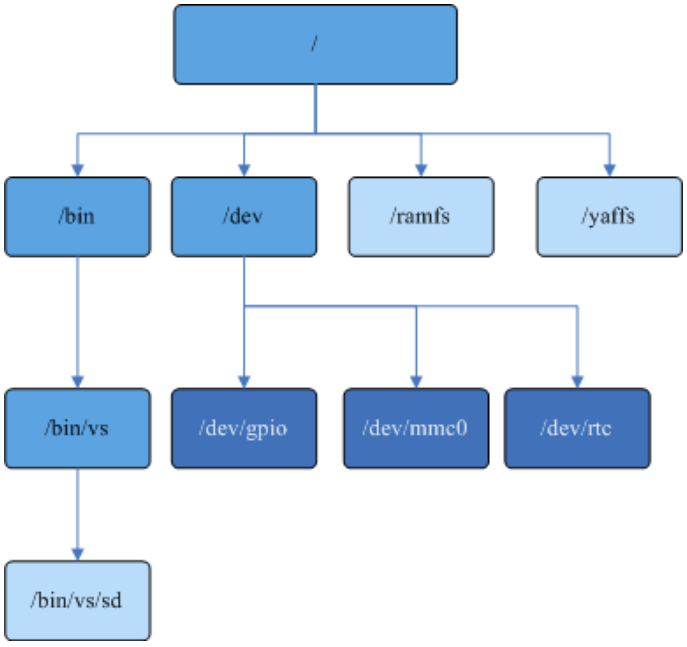
1、虚拟结点：作为vfs框架的虚拟文件，保持树的连续性，如/bin,/bin/vs

2、设备结点：/dev目录下，对应一个设备，如/dev/mmc0

3、挂载点：调用mount函数后生成，如/bin/vs/sd,/ramfs,/yaffs

inode的关键在于u和i_private字段，一个是函数方法结构体的指针，一个是数据指针

图 5-1 VFS 框架结构



版权所有 © 华为技术有限公司

5.2.2 开发指导

开发流程

推荐驱动开发人员使用VFS框架来注册/卸载设备，应用层使用open()、read()操作设备（字符设备）文件来调用驱动。

1. 系统调用了los_vfs_init()后，会将"/dev"作为root_inode。
2. 调用register_driver(),register_blockdriver()接口生成设备结点,调用mount()接口生成文件系统挂载结点
3. 生成结点的同时进行结构体信息的填充，然后根据结点名插入到树中合适的位置，保持有序。
4. 在调用时根据路径在树中进行查找，匹配到相应的设备或挂载点。
5. 通过查找到的结点指针可调用相应的函数。

文件描述符

本设计中采用全局数组的形式来对文件描述符进行管理。

文件描述符分两种：

- File描述符，普通文件描述符，其中0、1、2保留作为系统stdin标准输入、stdout标准输出和stderr标准错误。可以分配的文件描述符从3开始到CONFIG_NFILE_DESCRIPTORS - 1。
- Socket描述符，文件描述符的分配从CONFIG_NFILE_DESCRIPTORS开始。

两种文件描述符对应两个全局数组，内存上并不连续。

文件属性

使用chattr接口，改变文件系统文件属性，目前仅FAT文件系统支持。

int chattr(const char *path, mode_t mode)

path：待修改的文件；

mode：修改成的属性，目前支持4种（F_RDO：只读 F_HID：隐藏 F_SYS：系统文件 F_ARC：存档）；

说明

- 当前只是提供修改FAT文件系统文件属性的接口，各个系统对只读等属性有各自的处理方式。
- 在Huawei LiteOS中这4种属性并不冲突（可以任意修改）。
- 在Huawei LiteOS中只读属性文件/目录不允许被删除。
- 在Huawei LiteOS中只读属性文件/目录允许rename。
- 只读文件不允许以O_CREAT，O_TRUNC，以及有含有写的权限的方式打开。
- 在Huawei LiteOS中隐藏属性文件可见，在windows中不可见（不显示隐藏文件的属性情况下）。
- 在Huawei LiteOS中设置的系统文件加上隐藏属性，在windows中只能通过命令行找到（在显示，不显示隐藏文件的属性情况下都不能看到）。

版权所有 © 华为技术有限公司

5.2.3 注意事项

- VFS为文件系统的框架，为应用层开发者提供统一的文件系统调用。
- VFS下的所有文件系统，创建的目录名和文件名最多只可以有255个字节，名字长度超过255字节将会创建失败。
- VFS下的所有文件系统，默认未启用文件访问权限，此时文件权限默认创建为0666。
- inode_find()函数调用后会使查找到的inode节点连接数+1，调用完成后需要调用inode_release()使连接数-1，所以一般inode_find()要和inode_release()配套使用。
- 设备分为字符设备和块设备，为了块设备上的文件系统系统数据安全，VFS不提供对于块设备直接进行物理读写操作，需挂载相应文件系统后通过文件系统接口操作数据。
- los_vfs_init() 只能调用一次，多次调用将会造成文件系统异常。
- 关于挂载：挂载点必须为空目录，不能重复挂载至同一挂载点或挂载至其他挂载点下的目录或文件。
- 目前Huawei LiteOS所有的文件系统中的文件名和目录名中只可以出现- 与_两种特殊字符，使用其他特殊字符可能造成的后果不可预知，请谨慎为之。
- VFS可以直接递归创建多级目录。
- 挂载文件系统请严格按照手册进行，错误挂载可能损坏设备及系统。
- Shell端工作目录与系统工作目录是分开的，即通过Shell端cd pwd等命令是对Shell端工作目录进行操作，通过chdir getcwd等命令是对系统工作目录进行操作，两个工作目录相互之间没有联系。当文件系统操作命令入参是相对路径时要格外注意。
- 不要在休眠唤醒或者分散加载阶段挂载一个不存在的文件系统。
- vfs层能支持的全路径长度最长为259，超过这个路径长度的文件和目录无法创建。
- open打开一个文件时，参数O_RDWR、O_WRONLY、O_RDONLY互斥，只能出现一个，若出现2个或以上作为open的参数，会有不可预知的错误，不建议使用。

5.2.4 编程实例

无。

5.3 NFS

5.3.1 概述

基本概念

NFS是Network File System（网络文件系统）的缩写。它最大的功能是可以通过网络，让不同的机器、不同的操作系统彼此分享其他用户的文件。因此，用户可以简单地将它看做是一个文件系统服务，在一定程度上相当于Windows环境下的共享文件夹。

NFS客户端用户，能够将网络远程的NFS主机分享的目录挂载到本地端的机器或设备中，运行程序和共享文件，但不占用当前的系统资源，所以，在本地端的机器看起来，远程主机的目录就好像是自己的一个磁盘一样。

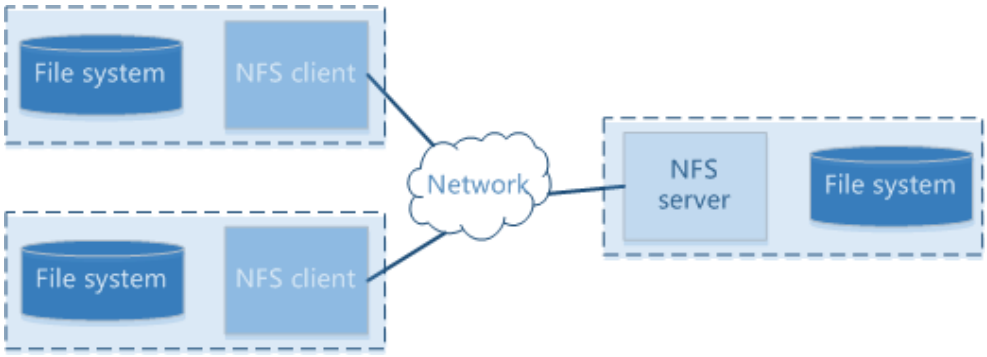
版权所有 © 华为技术有限公司

这样能够使本地工作站使用更少的磁盘空间，如软驱，CDROM，和 Zip（一种高储存密度的磁盘驱动器与磁盘）之类的存储设备可以在网络上被别的机器使用。这可以减少整个网络上的可移动介质设备的数量。

运作机制

一个典型的NFS组网图由图1所示。

图 5-2 NFS 组网图



Huawei LiteOS的NFS支持将主机端的NFS目录(/home/Huawei_LiteOS/nfs)，映射到客户端(/nfs)，两者内容相同并保持同步。

5.3.2 开发指导

开发流程

使用NFS功能，需要进行以下步骤（各步骤详细操作见下述分解）：

1. **搭建NFS服务器**
2. **设置单板为NFS客户端**
3. **利用NFS共享文件**

搭建 NFS 服务器

说明

这里以ubuntu操作系统为例，说明服务器端设置步骤。

步骤1 安装NFS服务器软件。

设置好Ubuntu系统的下载源，保证网络连接好的情况下执行：

```
sudo apt-get install nfs-kernel-server
```

步骤2 设置和启动NFS server。

修改NFS配置文件/etc/exports，添加如下一行：

```
/home/yourusername    *(rw,no_root_squash,async)。
```

其中/home/yourusername是NFS共享的根目录。

执行以下命令启动NFS server：

```
sudo /etc/init.d/nfs-kernel-server start
```


版权所有 © 华为技术有限公司

执行以下命令重启NFS server

```
sudo /etc/init.d/nfs-kernel-server restart
```

----结束

设置单板为 NFS 客户端

本指导中的NFS客户端指运行Huawei LiteOS的设备。

步骤1 硬件连接设置

Huawei LiteOS设备连接到和NFS服务器的网络。设置两者IP，使其处于同一网段。比如，设置NFS服务器的IP为10.67.212.178，设Huawei LiteOS设备（IPC板）IP为10.67.212.3。（IP的前三段设置成相同，最后一段不同，则这2个IP处于同一网段）。

Huawei LiteOS设备（IPC板）上的IP信息可通过ifconfig命令查看。

步骤2 启动网络，确保单板到NFS服务器之间的网络

启动以太网或者其他类型网络，使用ping命令检查到服务器的网络是否通畅。

```
Huawei LiteOS # ping 10.67.212.178
ping 4 packets start.
[0]Reply from 10.67.212.178: time=1ms TTL=63
[1]Reply from 10.67.212.178: time=0ms TTL=63
[2]Reply from 10.67.212.178: time=1ms TTL=63
[3]Reply from 10.67.212.178: time=1ms TTL=63
```

步骤3 客户端NFS初始化

运行命令：

```
Huawei LiteOS# mount 10.67.212.178:/home/sqbin/nfs /nfs nfs 1011 1000
```

将从串口得到如下回应信息，表明初始化NFS客户端成功。

```
Huawei LiteOS# mount 10.67.212.178:/home/sqbin/nfs /nfs nfs 1011 1000
Mount nfs on 10.67.212.178:/home/sqbin/nfs
Mount nfs finished.
```

该命令将服务器10.67.212.178上的/home/sqbin/nfs目录mount在Huawei LiteOS设备上的/nfs上。

mount命令的格式为

```
mount [SERVER_IP:SERVER_PATH] [CLIENT_PATH] nfs <uid gid>
```

其中SERVER_PATH表示服务器端NFS共享目录路径，注意该路径盘符小写，且不带冒号；SERVER_IP表示服务器的IP地址；CLIENT_PATH表示设备上的NFS路径。

uid是指Linux的用户ID，gid是指Linux的组用户ID。用于获取NFS服务器目录的权限。uid和gid可以在Linux的命令行使用ID命令查询。如果不想有NFS访问权限限制，请在Linux命令行将NFS根目录权限设置成777：

```
chmod -R 777 /home/sqbin/nfs
```

至此，NFS客户端设置完毕。NFS文件系统已成功挂载

----结束

利用 NFS 共享文件

在NFS服务器下新建目录dir，并保存。在Huawei LiteOS下运行ls命令

版权所有 © 华为技术有限公司

```
Huawei LiteOS# ls /nfs
```

则可从串口得到如下回应。

```
Huawei LiteOS#
Huawei LiteOS# ls /nfs
Directory /nfs:
.                 <DIR>
dir               <DIR>
```

可见，刚刚在NFS服务器上新建的dir目录已同步到客户端(Huawei LiteOS系统)的/nfs目录，两者保持同步。

同样地，在客户端(Huawei LiteOS系统)上创建文件和目录，在NFS服务器上也可以访问，读者可自行体验。

平台差异性

目前，NFS客户端仅支持NFS v3部分规范要求，因此对于规范支持不全的服务器，无法完全兼容。在开发测试过程中，建议使用Linux的NFS server，因为其对NFS支持很完善。

5.3.3 注意事项

- 当前NFS文件不支持权限控制，请在创建NFS目录和文件时使用777权限。
- 当前NFS文件不支读阻塞和写阻塞。
- 当前NFS文件不支持信号功能。
- 当前NFS文件不支持fcntl，ioctl，utime，chattr操作。
- 当前NFS不支持使用TCP做通信socket。
- open打开一个文件，参数有O_TRUNC时，必须同时拥有写的权限时，才会将文件中的内容清空。
- NFS属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

5.3.4 编程实例

无。

5.4 JFFS2

5.4.1 概述

基本概念

JFFS2是Journalling Flash File System Version 2(日志文件系统)的缩写。它的功能是管理在设备上实现的日志型文件系统。JFFS2主要应用于NOR FLASH，其特点是：可读写、支持数据压缩、提供了崩溃/掉电安全保护、提供“写平衡”支持等。

闪存与磁盘介质有许多差异，因此直接将磁盘文件系统运行在闪存上存在性能和安全性上的不足。为解决这一问题，需要实现一个特别针对闪存的文件系统，JFFS2就是这样一种文件系统。

版权所有 © 华为技术有限公司

Huawei LiteOS的JFFS2主要应用于对NOR Flash闪存的文件管理，并且支持多分区。

5.4.2 开发指导

开发流程

使用JFFS2功能，涉及以下几步骤（各步骤详细操作见下述分解）：

1. **添加JFFS2分区**
2. **挂载JFFS2**
3. **卸载JFFS2**
4. **删除JFFS2分区**

添加 JFFS2 分区

调用add_mtd_partition函数添加JFFS2分区，该函数会自动为设备节点命名，对于JFFS2，其命名规则是“/dev/spinorblk”加上分区号。

add_mtd_partition函数有四个参数，第一个参数表示介质，有“nand”和“spinor”两种，JFFS2分区在“spinor”上使用，而“nand”是提供给YAFFS2使用的。

第二个参数表示起始地址，第三个参数表示分区大小，这两个参数都以16进制的形式传入。

最后一个参数表示分区号，有效值为0~19。

成功后，在Shell中可以使用partition jffs命令查看jffs分区信息。

```
if(uwRet = add_mtd_partition("spinor", 0x100000, 0x800000, 0) != 0)
        dprintf("add jffs partition failed, return %d\n", uwRet);
else
{
    dprintf("Mount jffs2 on nand.\n");
    uwRet = mount("/dev/spinorblk0", "/jffs0", "jffs", 0, NULL);
    if(uwRet)
        dprintf("mount jffs err %d\n",uwRet);
        dprintf("Mount jffs2 on nor finished.\n");
}
if(uwRet = add_mtd_partition("spinor", 0x900000, 0x200000, 1) != 0)
        dprintf("add jffs partition failed, return %d\n", uwRet);
Huawei LiteOS# partition jffs
jffs partition num:0, dev name:/dev/spinorblk0, mountpt:/jffs0, startaddr:0x0100000,length:
0x0800000
jffs partition num:1, dev name:/dev/spinorblk1, mountpt:(null), startaddr:0x0900000,length:
0x0200000
```

挂载 JFFS2

调用mount()函数实现设备节点和挂载点的挂载。

该函数有四个参数，第一个参数表示设备节点，第二个参数表示挂载点，这两个参数需要和add_mtd_partition()函数对应起来。

第三个参数表示文件类型。

最后两个参数表示挂载标志和数据，默认为0和NULL；这一操作也可以在Shell中使用mount命令实现，最后两个参数不需要用户给出。

运行命令

版权所有 © 华为技术有限公司

```
Huawei LiteOS# mount /dev/spinorblk1 /jffs1 jffs
```

将从串口得到如下回应信息，表明挂载成功。

```
Huawei LiteOS# ls
Directory /:
bin                  <DIR>
dev                  <DIR>
jffs0                <DIR>
ramfs                <DIR>
yaffs0               <DIR>

Huawei LiteOS# mount /dev/spinorblk1 /jffs1 jffs

Huawei LiteOS# ls
Directory /:
bin                  <DIR>
dev                  <DIR>
jffs0                <DIR>
jffs1                <DIR>
ramfs                <DIR>
yaffs0               <DIR>
```

挂载成功后，用户就能对norflash进行读写操作。

卸载 JFFS2

调用umount()函数卸载分区，只需要正确给出挂载点即可。

运行命令

```
Huawei LiteOS# umount /jffs1
```

将从串口得到如下回应信息，表明卸载成功。

```
Huawei LiteOS# ls
Directory /:
bin                  <DIR>
dev                  <DIR>
jffs0                <DIR>
jffs1                <DIR>
ramfs                <DIR>
yaffs0               <DIR>

Huawei LiteOS# umount /jffs1
umount ok
```

删除 JFFS2 分区

调用delete_mtd_partition删除已经卸载的分区。

该函数有两个参数，第一个参数是分区号，第二个参数为介质类型，该函数与add_mtd_partition()函数对应。

```
uwRet =delete_mtd_partition(1,"spinor");
if(uwRet != 0)
    printf("delte jffs error\n");
else
    printf("delete jffs ok\n");
Huawei LiteOS# partition jffs
jffs partition num:0, dev name:/dev/spinorblk0, mountpt:/jffs0, startaddr:0x0100000,length:
0x0800000
```

制作 JFFS2 文件系统镜像

使用mkfs.jffs2工具，现在制作镜像默认命令为

版权所有 © 华为技术有限公司

./mkfs.jffs2 -s 0x1000 -e 0x10000 -p 0x100000 -d rootfs/ -o rootfs_64k.jffs2

（已制成脚本，自动执行生成）

其中：

表 5-2 指令含义表

指令	含义
-s	页大小
-e	eraseblock大小
-p	镜像大小
-d	要制作成文件系统镜像的源目录
-o	要制成的镜像名称

若实际中与上面参数不同时，修改相应参数。

5.4.3 注意事项

- 目前JFFS2文件系统用于NOR Flash，最终调用NOR Flash驱动接口，因此使用JFFS2文件系统之前要确保硬件上有NOR Flash，且驱动初始化成功(spinor_init()返回0)。
- 系统会自动对起始地址和分区大小根据block大小进行对齐操作。有效的分区号为0~19。
- 目前支持mkfs.jffs2工具，mkfs.jffs2命令在文件fsimage/MakeVersion.sh中，用户可根据自己实际情况修改命令中参数值。其它命令可通过查询mkfs.jffs2查看。
- open打开一个文件，参数有O_TRUNC时，会将文件中的内容清空。
- 目前JFFS2文件系统不支持ioctl，sync，dup，dup2，utime，chattr函数。

5.4.4 编程实例

无。

5.5 FAT

5.5.1 概述

基本概念

FAT文件系统是File Allocation Table（文件配置表）的简称，FAT文件系统有FAT12、FAT16、FAT32这几种类型。FAT文件系统将硬盘分为MBR区、DBR区、FAT区、DIR区、DATA区等5个区域。

FAT文件系统支持多种介质，特别在可移动存储介质(U盘、SD卡、移动硬盘等)上广泛使用。可以使嵌入式设备和Windows、Linux等桌面系统保持很好的兼容性，方便用户管理操作文件。

版权所有 © 华为技术有限公司

Huawei LiteOS的FAT文件系统具有代码量和资源占用小、可裁切、支持多种物理介质等特性，并且与Windows、Linux等系统保持兼容，支持多设备、多分区识别等功能。

5.5.2 开发指导

开发流程

使用FAT功能，涉及以下几步骤（各步骤详细操作见下述分解）：

1.设备识别

2.FAT文件系统的挂载

3.FAT文件系统的卸载

设备识别

```
Huawei LiteOS # ls
Directory /dev:
acodec                  0
adec                    0
aenc                    0
ai                      0
ao                      0
console                 0
fb0                     0
hi_gpio                 0
hi_mipi                 0
hi_rtc                  0
hi_tde                  0
i2c-0                   0
i2c-1                   0
i2c-2                   0
isp_dev                 0
lcd                     0
logmpp                  0
mem                     0
mmcblk0                 0
mmcblk0p0               0
mmcblk0p1               0
mmcblk1                 0
mmcblk1p0               0
```

- 在ffconf.h文件中配置_MULTI_PARTITION为1，可使用多分区功能。
- 在ffconf.h文件中配置_VOLUMES大于2时，可使用多设备功能

多设备，多分区功能开启后，系统对于插上的sd卡自动识别，自动注册设备节点如上图所示。mmcblk0和mmcblk1为卡0和 卡1，是独立的主设备，mmcblk0p0,mmcblk0p1为卡0的两个分区，可作为分区设备使用。在有分区设备存在的情况下，使用主设备将会自动调用第一个分区设备。

FAT 文件系统的挂载

运行命令

```
Huawei LiteOS# mount /dev/mmcblk0 /bin/vs/sd vfat
```

将从串口得到如下回应信息，表明挂载成功。

```
Huawei LiteOS# mount /dev/mmcblk0 /bin/vs/sd vfat
mount ok

Huawei LiteOS# ls
```


版权所有 © 华为技术有限公司

```
Directory /:
bin                 <DIR>
dev                 <DIR>
ramfs               <DIR>
yaffs0              <DIR>
```

FAT 文件系统的卸载

运行命令

```
Huawei LiteOS# umount /bin/vs/sd
```

将从串口得到如下回应信息，表明卸载成功。

```
Huawei LiteOS# umount /bin/vs/sd
umount ok
```

5.5.3 注意事项

- 默认配置可直接使用，用户也可根据实际需求自行裁切配置。
- FAT文件系统的配置项在ffconf.h文件中。
- 当_FS_READONLY 为0时可以读写，为1时只读。
- 当_USE_MKFS为1并且_FS_READONLY为1表示打开格式化功能。
- 当_FS_NORTC为1时没有实时时钟，创建文件将显示固定时间。
- _FS_LOCK为最多支持同时打开的文件（文件夹）数。
- 以读写方式打开一个文件后，未close前再次以读写方式打开会失败，只能以只读方式打开。长时间打开一个文件，没有close时数据会丢失，必须close才能保存。
- 单个文件不能大于4G。
- 文件名和挂载点之后的路径长度的大小总和不能大于253个字节。
- 当有两个SD卡插槽时，卡0和卡1不固定，先插上的为卡0，后插上的为卡1。
- 多分区功能打开，存在多分区的情况下，卡0注册的设备节点/dev/mmcblk0主设备和/dev/mmcblk0p0次设备是同一个设备，禁止对主设备进行操作。
- 双卡多分区功能打开，没有多分区的情况下，mmcblk0和mmcblk0p0相当于同一个设备，两者只能mount其一。
- fat文件系统暂不支持使用open() + O_DIRECTORY去打开一个目录，打开目录请使用opendir()。
- fat文件系统的读写指针没有分离，所以以O_APPEND（追加写）方式打开文件后，读指针也在文件尾，读文件前需要用户手动置位。
- fat文件系统的stat及lstat函数获取出来的文件时间只是文件的修改时间。暂不支持创建时间和最后访问时间。微软FAT协议不支持1980年以前的时间。
- open打开一个文件，参数有O_TRUNC时，会将文件中的内容清空。
- fat文件系统不支持ioctl，dup，dup2函数。

5.5.4 编程实例

无。

5.6 YAFFS2

版权所有 © 华为技术有限公司

5.6.1 概述

基本概念

YAFFS是Yet Another Flash File System的简称，是一种开源的、针对Nand flash的嵌入式文件系统。在YAFFS中，最小的存储单位为page。

目前YAFFS文件系统有YAFFS和YAFFS2两个版本，主要区别在于YAFFS2能够更好地支持大容量的Nand flash芯片，它支持2K大小的page，远高于YAFFS的512 bytes。此外YAFFS2还有64bytes的SPARE区域，用于存储坏块信息、ECC校验等。

- YAFFS2专为Nand flash而设计，适用于大容量的存储设备，同时也使得Nand flash具有高效性和健壮性。
- YAFFS2实现对2K page的支持，同时在内存空间占用、垃圾回收速度、读写速度等上面均有大幅提升。
- YAFFS2为文件系统提供了损耗平衡和掉电保护，保证数据在系统对文件系统修改的过程中发生意外而不损坏。

Huawei LiteOS的YAFFS2文件系统支持多分区。

5.6.2 开发指导

开发流程

Huawei LiteOS的YAFFS2多分区功能采用双向链表结构实现。使用YAFFS2功能，涉及到以下几个步骤（各步骤详细操作见下述分解）：

1. **调用add_mtd_partition创建分区**
2. **调用mount函数挂载分区**
3. **调用umount函数卸载分区**
4. **调用delete_mtd_partition函数删除分区**

具体实例可以在编程实例中查看。

调用 add_mtd_partition 创建分区

该函数会自动为设备节点命名，对于YAFFS2，其命名规则是“/dev/nandblk”加上分区号。

add_mtd_partition()函数有四个参数，第一个参数表示介质，有“nand”和“spinor”两种，YAFFS2分区在“nand”上使用，而“spinor”是提供给JFFS2使用的。

第二个参数表示起始地址，第三个参数表示分区大小，这两个参数都以16进制的形式传入。

最后一个参数表示分区号，有效值为0~19。

成功后，在Shell中可以使用partition yaffs命令查看yaffs分区信息。

创建分区。

```
if(uwRet = add_mtd_partition("nand", 0x900000, 0x200000, 0) < 0)
        dprintf("add yaffs partition failed, return %d\n", uwRet);

if(uwRet = add_mtd_partition("nand", 0xb00000, 0x200000, 1) < 0)
        dprintf("add yaffs partition failed, return %d\n", uwRet);
```


版权所有 © 华为技术有限公司

调用 mount 函数挂载分区

调用mount()函数实现设备节点和挂载点的挂载。

该函数有四个参数，第一个参数表示设备节点，第二个参数表示挂载点，这两个参数需要和add_mtd_partition()函数对应起来。

第三个参数表示文件类型。

最后两个参数表示挂载标志和私有数据，默认为0和NULL。

```
uwRet = mount("/dev/nandblk0", "/yaffs0", "yaffs", 0, NULL);
if(uwRet)
    dprintf("mount yaffs err %d\n",uwRet);
```

这一操作也可以在Shell中使用mount命令实现，最后两个参数不需要用户给出。

运行命令

```
Huawei LiteOS# mount /dev/nandblk1 /yaffs1 yaffs
```

将从串口得到如下回应信息，表明挂载成功。

```
start-blk:24, end-blk:39

Huawei LiteOS# partition yaffs
yaffs partition num:0, dev name:/dev/nandblk0, mountpt:/yaffs0, startaddr:0x0900000,length:
0x0200000
yaffs partition num:1, dev name:/dev/nandblk1, mountpt:/yaffs1, startaddr:0x0b00000,length:
0x0200000
```

调用 umount 函数卸载分区

调用umount()函数卸载分区，只需要正确给出挂载点即可。这一操作也可以在Shell中使用umount命令实现。

运行命令

```
Huawei LiteOS# umount /yaffs1
```

将从串口得到如下回应信息，表明卸载成功。

```
umount ok

Huawei LiteOS# umount /yaffs0
umount ok

Huawei LiteOS# partition yaffs
yaffs partition num:0, dev name:/dev/nandblk0, mountpt:(null), startaddr:0x0900000,length:0x0200000
yaffs partition num:1, dev name:/dev/nandblk1, mountpt:(null), startaddr:0x0b00000,length:0x0200000
```

调用 delete_mtd_partition 函数删除分区

删除分区前要卸载分区。

该函数有两个参数，第一个参数是分区号，第二个参数为介质类型，该函数与add_mtd_partition()函数对应。

```
uwRet =delete_mtd_partition(1, "nand");
if(uwRet != 0)
    printf("delte yaffs error\n");
else
    printf("delete yaffs ok\n");
```

卸载并删除分区后结果：

版权所有 © 华为技术有限公司

```
delete yaffs ok
```

5.6.3 注意事项

- Huawei LiteOS的YAFFS2文件系统支持多分区。分区起始地址可以灵活配置，但分区间地址不能重叠，用户需要根据自身flash使用情况，合理地配置空间，防止与其它文件系统或其它不适合挂载YFFAS2文件系统的空间发生冲突。
- 最小可添加的分区为一个block大小，但是最小可挂载的分区为9个block大小（由YAFFS2特性决定）。要注意区分这两个概念。
- Huawei LiteOS的YAFFS2文件系统，会自动对地址和分区大小根据block大小进行对齐操作。有效的分区号为0~19。
- Huawei LiteOS的YAFFS2文件系统,只允许连续打开20个目录。
- open打开一个文件，参数有O_TRUNC时，必须同时拥有写的权限时，才会将文件中的内容清空。
- Huawei LiteOS的YAFFS2文件系统不支持ioctl，utime，chattr函数。

5.6.4 编程实例

无。

5.7 RAMFS

5.7.1 概述

基本概念

RAMFS是一个可动态调整大小的基于RAM的文件系统。RAMFS没有后备存储源。向RAMFS中进行的文件写操作也会分配目录项和页缓存，但是数据并不写回到任何其他存储介质上，掉电后数据丢失。

RAMFS文件系统把所有的文件都放在 RAM 中，所以读/写 操作发生在RAM中，可以用RAMFS来存储一些临时性或经常要修改的数据，例如 /tmp 和/var 目录，这样既避免了对存储器的读写损耗，也提高了数据读写速度。

Huawei LiteOS的RAMFS是一个简单的文件系统，它是基于RAM的动态文件系统的一种缓冲机制。

Huawei LiteOS的RAMFS基于虚拟文件系统层（VFS），不能格式化。

5.7.2 开发指导

开发流程

RAMFS文件系统的使用，涉及以下三个步骤（各步骤具体操作见下述分解）：

1.**RAMFS文件系统的初始化**

2.**RAMFS文件系统的挂载**

3.**RAMFS文件系统的卸载**

版权所有 © 华为技术有限公司

RAMFS 文件系统的初始化

```
void ram_fs_init(void)
{
    int swRet=0;
    swRet = mount(NULL, RAMFS_DIR, "ramfs", 0, NULL);
    if (swRet) {
        dprintf("mount ramfs err %d\n", swRet);
        return;
    }
    dprintf("Mount ramfs finished.\n");
}
```

调用初始化函数，随后在Huawei LiteOS系统启动时可以看到如下显示，表示RAMFS文件系统已初始化成功：

```
Mount ramfs finished
```

RAMFS 文件系统的挂载

运行命令

```
Huawei LiteOS# mount 0 /ramfs ramfs
```

将从串口得到如下回应信息，表明挂载成功。

```
Huawei LiteOS# mount 0 /ramfs ramfs
mount ok
```

RAMFS 文件系统的卸载

运行命令

```
Huawei LiteOS# umount /ramfs
```

将从串口得到如下回应信息，表明卸载成功。

```
Huawei LiteOS# umount /ramfs
umount ok
```

5.7.3 注意事项

- RAMFS文件系统的读写指针没有分离，所以以O_APPEND（追加写）方式打开文件后，读指针也在文件尾，读文件前需要用户手动置位。
- 由于RAMFS所使用的内存空间是固定的，所以为了避免踩内存，RAMFS只能挂载一次，一次挂载成功后，后面不能继续挂载到其他目录。
- RAMFS文件系统的文件名和目录名长度不能超过20。
- open打开一个文件，参数有O_TRUNC时，会将文件中的内容清空。
- RAMFS文件系统不支持ioctl，sync，utime，dup，dup2，chattr函数。
- RAMFS属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

5.7.4 编程实例

无。

5.8 PROC

版权所有 © 华为技术有限公司

5.8.1 概述

基本概念

PROC文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

Huawei LiteOS的PROC是一个虚拟文件系统，不支持多线程操作。

5.8.2 开发指导

开发流程

PROC文件系统的使用涉及以下两个步骤（各步骤详细操作见下述分解）：

1.**PROC文件系统的初始化**

2.**PROC节点的创建**

3.**修改和查看PROC节点信息**

PROC 文件系统的初始化

```
void proc_fs_init(void)
{
    int swRet=0;
    swRet = mount(NULL, PROCFS_DIR, "procfs", 0, NULL);
    if (swRet) {
        dprintf("mount procfs err %d\n", swRet);
        return;
    }
    dprintf("Mount procfs finished.\n");
}
```

调用初始化接口，在Huawei LiteOS系统启动时可以看到如下显示，表示PROC文件系统已初始化成功：

```
Mount procfs finished
```

PROC 节点的创建

调用create_proc_entry函数创建文件节点，第一个参数为要创建的节点的名字，第二个参数为文件的模式（文件的类型和文件的权限），第三个参数表示要创建的节点的父节点，传NULL默认为父节点是“/proc”，函数的返回值就是创建出来的proc文件节点。

```
struct proc_dir_entry *pHandle;

pHandle = create_proc_entry("mounts", 0, NULL);
if (pHandle == NULL) {
    dprintf("creat mounts error!\n");
    return;
}
```

在节点创建成功后，可以给它赋予相应的操作函数：

```
pHandle->proc_fops = &mounts_proc_fops;
```

若没有则使用默认的操作函数proc_file_default_operations。

版权所有 © 华为技术有限公司

调用proc_mkdir函数创建目录节点，第一个参数为要创建的节点的名字，第二个参数表示要创建的节点的父节点，传NULL默认为父节点是“/proc”。函数的返回值就是创建出来的proc目录节点。

```
struct proc_dir_entry *pHandle;

pHandle = proc_mkdir("test", NULL);
if (pHandle == NULL) {
    dprintf("creat test error!\n");
    return;
}
```

修改和查看 PROC 节点信息

运行cat命令查看节点信息，如下举一个例子：

```
Huawei LiteOS # cat /proc/umap/logmpp
```

将从串口得到如下回应信息，表明cat成功。

```
Huawei LiteOS # cat /proc/umap/logmpp
Huawei LiteOS # -----LOG BUFFER STATE---------------------------------
MaxLen ReadPos WritePos ButtPos
 64(KB)      0      113   65536

-----CURRENT LOG LEVEL-----------------------------------------------
vb : 3
sys : 3
region : 3
chnl : 3
vpss : 3
venc : 3
vda : 3
h264e : 3
jpege : 3
vou : 3
viu : 3
rc : 3
aio : 3
ai : 3
ao : 3
aenc : 3
adec : 3
isp : 3
ive : 3
tde : 3
vgs : 3
h265e : 3
```

运行writeproc命令修改节点信息，如下举一个例子：

```
Huawei LiteOS # writeproc 'sys=2' >> /proc/umap/logmpp
```

将从串口得到如下回应信息：

```
sys=2 >> /proc/umap/logmpp
```

再次cat打印节点信息，会发现sys等级已修改。

```
Huawei LiteOS # cat /proc/umap/logmpp
Huawei LiteOS # -----LOG BUFFER STATE---------------------------------
MaxLen ReadPos WritePos ButtPos
 64(KB)      0        0   65536

-----CURRENT LOG LEVEL-----------------------------------------------
vb : 3
sys : 2
region : 3
```


版权所有 © 华为技术有限公司

```
chnl : 3
vpss : 3
venc : 3
vda : 3
h264e : 3
jpege : 3
vou : 3
viu : 3
rc : 3
aio : 3
ai : 3
ao : 3
aenc : 3
adec : 3
isp : 3
ive : 3
tde : 3
vgs : 3
h265e : 3
```

5.8.3 注意事项

- Huawei LiteOS的PROC文件系统在使用open打开文件时，仅支持读写属性。
- Huawei LiteOS的PROC文件系统在使用fseek函数时，不支持SEEK_END，即不支持从文件尾开始偏移。
- Huawei LiteOS的PROC文件系统初始化后，ls查看PROC下的文件，大小全是0，只有在去cat时会获取系统内核的实时信息，并将之显示出来。
- Huawei LiteOS的PROC文件系统不支持卸载，不支持创建或者删除文件或目录。
- Huawei LiteOS的PROC文件系统的文件和目录名长度最大为32。
- Huawei LiteOS的PROC文件系统不支持ioctl，sync，dup，dup2，utime，chattr，rename函数。
- PROC文件系统属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

5.8.4 编程实例

无。

版权所有 © 华为技术有限公司

6 驱动开发指导

版权所有 © 华为技术有限公司

6.1 概述

基本概念

驱动开发就是把硬件的功能按操作系统的规格实现并抽象出来，提供给应用程序开发人员调用。

当在新的芯片上移植系统时，需要根据该芯片特性对支持的外围设备进行驱动开发。

Huawei LiteOS的驱动初始化函数主要初始化设备对应的驱动结构，生成给上层用户用来注册设备驱动的控制节点。

6.2 开发指导

开发流程

驱动的开发涉及下面两个步骤：

1. 驱动初始化
2. 驱动节点与使用

驱动初始化

在Huawei LiteOS上进行驱动开发的第一个步骤，是编写驱动初始化函数。Huawei LiteOS的驱动初始化函数用于初始化设备所需的驱动结构，以及生成驱动设备的控制节点。

用户完成驱动初始化函数的编写后，需要在适当的地方引导设备初始化函数执行，简单的引导可以参考开发包中sample目录下sample_hi3516a.c中的代码。用户可以在app_init函数里调用已编写好的设备初始化函数引导设备初始化。

驱动初始化函数中需要调用用于注册并生成节点的设备驱动注册函数：

```
register_driver(FAR const char *path, FAR const struct file_operations_vfs *fops,mode_t mode, FAR void *priv)
register_blockdriver(FAR const char *path,FAR const struct block_operations *bops,mode_t mode, FAR void *priv)
```

参数说明如表1所示：

表 6-1 参数说明

参数	说明
path	驱动节点路径，应用程序可通过该路径访问到驱动节点，进而访问设备驱动提供的操作接口。
fops/bops	驱动操作结构体，为应用程序提供操作函数集。fops和bops分别表示字符设备和块设备
mode	应用读写该驱动节点的权限，暂时无效，后续或提供支持。
priv	驱动节点注册过程需要传入的参数。

版权所有 © 华为技术有限公司

用户编写好驱动程序，设备驱动初始化函数应该在系统初始化时被调用生成驱动节点以供应用访问。

驱动初始化工作完成后，指定路径会生成设备驱动节点，应用程序可通过该节点使用驱动程序提供的操作接口。

驱动节点与使用

驱动初始化后，生成的设备驱动节点为应用提供操作设备的接口，下面以i2c设备驱动程序为例，说明用户程序与驱动操作函数的调用关系。

1. **操作函数集**

   驱动操作函数集对于用户程序与驱动操作函数的调用关系非常重要。在编写驱动程序时，操作函数集需要完成硬件设备各项机制的实现，并在设备注册时传入并注册。操作函数集会成为应用函数需求的最终实现。表2为i2c设备驱动提供的函数操作集。

   表 6-2 i2c 设备函数操作集

函数操作集	对应的应用层接口
i2cdev_open	open
i2cdev_release	close
i2cdev_read	read
i2c_write	write
i2c_ioctl	ioctl

2. **open操作**

   应用程序打开节点文件时，系统最终会调用该驱动节点注册过程中的驱动操作函数集中的open函数。

   open函数完成对应驱动设备函数结构体实例化，并对驱动结构体进行必要的初始化。

3. **read/write操作**

   应用程序成功打开节点文件后，获取到对应驱动节点的文件描述符。程序可通过该文件描述符对驱动程序进行访问。

   read/write操作是常用的应用对驱动访问的接口。不同类型的设备与驱动提供的read/write操作的功能互有差异。对于i2c设备，用户程序调用read/write接口可以实现对i2c外围设备进行读写。

```
i2cdev_read(struct file * filep, char __user *buf, size_t count)
i2cdev_write(struct file * filep , const char __user *buf, size_t count)
```

   参数说明如表3所示：

   表 6-3 参数说明

参数	说明
filep	文件描述结构体指针。

版权所有 © 华为技术有限公司

参数	说明
buf	读出、写入数据缓冲区。
count	读出、写入数据长度。

4. **ioctl操作**

   ioctl操作提供对驱动设备函数的配置管理。通过执行相应的命令，完成对设备属性的配置或访问。

   i2c设备中，使用命令I2C_16BIT_REG、I2C_16BIT_DATA与I2C_TIMEOUT分别对应设置i2c设备驱动中传输寄存器位宽、传输数据位宽与超时时间。

   ```
   i2cdev_ioctl(struct file *filep, int cmd, unsigned long arg)
   ```

   参数说明如表4所示：

   表 6-4 参数说明

参数	说明
filep	文件描述结构体指针。
cmd	操作命令。
arg	附加参数。

5. **close操作**

   close操作对应着驱动操作函数集里的release函数。release函数提供对驱动程序的资源释放处理。

6.3 注意事项

无。

6.4 编程实例

无。

版权所有 © 华为技术有限公司

7 可维可测

版权所有 © 华为技术有限公司

7.1 Telnet

7.1.1 概述

基本概念

Telnet协议是TCP/IP协议族中的一员，是Internet远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。在终端使用者的电脑上使用Telnet程序，用它连接到服务器。终端使用者可以在Telnet程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。

Huawei LiteOS Telnet

Telnet是采用命令行进行系统调式的一种工具。用户通过配置电脑IP和网关，确保与板子在同一网段的局域网内后，通过windows下cmd.exe工具，执行Telnet IP（板子IP）后，进行网络连接，也可以说Telnet是串口的另一种调试方式

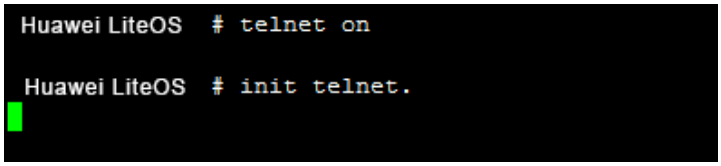
Huawei LiteOS Telnet采用简单的Telnet协议，暂时不需要登入用户名跟密码就可以连接到开发板，主要是做两件事：

- 读取用户在键盘上键入字符，通过TCP发送到开发板上。
- 把用户的数据经过处理后，回传到用户的终端上。

7.1.2 开发指导

开发流程

1. 在Huawei LiteOS Shell中键入命令telnet on启动telnet server



2. 在安装了telnet客户端的windows系统上，运行cmd.exe命令行工具，输入：telnet + 板子IP后可以连接到板子的系统。如下图：

3. 按下回车出现Huawei LiteOS#（Huawei LiteOS Shell提示符）表示已经成功连接到开发板，这时候可以进行Shell命令的相关操作。例如：task 查看系统当前所有任务状态。

版权所有 © 华为技术有限公司

```
Huawei LiteOS # task
Name                    PID    Priority   Status          StackSize    WaterLine    StackPoint
----                    ---    --------   --------        ---------    ---------    ----------
Swt_Task                0x0    0          QueuePend       0x6000       0x320        0x85121038
ui_MainApp              0x1    10         Pend            0x6000       0x16b0       0x85126f88
IdleCore000             0x2    31         Ready           0x400        0x154        0x851274a4
RcvAndDispatch          0x3    10         Ready           0x6000       0x25c        0x8522ed2c
SavePara2Flash          0x4    10         Ready           0x6000       0x244        0x8529fc94
RecMngMsgRcv            0x5    10         Ready           0x6000       0x984        0x852ba924
RecMngRecTimer          0x6    10         Ready           0x6000       0x284        0x852c08ec
RecMsgRcv               0x7    10         Ready           0x6000       0x23c        0x852e3d40
IspRun                  0x8    10         Ready           0x6000       0x151c       0x85301424
ProcessEncStreamThread  0x9    2          PendTimeOut     0x6000       0xbac        0x85339cfc
aenc_get                0xa    10         Pend            0x6000       0x3a4        0x8539d128
SendAFrameToAencThread  0xb    10         Ready           0x6000       0x4130       0x853a30e8
```

4. 在Huawei Liteos Shell中键入telnet off关闭telnet server。

```
Huawei LiteOS # telnet off
telnetd_accept_loop[412] Software caused connection abort
close telnet.

Huawei LiteOS #
```

7.1.3 注意事项

- 在windos下键入telnet+IP如果提示：telnet不是外部或内部命令，表示windows下的telnet服务没有开启，需要用户在本地电脑的控制面板--->程序和功能--->打开和关闭windows功能中选择开启telnet服务器和客户端。
- Telnet启动要确保以太网驱动已经初始化完成，且板子的以太网驱动打开。
- Telnet启动需要确保lwip已正常启动。使用lwip需要注册安全函数，需要初始化tcpip。
- 暂时无法支持多个客户端（telnet + IP）连接开发板。
- Telnet属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

7.1.4 编程实例

无。

7.2 Shell

7.2.1 概述

基本概念

Shell俗称壳（用来区别于核），是指“提供使用者使用界面”的软件（命令解析器）。它类似于DOS下的command和后来的cmd.exe。它接收用户命令，然后调用相应的应用程序。

同时它又是一种程序设计语言。作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

Shell可以管理你与操作系统之间的交互：等待你输入，向操作系统解释你的输入，并且处理各种各样的操作系统的输出结果。

版权所有 © 华为技术有限公司

Shell提供了你与操作系统之间通信的方式。这种通信可以以交互方式（从键盘输入，并且可以立即得到响应），或者以Shell script(非交互）方式执行。Shell script是放在文件中的一串Shell和操作系统命令，它们可以被重复使用。本质上，Shell script是命令行命令简单的组合到一个文件里面。

Huawei LiteOS嵌入式操作系统提供Shell作为在线调试工具，提供查询OS系统信息的功能。

Huawei LiteOS Shell

Huawei LiteOS提供的Shell支持调试常用的基本功能，包含系统、文件、网络和动态加载相关命令。同时Huawei LiteOS的Shell可以添加新的命令，可以根据需求来进行定制。

- 系统相关命令可以查询系统任务、内核信号量、系统软件定时器、CPU占用率、当前中断等相关信息。
- 文件相关命令除了支持基本的ls、cd等功能外，还支持sync。sync用来同步缓存数据（文件系统的数据）到sd卡或flash中。
- 网络相关命令可以查询连接到开发板的其他设备的IP，查询本机IP，测试网络连接，设置开发板的AP和station模式等相关功能。
- 动态加载相关命令从指定的路径加载obj文件，并通过查找已加载obj的函数地址，直接调用相关函数，同时用户也可以卸载指定路径加载的obj文件。

新增命令的详细流程可参见使用指导和编程实例。

7.2.2 开发指导

使用场景

Shell命令可以通过串口和telnet工具输入。新增定制的命令，需重新编译链接后运行。

功能

Huawei LiteOS系统的Shell具体包含以下命令

- 系统相关命令支持task、sem、swtmr、hwi、cpup等相关命令。
- PROC文件系统相关命令支持writeproc等相关命令。
- 文件相关命令支持ls、cd、uname、cat、touch、rm、rmdir等相关命令。
- 网络相关命令支持arp、ifconfig、ping、starthapd、stophapd等相关命令。

命令详细说明见**命令参考**，新增命令可参考**编程实例**。

开发流程

Shell的典型开发流程如下：

1. 增加Shell命令。

   #include "shell.h"

   #include "shcmd.h"

   - 注册命令项。

     SHELLCMD_ENTRY(ls_shellcmd, CMD_TYPE_EX, "ls", XARGS, (CMD_CBK_FUNC)osShellCmdLs);

版权所有 © 华为技术有限公司

- ls_shellcmd：命令项的名字；
- 命令项类型：
  1) 命令类型填写为：CMD_TYPE_EX；不支持标准命令参数输入，会把用户填写的命令关键字屏蔽掉，例如：输入ls /ramfs，传入给注册函数的参数只有/ramfs，而ls命令关键字并不会被传入。
  2) 命令类型填写为：CMD_TYPE_STD；支持的标准命令参数输入，所有输入的字符都会通过命令解析后被传入。
- "ls"：命令关键字，函数在Shell中访问的名称；
- XARGS：调用的执行函数的入参个数；
- (CMD_CBK_FUNC)osShellCmdLs：函数地址，即执行函数；

这个宏会封装注册一个命令项，在Shell中可以调用这个命令。

注意

- 需要在build/mk/liteos_tables_ldflags.mk中LITEOS_TABLES_LDFLAGS项下添加-uls_shellcmd。
- 命令关键字必须是唯一的，也即两个不同的命令项不能拥有相同的命令关键字，否则只会执行其中一个。Shell在执行用户命令时，如果存在多个命令关键字相同的命令，只会执行其中在"help"命令中排序在最前面的一个。

- 添加的内置命令函数原型。
  - UINT32 cmdHook（UINT32 argc, CHAR **argv）

  参数与c语言中带有的参数main原型类似。

注意

- argc：Shell命令中，参数个数。
- argv：为指针数组，每个元素指向一个字符串，可以根据选择命令类型，决定是否要把命令关键字传入给注册函数。

2. 输入Shell命令。

   有两种输入方式：
   - 在串口工具中直接输入Shell命令；
   - 在telnet工具中输入Shell命令；

7.2.3 注意事项

- 支持默认模式下英文输入。如果出现用户在UTF8格式下输入了中文字符的情况，只能通过回退三次来删除。
- Shell端工作目录与系统工作目录是分开的，即通过Shell端cd pwd等命令是对Shell端工作目录进行操作，通过chdir getcwd等命令是对系统工作目录进行操作，两个工作目录相互之间没有联系。当文件系统操作命令入参是相对路径时要格外注意。

版权所有 © 华为技术有限公司

- 网络的shell指令需要tcpip_init初始化后才能起作用，Huawei LiteOS默认不初始化tcpip_init。
- 若要运行动态加载相关的shell指令，需要事先完成动态加载模块的初始化，具体说明看扩展内核-动态加载一章。
- 不建议使用Shell命令对/dev目录下的设备文件进行操作，可能会引起不可预知的结果。
- Shell属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

7.2.4 编程实例

实例描述

在下面的例子中，演示如何新增一个Shell命令：settime。

1. 定义一个新增命令所要调用的执行函数cmd_datasettime。
2. 使用SHELLCMD_ENTRY函数添加新增命令项。
3. 在链接选项liteos_tables_ldflags.mk中添加链接该新增命令项参数。
4. 重新编译后运行。

编程示例

定义命令所要调用的函数cmd_datasettime：

```
#include "hi_rtc.h"
#include "shell.h"
#include "shcmd.h"
/*设置系统时间，只留作测试系统时间使用，*/
int cmd_datesettime(void)
{
rtc_time_t compositetime;

char date[7][10];
char *dateremind[] = {"year:","month:", "date:", "hour:", "minute:", "seconds:", "week:"};
int i = 0;
for(i = 0; i < 7; i++)
{
printf("%s", dateremind[i]);
gets(date[i]);
}
compositetime.year = atoi(date[1]);
compositetime.month= atoi(date[2]);
compositetime.date= atoi(date[3]);
compositetime.hour= atoi(date[4]);
compositetime.minute = atoi(date[5]);
compositetime.second= atoi(date[6]);
compositetime.weekday= atoi(date[7]);
hirtc_set_time(compositetime);
return 0;
}
```

添加新增命令项：

```
SHELLCMD_ENTRY(settime_shellcmd, CMD_TYPE_STD, "settime", 0, (CMD_CBK_FUNC)cmd_datesettime);
```

在链接选项中添加链接该新增命令项参数：

在build/mk/liteos_tables_ldflags.mk中LITEOS_TABLES_LDFLAGS项下添加-usettime_shellcmd。

版权所有 © 华为技术有限公司

重新编译：

```
make
```

7.2.5 命令参考

7.2.5.1 系统命令

7.2.5.1.1 task

命令功能

task命令用于查询系统任务信息。

命令格式

task [*ID*]

参数说明

表 7-1 参数说明

参数	参数说明	取值范围
ID	任务ID号	[0, 0xFFFFFFFF]

使用指南

- 参数缺省时默认打印全部运行任务信息。
- task后加ID，显示ID对应的任务名、任务PID、任务的调用栈信息（最大支持16层调用栈）以及当前系统运行的全部任务信息。

使用实例

举例：输入task 6

输出说明

图 7-1 查询 ID 号为 6 的任务信息

```
Huawei LiteOS # task 6
TaskName = cmdParesTask
TaskID = 0x6
*******backtrace begin*******
traceback 0 -- lr = 0x80057030
traceback 0 -- fp = 0x80536f2c
traceback 1 -- lr = 0x8005346c
traceback 1 -- fp = 0x80536f44
traceback 2 -- lr = 0x8006ee44
traceback 2 -- fp = 0x80536f5c
traceback 3 -- lr = 0x8006e294
traceback 3 -- fp = 0x80536f64
traceback 4 -- lr = 0x8006f5e4
traceback 4 -- fp = 0x80536f84
traceback 5 -- lr = 0x8005a224
traceback 5 -- fp = 0x80536f94
traceback 6 -- lr = 0x8006fa24
traceback 6 -- fp = 0x11111111

Name                 TID  Priority  Status      StackSize  WaterLine  StackPoint  TopOfStack  EventMask  SemID CPUUSE  CPUUSE10s  CPUUSE1s  MEMUSE
----                 ---  --------  ----------  ---------  ---------  ----------  ----------  ---------  ----- -------  ---------  --------  -------
Swt_Task             0x0  0         QueuePend   0x6000     0x2ec      0x801396f8  0x801338a0  0x0        0xffff   0.0      0.0        0.0       0
IdleCore000          0x1  31        Ready       0x400      0x164      0x80139b5c  0x801398c0  0x0        0xffff  99.8     98.2       99.9      0
tcpip_thread         0x3  5         PendTimeOut 0x6000     0x2f4      0x80174608  0x8016e838  0xf        0xffff   0.0      0.0        0.0       76
eth_irq_Task         0x4  3         Pend        0x20000    0x230      0x80532a90  0x80512c30  0xff       0xffff   0.0      0.0        0.0       0
shellTask            0x5  9         Running     0x3000     0x4b4      0x80535a54  0x80532f08  0xfff      0xffff   0.1      1.6        0.0       1656
cmdParesTask         0x6  9         Pend        0x1000     0x278      0x80536db0  0x80535f98  0x1        0xffff   0.0      0.0        0.0       108
```


版权所有 © 华为技术有限公司

图 7-2 查询所有任务信息

```
Huawei LiteOS # task

Name                 PID    Priority   Status      StackSize   WaterLine   StackPoint  TopOfStack  EventMask  SemID CPUUSE  CPUUSE10s  CPUUSE1s   MEMUSE
----                 ---    --------   --------    ---------   ---------   ----------  ----------  ---------  ----- -------  ---------  ---------  -------
Swt_Task             0x0    0          QueuePend   0x6000      0x38c       0x80cbc8c0  0x80cb6a68  0x0        0xffff   1.5      1.5        1.6        0
IdleCore000          0x1    31         Ready       0x400       0x230       0x80cbcd24  0x80cbca88  0x0        0xffff  78.1     78.2       78.2       0
system_wq            0x2    1          Pend        0x6000      0x240       0x80cc2d80  0x80cbcf30  0x0        0x2      0.0      0.0        0.0        0
app_Task             0x3    10         PendTimeOut 0x6000      0x34a4      0x80cc8ca8  0x80cc2f48  0x1        0xd5     0.7      0.6        0.7        26500944
shellTask            0x4    9          Running     0x3000      0x724       0x80cd0d4c  0x80cce460  0xfff      0xffff   0.0      0.4        0.0        0
cmdParesTask         0x5    9          Pend        0x1000      0x268       0x80cd2318  0x80cd14f0  0x1        0xffff   0.0      0.0        0.0        280
yaffs_bg_gc          0x6    10         Ready       0x6000      0x254       0x80cd8388  0x80cd2530  0x0        0xffff   0.0      0.0        0.0        0
tcpip_thread         0x7    5          Ready       0x6000      0x30c       0x80d9bc28  0x80d95e40  0xf        0xffff   0.0      0.0        0.0        96
eth_irq_Task         0x8    3          Pend        0x20000     0x228       0x81158090  0x81138228  0xff       0xffff   0.0      0.0        0.0        0
RcvAndDispatch       0x9    10         Ready       0x6000      0x260       0x811756b0  0x8116f880  0x0        0xffff   0.0      0.0        0.0        144
SavePara2Flash       0xa    10         Ready       0x6000      0x238       0x8117b788  0x81175930  0x0        0xffff   0.0      0.0        0.0        0
ProcessEncStreamThread 0xb  10         Ready       0x10000     0xc64       0x8118ddb8  0x8117e058  0x1        0xffff   4.0      4.3        4.7        38522608
UpdateTimeOsdThread  0xc    10         Ready       0x6000      0x88c       0x81193e70  0x8118e090  0x0        0xffff   0.1      0.1        0.1        12480
IspRun               0xd    10         Ready       0x6000      0x1614      0x8119d1a8  0x81197390  0x1        0xffff  10.2      9.8        9.6        0
aenc_get             0xe    10         Pend        0x12000     0x400       0x8121c868  0x8120abd8  0x1        0xffff   0.0      0.0        0.0        0
SendAFrameToAencThread 0xf  10         Ready       0x6000      0x5a64      0x812228e0  0x8121cbf0  0x0        0xffff   4.3      4.1        4.2        672
WORKSTATE_CHECK      0x10   10         Ready       0x6000      0x238       0x8206e960  0x82068b08  0x0        0xffff   0.0      0.0        0.0        0
STORAGE_Process_Thread_sd0 0x11 10      Pend        0x6000      0x544       0x8212afe0  0x821253f0  0xf        0xffff   0.0      0.0        0.0        659212
STORAGE_Listen_Thread_sd0 0x12 10       Ready       0x6000      0x5e8       0x821250d8  0x8211f3d0  0x0        0xffff   0.0      0.0        0.0        1012
himci_Task           0x13   10         Ready       0x800       0x248       0x82076b70  0x82076500  0x0        0xffff   0.0      0.0        0.0        0
TIMER_CHECK          0x14   10         Delay       0x6000      0x7b0       0x8207cb60  0x82076d18  0x0        0xffff   0.0      0.0        0.0        -720
mmc_data_thread      0x15   6          Pend        0x800       0x280       0x8207da18  0x8207d3c0  0x4        0xffff   0.0      0.0        0.0        0
Index_Adjust_Thread  0x16   10         Pend        0x6000      0x1338      0x821314a8  0x8212bce8  0xf        0xffff   0.0      0.0        0.0        0
pth14                0x17   10         Pend        0x6000      0xbb8       0x82137b28  0x82131d08  0xf        0xffff   0.0      0.0        0.0        2400016
pth15                0x18   10         Ready       0x6000      0x238       0x8213db78  0x82137d20  0x0        0xffff   0.0      0.0        0.0        0
hi_Timer             0x19   10         Ready       0x6000      0x298       0x82143b70  0x8213dd40  0x0        0xffff   0.0      0.0        0.0        0
rec_tsk_proc         0x1a   10         Pend        0x6000      0x270       0x8214b378  0x82145558  0xf        0xffff   0.0      0.0        0.0        0
rec_tsk_preproc      0x1b   10         Pend        0x6000      0x280       0x82151388  0x8214b578  0xf        0xffff   0.0      0.0        0.0        0
rec_tsk_proc         0x1c   10         Pend        0x6000      0x270       0x82158bb0  0x82152d90  0xf        0xffff   0.0      0.0        0.0        0
rec_tsk_preproc      0x1d   10         Pend        0x6000      0x280       0x8215ebe0  0x82158db0  0xf        0xffff   0.0      0.0        0.0        0
hi_Timer             0x1f   10         Pend        0x6000      0x278       0x823b4b10  0x823aecf8  0xf        0xffff   0.0      0.0        0.0        0
gui_petimer          0x20   10         Ready       0x6000      0x390       0x826f78f0  0x826f1b48  0x0        0xffff   0.3      0.3        0.2        -1099628
thttpdmain           0x22   10         PendTimeOut 0x20000     0x560       0x826dbd70  0x826bc240  0x0        0xdd     0.0      0.0        0.0        60264
lisnSvrProc          0x23   10         Ready       0x10000     0x3d8       0x82bafd08  0x82ba0050  0x0        0xffff   0.0      0.0        0.0        0
KEY_CHECK            0x24   10         Ready       0x6000      0x290       0x82827150  0x82821350  0x0        0xffff   0.0      0.0        0.0        0
AMngMsgRcv           0x25   10         Ready       0x6000      0x230       0x82bca7d8  0x82bc4978  0x0        0xffff   0.0      0.0        0.0        0
TIMER_CHECK          0x26   10         Ready       0x6000      0x5f0       0x82bd07e0  0x82bca998  0x0        0xffff   0.0      0.0        0.0        0
```

系统任务说明（Huawei LiteOS系统初始任务有以下几种）：

任务名	描述
Swt_Task	软件定时器任务，用于处理软件定时器超时回调函数
IdleCore000	系统空闲时执行的任务
system_wq	系统默认工作队列处理任务
cmdParesTask	从底层buf读取用户的输入，初步解析命令，例如tab补全，方向键等
shellTask	从cmdParesTask接收到传过来的命令进一步解析，并查找匹配相关的注册函数，进行调用

任务状态说明：

参数	说明
Ready	任务处于准备状态
Pend	任务处于阻塞状态
PendTimeOut	等待的任务处于超时状态
QueuePend	任务处于队列阻塞状态
Running	OS当前正在运行任务

版权所有 © 华为技术有限公司

参数	说明
Delay	任务处于延时等待状态

注意

如果在task命令中，发现任务是impossible状态，请确保pthread_create创建函数时候有进行如下操作之一，否则资源无法正常回收。

- 选择的是阻塞模式应该调用pthread_join()函数;
- 选择的是非阻塞模式选择调用pthread_detach()函数;
- 如果不想去调用前面两个之中一个接口，就要去设置pthread_attr_t 状态为PTHREAD_STATE_DETACHED，将attr参数传入pthread_create，此设置和调用pthread_detach函数一样，都是非阻塞模式;
- task命令的ID参数输入形式以十进制形式表示或十六进制形式表示皆可。
- task命令的ID参数在[0, 64]范围内时，返回对应ID的任务的状态（如果对应ID的任务未创建则进行提示）；其他取值时返回参数错误的提示。

参数说明：

参数	说明
Name	表示任务名
PID	表示任务ID
Priority	表示任务的优先级
Status	表示任务当前的状态
StackSize	表示任务堆栈的大小
WaterLine	表示栈使用的最大峰值
StackPoint	表示栈的起始地址
TopOfStack	表示栈顶的地址
EventMask	表示当前任务的事件掩码，没有使用事件，则默认任务事件掩码为0（如果任务中使用多个事件，则显示的是最近使用的事件掩码）
SemId	表示当前任务拥有的信号量ID，没有使用信号量，则默认0xFFFF(如果任务中使用了多个信号量，则显示的是最近使用信号量ID)
CPUUSE	表示开机到当前时间CPU的占用率
CPUUSE10s	表示当前时间前10秒CPU占用率

版权所有 © 华为技术有限公司

参数	说明
CPUUSE1s	表示当前时间前1秒CPU占用率
MEMUSE	表示任务到当前时间所申请的内存大小，是以字节为单位

说明

MEMUSE会有正值和负值的情况。

task申请内存，MEMUSE增加；task释放内存，MEMUSE减小。

MEMUSE为0，说明该task没有申请内存或者申请的内存和释放的内存相同；

MEMUSE为正值时，说明该task中有内存未释放；

MEMUSE为负值时，说明该task释放的内存大于申请的内存。

7.2.5.1.2 sem

命令功能

sem命令用于查询系统内核信号量相关信息。

命令格式

sem [*ID*]

参数说明

表 7-2 参数说明

参数	参数说明	取值范围
ID	信号ID号	[0, 0xFFFFFFFF]

使用指南

- 参数缺省时，显示所有的信号量的使用数及信号量总数。
- sem后加ID，显示对应ID信号量的使用数。

使用实例

举例：sem 12

输出说明

图 7-3 查询指令结果

```
Huawei LiteOS # sem 12

   SemID     Count
   -----     -----
   12          0x1
No task is pended on this semphore!
```


版权所有 © 华为技术有限公司

表 7-3 参数说明

参数	说明
SemID	信号量ID
Count	信号量使用数

说明

- sem命令的ID参数输入形式以十进制形式表示或十六进制形式表示皆可。
- sem命令的ID参数在[0, 1023]范围内时，返回对应ID的信号量的状态（如果对应ID的信号量未被使用则进行提示）；其他取值时返回参数错误的提示。

7.2.5.1.3 swtmr

命令功能

swtmr命令用于查询系统软件定时器相关信息。

命令格式

swtmr [*ID*]

参数说明

表 7-4 参数说明

参数	参数说明	取值范围
ID	软件定时器ID号。	[0，0xFFFFFFFF]

使用指南

- 参数缺省时，默认显示所有软件定时器的相关信息。
- swtmr后加ID号时，显示ID对应的软件定时器相关信息。

使用实例

举例：输入swtmr和swtmr 5

输出说明

图 7-4 查询软件定时器相关信息

```
SwTmrID  State    Mode   Interval   Count    Arg       pfnHandlerAddr
-------  -------  -------  ---------  -------  --------  --------
0        Ticking  Period   100        28       0x0       0x800f3a7c
1        Ticking  Period   10         7        0x0       0x8026d270
2        Ticking  Once     300        0        0x80b47cf0  0x8035d620
3        Ticking  Once     1          0        0x80b47cf0  0x8035d4b8
4        Ticking  Once     300        0        0x80b48008  0x8035d620
5        Ticking  Once     1          0        0x80b48008  0x8035d4b8
6        Ticking  Once     300        0        0x80b48320  0x8035d620
7        Ticking  Once     1          0        0x80b48320  0x8035d4b8
8        Ticking  Once     300        133      0x80b479d8  0x8035d620
9        Ticking  Once     1          1        0x80b479d8  0x8035d4b8
```


版权所有 © 华为技术有限公司

图 7-5 查询对应 ID 的软件定时器信息

```
SwTmrID  State    Mode   Interval   Count    Arg       pfnHandlerAddr
-------  -------  ------- --------- ------- --------  --------
5         Ticking  Once     1          0       0x80b48008   0x8035d4b8
```

表 7-5 参数说明

参数	说明
SwTmrID	软件定时器ID
State	软件定时器状态
Mode	软件定时器模式
Interval	软件定时器使用的Tick数
Count	剩余的Tick数
Arg	传入的参数
pfnHandlerAddr	回调函数的地址

说明

- swtmr命令的ID参数输入形式以十进制形式表示或十六进制形式表示皆可。
- swtmr命令的ID参数在[0, 当前软件定时器个数 - 1]范围内时，返回对应ID的软件定时器的状态；其他取值时返回错误提示。

7.2.5.1.4 hwi

命令功能

hwi命令查询当前中断信息

命令格式

hwi

参数说明

表 7-6 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- 输入hwi即显示当前中断号以及对应的中断次数。

版权所有 © 华为技术有限公司

使用实例

举例：输入hwi

输出说明

图 7-6 显示中断信息

```
Huawei LiteOS# hwi
 InterruptNo     Count
       35:     377946
       40:        152
       65:       3665
       67:     226768
       68:     226763
       69:     126137
       70:          1
       71:     188968
       75:        125
```

7.2.5.1.5 cpup

命令功能

cpup命令用于查询系统CPU的占用率。

命令格式

cpup [*mode*] [*taskID*]

参数说明

表 7-7 参数说明

参数	参数说明	取值范围
mode	● 缺省：显示系统10s前的CPU占用率 ● 0：显示系统10s前的CPU占用率 ● 1：显示系统1s前的CPU占用率 ● 其他数字：显示系统<1s前的CPU占用率	[0,0xFFFF] 或0xFFFFFFFF
taskID	任务ID号	[0,0xFFFF] 或0xFFFFFFFF

使用指南

- 参数缺省时，显示系统10s前的CPU占用率。
- 只有一个参数时，该参数为mode，显示系统相应时间前的CPU占用率。
- 输入两个参数时，第一个参数为mode，第二个参数为taskID，显示对应ID号任务的相应时间前的CPU占用率。

版权所有 © 华为技术有限公司

使用实例

举例：cpup 1 1

输出说明

图 7-7 指令输出结果

```
Huawei LiteOS # cpup 1 1
TaskId 1 CpuUsage in 1s: 78.7
```

7.2.5.1.6 memcheck

命令功能

检查动态申请的内存块是否完整，是否存在内存越界造成节点损坏。

命令格式

memcheck

参数说明

表 7-8 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- 当内存池所有节点完整时，输出"memcheck over, all passed!"。
- 当内存池存在节点不完整时，输出被损坏节点的内存块信息。

使用实例

举例：输入memcheck

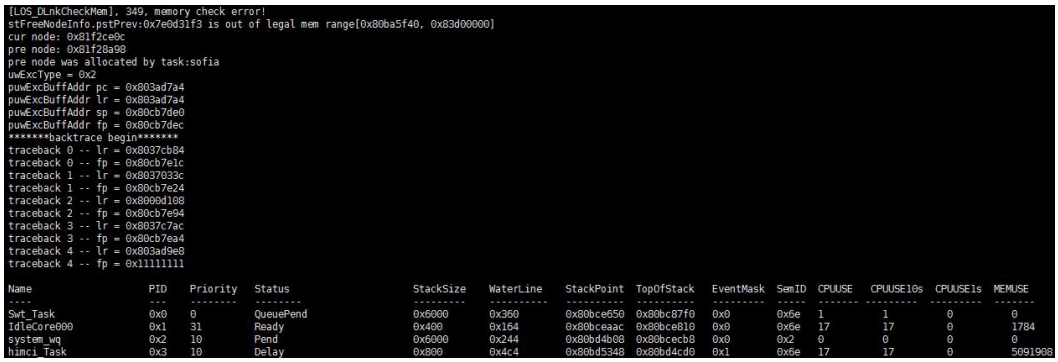
输出说明

图 7-8 当前没有内存越界

```
Huawei LiteOS # memcheck
memcheck over, all passed!
```


版权所有 © 华为技术有限公司

图 7-9 出现内存越界



7.2.5.1.7 writereg

命令功能

writereg命令用来按照给定的地址写入数据。

命令格式

writereg [*address*] [*value*]

参数说明

表 7-9 参数说明

参数	参数说明	取值范围
address	将要写入数据的地址	[0,0xFFFFFFFF]
value	将要写入的数据	[0,0xFFFFFFFF]

注意

用户需要自己保证写入的地址和值不要超过系统限制，且地址未被使用，否则可能会导致系统异常。

使用指南

- writereg命令用来按照给定的地址写入数据。
- 写入成功后会将写入的地址和数据打印在屏幕上。
- 该命令会对地址进行对齐操作，将输入的地址向下对齐到最近的4的倍数的地址上。注意地址是以16进制形式表示的。
- 免责声明：writereg为调测命令，不保证其安全性，任意写入会造成系统崩溃。

版权所有 © 华为技术有限公司

使用实例

举例：

输入 writereg 0x86412351 0x32。

输出说明

图 7-10 向指定地址写入内容时，地址被对齐在 0x86412350 处

```
Huawei LiteOS # writereg 0x86412351 0x32

The align-address:0x86412350,write value:0x00000032
Huawei LiteOS # readreg 0x86412350
86412350

The align-address:0x86412350  value:0x00000032
```

7.2.5.1.8 readreg

命令功能

readreg命令用于查询寄存器存储的内容。

命令格式

readreg [*address*] [*length*]

参数说明

表 7-10 参数说明

参数	参数说明	取值范围
address	要查询的起始寄存器地址	(0x0，0xFFFFFFFF)
length	要查询的长度	在有效地址范围内即可

注意

用户需要自己保证写入的地址和值不要超过系统限制，否则可能会导致系统异常。

使用指南

- readreg命令用于查询寄存器存储的内容。
- 寄存器地址以16进制形式给出，地址将向下对齐4字节的地址处，然后按照对齐后的地址查询，要查询的地址包含在对齐后的查询地址中。对于给出的查询长度，将自动向上按4对齐，在屏幕上将以其16进制形式打印出来。

版权所有 © 华为技术有限公司

- 免责声明：readreg为调测命令，不保证其安全性，任意写入会造成系统崩溃。

使用实例

举例：

输入readreg 0x86412351 0x32。

输出说明

图 7-11 从指定非对齐地址读取数据，会向下对齐在 0x86412350 处。

```
Huawei LiteOS # readreg 0x86412351 0x32
86412351

The address begin 0x86412350,length:0x34

  0x86412350  :0xd57f7282 0xf5bb69cf 0x6f79b5b4 0x177c9fc3
  0x86412360  :0x5e54f3ac 0xf1fdbb27 0x9b3f6b1f 0x06e27d56
  0x86412370  :0x2bd7f565 0x3f3a3f9a 0xf7277fdb 0xdfe7c91b
  0x86412380  :0xbfb3f5fa
The address end 0x86412384
```

7.2.5.1.9 free

命令功能

free命令可显示系统内存的使用情况，同时显示系统的text段、data段、rodata段、bss段大小。

命令格式

free [*-k* | *-m*]

参数说明

表 7-11 参数说明

参数	参数说明	取值范围
无参数	以Byte为单位显示	N/A
-k	以KB为单位显示	N/A
-m	以MB为单位显示	N/A

使用指南

- 输入free显示内存使用情况，total表示系统动态内存池总量，used表示已使用空间总量，free表示未被分配的内存大小。text表示代码段大小，data表示数据段大小，rodata表示只读数据段大小，bss表示未初始化全局变量占用内存大小。
- free命令可以以三种方式来显示内存使用情况，分别是以Byte为单位、以KB为单位和以MB为单位。

版权所有 © 华为技术有限公司

使用实例

举例：分别输入free、free -k、free -m.

输出说明

图 7-12 以三种方式显示内存使用情况

```
Huawei LiteOS# free

        total        used         free
Mem:    117631744    31826864     85804880

        text         data         rodata       bss
Mem:    4116480      423656       1204224      6659316

Huawei LiteOS#  free -k

        total        used         free
Mem:    114874       31080        83793

        text         data         rodata       bss
Mem:    4020         413          1176         6503

Huawei LiteOS#  free -m

        total        used         free
Mem:    112          30           81

        text         data         rodata       bss
Mem:    3            0            1            6
```

7.2.5.1.10 uname

命令功能

uname命令用于显示当前操作系统的名称，数据 创建时间，系统名称，版本信息等。

命令格式

uname[*-a* | *-s* | *-t* | *-v* | *--help*]

参数说明

参数	参数说明
-a	显示全部信息
-t	显示数据创建的时间
-s	显示操作系统名称
-v	显示版本信息
--help	显示uname指令格式提示

版权所有 © 华为技术有限公司

使用指南

- uname用于显示当前操作系统名称。语法uname -a | -t| -s| -v 描述uname 命令将正在使用的操作系统名写到标准输出中，这几个参数不能混合使用。

使用实例

举例：输入uname -a

输出说明

图 7-13 查看系统信息

```
Huawei LiteOS # uname -a
Huawei LiteOS KernelV100R002C00B111 1.1.3 May 12 2016 16:34:57
```

7.2.5.1.11 systeminfo

命令功能

systeminfo命令用于显示当前操作系统内资源使用情况，包括任务、信号量、互斥量、队列、定时器等。

命令格式

systeminfo

参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- systeminfo用于显示当前操作系统内资源使用情况。

使用实例

举例：输入systeminfo

输出说明

图 7-14 查看系统资源使用情况

```
Huawei LiteOS # systeminfo

   Module    Used      Total     Enabled
------------------------------------------
   Task      36        65        YES
   Sem       227       1024      YES
   Mutex     5         1024      YES
   Queue     1         1024      YES
   SwTmr     8         1024      YES
```


版权所有 © 华为技术有限公司

参数说明：

参数	说明
Module	模块名称
Used	当前使用量
Total	最大可用量
Enabled	模块裁剪情况

7.2.5.1.12 help

命令功能

help命令用于显示当前操作系统内所有操作指令。

命令格式

help

参数说明

参数	参数说明
N/A	N/A

使用指南

- help用于显示当前操作系统内所有操作指令。

使用实例

举例：输入help

输出说明

图 7-15 查看系统内所有操作指令。

```
Huawei LiteOS # help
******************shell commands:************************

arp           call          cat           cd            cp            cpup          dns           findsym
format        free          help          hwi           ifconfig      lddrop        ldinit        ls
mclose        memcheck      mkdir         mopen         mount         netstat       ntpdate       partition
ping          pwd           readreg       rm            rmdir         sem           statfs        swtmr
sync          systeminfo    task          telnet        tftp          touch         umount        uname
writeproc     writereg
```

7.2.5.2 文件

版权所有 © 华为技术有限公司

7.2.5.2.1 ls

命令功能

ls命令用来显示当前目录的内容。

命令格式

ls [*path*]

参数说明

表 7-12 参数说明

参数	参数说明	取值范围
path	path为空时，显示当前目录的内容。 path为文件名时显示失败，提示：No such directory path为有效目录路径时，会显示对应目录下的内容	可以为空也可以是 1.为空 2.有效的目录路径

使用指南

- ls命令显示当前目录的内容。
- ls可以显示文件的大小。
- proc下ls无法统计文件大小，显示为0。

使用实例

举例：输入ls

输出说明

图 7-16 查看当前系统路径下的目录，显示的内容如下

```
Huawei LiteOS# ls
Directory /:
.flash              153
bin                 <DIR>
font                <DIR>
etc                 <DIR>
lost+found          <DIR>
```

7.2.5.2.2 cd

命令功能

cd命令用来改变当前目录。

版权所有 © 华为技术有限公司

命令格式

cd [*path*]

参数说明

表 7-13 参数说明

参数	参数说明	取值范围
path	文件路径	用户必须具有指定目录中的执行（搜索）许可权

使用指南

- 未指定目录参数时，会跳转至根目录。
- cd后加路径名时，跳转至该路径。
- 路径名以 /（斜杠）开头时，表示根目录。
- .（点）表示当前目录。
- ..（点点）表示父目录。

使用实例

举例：cd ..

输出说明

图 7-17 显示结果如下

```
Huawei LiteOS# cd ..

Huawei LiteOS# ls
Directory /:
                    <DIR>
bin                 <DIR>
dev                 <DIR>
ramfs               <DIR>
yaffs0              <DIR>
```

7.2.5.2.3 pwd

命令功能

pwd命令用来显示当前路径。

命令格式

pwd

版权所有 © 华为技术有限公司

参数说明

表 7-14 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- pwd 命令将当前目录的全路径名称（从根目录）写入标准输出。全部目录使用 /（斜线）分隔。第一个 / 表示根目录，最后一个目录是当前目录。

使用实例

举例：输入pwd

输出说明

图 7-18 查看当前路径

```
Huawei LiteOS# pwd
/bin/vs
```

7.2.5.2.4 cp

命令功能

拷贝文件,创建一份副本。

命令格式

cp [*source path*] [*dest path*]

参数说明

表 7-15 参数说明

参数	参数说明	取值范围
source path	源文件路径	目前只支持文件,不支持目录
dest path	目的文件路径	支持目录以及文件

使用指南

- 同一路径下，源文件与目的文件不能重名。

版权所有 © 华为技术有限公司

- 源文件必须存在，且不为目录。
- 目的路径为目录时，该目录必须存在。此时目的文件以源文件命名。
- 目的路径为文件时，所在目录必须存在。此时拷贝文件的同时为副本重命名。
- 目前不支持多文件拷贝。参数大于2个时，只对前2个参数进行操作。
- 目的文件不存在时创建新文件，已存在则覆盖。
- 免责声明：cp为调测命令，不保证其安全性，拷贝系统重要资源时，会对系统造成影响，如用于拷贝/dev/uartdev-0文件时，会产生系统卡死现象。

使用实例

举例：cp 100HSCAM/FILE0087.MP4 .

输出说明

图 7-19 显示结果如下

```
Huawei LiteOS# ls
Directory /bin/vs/sd/dcim:
100HSCAM              <DIR>

Huawei LiteOS#  cp 100HSCAM/FILE0087.MP4 .

Huawei LiteOS# ls
Directory /bin/vs/sd/dcim:
100HSCAM              <DIR>
FILE0087.MP4          11332370
```

7.2.5.2.5 cat

命令功能

cat用于显示文本文件的内容。

命令格式

cat [*pathname*]

参数说明

表 7-16 参数说明

参数	参数说明	取值范围
pathname	文件路径	已存在的文件

使用指南

- cat用于显示文本文件的内容。

版权所有 © 华为技术有限公司

使用实例

举例：cat w //w为一个文件名

输出说明

图 7-20 查看 w 文件的信息

```
Huawei LiteOS# cat w
w open return Ux836be3fc
w size is 0
```

7.2.5.2.6 touch

命令功能

- touch命令用来在当前目录下创建一个不存在的空文件。
- touch命令操作已存在的文件会成功，不会更新时间戳。

命令格式

touch [*filename*]

参数说明

表 7-17 参数说明

参数	参数说明	取值范围
filename	需要创建文件的名称	N/A

使用指南

- touch命令用来创建一个空文件，该文件可读写。
- 使用touch命令一次只能创建一个文件。
- 免责声明：touch为调测命令，不保证其安全性，系统重要资源路径下创建文件，会对系统造成影响，如在/dev路径下执行touch uartdev-0，会产生系统卡死现象。

使用实例

举例：输入touch file.c

输出说明

图 7-21 创建一个名为 file.c 的文件

```
Huawei LiteOS# touch file.c

Huawei LiteOS# ls
Directory /bin/vs:
file.c              0
sd                  <DIR>

Huawei LiteOS#
```


版权所有 © 华为技术有限公司

7.2.5.2.7 rm

命令功能

rm命令用来删除文件。

命令格式

rm [-*r*] [dir*name/filename*]

参数说明

表 7-18 参数说明

参数	参数说明	取值范围
-r	可选参数，若是删除目录则需要该参数	N/A
dirname/filename	要删除文件的名称，支持输入路径	N/A

使用指南

- rm命令一次只能删除一个文件。
- rm -r命令可以删除非空目录。
- 免责声明：rm为调测命令，不保证其安全性，如用于删除系统关键资源（eg, /dev）。

使用实例

举例：

1. 输入rm 1.c
2. 输入rm -r dir

输出说明

图 7-22 用 rm 命令删除文件 1.c

```
Huawei LiteOS# ls
Directory /ramfs:
1.c                  0

Huawei LiteOS# rm 1.c

Huawei LiteOS# ls
Directory /ramfs:
```

图 7-23 用 rm -r 删除目录 dir

```
Huawei LiteOS# ls
Directory /ramfs:
dir                  <DIR>

Huawei LiteOS# rm -r dir

Huawei LiteOS# ls
Directory /ramfs:
```


版权所有 © 华为技术有限公司

7.2.5.2.8 sync

命令功能

sync命令用于同步缓存数据（文件系统的数据）到sd卡和nandflash。

命令格式

sync

参数说明

表 7-19 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- sync命令用来刷新缓存，当没有sd卡插入时不进行操作。
- 有sd卡插入时缓存信息会同步到sd卡和Nand flash，成功返回时无显示信息。

使用实例

举例：输入sync，有sd卡时同步到sd卡，无sd卡时不操作。

输出说明

无

7.2.5.2.9 statfs

命令功能

statfs命令用来打印文件系统的信息，如该文件系统类型、总大小、可用大小等信息。

命令格式

statfs [*directory*]

参数说明

表 7-20

参数	参数说明	取值范围
directory	文件系统的路径	必须是存在的文件系统，并且其支持statfs命令

版权所有 © 华为技术有限公司

使用指南

打印信息因文件系统而异。

使用实例

以YAFFS文件系统为例：

statfs yaffs0

输出说明

statfs yaffs0后的打印信息

statfs got:

f_type = 1497497427

cluster_size = 2048

total_clusters = 704

free_clusters = 640

avail_clusters = 640

f_namelen = 255

7.2.5.2.10 format

命令功能

format指令用于格式化磁盘。

命令格式

format [*dev_inodename*] [*sectors*]

参数说明

参数	参数说明
dev_inodename	设备名
sectors	分配的单元内存或扇区大小，如果输入0表示参数为空。（取值必须为0或2的幂数，最大128）

使用指南

- format指令用于格式化磁盘，设备名可以在dev目录下查找。format时必须安装存储卡。
- format只能格式化sd和mmc卡，对Nand flash和Nor flash 格式化不起作用。
- sectors参数必须传入合法值，传入非法参数可能引发异常。

版权所有 © 华为技术有限公司

使用实例

举例：输入format /dev/mmc0 0

输出说明

图 7-24 结果如下

```
Huawei LiteOS# format /dev/mmc0
0 format /dev/mmc0 Success
```

7.2.5.2.11 mount

命令功能

mount命令用来将设备挂载到到指定目录。

命令格式

mount [*device*] [*path*] [*name*]

参数说明

表 7-21 参数说明

参数	参数说明	取值范围
device	要挂载的设备（格式为设备所在路径）	系统拥有的设备
path	指定目录 用户必须具有指定目录中的执行（搜索）许可权。	N/A
name	文件系统的种类	vfat, yaffs, jffs, ramfs, nfs，procfs

使用指南

- mount后加需要挂载的设备信息、指定目录以及设备文件格式，就能成功挂载文件系统到指定目录。

使用实例

举例：mount /dev/mmc0 /bin/vs/sd vfat

输出说明

图 7-25 将/dev/mmc0 挂载到/bin/vs/sd 目录

```
Huawei LiteOS# mount /dev/mmc0 /bin/vs/sd vfat

Huawei LiteOS#
```


版权所有 © 华为技术有限公司

7.2.5.2.12 umount

命令功能

umount命令用来卸载指定文件系统。

命令格式

umount [*dir*]

参数说明

表 7-22 参数说明

参数	参数说明	取值范围
dir	需要卸载文件系统对应的目录	系统已挂载的文件系统的目录

使用指南

- umount后加上需要卸载的指定文件系统的目录，即将指定文件系统卸载。

使用实例

举例：umount /bin/vs/sd

输出说明

图 7-26 将已在/bin/vs/sd 挂载的文件系统卸载

```
Huawei LiteOS# umount /bin/vs/sd
```

7.2.5.2.13 rmdir

命令功能

rmdir命令用来删除一个目录。

命令格式

rmdir[*dir*]

版权所有 © 华为技术有限公司

参数说明

表 7-23 参数说明

参数	参数说明	取值范围
dir	需要删除目录的名称，删除目录必须为空，支持输入路径。	N/A

使用指南

- rmdir命令只能用来删除目录。
- rmdir一次只能删除一个目录。
- rmdir只能删除空目录。

使用实例

举例：输入rmdir dir

输出说明

图 7-27 删除一个名为 dir 的目录

```
Huawei LiteOS# ls
Directory /bin/vs:
dir                 <DIR>
sd                  <DIR>

Huawei LiteOS# rmdir dir

Huawei LiteOS# ls
Directory /bin/vs:
sd                  <DIR>
```

7.2.5.2.14 mkdir

命令功能

mkdir命令用来创建一个目录。

命令格式

mkdir [*directory*]

参数说明

表 7-24 参数说明

参数	参数说明	取值范围
directory	需要创建的目录。	N/A

版权所有 © 华为技术有限公司

使用指南

- mkdir后加所需要创建的目录名即创建目录成功。
- mkdir后加路径，再加上需要创建的目录名，即在指定目录下创建目录成功。

使用实例

举例：mkdir share

输出说明

图 7-28 创建 share 目录

```
Huawei LiteOS# mkdir share

Huawei LiteOS# ls
Directory /bin/vs:
share               <DIR>
sd                  <DIR>
```

7.2.5.2.15 partition

命令功能

partition命令用来查看文件系统分区信息。

命令格式

partition [*jffs* | *yaffs*]

参数说明

表 7-25 参数说明

参数	参数说明	取值范围
jffs	显示jffs文件系统分区信息	N/A
yaffs	显示yaffs文件系统分区信息	N/A

使用指南

- partition命令用来查看文件系统分区信息。
- 该命令只支持yaffs和jffs两种文件系统。

使用实例

举例：partition yaffs

版权所有 © 华为技术有限公司

输出说明

图 7-29 查看 yaffs 文件系统分区信息

```
Huawei LiteOS # partition yaffs
yaffs partition num:0, dev name:/dev/nandblk0,
mountpt:/yaffs0, startaddr:0x00e00000,
length:0x00200000
```

7.2.5.2.16 writeproc

命令功能

writeproc命令用于向指定的proc文件系统中的文件写入内容。

命令格式

writeproc [*pcval*] [*operational mark*] [*pcPath*]

参数说明

表 7-26 参数说明

参数	参数说明	取值范围
pcval	要写入文件的内容	字符串
operational mark	当操作符为">>",会写入内容到存在的文件。	只允许">>"
pcPath	将要写入内容的文件的路径	必须为绝对路径

使用指南

- writeproc用于将内容写入文件。
- 当操作符为">>"，文件存在时，将内容写入该文件。
- 对于该文件系统下非用户创建的部分文件，writeproc可实现修改其中记录的系统信息等功能。

使用实例

输入writeproc 'sys=2' >> /proc/umap/logmpp

版权所有 © 华为技术有限公司

输出说明

图 7-30 修改 logmpp 中 sys 的等级

```
Huawei LiteOS # writeproc 'sys=2' >> /proc/umap/logmpp
sys=2 >> /proc/umap/logmpp

Huawei LiteOS # cat /proc/umap/logmpp

Huawei LiteOS # LOG BUFFER STATE----------------------------------------------------------------
MaxLen  ReadPos WritePos ButtPos
 64(KB)       0      117   65536

-----CURRENT LOG LEVEL-------------------------------------------------------
vb      :  3
sys     :  2
chnl    :  3
vpss    :  3
venc    :  3
h264e   :  3
jpege   :  3
viu     :  3
rc      :  3
isp     :  3
vgs     :  3
```

7.2.5.3 网络

7.2.5.3.1 arp

命令功能

在以太网中，主机之前的通信是直接使用MAC地址（非IP地址）来通信的，所以，对于使用IP通信的协议，必须能够将IP地址转换成MAC地址，才能在局域网（以太网）内通信。解决这个问题的方法就是主机存储一张IP和MAC地址对应的表，即ARP缓存，主机要往一个局域网内的目的IP地址发送IP包时，就可以从ARP缓存表中查询到目的MAC地址。ARP缓存是由TCPIP协议栈维护的，用户可通过ARP命令查看和修改ARP表。

命令格式

arp

arp [-i IF] -s IPADDR HWADDR

arp [-i IF] -d IPADDR

参数说明

表 7-27 参数说明

参数	参数说明	取值范围
无	打印整个ARP缓存的内容。	N/A
-i IF	指定的网络接口（可选参数）	N/A
-s IPADDR HWADDR	增加一条ARP表项，后面的参数是局域网中另一台主机的IP地址及其对应的MAC地址	N/A

版权所有 © 华为技术有限公司

参数	参数说明	取值范围
-d IPADDR	删除一条ARP表项	N/A

使用指南

- arp命令用来查询和修改TCPIP协议栈的ARP缓存表，增加非局域网内的ARP表项是没有意义的，协议栈会返回失败。
- 命令需要启动TCPIP协议栈后才能使用。

使用实例

举例：

1. 输入arp
2. 输入arp -s 192.168.1.1 00:11:22:33:44:55

输出说明

图 7-31 打印整个 ARP 缓存表

```
Huawei LiteOS # arp
Address                 HWaddress               Iface
192.168.1.2             00:E0:4C:97:83:DB       eth0
```

表 7-28 参数说明

参数	说明
Address	连接到板子的网络设备IP地址
HWaddress	连接到板子的网络设备mac地址
Iface	表示该ARP表项使用的接口名

7.2.5.3.2 ifconfig

命令功能

ifconfig命令用来查询和设置网卡的IP地址、网络掩码、网关、硬件mac地址等参数。同时配置协议栈启用/关闭网卡的数据处理。

命令格式

ifconfig

[-a]

<interface> <address> [netmask <address>] [gateway <address>]

版权所有 © 华为技术有限公司

[hw ether <address>]

[up|down]

参数说明

表 7-29 参数说明

参数	参数说明	取值范围
不带参数	打印所有网卡的IP地址、网络掩码、网关、硬件mac地址、MTU、运行状态等信息	N/A
-a	打印协议栈数据收发统计数据	N/A
interface	指定网卡名，比如en0	N/A
address	设置IP地址，比如192.168.1.10，需指定网卡名	N/A
netmask	设置子网掩码，后面要掩码参数，比如255.255.255.0	N/A
gateway	设置网关，后面跟网关参数，比如192.168.1.1	N/A
hw ether	设置mac地址, 后面是MAC地址，比如00:11:22:33:44:55。目前只支持ether硬件类型。	N/A
up	启用网卡数据处理，需指定网卡名	N/A
down	关闭网卡数据处理，需指定网卡名	N/A

使用指南

- ifconfig可用于查询和设置网络模式（wifi、以太网、3G/4G网卡），IP地址，掩码，网关，mac地址等参数。
- 命令需要启动TCPIP协议栈后才能使用

使用实例

举例：输入命令 **ifconfig** *eth0 192.168.100.31 netmask 255.255.255.0 gateway 192.168.100.1 hw ether 00:49:cb:6c:a1:31*

设置板子IP为192.168.100.31，掩码255.255.255.0，网关192.168.100.1，硬件mac地址00:49:cb:6c:a1:31。

输出说明

设置网络参数

```
Huawei LiteOS # ifconfig
eth0    ip:192.168.1.2 netmask:255.255.255.0 gateway:192.168.1.1
```


版权所有 © 华为技术有限公司

```
       HWaddr d2:ba:f4:0d:fb:89 MTU:1500 Runing Default Link UP
lo     ip:127.0.0.1 netmask:255.0.0.0 gateway:127.0.0.1
       HWaddr 00 MTU:0 Runing Link Down

Huawei LiteOS # ifconfig eth0 192.168.100.31 netmask 255.255.255.0 gateway 192.168.100.1 hw ether
00:49:cb:6c:a1:31

Huawei LiteOS # ifconfig
eth0   ip:192.168.100.31 netmask:255.255.255.0 gateway:192.168.100.1
       HWaddr 00:49:cb:6c:a1:31 MTU:1500 Runing Default Link UP
lo     ip:127.0.0.1 netmask:255.0.0.0 gateway:127.0.0.1
       HWaddr 00 MTU:0 Runing Link Down
```

输出的各参数说明如下表所示：

表 7-30 参数说明

参数	说明
ip	板子IP地址
netmask	网络掩码
gateway	网关
HWaddr	板子硬件mac地址
MTU	网络最大传输单元
Running/Stop	网卡是否正在运行
Default	有这项说明此网卡连接到默认网关
Link UP/Down	网卡连接状态

7.2.5.3.3 ping

命令功能

ping命令用于测试网络连接是否正常。

命令格式

ping <*IP*> *[count]*

参数说明

表 7-31 参数说明

参数	参数说明	取值范围
IP	要测试是否网络连通的IP地址。	
count	执行的次数，不带本参数则默认为4次	1~65535

版权所有 © 华为技术有限公司

使用指南

- ping命令用来测试到目的IP的网络连接是否正常，参数为目的IP地址。
- 如果目的IP不可达，会显示请求超时。
- 如果显示发送错误，说明没有到目的IP的路由。
- 命令需要启动TCPIP协议栈后才能使用。

使用实例

举例：输入ping 192.168.0.2

输出说明

图 7-32 ping tftp 服务器地址

```
Huawei LiteOS# ping 192.168.0.2
Reply from 192.168.0.2: time=2ms TTL=128
Reply from 192.168.0.2: time=1ms TTL=128
Reply from 192.168.0.2: time=5ms TTL=128
Reply from 192.168.0.2: time=2ms TTL=128
```

7.2.5.3.4 tftp

命令功能

TFTP（Trivial File Transfer Protocol,简单文件传输协议）是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为69。

命令格式

tftp *<-g/-p> -l [FullPathLocalFile] -r [RemoteFile] [Host]*

参数说明

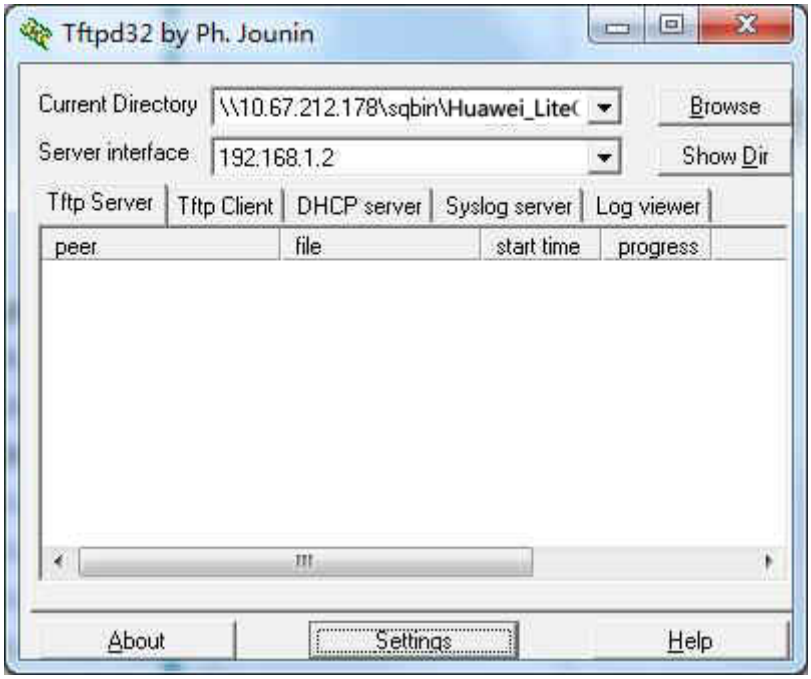
表 7-32 参数说明

参数	参数说明	取值范围
-g	从服务器获取文件	选择-g和-p中的一个
-p	上传文件到服务器	选择-g和-p中的一个
-l	本地文件名（Huawei LiteOS需全路径打开）	
-r	服务端文件名	
Host	服务端IP	

版权所有 © 华为技术有限公司

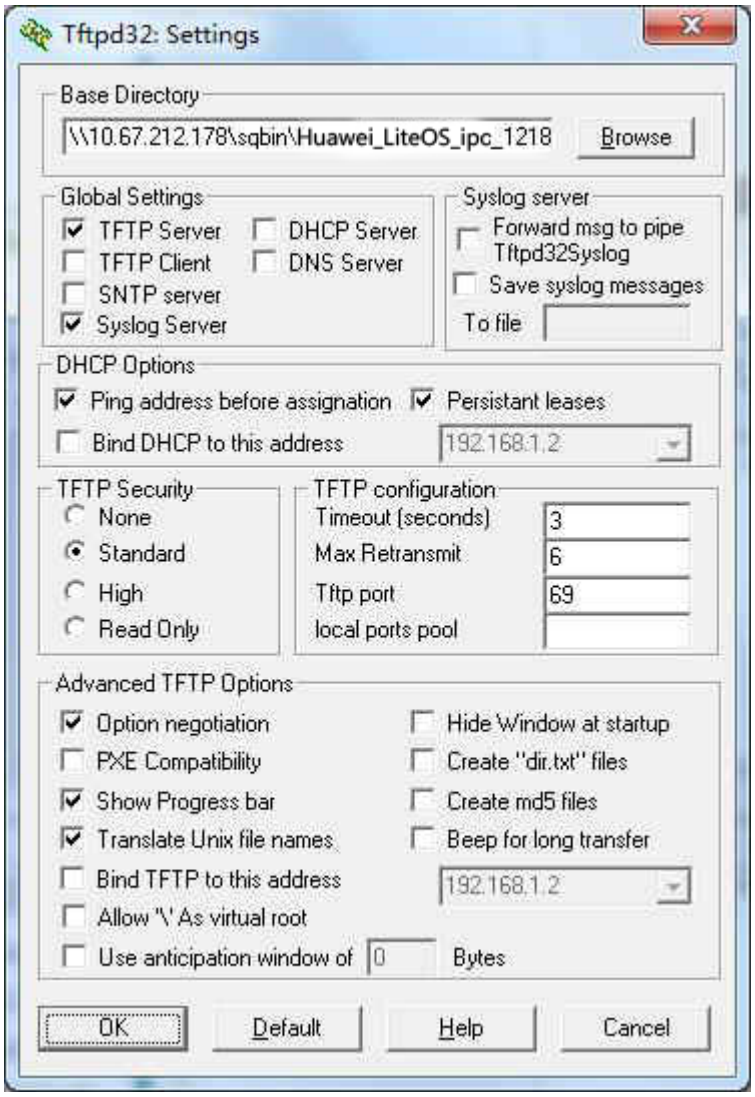
使用指南

1. 在服务器端搭建NFS TFTP服务器，首先需保证服务端已经安装了TFTP客户端，然后按照如下图示进行配置：



点Setting设置TFTP服务器：

版权所有 © 华为技术有限公司



这里主要设置Base Directory这项为TFTP目录，将其设置为镜像文件所在路径，然后点OK退出。

2. Huawei LiteOS单板使用tftp命令上传、下载文件。
3. 传输的文件大小是有限制的不能大于32M。
4. tftp属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
5. 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

使用实例

举例：从服务器下载vs_server.bin文件。

输出说明

```
Huawei LiteOS # tftp -g -l /nfs/vs_server.bin -r vs_server.bin 192.168.1.2
TFTP transfer finish
```

tftp命令执行后，传输正常完成会显示TFTP transfer finish， 失败的话会显示其他的打印信息帮助定位问题。

版权所有 © 华为技术有限公司

7.2.5.3.5 ntpdate

命令功能

命令用于从服务器同步系统时间。

命令格式

从NTP服务器获取系统时间：

ntpdate [*SERVER_IP1*] [*SERVER_IP2*]...

参数说明

表 7-33 参数说明

参数	参数说明	取值范围
SERVER_IP	NTP服务器IP	

使用指南

直接执行ntpdate [*SERVER_IP1*] [*SERVER_IP2*]...

ntpdate会获取第一个有效服务器IP的时间并显示。

使用实例

举例：

使用**ntpdate**命令更新系统时间。

输出说明

使用ntpdate命令更新系统时间：

```
Huawei LiteOS # ntpdate 192.168.1.3
time server 192.168.1.3: Mon Jun 13 09:24:25 2016
```

因为板子和服务器时区的不同，获取后的显示时间可能和服务器时间有数小时的差别。

7.2.5.3.6 dns

命令功能

命令用于设置单板dns服务器地址。

命令格式

dns <*1-2*> <*IP*>

dns -*a*

版权所有 © 华为技术有限公司

参数说明

表 7-34 参数说明

参数	参数说明	取值范围
<1-2>	选择设置第一个还是第二个DNS服务器	1~2
<IP>	服务器IP地址	
-a	显示当前设置状态	

使用指南

直接输入命令。

使用实例

举例：

1. 检查当前DNS设置。
2. 设置第二个DNS服务器IP。
3. 检查DNS设置是否成功。

输出说明

1. 检查当前DNS设置：
```
Huawei LiteOS # dns -a
dns1: 208.67.222.222
dns2: 0.0.0.0
```
2. 设置第二个DNS服务器IP：
```
Huawei LiteOS # dns 2 192.168.1.2
```
3. 检查DNS设置是否成功：
```
Huawei LiteOS # dns -a
dns1: 208.67.222.222
dns2: 192.168.1.2
```

7.2.5.3.7 netstat

命令功能

netstat是控制台命令,是一个监测TCP/IP网络的非常有用的工具，它可以显示实际的网络连接以及每一个网络接口设备的状态信息。Netstat用于显示与TCP、UDP协议相关的统计数据，一般用于检验本设备（单板）各端口的网络连接情况。

命令格式

netstat

版权所有 © 华为技术有限公司

参数说明

表 7-35 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

直接输入命令。

使用实例

举例：输入netstat

输出说明

```
Huawei LiteOS # netstat
Proto  Local Address          Foreign Address     State
udp    0.0.0.0:49153          0.0.0.0:0           ---
tcp    192.168.1.10:5001      192.168.1.2:60263   ESTABLISHED
tcp    0.0.0.0:5001           0.0.0.0:0           LISTEN
```

表 7-36 参数说明

参数	说明
Proto	协议类型
Local Address	本地地址和端口
Foreign Address	远程地址和端口
State	TCP连接状态，UDP此项无意义

7.2.5.3.8 telnet

命令功能

telnet用于从终端使用者的电脑上通过网络连接到服务器。

命令格式

telnet [*on* | *off*]

版权所有 © 华为技术有限公司

参数说明

表 7-37 参数说明

参数	参数说明	取值范围
on	启动server服务端	N/A
off	关闭server服务端	N/A

使用指南

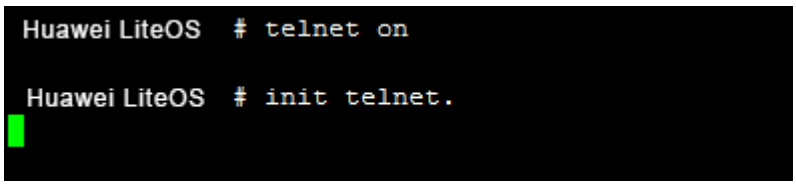
- telnet用于从终端使用者的电脑上通过网络连接到服务器。
- telnet启动要确保以太网驱动已经初始化完成，且板子的以太网驱动打开。
- 暂时无法支持多个客户端（telnet + IP）链接开发板。
- telnet属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。
- 免责声明：华为不承担在正式商用产品中使用该功能的任何风险。

使用实例

举例：输入telnet on

输出说明

图 7-33 输入 telnet on



7.2.5.4 动态加载

7.2.5.4.1 ldinit

命令功能

初始化动态加载模块。

命令格式

ldinit *symbol_path*

版权所有 © 华为技术有限公司

参数说明

表 7-38 参数说明

参数	参数说明
symbol_path	符号表所在目标文件路径

使用指南

- symbol_path是系统符号表所在路径。

使用实例

举例：以/yaffs/bin/dynload/elf_symbol.so为符号表目标文件初始化动态加载模块。

输出说明

```
Huawei LiteOS# ldinit /yaffs/bin/dynload/elf_symbol.so
Symbol lflush redefined!

Huawei LiteOS#
```

未提示错误表示初始化成功（出现符号重定义时，函数返回成功，同时会打印符号重定义的信息）。

7.2.5.4.2 mopen

命令功能

加载一个用户模块。

命令格式

mopen *module_path*

参数说明

表 7-39 参数说明

参数	参数说明
module_path	用户模块所在路径

使用指南

- module_path可以是.o文件，也可以是.so文件。

使用实例

举例：加载/yaffs/bin/dynload/foo.o

版权所有 © 华为技术有限公司

输出说明

```
Huawei LiteOS# mopen /yaffs/bin/dynload/foo.o
module handle: 0x80391928

Huawei LiteOS#
```

加载成功返回模块句柄，本例中返回的模块句柄为0x80391928。

7.2.5.4.3 findsym

命令功能

查找一个符号所在地址。

命令格式

findsym *handle symbol_name*

参数说明

表 7-40 参数说明

参数	参数说明
hanlde	模块句柄
symbol_name	所要查找的符号名

使用指南

- handle传入0时在全局符号表中查找符号地址（全局符号表中包含了内核符号以及用户模块提供的符号）。
- handle不为0且是个合法句柄时，在该handle指定的模块中查找符号。

使用实例

举例：演示在全局符号表中查找printf符号，以及在mopen中打开的句柄为0x80391928的用户模块中查找test_0符号地址。

输出说明

```
Huawei LiteOS# findsym 0 printf
symbol address:0x8004500c

Huawei LiteOS#
Huawei LiteOS# findsym 0x80391928 test_0
symbli address:0x8030f241

Huawei LiteOS#
```


版权所有 © 华为技术有限公司

7.2.5.4.4 call

命令功能

调用一个无参函数。

命令格式

call *func_address*

参数说明

表 7-41 参数说明

参数	参数说明
func_address	函数所在内存地址

使用指南

- 当在findsym中查找到一个函数符号所在内存地址后，可以调用该函数。
- 免责声明：call为调测命令，不保证其安全性，call任意内存时，会造成系统崩溃。

使用实例

举例：调用在findsym中查找到的地址为0x8030f241的test_0函数。

输出说明

```
Huawei LiteOS# call 0x8030f241
test_0

Huawei LiteOS#
```

7.2.5.4.5 mclose

命令功能

卸载一个模块。

命令格式

mclose *module_handle*

版权所有 © 华为技术有限公司

参数说明

表 7-42 参数说明

参数	参数说明
module_handle	在mopen中打开的模块返回的句柄值

使用指南

- 卸载了一个指定句柄的模块后，无法再从该模块中查找符号。

使用实例

举例：卸载句柄为0x80391928的用户模块。

输出说明

```
Huawei LiteOS# mclose 0x80391928

Huawei LiteOS#
```

无错误信息返回表示卸载成功。

7.2.5.4.6 lddrop

命令功能

卸载动态加载模块。

命令格式

lddrop

参数说明

表 7-43 参数说明

参数	参数说明	取值范围
N/A	N/A	N/A

使用指南

- 卸载动态加载模块后无法再使用动态加载特性。如若再使用动态加载特性需要重新初始化动态加载模块。

使用实例

举例：卸载动态加载模块。

版权所有 © 华为技术有限公司

输出说明

```
Huawei LiteOS# lddrop

Huawei LiteOS#
```

无错误信息返回表示卸载成功。

版权所有 © 华为技术有限公司

8 调试指南

版权所有 © 华为技术有限公司

8.1 踩内存定位方法

8.1.1 通过异常信息定位问题

系统异常被挂起后，会在串口看到一些关键寄存器的信息，如图1所示。

图 8-1

```
uwExcType = 0x4
puwExcBuffAddr pc = 0x80041a50
puwExcBuffAddr lr = 0x8
puwExcBuffAddr sp = 0x80146328
puwExcBuffAddr fp = 0x8014835c
*******backtrace begin*******
R0 = 0x1
R1 = 0x80146366
R2 = 0x1d
R3 = 0x6
R4 = 0x80146349
R5 = 0x80146338
R6 = 0x80146345
R7 = 0x4
```

通常，调试人员可以对比Huawei_LiteOS\out\<platform>\vs_server.asm文件，并通过pc指针找到当前所执行的操作，该操作就是导致异常的直接原因。以下图为例：

图 8-2

```
80041a44: e3a0201d mov r2, #29
80041a48: e3a03006 mov r3, #6
80041a4c: e0841002 add r1, r4, r2
80041a50: e7868007 str r8, [r6, r7]
```

查看pc指针0x80041a50，发现异常发生时，正在执行指令(str r8,[r6,r7])。尝试通过分析该指令，初步定位异常发生的原因。

8.1.2 内存池节点完整性验证

仅凭上节异常信息定位的基本方法，常常无法直接定位问题所在。并且经常会发生这些寄存器本身的值异常而无法定位。如果怀疑是内存越界相关的问题导致，可以调用内存池完整性检测函数osShellCmdMemcheck进行check。

函数osShellCmdMemcheck将会对系统动态内存池所有的节点进行遍历，如果所有节点正常则会打印log：“memcheck over, all passed!”，否则将打印相关的错误信息。

为了更好的说明，这里设置了一个测试场景，代码如下：

```
VOID sampleFunc(VOID *p)
{
memset(p, 0, 0x110);//超出长度的memset,设置踩内存场景
}

#include "los_dlinkmem.h"
UINT32 test(UINT32 argc, CHAR **args)
{
```


版权所有 © 华为技术有限公司

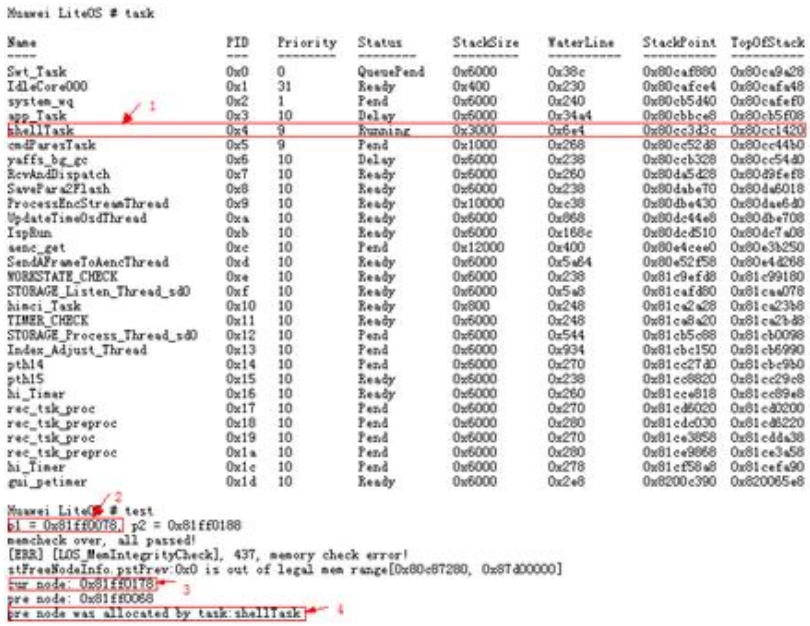
```
void *p1,*p2;

p1 = LOS_MemAlloc((void*)OS_SYS_MEM_ADDR,0x100);
p2 = LOS_MemAlloc((void*)OS_SYS_MEM_ADDR,0x100);
dprintf("p1 = %p, p2 = %p \n", p1, p2);

osShellCmdMemcheck(0,NULL); //内存完整性检测
sampleFunc(p1); // 假设此处可能存在踩内存的操作
osShellCmdMemcheck(0,NULL); //内存完整性检测

LOS_MemFree(OS_SYS_MEM_ADDR, (void *)p1);
LOS_MemFree(OS_SYS_MEM_ADDR, (void *)p2);

return 0;
}
```

操作步骤：运行“task”shell命令打印task状态；运行“test ”shell命令，执行以上程序。

对应的log信息如下：

1. 可以看到第一次运行osShellCmdMemcheck时，打印“memcheck over, all passed!”，表示此时没有踩内存。
2. 在第二次运行osShellCmdMemcheck时，打印了错误信息，表示是两次osShellCmdMemcheck之间的操作导致的踩内存。并且打印log，可以看到标记3所指“cur node：0x81ff0078”，表示该节点内存被踩。图1可以看到“p2 = 0x81ff0188”，减去控制头大小0x10后，即p2-0x10=cur node。（前提是p1,p2是两个相连的节点，这个可以对比p1, p2打印出来的地址得到验证，p2-p1=0x110,0x100是size大小，0x10是控制头的大小）
3. 标记4所示“pre node was allocated by task:shellTask”这个踩内存的操作发生在shellTask任务中。

图 8-3



8.1.3 memset 和 memcpy 可用长度检测

为了加快定位速度，所以在最有可能导致内存越界的操作memset和memcpy中增加长度检测，若检测到将操作的长度超出了被操作节点的可用长度，将打印log进行提示，并

版权所有 © 华为技术有限公司

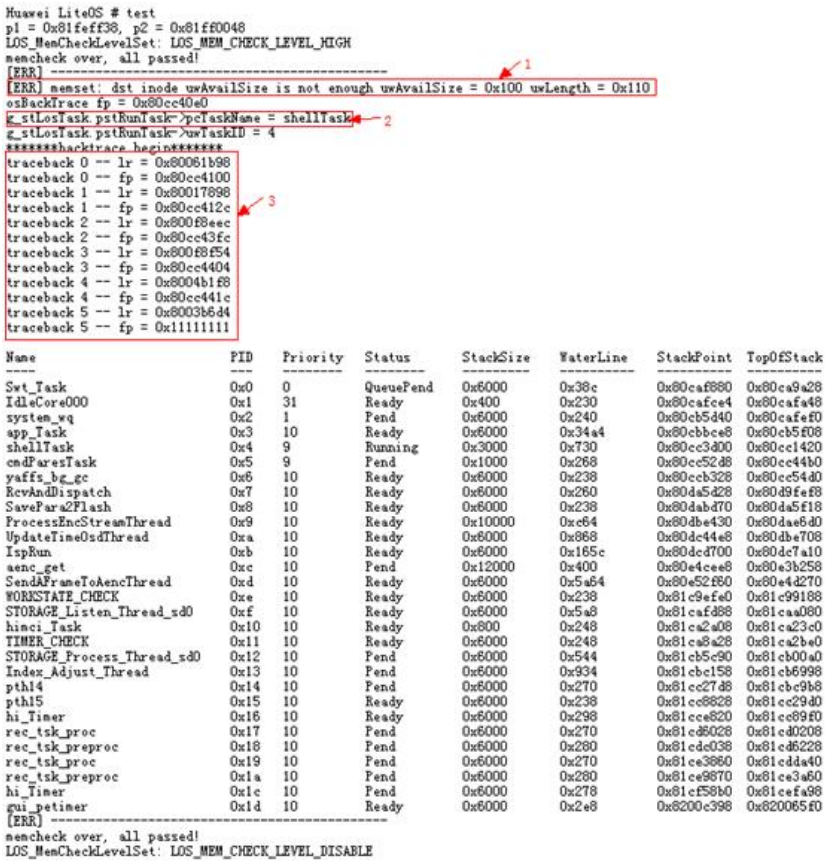
且该次memset或memcpy操作被取消（否则可能无法看到完整log），可依此判断是否memset或者memcpy而导致踩内存。相比上述场景开启了memset&memcpy长度检测，代码如下：

```
VOID sampleFunc(VOID *p)
{
memset(p, 0, 0x110);//超出长度的memset,设置踩内存场景
}

#include "los_dlinkmem.h"
UINT32 test(UINT32 argc, CHAR **args)
{
void *p1,*p2;

p1 = LOS_MemAlloc((void*)OS_SYS_MEM_ADDR,0x100);
p2 = LOS_MemAlloc((void*)OS_SYS_MEM_ADDR,0x100);
dprintf("p1 = %p, p2 = %p \n", p1, p2);

LOS_MemCheckLevelSet(LOS_MEM_CHECK_LEVEL_HIGH); //开启对memset&memcpy 的长度检测

osShellCmdMemcheck(0,NULL); //内存完整性检测
sampleFunc(p1); // 假设此处可能存在踩内存的操作
osShellCmdMemcheck(0,NULL); //内存完整性检测

LOS_MemCheckLevelSet(LOS_MEM_CHECK_LEVEL_DISABLE);//关闭对memset&memcpy 的长度检测

LOS_MemFree(OS_SYS_MEM_ADDR, (void *)p1);
LOS_MemFree(OS_SYS_MEM_ADDR, (void *)p2);

return 0;
}
```

运行结果如下图所示，由于开启检测，非法的memset操作被取消，打印了完整log。标记1所示表示此时memset&memcpy长度超出。标记2显示了该非法操作发生在shellTask中。标记3打印了寄存器lr和fp，此时可以打开Huawei_LiteOS\out\<platform>\vs_server.asm文件，通过对比“寄存器lr”的值，查看函数的嵌套调用。

版权所有 © 华为技术有限公司

图 8-4



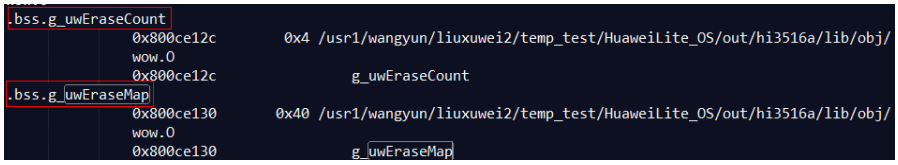
8.1.4 全局变量踩内存定位方法

如果已知一个全局变量被踩内存，可在Huawei_LiteOS\build\<platform>\vs_server.map文件中找到该全局变量所在的地址。并且特别注意该地址之前最近被使用的变量，有极大概率，是前面的变量在使用的过程中内存越界，踩掉了这个全局变量。这里例举一个测试时的例子：

在erase.c文件中定义了两全局变量，并且初始化。

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

UINT32 g_uwEraseMap[16] = {0};

UINT32 g_uwEraseCount = 0;

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

图 8-5



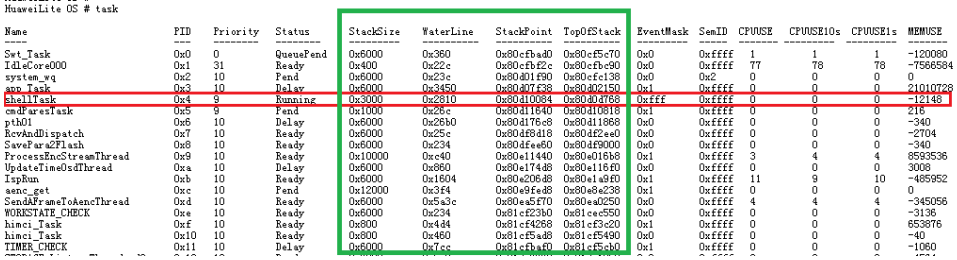
版权所有 © 华为技术有限公司

然后在.map中可以找到该全局变量在bss段对应的位置。若表现的现象为g_uwEraseMap被踩，则需要特别注意分析g_uwEraseCount这个全局变量的使用情况，观察是否存在某处，对全局变量g_uwEraseCount进行了越界操作。

8.1.5 task 状态判断是否踩内存

起业务后，串口中运行“task”shell指令。可以查看当前系统所有任务的状态。stackSize、WaterLine、StackPoint、Top0fStack将作为一个指标判断任务栈是否踩内存.

图 8-6



```
HuaweiLite OS # task
Name        PID  Priority  Status     StackSize  WaterLine  StackPoint  Top0fStack  EventMask  SemID  CPUUSE  CPUUSE10s  CPUUSE1s  MEMUSE
----        ---  --------  ---------  ---------  ---------  ----------  ----------  ---------  -----  ------  ---------  --------  ------
Swt_Task    0x0  0         QueuePend  0x6000     0x360      0x80cfbad0  0x80cf5c70  0x0        0xffff 1       1          1         -120080
IdleCore000 0x1  31        Ready      0x400      0x22c      0x80cfbf2c  0x80cfbe90  0x0        0xffff 77      78         78        -7566584
system_wq   0x2  10        Pend       0x6000     0x23c      0x80d01f90  0x80cfc138  0x0        0x2    0       0          0         0
app_Task    0x3  10        Delay      0x6000     0x2450     0x80d07f38  0x80d02150  0x1        0xffff 0       0          0         21010728
shellTask   0x4  9         Running    0x3000     0x2810     0x80d10084  0x80d0d768  0xffff     0xffff 0       0          0         -12148
cmdFaresTask 0x5 9         Pend       0x1000     0x26c      0x80d11640  0x80d10818  0x1        0xffff 0       0          0         216
pth01       0x6  10        Delay      0x6000     0x26b0     0x80d176c8  0x80d11868  0x0        0xffff 0       0          0         -340
RcvAndDispatch 0x7 10        Ready      0x6000     0x25c      0x80d68d18  0x80d62ea0  0x0        0xffff 0       0          0         -2704
SaveFar2Flash 0x8  10        Ready      0x6000     0x234      0x80dfee60  0x80df9000  0x0        0xffff 0       0          0         -340
ProcessEncStreamThread 0x9 10 Ready      0x10000    0xc40      0x80e11440  0x80e016b8  0x1        0xffff 3       4          4         8593536
UpdateTimeOsdThread 0xa 10    Delay      0x6000     0x860      0x80e174d8  0x80e116f0  0x0        0xffff 0       0          0         3008
IspRun      0xb  10        Ready      0x6000     0x1604     0x80e206d8  0x80e1a9f0  0x1        0xffff 11      9          10        -485952
aenc_get    0xc  10        Pend       0x12000    0x3f4      0x80e9fed8  0x80e8e238  0x1        0xffff 0       0          0         0
SendAFrameToAencThread 0xd 10 Ready      0x6000     0x5a3c     0x80ea5f70  0x80ead250  0x0        0xffff 4       4          4         -345056
WORKSTATE_CHECK 0xe 10        Ready      0x6000     0x234      0x81cf23b0  0x81cec550  0x0        0xffff 0       0          0         -3136
himci_Task  0xf  10        Ready      0x800      0x4d4      0x81cf4268  0x81cf3c20  0x1        0xffff 0       0          0         653876
himci_Task  0x10 10        Ready      0x800      0x460      0x81cf5ad8  0x81cf5490  0x0        0xffff 0       0          0         -40
TIMER_CHECK 0x11 10        Delay      0x6000     0x7cc      0x81cfbaf0  0x81cf5cb0  0x1        0xffff 0       0          0         -1060
```

结果：举例，任务名为shellTask，

StackSize = 0x3000（创建该任务时分配的栈大小）

WaterLine = 0x2810（水线，目前为止该任务栈已经被使用的内存大小）

StackPoint = 0x80d10084 （任务栈指针， 指向该任务当前的地址）

Top0fStack = 0x80d0d768（栈顶）

MaxStackPoint = Top0fStack + StackSize = 0x80d10768(得到该任务栈最大的可访问地址)

对比WaterLine和StackSize， 若WaterLine大于StackSize，则说明该任务踩内存

观察StackPoint ， 理论上 StackPoint<= MaxStackPoint && StackPoint >= Top0fStack,

若不在这个范围内，则说明任务栈踩内存。

8.2 踩内存解决案例

8.2.1 音频库踩内存

基本信息

表 8-1

适配	描述
适用操作系统	Huawei LiteOS

背景及现象

在BVT提供给雄迈的版本上面，概率性出现系统挂死异常。

版权所有 © 华为技术有限公司

原因分析

AEC算法处理时需要8字节对齐的buffer，而海思封装的时候传递的buffer做8字节对齐处理方式有问题（(state->pAEC_buffer) & 0xFFFFFFF8)），如果申请的buffer非8字节对齐，那处理过后会导致buffer地址回退最多8字节，从而导致越界；在linux下未发现问题，与使用中malloc出来地址已经对齐或者越界的地址刚好没有被使用，所以一直未触发该问题。

解决方法

针对AEC算法特殊8字节对齐的需求，malloc申请时做正确8字节对齐处理：

```
state->pAEC_buffer = malloc(s32AecSize + 8);
if(HI_NULL == state->pAEC_buffer)
{
…;
}
/*pAEC_buffer should be 8 byte alignment*/
if((state->pAEC_buffer - (HI_VOID*)HI_NULL) & 0x7)
{
s32AlignNum = 8 - ((state->pAEC_buffer - (HI_VOID*)HI_NULL) & 0x7);
}
else
{
s32AlignNum = 0;
}
…
(void *)((HI_U8 *)(state->pAEC_buffer) + s32AlignNum)
```

建议与总结

在内存的操作，特别是指针的操作，需要更加仔细，防止越界。

8.2.2 音频任务名乱码

基本信息

表 8-2

适配	描述
适用操作系统	Huawei LiteOS

版权所有 © 华为技术有限公司

背景及现象

在LiteOS 的shell命令中敲入task命令后出现音频线程名称乱码。

原因分析

任务名称乱码初步分析有两个可能性导致：

1. 内存越界
2. 野指针继续使用

首先分析第一种可能性，如果存在内存越界的问题，而且任务名严重溢出，则很可能是大规模的越界导致，但是在task命令中保存的信息除任务名外均为正常信息，而任务名在内核中仅仅是一个指针传入，这更增大了野指针的可疑性。另外使用内存检查函数，在shell中随机敲入memcheck命令检查系统内存，不管是task任务名异常与否，该检查均pass。基本排除内存越界的可能性。

再来分析第二种可能性，任务控制块（TCB）中保存任务名的仅仅是一个字符串指针，也就是说里面只是存了任务名所在字符串的地址；再看该任务启动线程里面，发现函数的一个局部变量

```
HI_CHAR aszThreadName[16] = "adec_sendao";
```

该局部变量在该函数执行完成后，随即退出，函数栈随之释放，即那块内存可以被其他操作访问与使用。导致在task命令查看的时候，指向的地址数据有可能已被覆盖（如果恰巧未被覆盖，则显示正常，但访问已经被释放的内存区域仍属非法操作）。

解决方法

在任务名的传递中，直接使用字符串本身进行传递即可。

建议与总结

无

8.2.3 全局变量踩内存

基本信息

表 8-3

适配	描述
适用操作系统	Huawei LiteOS

背景及现象

客户调试过程中，发现一个全局变量只在一处赋值为0，但使用时打印发现变成一个非零异常值。

版权所有 © 华为技术有限公司

原因分析

大概率是该全局变量被踩。对该全局变量前面的变量进行memcpy, memset操作时越界，溢出覆盖了当前全局变量。

解决方法

检查map文件，查看该全局变量前面的变量，排查是否前面变量操作不当引发踩内存。

建议与总结

此法对于踩全局变量屡试不爽。

版权所有 © 华为技术有限公司

9 标准库

华为专有和保密信息
版权所有 © 华为技术有限公司

9.1 POSIX 接口

9.1.1 POSIX 适配接口

Huawei LiteOS提供一套POSIX适配接口，具体的规格参见下表。

头文件	名称	类型	说明	备注
sys/socket.h	accept	函数	接受socket连线	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_accept函数,与lwip_accept函数实现一致
sys/socket.h	bind	函数	对socket定位	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_bind函数,与lwip_bind函数实现一致
time.h	clock	函数	返回处理器调用某个进程或函数所花费的时间	
time.h	clock_getres	函数	获取指定时钟的精度	
time.h	clock_gettime	函数	获取指定时钟的时间	
time.h	clock_settime	函数	设置指定时钟的时间	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
sys/socket.h	connect	函数	建立socket连线	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_connect函数,与lwip_connect函数实现一致
time.h	difftime	函数	计算两个时刻之间的时间差	
dlfcn.h	dlclose	宏	卸载打开的库	
dlfcn.h	dlopen	宏	以指定模式打开指定的动态链接库文件	标准：dlopen第二个参数是符号解析模式。 RTLD_LAZY：在dlopen返回前，对于动态库中的未定义的符号不执行解析。 RTLD_NOW：需要在dlopen返回前，解析出所有未定义符号，如果解析不出来，在dlopen会返回NULL。 Huawei LiteOS：打开动态链接库文件时不支持符号解析模式，行为与标准的RTLD_NOW模式一致。
dlfcn.h	dlsym	宏	通过句柄和链接符名称获取函数地址或者变量地址	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
sys/socket.h	getpeername	函数	获取与某个套接字关联的外地协议地址	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_getpeername函数,与lwip_getpeername函数实现一致
unistd.h	getpid	函数	获取进程标识码	返回类型不一样
sys/socket.h	getsockname	函数	获取一个套接口的名字	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_getsockname函数,与lwip_getsockname函数实现一致
sys/socket.h	getsockopt	函数	取得socket状态	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_getsockopt函数,与lwip_getsockopt函数实现一致
sys/time.h	gettimeofday	函数	取得目前的时间	
in.h	htonl	函数	将主机的无符号长整型数转换成网络字节顺序	与lwip_htonl实现一致
in.h	htons	函数	将主机的无符号短整型数转换成网络字节顺序	与lwip_htons实现一致

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
arpa/inet.h	inet_aton	函数	将网络地址转换成“.”点隔的字符串格式	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的inet_aton函数
arpa/inet.h	inet_addr	函数	将一个点间隔地址转换成一个in_addr	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的inet_addr函数
mqueue.h	mq_getattr	函数	获取消息队列属性	
mqueue.h	mq_open	函数	打开消息队列	
mqueue.h	mq_receive	函数	接受一个消息队列中的消息	
mqueue.h	mq_send	函数	发送一个消息到消息队列	
mqueue.h	mq_setattr	函数	设置消息队列属性	
mqueue.h	mq_timedreceive	函数	定时接收消息	标准：超时时间为绝对时间，可以处理早已超时的情况Huawei LiteOS：超时等待时间为相对时间，不能处理早已超时的情况
mqueue.h	mq_timedsend	函数	定时发送消息	标准：超时时间为绝对时间，可以处理早已超时的情况Huawei LiteOS：超时等待时间为相对时间，不能处理早已超时的情况
mqueue.h	mq_unlink	函数	移除消息队列	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
in.h	ntohl	函数	将一个无符号长整型数从网络字节顺序转换为主机字节顺序	与lwip_ntohl实现一致
in.h	ntohs	函数	将一个无符号短整型数从网络字节顺序转换为主机字节顺序	与lwip_ntohs实现一致
pthread.h	pthread_attr_getinheritsched	函数	获取任务调度方式	
pthread.h	pthread_attr_getschedparam	函数	获取任务调度优先级属性	
pthread.h	pthread_attr_getschedpolicy	函数	获取任务调度策略属性	标准：调度策略支持SCHED_OTHERSCHED_FIFOSCHED_RRHuawei LiteOS：仅支持SCHED_RR调度策略
pthread.h	pthread_attr_getscope	函数	获取任务范围属性	标准：任务使用范围支持PTHREAD_SCOPE_S完全兼容STEM与PTHREAD_SCOPE_PROCESSHuawei LiteOS任务使用范围只支持PTHREAD_SCOPE_S完全兼容STEM，不支持PTHREAD_SCOPE_PROCESS
pthread.h	pthread_attr_getstacksize	函数	获取任务属性堆栈大小	
pthread.h	pthread_attr_init	函数	初始化任务属性	
pthread.h	pthread_attr_setdetachstate	函数	设置任务属性分离状态	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
pthread.h	pthread_attr_setinheritsched	函数	设置任务调度方式	
pthread.h	pthread_attr_setschedparam	函数	设置任务调度优先级属性	标准：设置参数值越大，优先级越高。Huawei LiteOS：设置任务优先级的参数值越小，任务在系统中的优先级就，与标准相反
pthread.h	pthread_attr_setschedpolicy	函数	设置任务调度策略属性	标准：调度策略支持SCHED_OTHERSCHED_FIFOSCHED_RRHuawei LiteOS：仅支持SCHED_RR调度策略
pthread.h	pthread_attr_setscope	函数	设置任务范围属性	标准：任务使用范围支持PTHREAD_SCOPE_S完全兼容STEM与PTHREAD_SCOPE_PROCESSHuawei LiteOS任务使用范围只支持PTHREAD_SCOPE_S完全兼容STEM，不支持PTHREAD_SCOPE_PROCESS
pthread.h	pthread_attr_setstacksize	函数	设置任务属性堆栈大小	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
pthread.h	pthread_cancel	函数	取消任务	标准：可以在阻塞点实现任务取消Huawei LiteOS：仅支持先设置PTHREAD_CANCEL_AS完全兼容NCHRONOUS状态，再调用pthread_cancel取消任务
pthread.h	pthread_cond_broadcast	函数	唤醒所有被阻塞在条件变量上的线程	
pthread.h	pthread_cond_destroy	函数	释放条件变量	
pthread.h	pthread_cond_init	函数	初始化条件变量	
pthread.h	pthread_cond_signal	函数	释放被阻塞在条件变量上的一个线程	
pthread.h	pthread_cond_timedwait	函数	超时时限内等待一个条件变量	标准：超时时间为绝对时间，可以处理早已超时的情况Huawei LiteOS：超时等待时间为相对时间，不能处理早已超时的情况
pthread.h	pthread_cond_wait	函数	等待一个条件变量	
pthread.h	pthread_condattr_getpshared	函数	获取条件变量属性	标准：条件变量支持PTHREAD_PROCESS_PRIVATEPTHREAD_PROCESS_SHAREDHuawei LiteOS：条件变量状态只支持PTHREAD_PROCESS_PRIVATE

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
pthread.h	pthread_condattr_setpshared	函数	设置条件变量属性	标准：条件变量支持PTHREAD_PROCESS_PRIVATEPTHREAD_PROCESS_SHAREDHuawei LiteOS：条件变量状态只支持PTHREAD_PROCESS_PRIVATE
pthread.h	pthread_create	函数	创建任务	
pthread.h	pthread_detach	函数	分离任务	
pthread.h	pthread_equal	函数	判断是否为同一任务	
pthread.h	pthread_exit	函数	任务退出	
pthread.h	pthread_getschedparam	函数	获取任务优先级及调度策略	标准：调度策略支持SCHED_OTHERSCHED_FIFOSCHED_RRHuawei LiteOS：仅支持SCHED_RR调度策略
pthread.h	pthread_join	函数	阻塞任务	
pthread.h	pthread_mutex_destroy	函数	销毁互斥锁	
pthread.h	pthread_mutex_getprioceiling	函数	获取互斥锁优先级上限	
pthread.h	pthread_mutex_init	函数	初始化互斥锁	
pthread.h	pthread_mutex_lock	函数	互斥锁加锁操作	
pthread.h	pthread_mutex_setprioceiling	函数	设置互斥锁优先级上限	
pthread.h	pthread_mutex_trylock	函数	互斥锁尝试加锁操作	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
pthread.h	pthread_mutex_unlock	函数	互斥锁解锁操作	
pthread.h	pthread_mutexattr_destroy	函数	销毁互斥锁属性	
pthread.h	pthread_mutexattr_getprioceiling	函数	获取互斥锁属性优先级上限	
pthread.h	pthread_mutexattr_getprotocol	函数	获取互斥锁属性中的协议	
pthread.h	pthread_mutexattr_gettype	函数	获取互斥锁属性中的类型	
pthread.h	pthread_mutexattr_init	函数	初始化互斥锁属性	
pthread.h	pthread_mutexattr_setprioceiling	函数	设置互斥锁属性优先级上限	
pthread.h	pthread_mutexattr_setprotocol	函数	设置互斥锁属性中的协议	
pthread.h	pthread_mutexattr_settype	函数	设置互斥锁属性中的类型	
pthread.h	pthread_once	函数	一次性操作任务	
pthread.h	pthread_self	函数	获取任务ID	
pthread.h	pthread_setcancelstate	函数	任务cancel功能开关	
pthread.h	pthread_setcanceltype	函数	设置任务cancel类型	
pthread.h	pthread_setschedparam	函数	设置任务优先级及调度策略	标准：调度策略支持SCHED_OTHERSCHED_FIFOSCHED_RRHuawei LiteOS：仅支持SCHED_RR调度策略
pthread.h	pthread_testcancel	函数	取消任务	

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
sys/socket.h	recv	函数	经socket接收数据	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_recv函数,与lwip_recv函数实现一致
sys/socket.h	recvfrom	函数	经socket接收数据	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_recvfrom函数,与lwip_recvfrom函数实现一致
sched.h	sched_get_priority_max	函数	获取系统支持的最大的优先级值	
sched.h	sched_get_priority_min	函数	获取系统支持的最小的优先级值	
sched.h	sched_yield	函数	使当前线程放弃占有处理机	
semaphore.h	sem_destroy	函数	销毁无名信号量	
semaphore.h	sem_getvalue	函数	获取指定信号量的值	
semaphore.h	sem_init	函数	初始化无名信号量	
semaphore.h	sem_post	函数	释放一个指定的无名信号量	
semaphore.h	sem_timedwait	函数	申请一个超时等待的无名信号量	标准：超时时间为绝对时间，可以处理早已超时的情况Huawei LiteOS：超时等待时间为相对时间，不能处理早已超时的情况

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
semaphore.h	sem_trywait	函数	尝试申请一个无名信号量	
semaphore.h	sem_wait	函数	申请等待一个无名信号量	
sys/socket.h	send	函数	经socket传送数据	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_send函数,与lwip_send函数实现一致
sys/socket.h	sendto	函数	经socket传送数据	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_sendto函数,与lwip_sendto函数实现一致
sys/socket.h	setsockopt	函数	设置socket状态	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_setscokopt函数,与lwip_setsockopt函数实现一致
sys/socket.h	socket	函数	建立一个socket通信	函数具体描述请参考《Huawei LiteOS LwIP API Reference》pdf文档中的lwip_socket函数,与lwip_socket函数实现一致

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	备注
time.h	timer_create	函数	创建一个新的Timer，并且指定定时器到时通知机制	
time.h	timer_delete	函数	删除一个Timer	标准：一次性软件定时器运行后，不会自删除。Huawei LiteOS：一次性软件定时器，运行完毕后，系统会自动删除该软件定时器，通过该接口只能创建周期性定时器
time.h	timer_getoverrun	函数	获取丢失的定时通知个数	标准：通过这个接口来确定一个特定的定时器出现这种超限的次数 Huawei LiteOS：返回值为周期定时器已执行的周期个数
time.h	timer_gettime	函数	Get the time remaining on a POSIX.1b interval timer	
time.h	timer_settime	函数	开始或者停止某个定时器	
sys/utsname.h	uname	函数	获取当前内核名称和其他信息	Huawei LiteOS中为utsname.h
stdarg.h	va_arg	宏	返回可变的参数	
stdarg.h	va_copy	宏	复制初始化过的va_list到目的参数列表	
stdarg.h	va_end	宏	结束可变参数的获取	
stdarg.h	va_start	宏	初始化变量arg_ptr	

版权所有 © 华为技术有限公司

9.1.2 POSIX 不支持接口

Huawei LiteOS的POSIX接口中，有一些未支持，具体参见下表。

文件名	函数接口名	类型	说明	是否支持
dirent.h	fdopendir	函数	将文件描述符转换成目录结构指针	不支持
mqueue.h	mq_notify	函数	通知任务在消息队列中有可获取的消息	不支持
mqueue.h	mq_unlink	函数	移除消息队列	不支持
pthread.h	pthread_attr_destroy	函数	销毁任务属性	不支持
pthread.h	pthread_condattr_destroy	函数	释放条件变量属性	不支持
pthread.h	pthread_condattr_init	函数	条件变量属性初始化	不支持
pthread.h	pthread_getspecific	函数	获取任务私有数据的共享地址	不支持
pthread.h	pthread_key_create	函数	创建任务私有数据	不支持
pthread.h	pthread_key_delete	函数	释放任务私有数据	不支持
pthread.h	pthread_mutex_timedlock	函数	超时时限内锁住互斥锁	不支持
pthread.h	pthread_setspecific	函数	存储任务私有数据的地址	不支持
semaphore.h	sem_close	函数	关闭有名信号量	不支持
semaphore.h	sem_open	函数	打开有名信号量	不支持
semaphore.h	sem_unlink	函数	移除有名信号量	不支持
signal.h	kill	函数	传送信号给指定的进程	不支持
signal.h	pthread_kill	函数	传送信号给指定的线程	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
signal.h	pthread_sigmask	函数	屏蔽某个线程对某些信号的响应处理	不支持
signal.h	raise	函数	向当前进程发送信号	不支持
signal.h	sigaction	函数	查询或设置信号处理方式	不支持
signal.h	sigaddset	函数	增加一个信号至信号集	不支持
signal.h	sigdelset	函数	从信号集里删除一个信号	不支持
signal.h	sigemptyset	函数	初始化信号集	不支持
signal.h	sigfillset	函数	将所有信号加入至信号集	不支持
signal.h	sigismember	函数	测试某个信号是否已加入至信号集里	不支持
signal.h	signal	函数	设置信号处理方式	不支持
signal.h	sigpending	函数	查询被阻塞的信号	不支持
signal.h	sigprocmask	函数	查询或设置信号掩码	不支持
signal.h	sigqueue	函数	信号发送函数，支持信号带有参数	不支持
signal.h	sigsuspend	函数	挂起进程直到捕捉到信号	不支持
signal.h	sigtimedwait	函数	在最大阻塞时间内阻塞线程，直到信号集中的某个信号被递送为止	不支持
signal.h	sigwait	函数	阻塞线程，直到信号集中的某个信号被递送为止	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
signal.h	sigwaitinfo	函数	阻塞线程，直到信号集中的某个信号被递送为止	不支持
stdio.h	getdelim	函数	定制读文件的结束标志位	不支持
stdio.h	getline	函数	从输入流中读取一行字符	不支持
stdio.h	tempnam	函数	产生一个唯一的包含路径的文件名	不支持
stdio.h	tmpfile	函数	以二进制的形式创建一个临时文件	不支持
stdio.h	vsscanf	函数	格式化字符串输入	不支持
stdlib.h	abort	函数	直接从调用的地方跳出	不支持
stdlib.h	atexit	函数	设置程序正常结束前调用的函数	不支持
stdlib.h	div	函数	将两个整数相除，返回商和余数用法	不支持
stdlib.h	exit	函数	正常结束进程	不支持
stdlib.h	getenv	函数	取得环境变量内容	不支持
stdlib.h	ldiv	函数	将两个整数相除，返回商和余数用法	不支持
stdlib.h	mblen	函数	返回多字节字符的长度	不支持
stdlib.h	mbstowcs	函数	多字节序列转换为宽字符	不支持
stdlib.h	mbtowc	函数	多字节序列转换为宽字符	不支持
stdlib.h	system	函数	执行shell 命令	不支持
stdlib.h	wctomb	函数	检查多字节字符的编码	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
sys/wait.h	waitpid	函数	等待子进程中断或结束	不支持
syslog.h	closelog	函数	关闭一个到系统日志记录程序的连接	不支持
syslog.h	setlogmask	函数	设置syslog记录屏蔽字	不支持
syslog.h	syslog	函数	把日志消息发给系统程序syslogd去记录	不支持
time.h	nanosleep	函数	提供高精度（纳秒级）休眠时间	不支持
unistd.h	alarm	函数	设置信号传送闹钟	不支持
unistd.h	_exit	函数	结束进程执行	不支持
unistd.h	execve	函数	执行文件函数	不支持
unistd.h	fchown	函数	改变文件的所有者	不支持
unistd.h	fork	函数	创建一个子进程	不支持
unistd.h	gethostname	函数	获取主机名	不支持
unistd.h	isatty	函数	判断文件描述词是否是为终端机	不支持
unistd.h	nice	函数	改变进程优先顺序	不支持
unistd.h	pipe	函数	创建一个管道	不支持
unistd.h	readlink	函数	取得符号链接所指的文件	不支持
wchar.h	fgetws	函数	从文件流中读取宽字符串	不支持
wchar.h	fputws	函数	写入宽字符串到文件流	不支持
wchar.h	fwide	函数	修改流为面向字节/面向宽字符	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
wchar.h	fwprintf	函数	格式化输出宽字符数据至文件	不支持
wchar.h	fwscanf	函数	格式化宽字符串输入	不支持
wchar.h	getwchar	函数	由标准输入设备内读进一宽字符	不支持
wchar.h	mbrlen	函数	返回多字节字符的长度	不支持
wchar.h	mbsrtowcs	函数	多字节序列转换为宽字符	不支持
wchar.h	putwchar	函数	将指定的宽字符写到标准输出设备	不支持
wchar.h	swprintf	函数	格式化宽字符串复制	不支持
wchar.h	swscanf	函数	格式化宽字符串输入	不支持
wchar.h	vfwprintf	函数	格式化输出宽字符数据至文件	不支持
wchar.h	vfwscanf	函数	格式化宽字符串输入	不支持
wchar.h	vswprintf	函数	格式化宽字符串复制	不支持
wchar.h	vswscanf	函数	格式化宽字符串输入	不支持
wchar.h	vwprintf	函数	格式化输出宽字符数据	不支持
wchar.h	vwscanf	函数	格式化宽字符输入	不支持
wchar.h	wcscat	函数	连接两个宽字符串	不支持
wchar.h	wcschr	函数	查找宽字符串中第一个出现的指定宽字符	不支持
wchar.h	wcscpy	函数	拷贝宽字符串	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
wchar.h	wcscspn	函数	返回宽字符串中连续不含指定宽字符串内容的字符数	不支持
wchar.h	wcsncat	函数	连接两个宽字符串	不支持
wchar.h	wcsncmp	函数	宽字符串比较	不支持
wchar.h	wcsncpy	函数	拷贝宽字符串	不支持
wchar.h	wcspbrk	函数	返回两个宽字符串中首个相同宽字符的位置	不支持
wchar.h	wcsrchr	函数	查找宽字符串中最后出现的指定宽字符	不支持
wchar.h	wcsrtombs	函数	把数组中存储的编码转换为多字节字符	不支持
wchar.h	wcsspn	函数	返回宽字符串中连续不含指定宽字符串内容的字符数	不支持
wchar.h	wcsstr	函数	在一宽字符串中查找指定的字符串	不支持
wchar.h	wcstod	函数	将宽字符串转换为一个双精度浮点型	不支持
wchar.h	wcstof	函数	将宽字符串转换为一个单精度浮点型	不支持
wchar.h	wcstok	函数	分割宽字符串	不支持
wchar.h	wcstol	函数	将宽字符串转换为一个长整数	不支持
wchar.h	wcstoul	函数	将宽字符串转换为一个无符号长整数	不支持
wchar.h	wprintf	函数	格式化输出宽字符数据	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
wchar.h	wscanf	函数	格式化宽字符串输入	不支持
wctype.h	towctrans	函数	宽字符转换函数	不支持
wctype.h	wctrans	函数	根据命名返回宽字符映射表	不支持

9.2 Libc/Libm 接口

9.2.1 Libc 适配接口

Huawei LiteOS还提供了Libc的适配接口，具体如下表所示。

头文件	名称	类型	说明
local.h	__vfprintf	文件操作函数	格式化输出数据至文件
thread_private.h	_thread_atexit_lock	锁操作函数	互斥锁加锁操作
thread_private.h	_thread_atexit_unlock	锁操作函数	互斥锁解锁操作
stdlib.h	arc4random_uniform	随机数函数	产生0~(x-1)范围内的随机数
time.h	asctime_r	时间函数	将时间和日期以字符串格式表示（入参指向的tm结构体在使用前要检查合理性，确保星期范围[0,6]和月份范围[0,11]）
string.h	bzero	字符串处理函数	将内存区域前n个字节赋零
stdlib.h	calloc	内存配置函数	配置内存空间
checksum.h	csum_fold	数据校验函数	将32位的累加和转化成16位的校验和
time.h	ctime	时间函数	将时间和日期以字符串形式表示
stdio.h	fgets	标准I/O函数	由文件中读取一字符串
stdio.h	fopen64	标准I/O函数	打开文件

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
stdlib.h	free	内存配置函数	释放原先配置的内存
stdio.h	freopen	标准I/O函数	重定向一个文件流。此接口内部调用的dup2()函数部分文件系统不支持（ramfs）,因此该接口返回值中的fd参数可能不同于标准；
stdio.h	fseeko64	标准I/O函数	移动文件流的读写位置
stdio.h	ftello64	标准I/O函数	取得文件流的读取位置
stdio.h	getc_unlocked	标准I/O函数	无锁stdio getc操作
stdio.h	getchar	标准I/O函数	由标准输入设备内读进一个字符
stdio.h	getchar_unlocked	标准I/O函数	无锁stdio getchar操作
stdio.h	gets	标准I/O函数	由标准输入设备内读进一字符串（建议使用fgets）
time.h	localtime	时间函数	取得当地目前的时间和日期
stdlib.h	malloc	内存配置函数	配置内存空间
stdlib.h	memalign	内存处理函数	申请内存（入参对齐的字节数，必须是2的正整数次幂）
libcmini.h	memchr	内存处理函数	在某一内存范围中查找一特定字符
libcmini.h	memcpy	内存处理函数	内存拷贝
libcmini.h	memmove	内存处理函数	从src拷贝count个字节到dest
libcmini.h	memset	内存处理函数	在一段内存块中填充某个给定的值
stdlib.h	nocache_calloc	内存处理函数	配置内存空间（申请的空间大小为nitems*size）

版权所有 © 华为技术有限公司

头文件	名称	类型	说明
stdlib.h	nocache_free	内存处理函数	释放内存
stdlib.h	nocache_malloc	内存处理函数	配置内存空间
stdlib.h	nocache_realloc	内存处理函数	配置内存空间
stdlib.h	nocache_zalloc	内存处理函数	配置内存空间
stdio.h	perror	错误处理函数	打印出错误信息
arc4random.h	pthread_atfork	线程处理函数	注册fork（）函数处理函数
stdlib.h	realloc	内存配置函数	更改已配置的内存空间
locale.h	setLocaleInit	地域函数	初始化地域化信息
string.h	strcasestr	字符串处理函数	判断字符串str2是否是str1的子串，忽略大小写
string.h	strcat	字符串处理函数	连接两字符串
string.h	strcoll	字符串处理函数	采用目前区域的字符排列次序来比较字符串
string.h	strerror	错误处理函数	返回错误原因的描述字符串
stdio.h	strerror_r	格式化输入输出函数	返回字符串描述错误码
string.h	strncat	字符串处理函数	连接两个字符串
string.h	strxfrm	字符串处理函数	转换字符串
ctype.h	tolower	数据转换函数	将大写字母转换成小写字母
ctype.h	toupper	数据转换函数	将小写字母转换成大写字母
time.h	tzset	时间函数	初始化时间转换信息
stdlib.h	zalloc	内存处理函数	配置内存空间

9.2.2 Libc 开源接口

Huawei LiteOS还提供了Libc的开源接口，具体如下表所示。

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
libcmini.h	__fpclassifyd	浮点数计算函数	归类浮点类型	bionic 5.0
floatio.h	__hdtoa	浮点数计算函数	将IEEE的双精度数值转化成十六进制数值字符串	bionic 5.0
floatio.h	__hldtoa	浮点数计算函数	将IEEE的浮点数值转化成十六进制数值字符串	bionic 5.0
libcmini.h	__isnan	浮点数计算函数	判断浮点数是否为不可表示的值	bionic 5.0
floatio.h	__ldtoa	浮点数计算函数	gdtoa()函数的封装，使得其功能与dtoa（）函数相似	bionic 5.0
local.h	__sclose	文件操作函数	关闭文件	bionic 5.0
local.h	__sflags	文件操作函数	返回操作模式的标志	bionic 5.0
local.h	__sflush	文件操作函数	清除读写缓冲区	bionic 5.0
local.h	__sflush_locked	文件操作函数	添加锁保护的清除读写缓冲区	bionic 5.0
local.h	__sfp	文件操作函数	寻找空的FILE	bionic 5.0
fvwrite.h	__sfvwrite	内存处理函数	写内存函数	bionic 5.0
local.h	__sinit	文件操作函数	初始化stdio中的内部变量	bionic 5.0
local.h	__smakebuf	文件操作函数	分配一个文件缓存	bionic 5.0
local.h	__sread	文件操作函数	读取文件	bionic 5.0
local.h	__srefill	文件操作函数	重新填充一个stdio缓存	bionic 5.0
local.h	__sseek	文件操作函数	重置文件中指针位置	bionic 5.0
local.h	__swhatbuf	文件操作函数	确定一个文件中适当的缓存	bionic 5.0
local.h	__swrite	文件操作函数	写文件	bionic 5.0
local.h	__swsetup	文件操作函数	检查文件是否可以安全写入	bionic 5.0
local.h	_cleanup	文件操作函数	清理文件	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
local.h	_fwalk	文件操作函数	对于系统中的每一个打开的文件，给定函数被调用，传递文件指针作为其唯一参数	bionic 5.0
stdlib.h	abs	数学计算函数	求整形绝对值	nuttx 7.8
stdlib.h	arc4random	随机数函数	产生随机数	bionic 5.0
time.h	asctime	时间函数	将时间和日期以字符串形式表示	bionic 5.0
time64.h	asctime64	时间函数	将时间和日期以字符串格式表示	bionic 5.0
time64.h	asctime64_r	时间函数	将时间和日期以字符串格式表示	bionic 5.0
assert.h	assert	诊断宏函数	在C程序中添加诊断	nuttx 7.8
stdlib.h	atoi	数据转换函数	将字符串转换成整型数	bionic 5.0
stdlib.h	atol	数据转换函数	将字符串转换成长整型数	bionic 5.0
stdlib.h	atoll	数据转换函数	将字符串转换成长长整型数	bionic 5.0
stdlib.h	bsearch	数据结构函数	二元搜索	bionic 5.0
wchar.h	btowc	宽字符处理函数	把单个字节转换为宽字符	bionic 5.0
stdio.h	clearerr	标准I/O函数	清除文件流的错误旗标	bionic 5.0
fcntl.h	creat	文件操作函数	创建一个文件	bionic 5.0
time64.h	ctime64	时间函数	将时间和日期以字符串格式表示	bionic 5.0
time64.h	ctime64_r	时间函数	将时间和日期以字符串格式表示	bionic 5.0
stdio.h	fclose	标准I/O函数	关闭文件	bionic 5.0
stdio.h	fdopen	标准I/O函数	取一个现存的文件描述符，并使一个标准的I / O流与该描述符相结合	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
stdio.h	feof	标准I/O函数	检查文件流是否读到了文件尾	bionic 5.0
stdio.h	ferror	错误处理函数	检查文件流是否有错误	bionic 5.0
stdio.h	fflush	标准I/O函数	更新缓冲区	bionic 5.0
stdio.h	fgetc	标准I/O函数	从文件中读取一个字符	bionic 5.0
stdio.h	fgetpos	标准I/O函数	取得文件流的读取位置	bionic 5.0
wchar.h	fgetwc	宽字符处理函数	把单个字节转换为宽字符	bionic 5.0
stdio.h	fileno	标准I/O函数	取得参数stream指定的文件流所使用的文件描述符	bionic 5.0
libcmini.h	finite	浮点数计算函数	判断浮点数是否为有限的	bionic 5.0
stdio.h	flockfile	标准I/O函数	锁文件	bionic 5.0
stdio.h	fopen	标准I/O函数	打开文件	bionic 5.0
stdio.h	fprintf	格式化输入输出函数	格式化输出数据至文件	bionic 5.0
stdio.h	fputc	标准I/O函数	将一指定字符写入文件流中	bionic 5.0
stdio.h	fputs	标准I/O函数	将一指定字符串写入文件流中	bionic 5.0
wchar.h	fputwc	宽字符处理函数	写入一个宽字符到文件流	bionic 5.0
stdio.h	fread	标准I/O函数	从文件流读取数据	bionic 5.0
stdio.h	fscanf	格式化输入输出函数	读取格式化输入	bionic 5.0
stdio.h	fseek	标准I/O函数	移动文件流的读取位置	bionic 5.0
stdio.h	fseeko	标准I/O函数	移动文件流的读写位置	bionic 5.0
stdio.h	fsetpos	标准I/O函数	移动文件流的读取位置	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
stdio.h	ftell	标准I/O函数	取得文件流的读取位置	bionic 5.0
stdio.h	ftello	标准I/O函数	取得文件流的读取位置	bionic 5.0
stdio.h	ftrylockfile	标准I/O函数	stdio锁文件操作	bionic 5.0
stdio.h	funlockfile	标准I/O函数	stdio解锁文件操作	bionic 5.0
stdio.h	fwrite	标准I/O函数	将数据写入文件流	bionic 5.0
stdio.h	getc	标准I/O函数	由文件中读取一个字符	bionic 5.0
wchar.h	getwc	宽字符处理函数	有文件中读取一个宽字符	bionic 5.0
time.h	gmtime	时间函数	取得目前的时间和日期	nuttx 7.8
time.h	gmtime_r	时间函数	取得目前时间和日期	nuttx 7.8
time64.h	gmtime64	时间函数	取得目前时间和日期	bionic 5.0
time64.h	gmtime64_r	时间函数	取得目前时间和日期	bionic 5.0
ctype.h	isalnum	字符测试函数	测试字符是否为英文字母或数字	bionic 5.0
ctype.h	isalpha	字符测试函数	测试字符是否为英文字母	bionic 5.0
ctype.h	isascii	字符测试宏	判断字符是否是ASCII码	bionic 5.0
ctype.h	isblank	字符测试函数	测试字符是否为空格或者TAB	bionic 5.0
ctype.h	iscntrl	字符测试函数	测试字符是否为控制字符	bionic 5.0
ctype.h	isdigit	字符测试函数	测试字符是否为十进制数字	bionic 5.0
ctype.h	Ise	字符测试宏	判断字符是否是e	bionic 5.0
ctype.h	isgraph	字符测试函数	测试字符是否有图形表示法	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
ctype.h	islower	字符测试函数	测试字符是否为小写英文字母	bionic 5.0
libcmini.h	isnan	浮点数计算函数	判断浮点数是否为不可表示的值	bionic 5.0
ctype.h	isprint	字符测试函数	测试字符是否为可打印字符	bionic 5.0
ctype.h	ispunct	字符测试函数	测试字符是否为标点符号字符	bionic 5.0
ctype.h	Issign	字符测试宏	判断字符是否是加减号	bionic 5.0
ctype.h	isspace	字符测试函数	测试字符是否为空白字符	bionic 5.0
ctype.h	isupper	字符测试函数	测试字符是否为大写英文字母	bionic 5.0
wctype.h	iswalnum	宽字符处理函数	判断宽字符是否字母或数字	bionic 5.0
wctype.h	iswalpha	宽字符处理函数	判断宽字符是否为英文字母	bionic 5.0
wctype.h	iswblank	宽字符处理函数	判断宽字符是否为TAB或者空格	bionic 5.0
wctype.h	iswcntrl	宽字符处理函数	判断宽字符是否为控制字符	bionic 5.0
wctype.h	iswctype	宽字符处理函数	判断是否为宽字符类型	bionic 5.0
wctype.h	iswdigit	宽字符处理函数	判断宽字符是否为阿拉伯字符0到9	bionic 5.0
wctype.h	iswgraph	宽字符处理函数	判断宽字符是否为除空格外的可打印字符	bionic 5.0
wctype.h	iswlower	宽字符处理函数	判断宽字符是否为小写英文字母	bionic 5.0
wctype.h	iswprint	宽字符处理函数	判断宽字符是否为可打印字符	bionic 5.0
wctype.h	iswpunct	宽字符处理函数	判断宽字符是否为标点符号或特殊符号	bionic 5.0
wctype.h	iswspace	宽字符处理函数	判断宽字符是否为空格字符	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
wctype.h	iswupper	宽字符处理函数	判断宽字符是否为大写英文字母	bionic 5.0
wctype.h	iswxdigit	宽字符处理函数	判断宽字符是否为16进制数字	bionic 5.0
ctype.h	isxdigit	字符测试函数	测试字符是否为十六进制数字	bionic 5.0
stdlib.h	labs	数学计算函数	求长整形绝对值	nuttx 7.8
stdlib.h	llabs	数学计算函数	求长整形绝对值	nuttx 7.8
time64.h	localtime64	时间函数	取得当地目前时间和日期	bionic 5.0
time64.h	localtime64_r	时间函数	取得当地目前时间和日期	bionic 5.0
wchar.h	mbrtowc	宽字符处理函数	多字节序列转换为宽字符	bionic 5.0
wchar.h	mbsinit	宽字符处理函数	测试初始的转换状态	bionic 5.0
bionic_mbstate.h	mbstate_bytes_so_far	其它函数	获取不为零流状态位	bionic 5.0
bionic_mbstate.h	mbstate_get_byte	其它函数	获取流状态	bionic 5.0
bionic_mbstate.h	mbstate_set_byte	其它函数	设置流状态	bionic 5.0
string.h	memchr	字符串处理函数	在某一内存范围内查找一特定字符	nuttx 7.8
string.h	memcmp	字符串处理函数	比较内存内容	bionic 5.0
string.h	memcpy	字符串处理函数	拷贝内存内容	bionic 5.0
string.h	memmove	字符串处理函数	拷贝内存内容	bionic 5.0
string.h	memset	字符串处理函数	将一段内存空间填入某值	bionic 5.0
stdlib.h	mkstemp	内存处理函数	建立唯一的临时文件	bionic 5.0
time.h	mktime	时间函数	将时间结构数据转换成经过的秒数	nuttx 7.8

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
time64.h	mktime64	时间函数	将时间结构数据转换成经过的秒数	bionic 5.0
stddef.h	offsetof	获取偏移	获取结构成员相对于结构体开头的字节偏移量	nuttx7.8
stdio.h	printf	格式化输入输出函数	格式化输出数据	bionic 5.0
stdio.h	putc	标准I/O函数	将一指定字符写入文件	bionic 5.0
stdio.h	putc_unlocked	标准I/O函数	无锁stdio putc操作	bionic 5.0
stdio.h	putchar	标准I/O函数	将指定的字符写到标准输出设备	bionic 5.0
stdio.h	putchar_unlocked	标准I/O函数	无锁stdio putchar操作	bionic 5.0
stdio.h	puts	标准I/O函数	将指定的字符串写到标准输出设备	bionic 5.0
wchar.h	putwc	宽字符处理函数	将一指定宽字符写入文件中	bionic 5.0
stdlib.h	qsort	数据结构函数	利用快速排序法排列数组	bionic 5.0
stdlib.h	rand	随机数函数	产生随机数	bionic 5.0
stdio.h	remove	文件及目录函数	删除文件	bionic 5.0
bionic_mbstate.h	reset_and_return	其它函数	流状态初始化	bionic 5.0
bionic_mbstate.h	reset_and_return_illegal	其它函数	流状态初始化并返回-1	bionic 5.0
stdio.h	rewind	标准I/O函数	重设文件流的读取位置为文件开头	bionic 5.0
stdio.h	scanf	格式化输入输出函数	格式化字符串输入	bionic 5.0
stdio.h	setbuf	标准I/O函数	设置文件流的缓冲区	bionic 5.0
locale.h	setlocale	地域函数	设置或读取地域化信息	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
stdio.h	setvbuf	标准I/O函数	设置文件流的缓冲区	bionic 5.0
stdio.h	snprintf	格式化输入输出函数	格式化字符串复制	bionic 5.0
string.h	snprintf	格式化输入输出函数	格式化字符串复制	bionic 5.0
stdio.h	sprintf	格式化输入输出函数	格式化字符串复制	bionic 5.0
stdlib.h	srand	随机数函数	设置随机数种子	bionic 5.0
stdlib.h	srandom	随机数函数	产生随机数种子	bionic 5.0
stdio.h	sscanf	格式化输入输出函数	格式化字符串输入	bionic 5.0
string.h	strcasecmp	字符串处理函数	字符串比较	bionic 5.0
string.h	strchr	字符串处理函数	查找字符串中第一个出现的指定字符	nuttx7.8
string.h	strcmp	字符串处理函数	比较字符串	bionic 5.0
string.h	strcpy	字符串处理函数	拷贝字符串	bionic 5.0
string.h	strcspn	字符串处理函数	返回字符串中连续不含指定字符串内容的字符数	bionic 5.0
string.h	strdup	字符串处理函数	复制字符串	bionic 5.0
time.h	strftime	时间函数	格式化日期和时间	nuttx7.8
string.h	strlcpy	字符串处理函数	拷贝字符串	bionic 5.0
string.h	strlen	字符串处理函数	返回字符串长度	bionic 5.0
libcmini.h	strlen	字符串处理函数	计算给定字符串的长度	bionic 5.0
string.h	strncasecmp	字符串处理函数	字符串比较	bionic 5.0
string.h	strncmp	字符串处理函数	比较字符串	bionic 5.0
string.h	strncpy	字符串处理函数	拷贝字符串	bionic 5.0
libcmini.h	strncpy	字符串处理函数	字符串拷贝	bionic 5.0
string.h	strpbrk	字符串处理函数	查找字符串中第一个出现的指定字符	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
string.h	strrchr	字符串处理函数	查找字符串中最后出现的指定字符	nuttx7.8
string.h	strsep	字符串处理函数	分解字符串为一组字符串	bionic 5.0
string.h	strspn	字符串处理函数	返回字符串中连续不含指定字符串内容的字符数	bionic 5.0
string.h	strstr	字符串处理函数	在一字符串中查找指定字符串	bionic 5.0
stdlib.h	strtod	数据转换函数	将字符串转换成浮点型数	bionic 5.0
string.h	strtok	字符串处理函数	分割字符串	nuttx7.8
string.h	strtok_r	字符串处理函数	分割字符串	nuttx7.8
stdlib.h	strtol	数据转换函数	将字符串转换成长整型数	bionic 5.0
stdlib.h	strtoul	数据转换函数	将字符串转换成无符号长整型数	bionic 5.0
time.h	time	时间函数	取得目前的时间	nuttx7.8
time64.h	timegm64	时间函数	将struct tm结构转成time_t结构	bionic 5.0
time64.h	timelocal64	时间函数	获取当前时间和日期并转换为本地时间	bionic 5.0
time.h	timer_create	时间函数	创建定时器	nuttx7.8
time.h	timer_delete	时间函数	删除定时器	nuttx7.8
time.h	timer_gettime	时间函数	获得一个定时器剩余时间	nuttx7.8
time.h	timer_settime	时间函数	初始化或者撤销定时器	nuttx7.8
time.h	times	时间函数	填充tms结构所指的缓冲区的计时信息	nuttx7.8
stdio.h	tmpfile	标准I/O函数	建立临时二进制文件	bionic 5.0
stdio.h	tmpnam	标准I/O函数	返回指向唯一文件名的指针	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
ctype.h	toascii	字符测试宏	将字符转换成对应的ASCII码	bionic 5.0
wctype.h	towlower	宽字符处理函数	将宽字符转换成小写字母	bionic 5.0
wctype.h	towupper	宽字符处理函数	将宽字符转换成大写字母	bionic 5.0
stdio.h	ungetc	标准I/O函数	将一指定字符写回文件流中	bionic 5.0
wchar.h	ungetwc	宽字符处理函数	将款字符写回指定的文件流	bionic 5.0
stdarg.h	va_arg	获取参数	检索参数列表中下一个参数	bionic 5.0
stdarg.h	va_copy	获取参数宏	复制宏	bionic 5.0
stdarg.h	va_end	获取参数	获取va_start返回	bionic 5.0
stdarg.h	va_start	获取参数	初始化变量	bionic 5.0
ctype.h	Val	字符测试宏	返回字符	bionic 5.0
stdio.h	vfscanf	格式化输入输出函数	从文件流读取字符串，根据参数format转化并格式化数据	bionic 5.0
stdio.h	vfprintf	格式化输入输出函数	格式化输出数据至文件	bionic 5.0
stdio.h	vprintf	格式化输入输出函数	格式化输出	bionic 5.0
stdio.h	vscanf	格式化输入输出函数	格式化字符串输入	bionic 5.0
stdio.h	vsnprintf	格式化输入输出函数	格式化字符串复制	bionic 5.0
stdio.h	vsprintf	格式化输入输出函数	格式化字符串复制	bionic 5.0
wchar.h	wcrtomb	宽字符处理函数	检查多字节字符的编码	bionic 5.0
wchar.h	wcscmp	宽字符处理函数	比较宽字符串	bionic 5.0
wchar.h	wcscoll	宽字符处理函数	采用目前区域的字符排列次序来比较宽字符串	bionic 5.0
wchar.h	wcsftime	宽字符处理函数	格式化时间	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
wchar.h	wcslcpy	宽字符处理函数	拷贝宽字符串	bionic 5.0
wchar.h	wcslen	宽字符处理函数	返回宽字符串长度	bionic 5.0
wctype.h	wcslen	宽字符处理函数	返回宽字符串长度	bionic 5.0
stdlib.h	wcstombs	转换函数	将数组存储的编码转换为多字节字符	bionic 5.0
wctype.h	wcstombs	宽字符处理函数	把数组中存储的编码转换为多字节字符	bionic 5.0
wchar.h	wcsxfrm	宽字符处理函数	根据程序当前的区域选项中的LC_COLLATE来转换宽字符串的前n个字符	bionic 5.0
wchar.h	wctob	宽字符处理函数	把宽字符转换为单字节字符	bionic 5.0
wctype.h	wctype	宽字符处理函数	判断是否为宽字符类型	bionic 5.0
wchar.h	wmemchr	宽字符处理函数	在一个宽字符数组中搜索	bionic 5.0
wctype.h	wmemchr	宽字符处理函数	在一个宽字符数组中搜索	bionic 5.0
wchar.h	wmemcmp	宽字符处理函数	比较两个宽字符数组	bionic 5.0
wctype.h	wmemcmp	宽字符处理函数	比较两个宽字符数组	bionic 5.0
wchar.h	wmemcpy	宽字符处理函数	拷贝宽字符数组	bionic 5.0
wctype.h	wmemcpy	宽字符处理函数	拷贝宽字符数组	bionic 5.0
wchar.h	wmemmove	宽字符处理函数	拷贝宽字符数组	bionic 5.0
wctype.h	wmemmove	宽字符处理函数	拷贝宽字符数组	bionic 5.0
wchar.h	wmemset	宽字符处理函数	填充一个宽字符数组	bionic 5.0
wctype.h	wmemset	宽字符处理函数	填充一个宽字符数组	bionic 5.0

版权所有 © 华为技术有限公司

9.2.3 Libm 开源接口

Huawei LiteOS提供一套Libm开源接口，具体的规格参见下表。

注意

Libm不支持设置错误返回码。

头文件	名称	类型	说明	开源出处
float.h	__ieee754_exp	浮点数计算函数	指数计算（双精度类型）	bionic 5.0
float.h	__ieee754_expf	浮点数计算函数	指数计算（浮点类型）	bionic 5.0
float.h	__ieee754_log	浮点数计算函数	对数计算（双精度类型）	bionic 5.0
float.h	__ieee754_logf	浮点数计算函数	对数计算（浮点类型）	bionic 5.0
float.h	__ieee754_rem_pio2	浮点数计算函数	return the remainder of x rem pi/2 in y[0]+y[1]	bionic 5.0
float.h	__ieee754_rem_pio2f	浮点数计算函数	return the remainder of x rem pi/2 in y[0]+y[1]	bionic 5.0
float.h	__ieee754_sqrt	浮点数计算函数	平方根计算（双精度类型）	bionic 5.0
float.h	__ieee754_sqrtf	浮点数计算函数	平方根计算（浮点类型）	bionic 5.0
float.h	__kernel_cos	浮点数计算函数	余弦计算	bionic 5.0
float.h	__kernel_rem_pio2	浮点数计算函数	__kernel_rem_pio2 return the last three digits of N with y = x - N*pi/2	bionic 5.0
float.h	__kernel_sin	浮点数计算函数	正弦计算	bionic 5.0
float.h	__kernel_tan	浮点数计算函数	正切计算	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
float.h	__kernel_tandf	浮点数计算函数	正切计算（浮点类型）	bionic 5.0
math.h	acos	数学计算函数	取反余弦函数值	bionic 5.0
math.h	acosf	数学计算函数	求反余弦函数	bionic 5.0
math.h	acosh	数学计算函数	求反双曲余弦值	bionic 5.0
math.h	acoshf	数学计算函数	求反双曲余弦值	bionic 5.0
math.h	acoshl	数学计算函数	求反双曲余弦值	nuttx7.8
math.h	acosl	数学计算函数	求反余弦函数	nuttx7.8
math.h	asin	数学计算函数	取反正弦函数值	bionic 5.0
math.h	asinf	数学计算函数	求反正弦函数	bionic 5.0
math.h	asinh	数学计算函数	求反双曲正弦值	bionic 5.0
math.h	asinhf	数学计算函数	求反双曲正弦值	bionic 5.0
math.h	asinhl	数学计算函数	求反双曲正弦值	bionic 5.0
math.h	asinl	数学计算函数	求反正弦函数	bionic 5.0
math.h	atan	数学计算函数	取反正切函数值	bionic 5.0
math.h	atan2	数学计算函数	取得反正切函数值	bionic 5.0
math.h	atan2f	数学计算函数	求反正切的值(以弧度表示)	bionic 5.0
math.h	atan2l	数学计算函数	求反正切的值(以弧度表示)	bionic 5.0
math.h	atanf	数学计算函数	求反正切函数	bionic 5.0
math.h	atanh	数学计算函数	求反双曲线正切函数	bionic 5.0
math.h	atanhf	数学计算函数	求反双曲线正切函数	bionic 5.0
math.h	atanhl	数学计算函数	求反双曲线正切函数	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
math.h	atanl	数学计算函数	求反正切函数	bionic 5.0
math.h	cbrt	数学计算函数	求立方根函数	bionic 5.0
math.h	cbrtf	数学计算函数	求立方根函数	bionic 5.0
math.h	cbrtl	数学计算函数	求立方根函数	bionic 5.0
math.h	ceil	数学计算函数	计算最小整数值	bionic 5.0
math.h	ceilf	数学计算函数	返回不小于x的最小整数值	bionic 5.0
math.h	ceill	数学计算函数	返回不小于x的最小整数值	bionic 5.0
math.h	copysign	数学计算函数	返回采用y的符号后的x	bionic 5.0
math.h	copysignl	数学计算函数	返回采用y的符号后的x	bionic 5.0
math.h	cos	数学计算函数	取余弦函数值	bionic 5.0
math.h	cosf	数学计算函数	求余弦函数	bionic 5.0
math.h	cosh	数学计算函数	取双曲线余弦函数值	nuttx7.8
math.h	coshf	数学计算函数	求双曲线余弦函数	nuttx7.8
math.h	coshl	数学计算函数	求双曲线余弦函数	nuttx7.8
math.h	cosl	数学计算函数	求余弦函数	nuttx7.8
math.h	erf	数学计算函数	求误差函数	bionic 5.0
math.h	erfc	数学计算函数	求误差补函数	bionic 5.0
math.h	erfcf	数学计算函数	求误差补函数	bionic 5.0
math.h	erfcl	数学计算函数	求误差补函数	bionic 5.0
math.h	erff	数学计算函数	求误差函数	bionic 5.0
math.h	erfl	数学计算函数	求误差函数	bionic 5.0
math.h	exp	数学计算函数	计算指数	bionic 5.0
math.h	expf	数学计算函数	计算以e为底的x次方值	bionic 5.0
math.h	expl	数学计算函数	计算以e为底的x次方值	nuttx7.8

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
math.h	expm1f	数学计算函数	exp(x) - 1	bionic 5.0
math.h	fabs	数学计算函数	计算浮点型数的绝对值	bionic 5.0
math.h	fabsf	数学计算函数	求浮点数的绝对值	nuttx7.8
math.h	fabsl	数学计算函数	求浮点数的绝对值	nuttx7.8
math.h	floor	数学计算函数	计算最大整数值	bionic 5.0
math.h	floorf	数学计算函数	返回不大于x的最大整数值	bionic 5.0
math.h	floorl	数学计算函数	返回不大于x的最大整数值	bionic 5.0
float.h	FLT_DIG DBL_DIG LDBL_DIG	浮点相关常量	定义舍入后不会改变表示的十进制数字的最大值（基数10）	bionic 5.0
float.h	FLT_EPSILON DBL_EPSILON LDBL_EPSILON	浮点相关常量	定义可表示的最小有效数字	bionic 5.0
float.h	FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	浮点相关常量	定义FLT_RADIX基数中的位数	bionic 5.0
float.h	FLT_MAX 1E DBL_MAX 1E LDBL_MAX 1E	浮点相关常量	定义最大的有限浮点值	bionic 5.0
float.h	FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	浮点相关常量	定义基数为10时的指数的最大整数值	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
float.h	FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	浮点相关常量	定义基数为FLT_RADIX时的指数的最大整数值	bionic 5.0
float.h	FLT_MIN DBL_MIN LDBL_MIN	浮点相关常量	定义最小的浮点值	bionic 5.0
float.h	FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	浮点相关常量	定义基数为10时的指数的最小负整数值	bionic 5.0
float.h	FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	浮点相关常量	定义基数为FLT_RADIX时的指数的最小负整数值	bionic 5.0
float.h	FLT_RADIX	浮点相关常量	定义指数表示基数	bionic 5.0
math.h	fmod	数学计算函数	计算余数	bionic 5.0
math.h	fmodf	数学计算函数	对浮点数进行取模（求余）	bionic 5.0
math.h	fmodl	数学计算函数	对浮点数进行取模（求余）	bionic 5.0
math.h	frexp	数学计算函数	将浮点型数分为底数与指数	bionic 5.0
math.h	frexpf	数学计算函数	把浮点数分解成尾数和指数	bionic 5.0
math.h	frexpl	数学计算函数	把浮点数分解成尾数和指数	bionic 5.0
math.h	HUGE_VAL	变量	结果的幅度太大以致于无法表示	bionic 5.0
math.h	hypot	数学计算函数	返回直角三角形斜边长度	bionic 5.0
math.h	isnan	数学计算函数	判断浮点数是否为不可表示的值	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
math.h	ldexp	数学计算函数	计算2的次方值	nuttx7.8
math.h	ldexpf	数学计算函数	返回x乘以2的exp次幂	nuttx7.8
math.h	ldexpl	数学计算函数	返回x乘以2的exp次幂	nuttx7.8
math.h	llrint	数学计算函数	返回整数类型的舍入值	bionic 5.0
math.h	log	数学计算函数	计算以e为底的对数值	bionic 5.0
math.h	log10	数学计算函数	计算以10为底的对数值	bionic 5.0
math.h	log10f	数学计算函数	计算以10为底的x对数值	bionic 5.0
math.h	log10l	数学计算函数	计算以10为底的x对数值	bionic 5.0
float.h	log1p	浮点数计算函数	自然对数计算（双精度类型）	bionic 5.0
math.h	log1p	数学计算函数	log(1+x)	bionic 5.0
float.h	log1pf	浮点数计算函数	自然对数计算（浮点类型）	bionic 5.0
math.h	log1pf	数学计算函数	log(1+x)	bionic 5.0
math.h	log2	数学计算函数	计算以2为底的x对数值	bionic 5.0
math.h	log2f	数学计算函数	计算以2为底的x对数值	bionic 5.0
math.h	log2l	数学计算函数	计算以2为底的x对数值	bionic 5.0
math.h	logf	数学计算函数	计算以e为底的x对数值	bionic 5.0
math.h	logl	数学计算函数	计算以e为底的x对数值	bionic 5.0
math.h	modf	数学计算函数	将浮点型数分解成整数与小数，返回小数部分	bionic 5.0

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
math.h	modff	数学计算函数	将一个浮点值分为整数和小数部分。	bionic 5.0
math.h	modfl	数学计算函数	将一个浮点值分为整数和小数部分。	bionic 5.0
math.h	pow	数学计算函数	计算次方值	bionic 5.0
math.h	powf	数学计算函数	求 x 的 y 次幂（次方）	bionic 5.0
math.h	powl	数学计算函数	求 x 的 y 次幂（次方）	bionic 5.0
math.h	rint	数学计算函数	将浮点数舍入到最接近的整数	bionic 5.0
math.h	rintf	数学计算函数	将浮点数舍入到最接近的整数	bionic 5.0
math.h	rintl	数学计算函数	将浮点数舍入到最接近的整数	nuttx7.8
math.h	round	数学计算函数	返回x的四舍五入整数值	nuttx7.8
math.h	roundf	数学计算函数	返回x的四舍五入整数值	nuttx7.8
math.h	roundl	数学计算函数	返回x的四舍五入整数值	nuttx7.8
math.h	scalbn	数学计算函数	返回x乘以FLT_RADIX的n次幂	bionic 5.0
float.h	scalbnf	浮点数计算函数	返回x乘以FLT_RADIX的n次幂	bionic 5.0
math.h	scalbnf	数学计算函数	返回x乘以FLT_RADIX的n次幂	bionic 5.0
math.h	sin	数学计算函数	取正弦函数值	bionic 5.0
math.h	sinf	数学计算函数	求正弦函数	nuttx7.8
math.h	sinh	数学计算函数	取双曲线正弦函数值	nuttx7.8

版权所有 © 华为技术有限公司

头文件	名称	类型	说明	开源出处
math.h	sinhf	数学计算函数	求双曲线正弦函数	nuttx7.8
math.h	sinhl	数学计算函数	求双曲线正弦函数	nuttx7.8
math.h	sinl	数学计算函数	求正弦函数	nuttx7.8
math.h	sqrt	数学计算函数	计算平方根值	bionic 5.0
math.h	sqrtf	数学计算函数	求开方函数	bionic 5.0
math.h	sqrtl	数学计算函数	求开方函数	bionic 5.0
math.h	tan	数学计算函数	求正切函数	bionic 5.0
math.h	tanf	数学计算函数	求正切函数	bionic 5.0
math.h	tanh	数学计算函数	取双曲线正切函数值	bionic 5.0
math.h	tanhf	数学计算函数	求双曲线正切函数	bionic 5.0
math.h	tanhl	数学计算函数	求双曲线正切函数	bionic 5.0
math.h	tanl	数学计算函数	求正切函数	bionic 5.0
math.h	trunc	数学计算函数	截取日期或数字，返回指定的值	bionic 5.0
math.h	truncf	数学计算函数	截取日期或数字，返回指定的值	bionic 5.0
math.h	truncl	数学计算函数	截取日期或数字，返回指定的值	bionic 5.0

9.2.4 Libc/Libm 不支持接口

Huawei LiteOS的Libc/Libm接口中，有一些未支持，具体参见下表。

文件名	函数接口名	类型	说明	是否支持
locale.h	localeconv	地域函数	设置或读取地域化信息	不支持
bionic_time.h	localtime_tz	时间函数	取得当地目前时间和日期	不支持

版权所有 © 华为技术有限公司

文件名	函数接口名	类型	说明	是否支持
bionic_time.h	mktime_tz	时间函数	将时间结构数据转换成经过的秒数	不支持
bionic_time.h	strftime_tz	时间函数	格式化时间	不支持
checksum.h	csum_partial	数据校验函数	计算数据的验校和	不支持
statfs.h	fstatfs	文件操作函数	查看挂载文件系统信息	不支持
statfs.h	statfs64	文件操作函数	查询文件系统相关的信息	不支持
time.h	posix2time	时间函数	将posix time_t转换为local time_t	不支持
time.h	time2posix	时间函数	将local time_t转换为posix time_t	不支持
time.h	timegm	时间函数	将struct tm结构转成time_t结构	不支持
time.h	timelocal	时间函数	获取当前时间和日期并转换为本地时间	不支持
unistd.h	isatty	文件操作函数	检查设备类型， 判断文件描述词是否是为终端机	不支持
unistd.h	lseek64	文件操作函数	移动文件读/写指针	不支持

9.3 C++兼容规格

下述表格为C++标准库/STL兼容规格。

说明

不支持异常特性，其他特性由编译器支持。

- 支持语言支持功能

版权所有 © 华为技术有限公司

头文件	描述
<limits>	提供与基本数据类型相关的定义。例如，对于每个数值数据类型，它定义了可以表示出来的最大值和最小值以及二进制数字的位数
<new>	支持动态内存分配

- 支持工具函数

头文件	描述
<utility>	定义重载的关系运算符，简化关系运算符的写入，它还定义了pair类型，该类型是一种模板类型，可以存储一对值。这些功能在库的其他地方使用
<functional>	定义了许多函数对象类型和支持函数对象的功能，函数对象是支持operator函数调用运算符的任意对象
<memory>	给容器、管理内存的函数和auto_ptr模板类定义标准内存分配器。

- 支持字符串处理

头文件	描述
<string>	为字符串类型提供支持和定义，包括单字节字符串(由char组成)的string和多字节字符串(由wchar_t组成)

- 支持容器类模板

头文件	描述
<vector>	定义vector序列模板，这是一个大小可以重新设置的数组类型，比普通数组更安全、更灵活
<list>	定义list序列模板，这是一个序列的链表，常常在任意位置插入和删除元素
<deque>	定义deque序列模板，支持在开始和结尾的高效插入和删除操作
<queue>	为队列(先进先出)数据结构定义序列适配器queue和priority_queue

版权所有 © 华为技术有限公司

<stack>	为堆栈(后进先出)数据结构定义序列适配器stack
<map>	map是一个关联容器类型，允许根据唯一键值，按照升序存储。
<set>	set是一个关联容器类型，用于以升序方式存储唯一值。
<bitset>	为固定长度的位序列定义bitset模板，它可以看作固定长度的紧凑型bool数组

- 支持迭代器

头文件	描述
<iterator>	给迭代器提供定义和支持

- 支持算法

头文件	描述
<algorithm>	提供一组基于算法的函数，包括置换、排序、合并和搜索

- 支持数值操作

头文件	描述
<complex>	支持复杂数值的定义和操作
<valarray>	支持数值矢量的操作
<numeric>	在数值序列上定义一组一般数学操作，例如accumulate和inner_product

注意

HUAWEI_LITEOS提供的memory，uninitialized_fill函数存在内存泄露，慎用。

版权所有 © 华为技术有限公司

10 配置参考

版权所有 © 华为技术有限公司

10.1 配置工具使用说明

工具介绍

Menuconfig是基于menu的config，即菜单式的配置。所用的Kconfig语言是一种菜单配置语言，Config.in和Kconfig都是用该语言编写而成。

使用步骤

在Huawei_LiteOS目录下执行make menuconfig即可。

使用说明

menuconfig的使用方式，主要是：

上下键：选择不同的行，即移动到不同的（每一行的）选项上。

空格键：用于在选择该选项，取消选择该选项，之间来回切换。

1. 选择该（行所在的）选项：则对应的该选项前面就变成了“中括号里面一个星号，即 [*]，表示被选中了”。
2. 如果是取消该选项，就变成了“只有一个中括号，里面是空的，即：[ ]”

左右键：用于在Select/Exit/Help之间切换

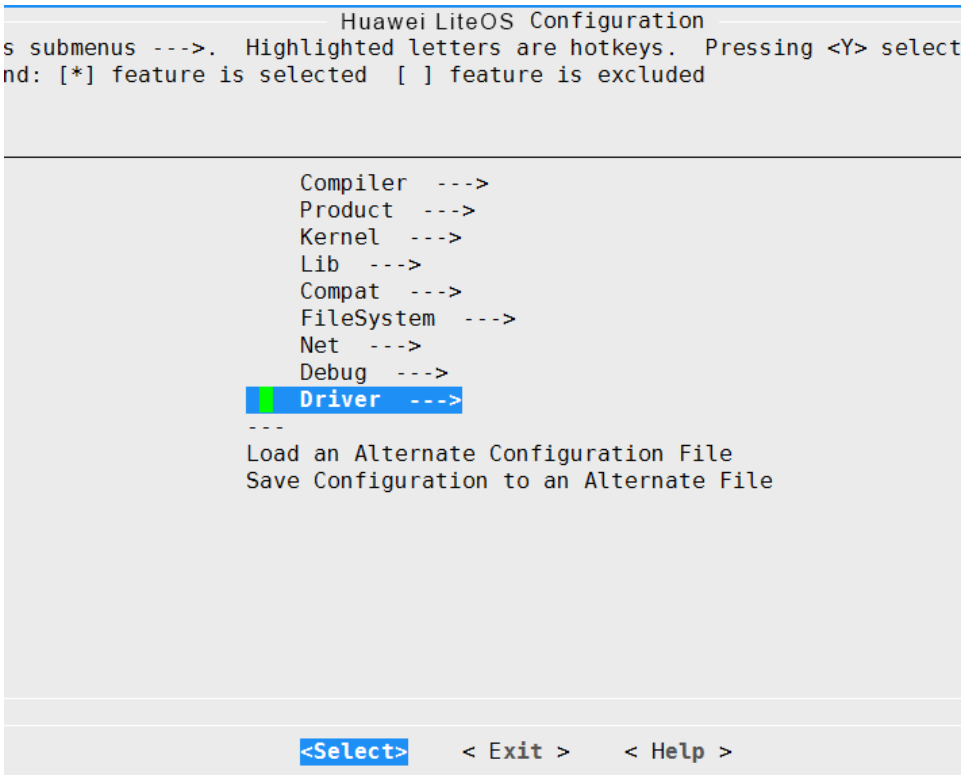
回车键：左右键切换到了某个键上，此时敲回车键，就执行相应的动作：

1. Select：此时一般都是所在（行的）选项，后面有三个短横线加上一个右箭头，即--->，表示此项下面还有子选项，即进入子菜单
2. Exit：直接退出当前的配置

   当你更改了一些配置，但是又没有去保存，此时一般都会询问你是否要保存当前（已修改后的最新的）配置，然后再退出。
3. Help：针对当前所在某个（行的）选项，查看其帮助信息。

menuconfig界面如下图所示。

版权所有 © 华为技术有限公司



注意事项

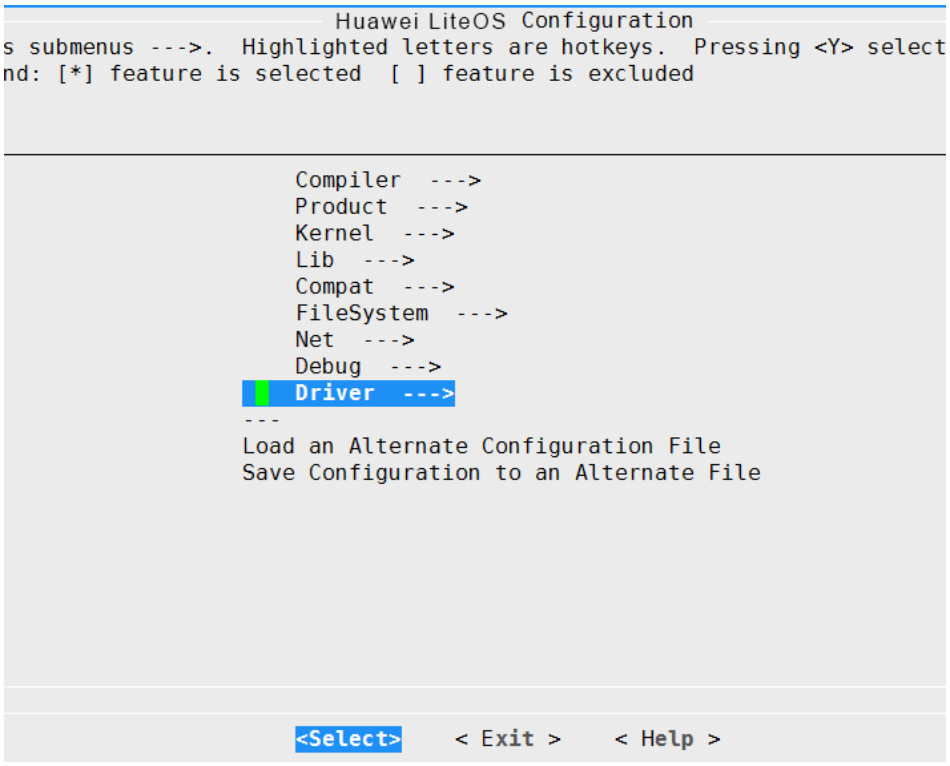
1、在使用本工具时请确保已经安装交叉编译器工具链，arm-liteos-linux-uclibcgnueabi-系列、arm-hisiv500-linux-uclibcgnueabi-系列、arm-hisiv300-linux-uclibcgnueabi-系列或者arm-hisiv600-linux-gnueabi-系列。

2、如果copy了一份Huawei_LiteOS的源码，make menuconfig 弹不出菜单，请删除tools/menuconfig/extra/config目录下面所有的二进制文件，再重新在顶层目录下执行make menuconfig。

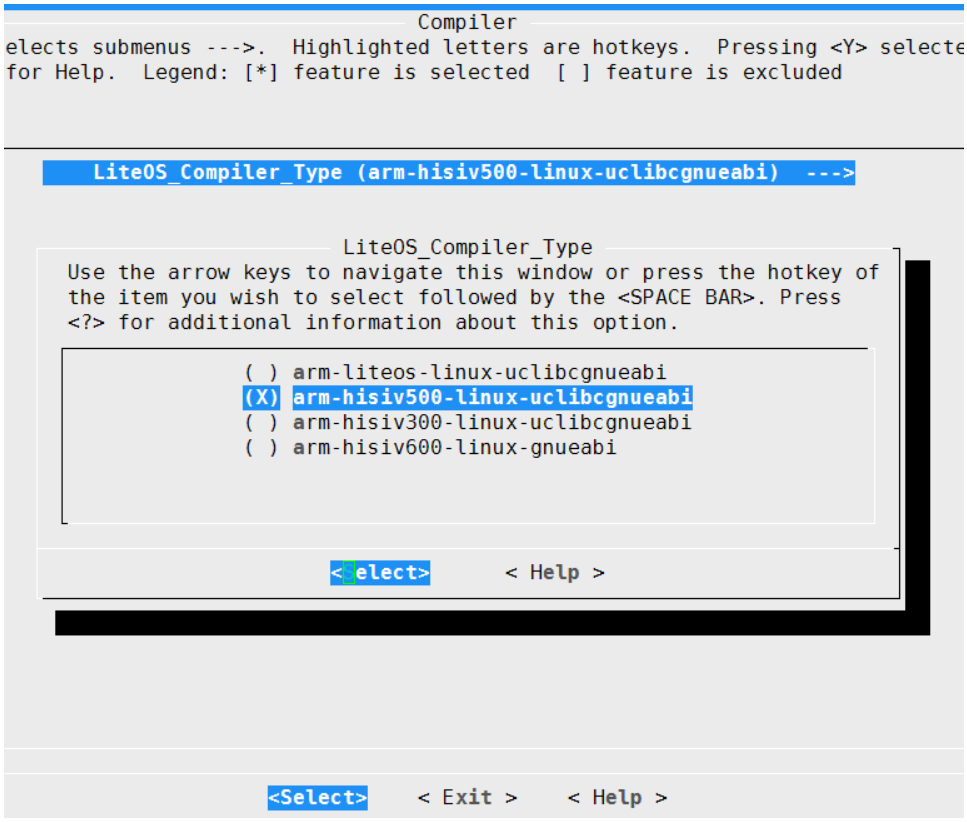
配置说明

1. 运行make menuconfig。进入Configuration主界面，目前有Compiler，Product，Kernel，Lib，Compat，FileSystem，Net，Debug和Driver这几个选项

版权所有 © 华为技术有限公司

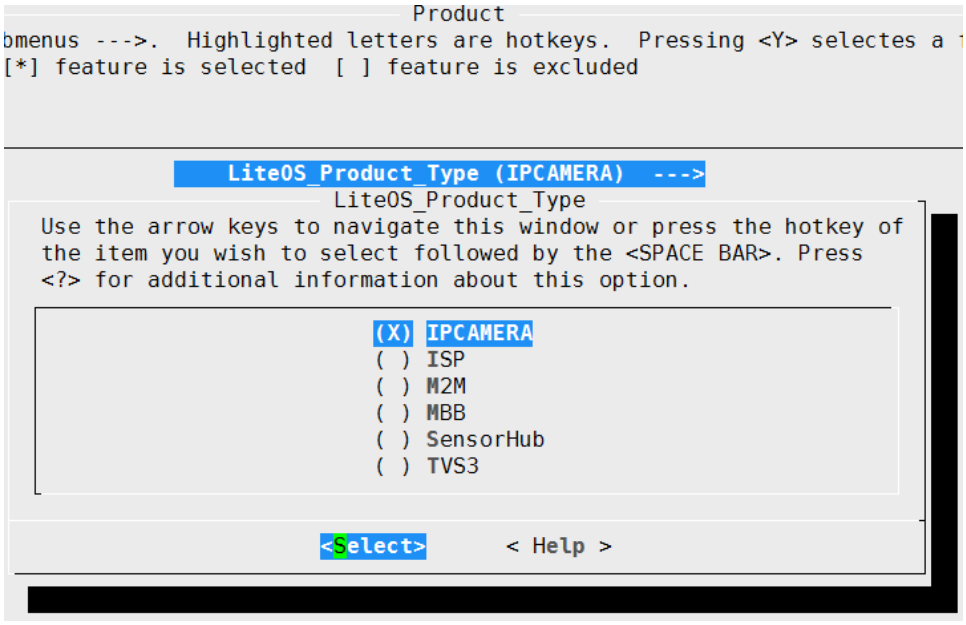


2. 选择Compiler选项。Compiler表示交叉编译器工具链类型，进入Compiler后，需要对LiteOS_Compiler_Typer进行选择，现有四种交叉编译器可选。默认选择arm-hisiv500-linux-uclibcgnueabi。



3. 选择Product选项。Product表示产品类型，进入Product后，需要对LiteOS_Product_Type进行选择，默认选择IPCAMERA，目前还支持TV系列。

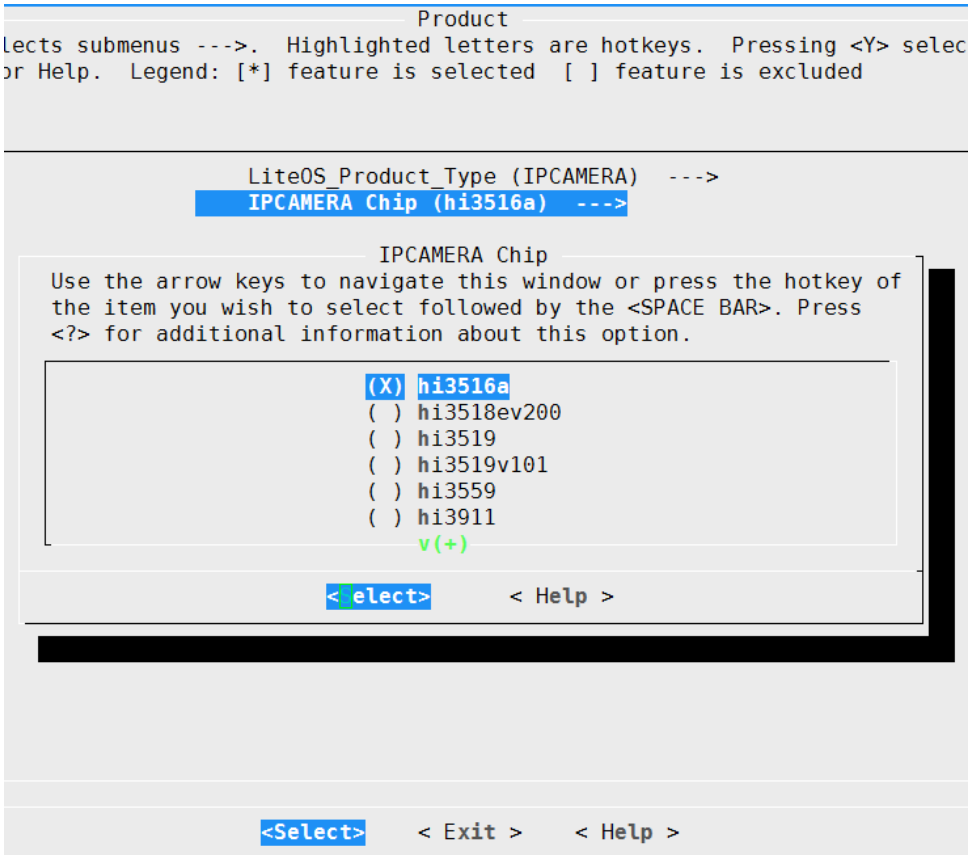
版权所有 © 华为技术有限公司



- 选择IPCAMERA之后，需要在IPCAMERA Chip选项再选择具体芯片，目前有hi3516a，hi3518ev200、hi3519、hi3519v101、hi3559、hi3911和him5v100。默认选择hi3516a。

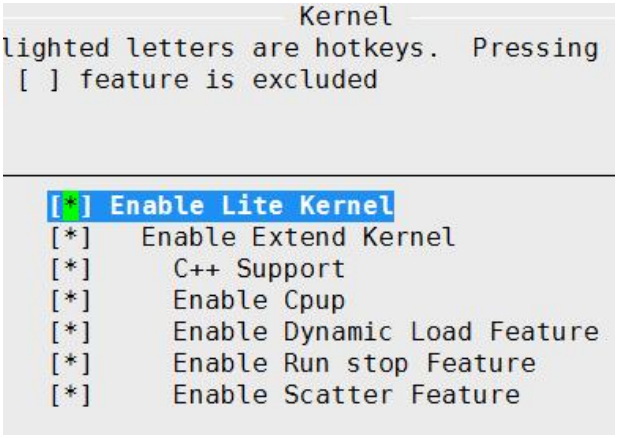
  说明

  图中选择v(+)可以展开更多选项，包括him5v100。

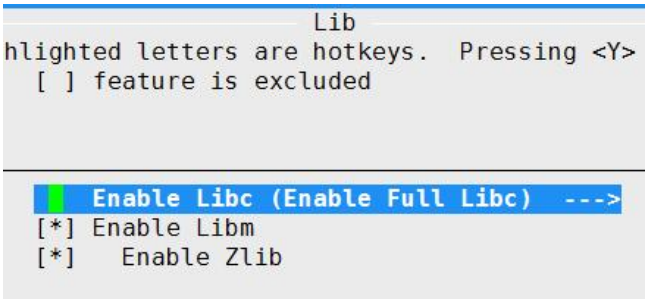


版权所有 © 华为技术有限公司

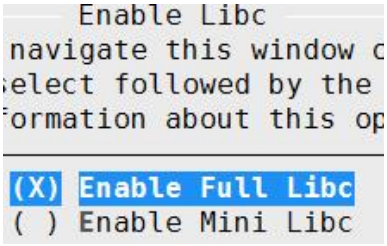
4. 选择Kernel选项。在该选项内，Lite Kernel为基础kernel，一般必选。扩展Kernel目前有五个选项（C++支持、CPU占用率、动态加载、Run-Stop（wifi唤醒）和分散加载），可根据需要选择。



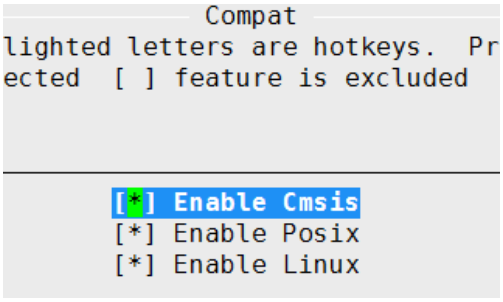
5. 选择Lib选项。在该选项内，有Libc，Libm和Zlib可供需要选择，Lib选项一般必选。Full Libc一般与全代码一起使用，Mini Libc一般与Litekernel一起使用。



在编译全部代码时需要选择Full libc ，在只编译内核时需要选择Mini libc。

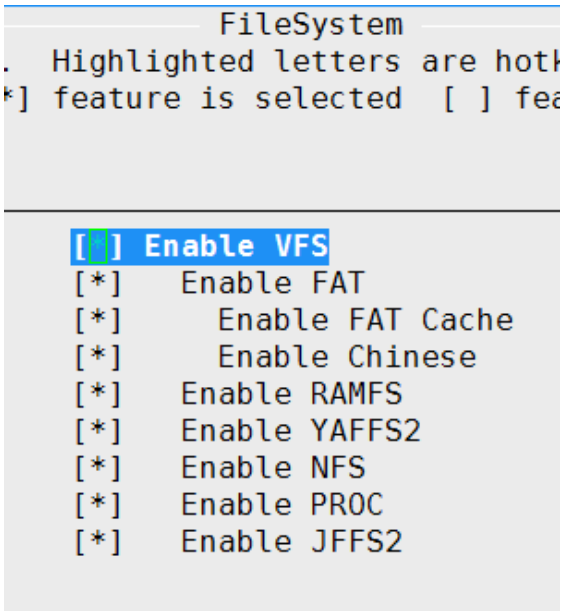


6. 选择Compat选项。在该选项内，有Cmsis，Posix和Linux可选择，其中Posix一般必选。

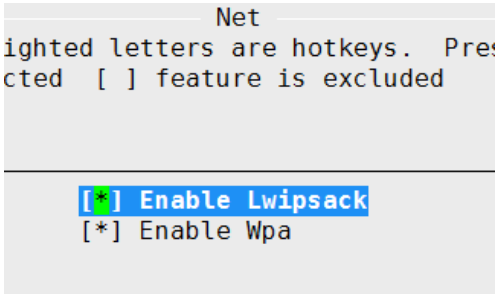


7. 选择FileSystem选项。在该选项内，有多种文件系统可供选择：FAT，RAMFS，NFS，PROC，YAFFS2和JFFS2，其中FAT中还有启用cache和支持中文功能。所有文件系统都必须选择VFS作为基础。ramfs，nfs需要开启debug版本后才可以使用。

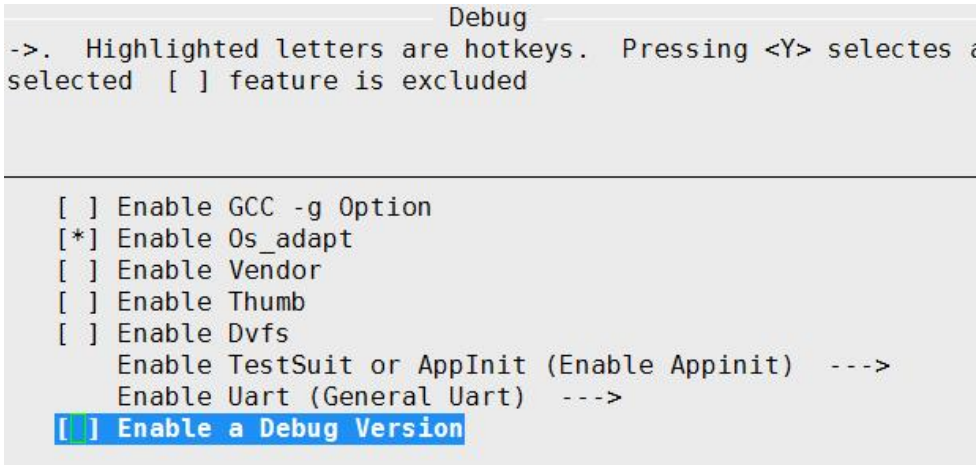
版权所有 © 华为技术有限公司



8. 选择Net选项。在该选项中，可供选择的是与网络相关的LWIP和与WIFI安全相关的WPA。WPA需要开启debug版本后才可以使用。



9. 选择Debug选项。该选项中，可选择是否使用-g编译选项，是否需要对客户代码进行必要的适配（Os_adapt和AppInit），是否需要链接客户库（Vendor），是否需要Test代码（TestSuit），是否需要thumb指令集（THUMB），是否需要低功耗（Dvfs），是否需要Uart（如果只选择Litekernel 可选Simple Uart，如果选择全部选项，可选General Uart）。最后一个选项为版本选项，是编译发布版本还是调测版本。如果是调测版本，则包含是否启用Shell功能，是否启用Telnet功能，，是否使用tftp工具，是否需要使用Iperf工具，以及是否启用内存检测功能（0为关闭，1为开启）。



版权所有 © 华为技术有限公司

```
[*] Enable a Debug Version
[*]   Enable Shell (NEW)
[*]     Enable Tftp (NEW)
[*]     Enable Telnet (NEW)
[*]     Enable Iperf-2.0.5 (NEW)
[*]   Enable Memory Check (NEW)
(0)     Enable integrity check or not (0,1) (NEW)
(1)     Enable size check or not (0.1) (NEW)
```

10. 选择Driver选项。该选项内包含多种硬件驱动，nand_flash芯片有2种可选，WIFI芯片可根据需要QRD或BCM。其他跟芯片强相关的可选驱动menuconfig会自动选择。高通的WiFi驱动i依赖于wpa。

    高通WiFi驱动需要在开启debug版本后使用，博通WiFi驱动可在release版本中使用。

```
                                Driver
submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> selectes
i: [*] feature is selected  [ ] feature is excluded


  ^(-)
  [*]   Enable Cellwise
  [*]   Enable GPIO
  [*]   Enable HIDMAC
  [*]   Enable Net Device
          MAC (Enable higmac for hi3516a and hi3519)  --->
  [*]   Enable I2C
  [*]   Enable LCD and LCD_PWM
  [*]   Enable MEM
  [*]   Enable MMC
          MMC Chips (Enable himmcv100)  --->
  [*]   Enable MTD
  [*]     Enable MTD nandflash
            NandFlash Chips (Enable hinfc620 for parallel nand)  --->
  [*]     Enable MTD spi_nor flash
            SpiNorFlash Chips (Enable hisfc350)  --->
  [*]   Enable random
  [*]   Enable RTC
  [*]   Enable SPI
  [*]   Enable USB
  v(+)

                  <Select>    < Exit >    < Help >
```

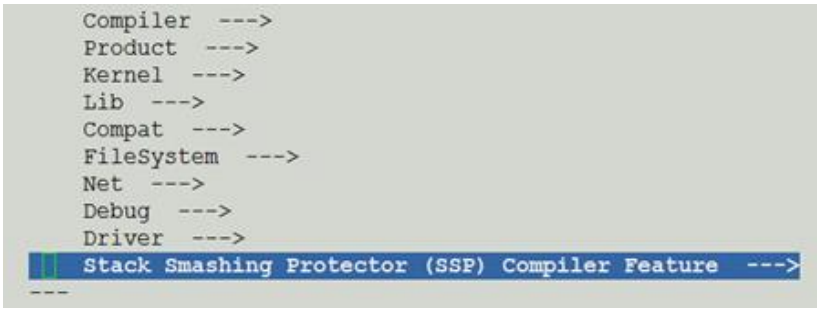
```
  [*]   Enable USB
  [*]   Enable Video
  [*]   Enable WIFI
          WIFI Chips (Enable Qualcomm WIFI)  --->
  [*]   Enable Watch Dog

                    <Select>    < Exit >    < Help >
```

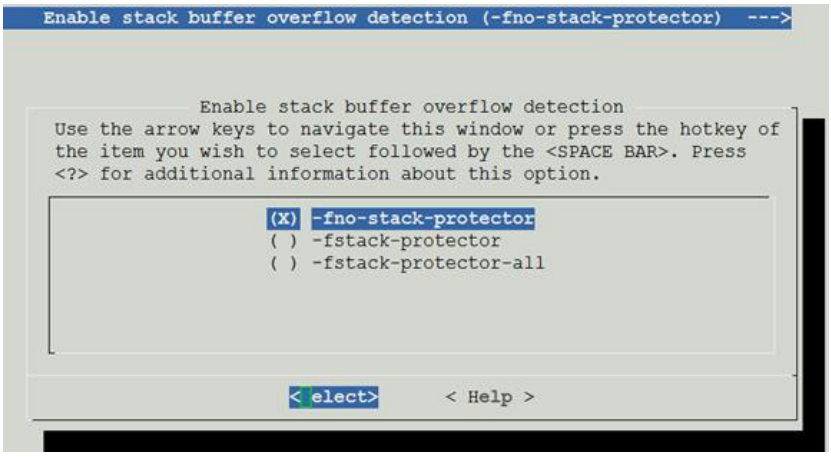
11. 选择Stack Smashing Protector (SSP) Compiler Feature选项，该选项可以打开和关闭堆栈保护功能。

    - -fno-stack-protector为关闭堆栈保护（默认）；
    - -fstack-protector为启用堆栈保护，但只为局部变量中含有字符（char）数组(数组大小大于等于4个字节的字符数组)的C函数插入保护代码；
    - -fstack-protector-all为所有C函数插入保护代码，相比上一个，可能会大幅度的增加性能上的开销；

版权所有 © 华为技术有限公司

推荐使用"-fstack-protector"选项，在兼顾性能的同时，增强了安全性。





10.2 时间管理配置参数

配置项说明

时间管理不是单独的功能模块，依赖于sys模块中的OS_SYS_CLOCK和Tick模块中的LOSCFG_BASE_CORE_TICK_PER_SECOND两个配置项。该配置项默认值基于hi3516a芯片配置。

配置项	含义	取值范围	默认值	依赖	可否动态刷新
OS_SYS_CLOCK	系统主频	(0, n)	50000000	无	否
LOSCFG_BASE_CORE_TICK_PER_SECOND	每秒Tick数	(0, n)	100	无	否

版权所有 © 华为技术有限公司

10.3 内存管理配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖	可否动态刷新
OS_SYS_MEM_ADDR	系统动态内存起始地址	[0, n)	&m_aucSysMem0[0]	无	否
OS_SYS_MEM_SIZE	内存的大小（DDR自适应配置）	[0, n)	从bss段末尾至系统DDR末尾	无	是

10.4 内存维测配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖	可否动态刷新
LOSCFG_BASE_MEM_NODE_INTEGRITY_CHECK	是否打开系统内存节点完整性检测	YES/NO	NO	无	否
LOSCFG_BASE_MEM_NODE_SIZE_CHECK	是否打开系统内存节点大小检测	YES/NO	YES	无	否

这两个配置项可以通过menuconfig，由用户输入来控制，无需修改头文件。

10.5 任务配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖
LOSCFG_BASE_CORE_TSK_LIMIT	任务最大数	[0,n)	64	无

版权所有 © 华为技术有限公司

配置项	含义	取值范围	默认值	依赖
LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE	IDLE任务栈大小	[0,n)	0x400	无
LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE	默认任务栈大小	[0,n)	0x6000	无
LOSCFG_BASE_CORE_TIMESLICE	任务分时开关	YES/NO	YES	无
LOSCFG_BASE_CORE_TIMESLICE_TIMEOUT	同优先级任务最长执行时间（单位：Tick）	[1,n)	2	无
LOSCFG_BASE_CORE_TSK_MONITOR	任务栈溢出检查和轨迹开关	YES/NO	YES	无
LOSCFG_BASE_CORE_CPUP	cpu使用率	YES/NO	YES	无

10.6 软件定时器配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖	可否动态刷新
LOSCFG_BASE_CORE_SWTMR	软件定时器裁剪开关	YES/NO	YES	LOSCFG_BASE_IPC_QUEUE	否
LOSCFG_BASE_CORE_SWTMR_LIMIT	最大支持软件定时器数	[0, n)	1024	LOSCFG_BASE_IPC_QUEUE	否
OS_SWTMR_HANDLE_QUEUE_SIZE	软件定时器处理队列大小	[0, n)	1024	LOSCFG_BASE_CORE_SWTMR_LIMIT	否

版权所有 © 华为技术有限公司

10.7 信号量配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖
LOSCFG_BASE_IPC_SEM	信号量模块裁剪开关	YES/NO	YES	无
LOSCFG_BASE_IPC_SEM_LIMIT	信号量最大数	[0,n)	1024	无

10.8 互斥锁配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖
LOSCFG_BASE_IPC_MUX	互斥锁模块裁剪开关	YES/NO	YES	无
LOSCFG_BASE_IPC_MUX_LIMIT	互斥锁最大数	[0,n)	1024	无

10.9 硬中断裁剪开关

配置项说明

配置项	含义	取值范围	默认值	依赖	可否动态刷新
LOSCFG_PLATFORM_HWI	硬中断裁剪开关	YES/NO	YES	无	否
LOSCFG_PLATFORM_HWI_LIMIT	硬中断使用最大数	根据芯片手册适配	根据芯片手册适配	无	否

版权所有 © 华为技术有限公司

10.10 队列配置参数

配置项说明

配置项	含义	取值范围	默认值	依赖	可否动态刷新
LOSCFG_BASE_IPC_QUEUE	队列模块静态裁剪开关	YES/NO	YES	无	否
LOSCFG_BASE_IPC_QUEUE_LIMIT	系统支持的最大队列数（含Huawei LiteOS定时器模块占用的一个队列）	[0,n)	1024	无	否

10.11 模块裁剪配置参数

用户可以根据自己不同的需求，对模块进行裁剪。

动态加载

裁剪开关：

LOSCFG_KERNEL_DYNLOAD

裁剪方法：

在项目根目录下.config，将LOSCFG_KERNEL_DYNLOAD赋值成n，即LOSCFG_KERNEL_DYNLOAD=n，或通过make menuconfig在kernel选项里取消选中动态加载特性。

依赖：无。

注意：无。

分散加载

裁剪开关：

LOSCFG_KERNEL_SCATTER

裁剪方法：

在项目根目录下的.config，将LOSCFG_KERNEL_SCATTER赋值成n，即LOSCFG_KERNEL_SCATTER=n，或通过make menuconfig在kernel选项里取消选中分散加载特性。

依赖：无。

注意：关掉LOSCFG_KERNEL_SCATTER会影响快速启动性能。

版权所有 © 华为技术有限公司

文件系统

JFFS2

裁剪开关：

LOSCFG_FS_JFFS

裁剪方法：

在项目根目录下的.config，将LOSCFG_FS_JFFS赋值成n，即LOSCFG_FS_JFFS=n，或通过make menuconfig在FileSystem选项里取消选中JFFS文件系统。

依赖：无。

注意：无。

FAT

裁剪开关：

LOSCFG_FS_FAT

裁剪方法：

在项目根目录下的.config，将LOSCFG_FS_FAT赋值成n，即LOSCFG_FS_FAT=n，或通过make menuconfig在FileSystem选项里取消选中FAT文件系统。

依赖：无。

注意：无。

YAFFS2

裁剪开关：

LOSCFG_FS_YAFFS

裁剪方法：

在项目根目录下的.config，将LOSCFG_FS_YAFFS赋值为n，即LOSCFG_FS_YAFFS=n，或通过make menuconfig在FileSystem选项里取消选中YAFFS文件系统。

依赖：无。

注意：无。

RAMFS

裁剪开关：

LOSCFG_FS_RAMFS

裁剪方法：

在项目根目录下的.config中，将LOSCFG_FS_RAMFS赋值为n，即LOSCFG_FS_RAMFS=n，或通过make menuconfig在FileSystem选项里取消选中RAMFS文件系统。

版权所有 © 华为技术有限公司

依赖：无。

注意：无。

PROCFS

裁剪开关：

LOSCFG_FS_PROC

裁剪方法：

在项目根目录下的.config中，将LOSCFG_FS_PROC赋值为n，即LOSCFG_FS_PROC=n，或通过make menuconfig在FileSystem选项里取消选中PROC文件系统。

依赖：无。

注意：无。

10.12 动态加载配置参数

符号配置

在Huawei_LiteOS/tools/scripts/dynload_tools目录下的symbol.list文件中存有用户希望OS提供的符号（函数）列表，如下截取的symbol.list片段：

```
458 sin
459 sinf
460 sinh
461 sinhf
462 sqrt
463 sqrtf
464 sqrtl
465 tan
466 tanf
467 tanh
468 tanhf
469 tanhl
```

“sin”被包含在该文件中，表示用户希望OS能提供sin符号（譬如用户模块需要调用sin函数，前提是OS中有该符号的定义）。用户可以利用make_symlst.py脚本生成自己的模块所需要的符号。

make_symlst.py接受多个参数，可以是包含用户模块的目录路径，脚本会递归搜索该目录下所有的.o及.so文件，并将这些模块所依赖的系统符号提取到symbol.list文件中；也可以是用户模块的路径，脚本会提取此模块所依赖的系统符号到symbol.list文件。脚本的参数可以是模块目录路径和模块路径的组合。

在Linux Shell下敲入如下命令使用：

```
./make_symlst.py modules1_dir module1_path module2_path modules2_dir
```


版权所有 © 华为技术有限公司

表 10-1 参数说明

参数	参数说明
modules1_dir	包含模块的目录路径（脚本会递归搜索该目录下的所有模块）
module1_path	指定一个模块路径
modules2_dir	包含模块的目录路径（脚本会递归搜索该目录下的所有模块）
module2_path	指定一个模块路径

注意

原则上，symbol.list最好是可以支持用户模块运行的最小符号集。symbol.list中的符号会在动态加载模块初始化时分配内存，应尽量减少不必要的符号从而减少内存的使用和加快动态加载模块的初始化。

得到symbol.list后，调用make_symbol.sh脚本得到符号表源文件。

make_symbol.sh接受三个参数，第一个参数是系统镜像的路径，第二个参数是上一步生成的symbol.list的文件路径，第三个参数是用户希望导出的符号表源文件。

在Linux Shell下敲入如下命令使用：

```
./make_symbol.sh build/vs_server tools/scripts/dynload_tools/symbol.list kernel/dynload/src/
elf_symbol.c
```

表 10-2 参数说明

参数	参数说明
build/vs_server	系统镜像文件
tools/scripts/dynload_tools/symbol.list	symbol.list文件路径
kernel/dynload/src/elf_symbol.c	希望导出的符号表源文件

在得到符号表源文件（假设为elf_symbol.c）后，编译该源文件得到符号表目标文件elf_symbol.so，该目标文件作为后续初始化动态加载模块的参数。

版权所有 © 华为技术有限公司

11 附录

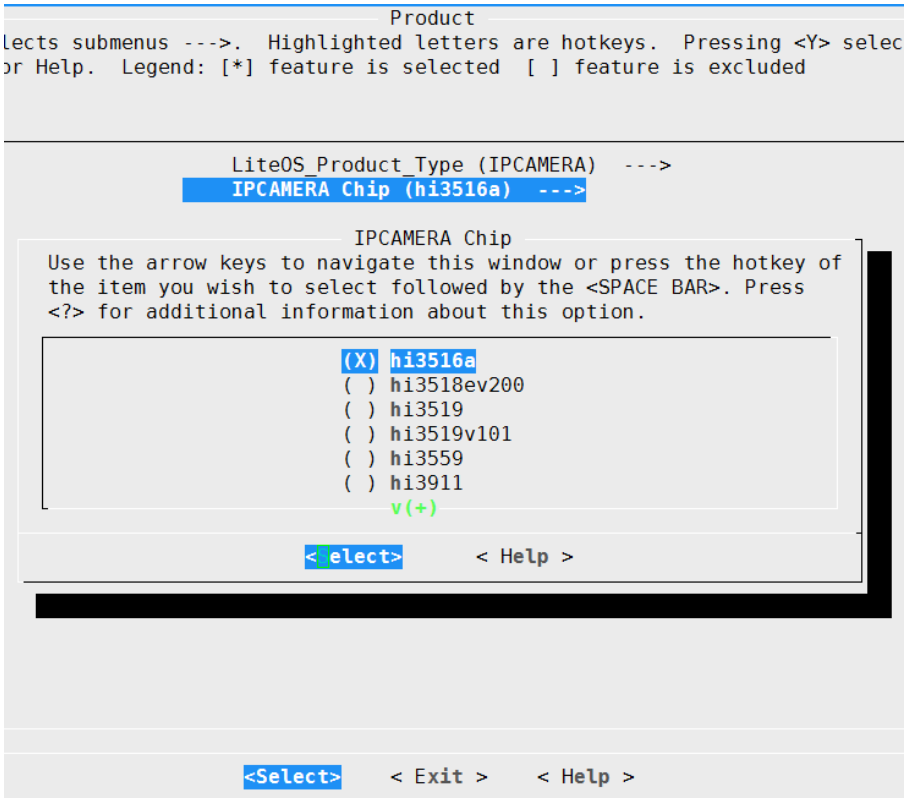
版权所有 © 华为技术有限公司

11.1 OS 内存占用情况

为了方便实时统计OS内存的占用情况，我们内置了统计工具方便用户统计。

具体使用步骤如下：

步骤1 在menuconfig配置界面选择编译平台，选择Product后，选择芯片类型：



步骤2 编译：

```
make
```

步骤3 运行脚本：

```
./tools/scripts/mem_statistic/mem_statistic.py hi3516a
```

其中hi3516a必须是在步骤一中选择并已经编译好的平台。

版权所有 © 华为技术有限公司

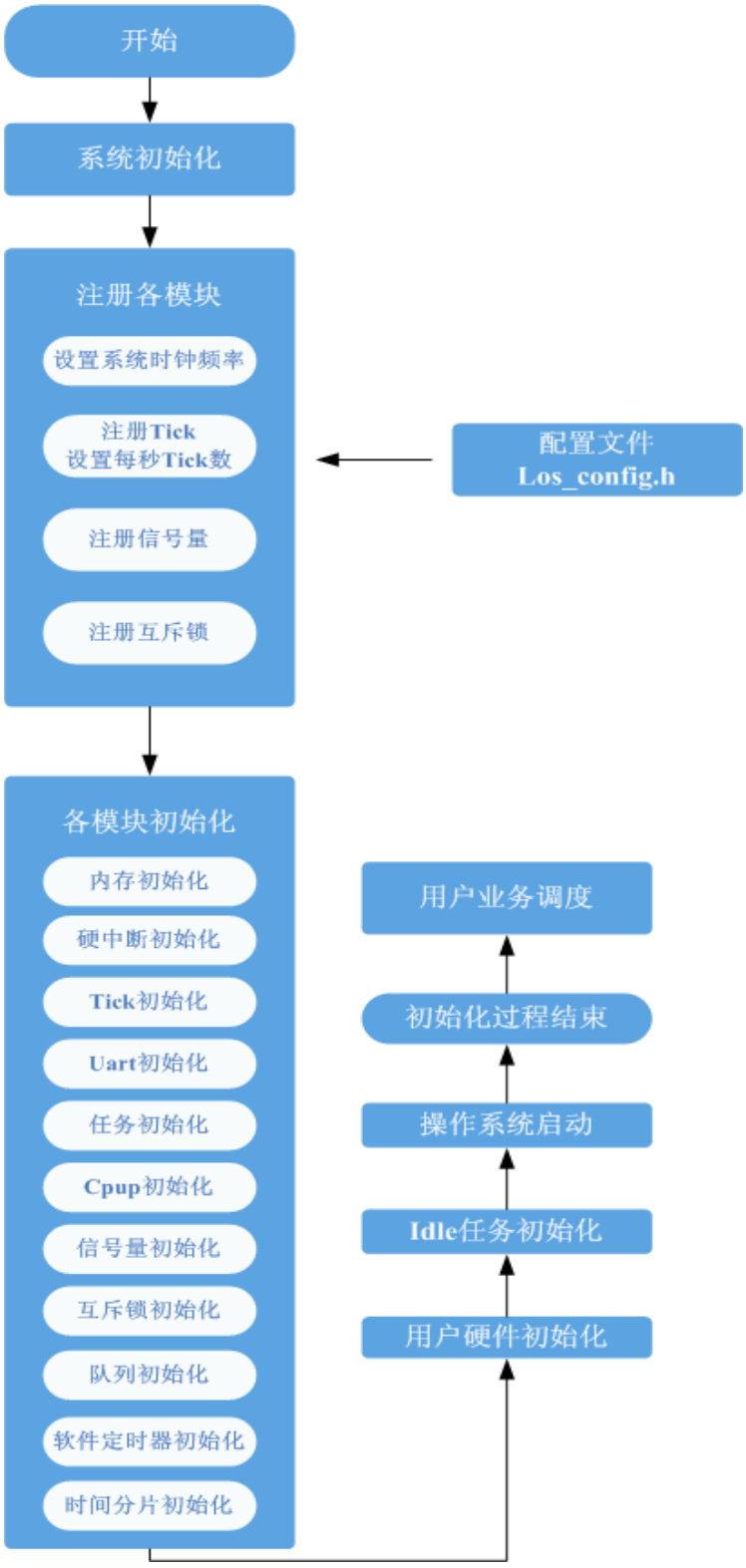
```
~/workspace/MEM/Huawei_LiteOS_ipc(branch:working*) » ./tools/scripts/mem_statistic.py hi3516a

hi3516a
===================
runstop:
total:      2.2k
.text:      2.0k        .data:      0.0k        .bss:       0.0k        .rodata:    0.1k

kernel:
total:     28.4k
.text:     27.4k        .data:      0.0k        .bss:       0.0k        .rodata:    1.0k

shell:
total:     30.1k
.text:     16.6k        .data:      0.3k        .bss:       8.6k        .rodata:    4.6k

lwip:
total:    243.6k
.text:    106.1k        .data:      0.0k        .bss:     122.1k        .rodata:   15.4k

usb:
total:   1203.3k
.text:    145.8k        .data:      8.5k        .bss:    1034.2k        .rodata:   14.9k
```

----结束

版权所有 © 华为技术有限公司

11.2 Kernel 启动流程介绍

Huawei LiteOS Kernel 启动流程



版权所有 © 华为技术有限公司