龙芯 LS2K SDK 参考手册

目录

前言

LS2K SDK 是 LoongIDE 配套的、为 LS2K0300、LS2K0301、LS2K0500、LS2K1000LA 等芯片定制的软件开发工具包，有以下特点：

- 实现 RTThread、FreeRTOS、μCOS 的移植，供不同用户选择；
- 移植高可靠性航天级嵌入式实时操作系统 RTEMS；
- 裸机编程使用**原创**的 PesudoOS 软件框架；
- 提供一致的 osal 接口用于裸机和 RTOS 编程；
- 设备驱动的实现采用统一的驱动程序模型；
- 块设备实现 FAT、EXT 和 yaffs2 文件系统的支持；
- 实现 posix 标准函数读写字符设备和块设备；
- loongarch 工具链使用龙芯官方 gcc 8.3.0 rc1.6 制作；

LS2K SDK 将项目开发统一到 osal 和 posix 函数的调用。

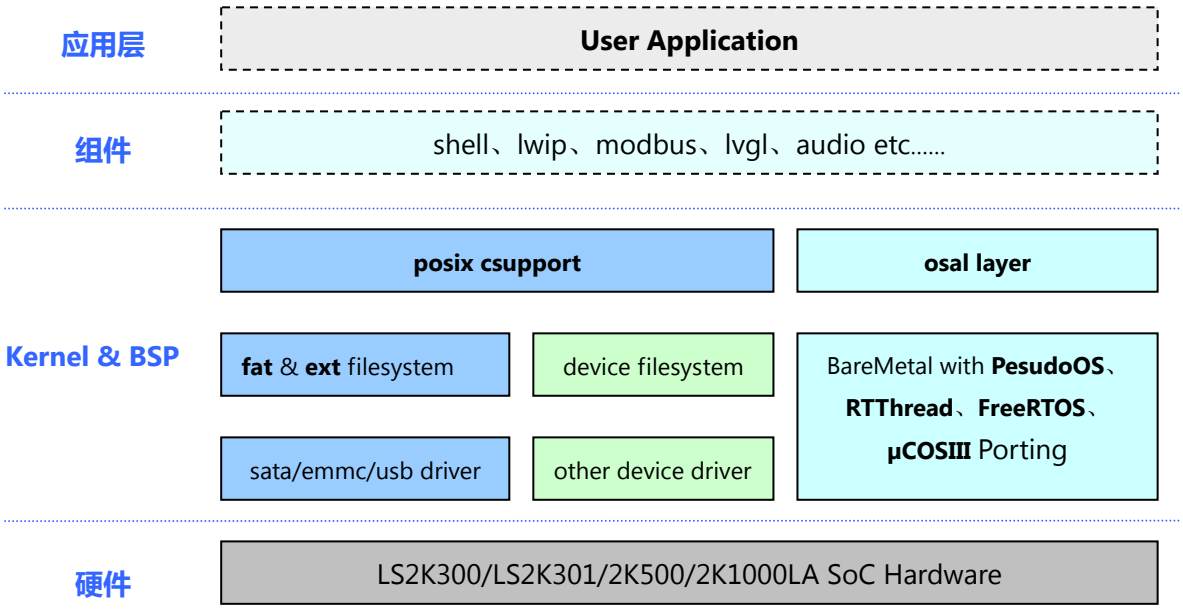
使用 osal 和 posix 编写实现的应用程序，同一套源代码可以在裸机和不同 RTOS 下无缝编译、运行。

第一章 基础篇

基础篇介绍 LS2K SDK 的各个组成部分以及实现的功能。

第一节、SDK 架构

LS2K 软件开发工具包（Software Development Kit）架构如下：



- 硬件层的驱动程序使用 osal 实现；
- posix 层封装了内存管理和文件系统相关的函数；
- 组件层主要使用 osal 和 posix 函数实现。

第二节、操作系统和 OSAL

LS2K 系列芯片是高集成度、高性能的 SoC 芯片，适合高端的嵌入式应用场景，所以为 LS2K 移植应用最广泛的、受众用户最多的操作系统是 SDK 需要实现的基本目标。

1、RTOS

LS2K SDK 移植了以下 RTOS：

- RTThread 4.10
- FreeRTOS 10.2.1
- μCOSIII 3.8.1
- RTEMS 4.11

2、PesudoOS

PesudoOS 是本公司研发的一套裸机编程框架，该框架实现了可以使用 Task、TaskSleep、Event、Semaphore、Mutex、MessageQueue、Timer 等 RTOS 的高级机制来进行裸机编程。之所以称为 PesudoOS（伪操作系统），是因为 PesudoOS 仍然使用裸机大循环来**顺序调用**各个函数（PesudoOS 将**这些函数称为 Task**）、而没有使用 RTOS 的线程、优先级的机制通过任务切换来实现各个函数的执行。

PesudoOS 为裸机编程的 osal 实现提供了底层基础。

3、osal

操作系统抽象层（Operating System Abstraction Layer）的作用是为不同操作系统提供一个统一的 API 接口，以掩盖底层操作系统的差异。它通常在应用层与操作系统之间充当中介角色。通过使用 OSAL，开发人员只需开发一次应用逻辑，而无需关心底层操作系统的具体细节。

Task --- 任务相关函数		**参数**
创建	osal_task_t **osal_task_create**(const char *name,	任务名称
	uint32_t stack_size,	堆栈大小
	uint32_t prio,	优先级
	uint32_t slice,	时间片
	osal_task_entry_t entry,	入口函数
	void *args);	函数参数
删除	void **osal_task_delete**(osal_task_t task);	任务句柄
挂起	void **osal_task_suspend**(osal_task_t task);	任务句柄
恢复	void **osal_task_resume**(osal_task_t task);	任务句柄

休眠	void **osal_task_sleep**(uint32_t ms);	休眠毫秒数
休眠到	void **osal_task_sleep_until**(uint32_t *prev_ticks, uint32_t inc_ticks);	上次休眠 ticks 的指针 本次休眠的 ticks *执行后*prev_ticks* 自动加上 *inc_ticks*

Event --- 事件相关函数		**参数**
创建	osal_event_t **osal_event_create**(const char *name, uint32_t opt);	事件名称 OSAL_OPT_FIFO 等选项
删除	void **osal_event_delete**(osal_event_t event);	事件句柄
发送	int **osal_event_send**(osal_event_t event, uint32_t bits);	事件句柄 发送的事件
接收	uint32_t **osal_event_receive**(osal_event_t event, uint32_t bits, uint32_t flag, uint32_t timeout_ms);	事件句柄 待接收事件 OSAL_EVENT_FLAG_AND 等 等待时间，0=立即返回
设置 osal 事件	void **osal_event_set_bits**(osal_event_t event, uint32_t bits);	事件句柄 直接设置事件标志位

Semphore --- 信号量相关函数		**参数**
创建	osal_sem_t **osal_sem_create**(const char *name, uint32_t opt, uint32_t initial_count);	信号量名称 OSAL_OPT_FIFO 等选项 初始化值
删除	void **osal_sem_delete**(osal_sem_t sem);	信号量句柄
获取	int **osal_sem_obtain**(osal_sem_t sem, uint32_t timeout);	信号量句柄 等待时间，0=立即返回
释放	int **osal_sem_release**(osal_sem_t sem);	信号量句柄
重置 osal 信号量	void **osal_sem_reset**(osal_sem_t sem);	信号量句柄

Mutex --- 互斥量相关函数		**参数**
创建	osal_mutex_t **osal_mutex_create**(const char *name, uint32_t opt);	互斥量名称 OSAL_OPT_FIFO 等选项
删除	void **osal_mutex_delete**(osal_mutex_t mutex);	互斥量句柄
获取	int **osal_mutex_obtain**(osal_mutex_t mutex, uint32_t timeout_ms);	互斥量句柄 等待时间，0=立即返回
释放	int **osal_mutex_release**(osal_mutex_t mutex);	互斥量句柄

Message Queue --- 消息相关函数		**参数**
创建	osal_mq_t **osal_mq_create**(const char *name, uint32_t opt, uint32_t item_size, uint32_t max_msgs);	消息名称 OSAL_OPT_FIFO 等选项 消息项大小 消息个数
删除	void **osal_mq_delete**(osal_mq_t mq);	消息句柄
发送	int **osal_mq_send**(osal_mq_t mq, const void *msg, int size);	消息句柄 待发送消息指针 消息项大小
接收	int **osal_mq_receive**(osal_mq_t mq, void *msg, int size, uint32_t timeout);	消息句柄 待接收消息缓冲区指针 消息项大小 等待时间，0=立即返回
是否满	int osal_mq_is_full(osal_mq_t mq);	消息句柄
清理	int osal_mq_flush(osal_mq_t mq);	消息句柄

Timer --- 定时器相关函数		**参数**
创建	osal_timer_t **osal_timer_create**(const char *name, osal_task_entry_t handler, void *argument, uint32_t timeout_ms, bool is_period);	定时器名称 定时函数句柄 定时函数参数 定时时间间隔 是否重复
删除	void **osal_timer_delete**(osal_timer_t timer);	定时器句柄
启动	void **osal_timer_start**(osal_timer_t timer, uint32_t timeout_ms);	定时器句柄 定时时间间隔
停止	void **osal_timer_stop**(osal_timer_t timer);	定时器句柄

Other --- 其它函数		**说明**
os 是否运行	int **osal_is_osrunning**(void);	返回 1 表示正在运行
进入临界区	size_t **osal_enter_critical_section**(void);	
离开临界区	void **osal_leave_critical_section**(size_t flag);	osal_enter_critical_section() 的返回值
osal 延时	void **osal_msleep**(uint32_t ms);	延时时间
申请内存	void ***osal_malloc**(size_t size);	
释放内存	void **osal_free**(void *ptr);	

注：1、返回类型 int 的函数，当返回 0 表示函数执行成功；
2、无论裸机还是 RTOS，均配置 1 tick 等于 1ms，所以没有实现转换。

osal 实现的头文件：osal.h

osal 实现的源代码文件：

- RTThread osal_rtthread.c
- FreeRTOS osal_freertos.c
- µCOSIII osal_ucos.c
- PesudoOS osal_pesudoos.c

第三节、驱动程序

设备驱动程序是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件只是个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。

设备驱动程序是内核的一部分，它主要完成 4 个功能：

- 对设备初始化和释放；
- 把数据从内核传送到硬件，从硬件读取数据；
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据；
- 检测和处理设备出现的错误。

1、设备驱动模型

芯片上连接的设备，无论是片上设备还是片外设备，其驱动程序都编写成以下形式，供系统调用。

```
typedef int (*driver_init_t)(const void *dev, void *arg);
typedef int (*driver_open_t)(const void *dev, void *arg);
typedef int (*driver_close_t)(const void *dev, void *arg);
typedef int (*driver_read_t)(const void *dev, void *buf, int size, void *arg);
typedef int (*driver_write_t)(const void *dev, void *buf, int size, void *arg);
typedef int (*driver_ioctl_t)(const void *dev, int cmd, void *arg);

typedef struct driver_ops                       // 对象封装
{
    driver_init_t       init_entry;             // 初始化
    driver_open_t       open_entry;             // 打开
    driver_close_t      close_entry;            // 关闭
    driver_read_t       read_entry;             // 读操作
    driver_write_t      write_entry;            // 写操作
    driver_ioctl_t      ioctl_entry;            // 控制
} driver_ops_t;
```

参数：

- ***dev**：　待操作的设备指针，例如：devUART0、devCAN0 etc；
- ***arg**：　操作使用的参数。

2、I2C/SPI 总线驱动模型

一个 I2C 和 SPI 控制器上可以挂接多个设备，从设备以 **I2C 地址**或者 **SPI 片选**来区分；所以在驱动实现上使用二级驱动程序的方式，本驱动也称为**总线驱动**程序。

```
typedef int (*I2C_init_t)(const void *bus);
typedef int (*I2C_send_start_t)(const void *bus, unsigned Addr);
typedef int (*I2C_send_stop_t)(const void *bus, unsigned Addr);
typedef int (*I2C_send_addr_t)(const void *bus, unsigned Addr, int rw);
typedef int (*I2C_read_bytes_t)(const void *bus, unsigned char *bytes, int nbytes);
typedef int (*I2C_write_bytes_t)(const void *bus, unsigned char *bytes, int nbytes);
typedef int (*I2C_ioctl_t)(const void *bus, int cmd, void *arg);

typedef struct libi2c_ops                       // 对象封装
{
    I2C_init_t          init;                   // 初始化
    I2C_send_start_t    send_start;             // 发送 I2C 开始信号
    I2C_send_stop_t     send_stop;              // 发送 I2C 结束信号
    I2C_send_addr_t     send_addr;              // 发送 I2C 从设备地址
    I2C_read_bytes_t    read_bytes;             // 读数据
    I2C_write_bytes_t       write_bytes;        // 写数据
    I2C_ioctl_t         ioctl;                  // 控制
} libi2c_ops_t;

typedef libi2c_ops_t  libspi_ops_t;
```

参数：

- ***bus**：　待操作的设备指针，例如：busI2C0、busSPI0 etc；
- **Addr**：　I2C 从设备地址、或者 SPI 从设备片选；
- **rw**：　发送 I2C 从设备地址后执行读或者写操作；
- ***arg**：　控制使用的参数。

SPI、I2C 总线上挂接的从设备，**通过调用总线驱动程序来实现具体的从设备驱动程序**。

3、驱动实现

驱动程序使用 osal 来实现，具体实现请参阅项目目录：
ls2k300/drivers、ls2k500/drivers、ls2k1000la/drivers 下各个设备驱动程序的代码。

第四节、文件系统

LS2K SDK 实现了用于管理 I/O 设备的 device filesystem 和管理块设备的 disk filesystem，通过设备抽象层将两种设备转换为文件系统可访问的接口，并实现统一的 POSIX 接口来访问设备。

块设备支持三种类型的文件系统挂载：

- Windows 文件系统：　FAT16、FAT32
- Linux 文件系统：　EXT2、EXT3、EXT4
- NAND Flash 文件系统：　yaffs2

1、文件系统根目录：　**/**

2、I/O 设备的目录：　**/dev**

设备名称如：

```
/dev/uart0          // 串口设备 0
/dev/uart4          // 串口设备 4
/dev/can0           // CAN 设备 0
/dev/spi0.norflash  // SPI0 上挂接的 norflash 芯片
```

3、SATA 盘根目录：　**/hda**、**/hdb** ...

自动设别是否有磁盘、是否存在 mbr；如果分区格式是 FAT 或者 EXT，自动挂载磁盘。

4、EMMC 盘：　**/mmc0**

自动设别是否有 emmc 芯片、是否存在 mbr；如果分区格式是 FAT 或者 EXT，自动挂载磁盘。

5、U 盘：　**/usbd0**、**/usbd1** ...

协议栈使用：CherryUSB

支持 USB 控制器：EHCI、DWC2

当 U 盘插入时，自动设别是否存在 mbr；如果分区格式是 FAT 或者 EXT，自动挂载磁盘；当拔掉 U 盘时，自动卸载磁盘。

6、NAND 盘：　**/ndd**

bsp 启动时判断 NAND 总线上是否挂接有 nand-flash 芯片；如果有 flash 芯片，系统将根据用户配置的 flash 芯片参数把该 flash 挂载为 yaffs2 的文件系统。

注：1、上述文件系统是否使用，用户可裁剪；

2、文件系统以静态库 libbsp.a 供用户链接使用；

第五节、posix 接口函数

POSIX 是 Portable Operating System Interface (可移植操作系统接口)的简称，是软件业界的国际标准。使用 posix 接口编写的软件方便移植，其封装层的开销在现代高速 cpu 下可以忽略不计。

POSIX 标准同样适用于嵌入式软件。

1、SDK 实现的 posix 接口函数

BSP 实现的函数	说明
void ***malloc**(size_t size)	申请内存
void ***calloc**(size_t nmemb, size_t size)	申请内存并清零
void ***realloc**(void *ptr, size_t size)	重新申请内存
void **free**(void *ptr)	释放申请的内存
void ***aligned_malloc**(size_t size, unsigned int align)	*对齐申请内存
void ***aligned_realloc**(void *ptr, size_t size, unsigned int align)	*对齐重新申请内存
void **aligned_free**(void *addr)	*释放对齐内存
int **open**(const char *file, int flags, ...)	打开文件（设备）
int **close**(int fd)	关闭文件（设备）
int **read**(int fd, void *buf, size_t len)	读文件（设备）
int **write**(int fd, const void *buf, size_t len)	写文件（设备）
int **ioctl**(int fd, unsigned cmd, ...)	控制（设备）
int **mkdir**(const char *path, mode_t mode)	创建目录
int **rmdir**(const char *path)	删除目录
DIR ***opendir**(const char *path)	打开目录
struct dirent ***readdir**(DIR *dir)	读取目录内容（下一个）
int **closedir**(DIR *dir)	关闭目录
int **fsize**(int fd)	文件大小
int **tell**(int fd)	文件当前读写位置
int **stat**(const char *path, struct stat *sbuf)	文件或者目录状态
int **fstat**(int fd, struct stat *sbuf)	文件状态
int **chmod**(const char *path, mode_t mode)	改变文件或者目录权限
int **fchmod**(int fd, mode_t mode)	改变文件权限
int **unlink**(const char *name)	删除文件
int **rename**(const char *oldname, const char *newname)	改名文件或者目录
off_t **lseek**(int fd, off_t offset, int whence)	设置文件当前读写位置

int **ftruncate**(int fd, off_t length)	截断文件设置成新的大小
int **fsync**(int fd)	如果有写缓冲区，执行写入
int **isatty**(int fd)	是否 tty
int **access**(const char *path, int amode)	文件或者目录是否存在
int **chdir**(const char *path)	设置文件系统的当前目录
char ***getcwd**(char *buf, size_t size)	获取文件系统的当前目录
int **gettimeofday**(struct timeval *__p, void *__tz)	从 RTC 获取当前日期

2、newlibc 内置的 posix 函数

newlib 库内置丰富的 posix 接口函数。

第六节、组件

LS2K SDK 包含以下组件：

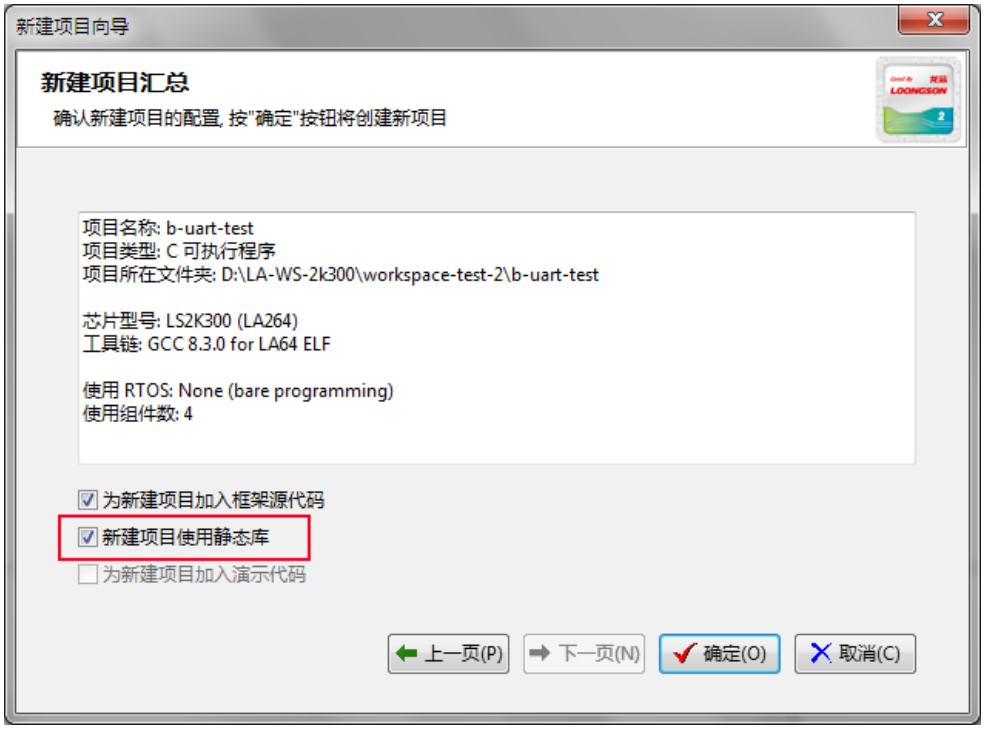
- lwip 1.4.1
- lwip 2.1.3
- modbus master & slave library
- fat filesystem
- lwext filesystem
- yaffs2 filesystem
- Cherry USB library
- command shell library
- mp3 audio player
- etc.

第二章 应用篇

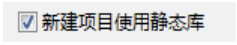
应用篇以设计一个自定义协议的串口通信程序来说明 SDK 的应用。
开发板：LS2K300

第一节 新建项目

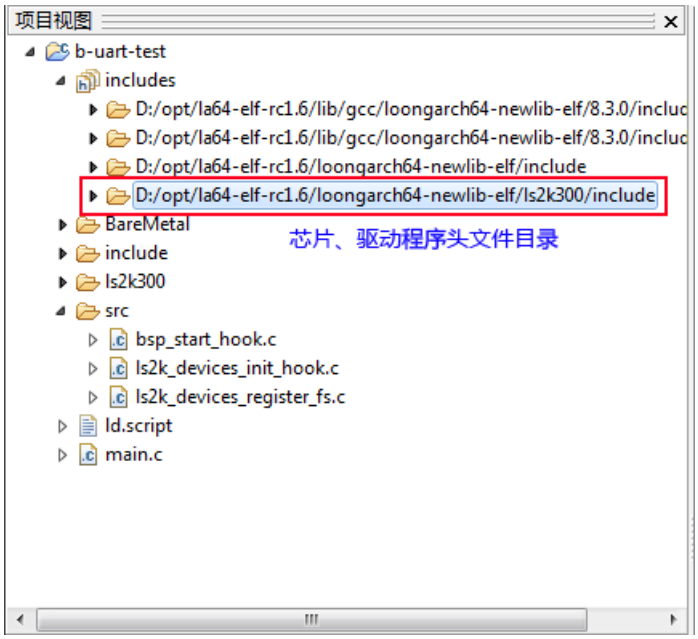
使用“新建项目”向导新建一个项目。
芯片：LS2K300；
开发板：General；
操作系统：裸机。



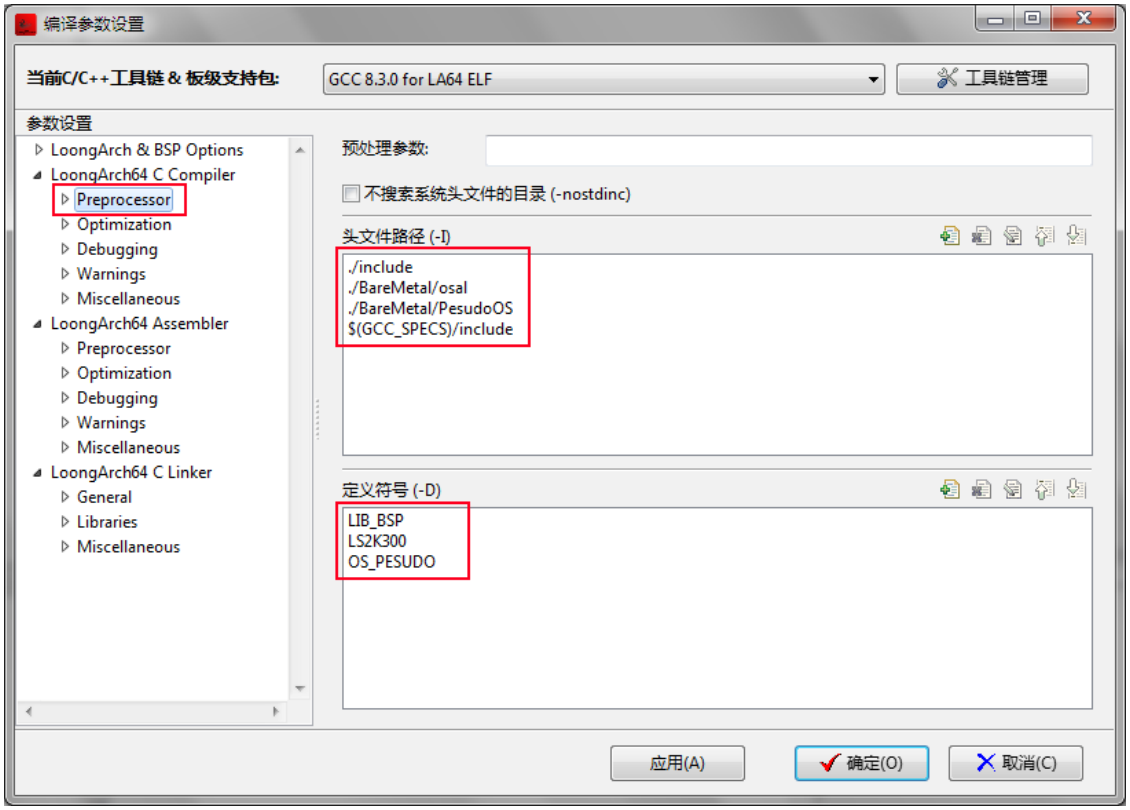
1、使用静态库

当 

时，生成的项目目录如下：



除了项目配置相关源代码，项目全部使用 **libbsp.a** 静态库。
按“F2”打开编译参数设置窗口：



头文件使用“$(GCC_SPECS)/include”搜索路径。

2、不使用静态库

当 新建项目使用静态库 时，生成的项目目录如下：

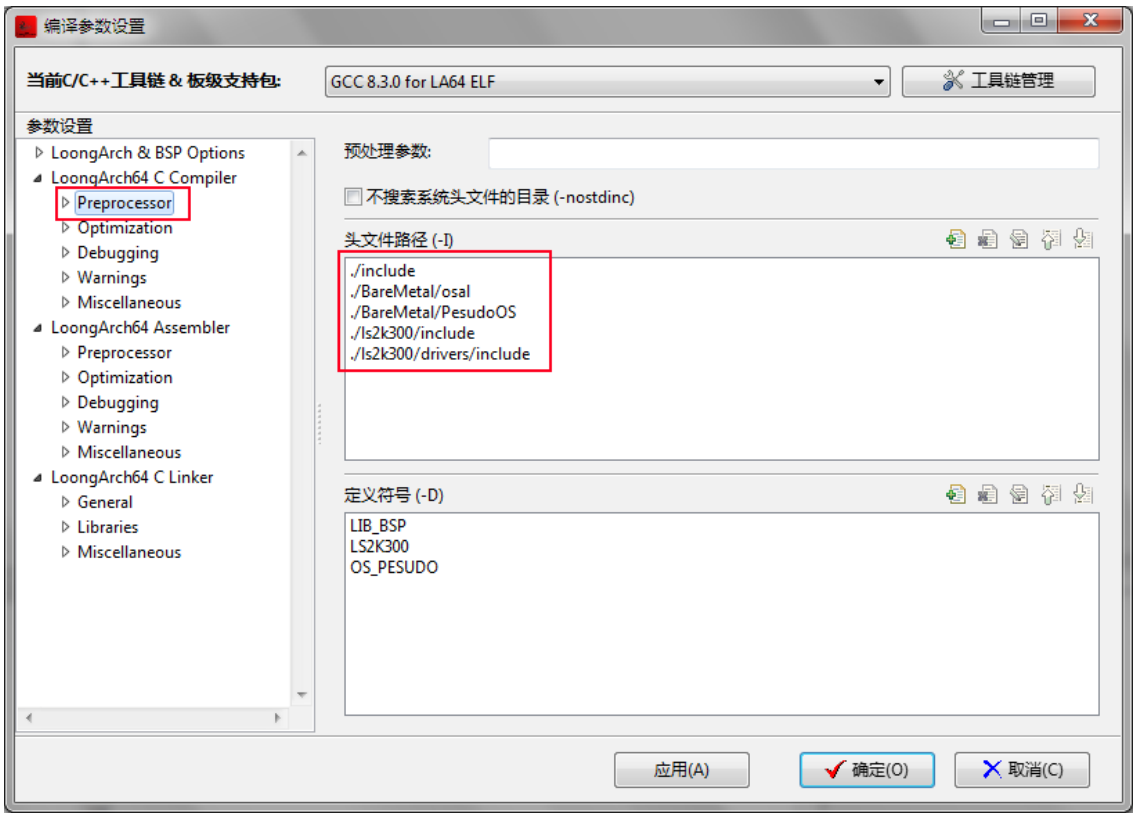
项目加入的源代码：

- OS 源代码；
- 底层移植源代码；
- OSAL 实现源代码；
- 驱动源代码；
- 项目配置源代码。

使用 libbsp.a 静态库的功能模块：

- sata、emmc 块设备驱动；
- cherry usb 协议栈；
- fat、ext、yaffs2 文件系统；
- posix c 接口模块；
- mp3 音频播放模块；
- shell 功能模块。

按“F2”打开编译参数设置窗口：



头文件使用本地搜索路径。

注：新建项目可以直接编译、运行。

第二节 配置项目

1、bsp.h 文件

通过 bsp.h 宏定义的设置，对项目使用哪些设备和库文件进行裁剪。

①、是否使用 OS

```
#define BSP_USE_OS          1       // 1=使用 OS
```

注：裸机编程有 PesodoOS 的支持，也使用 OS

②、驱动设备的裁剪

```
#define BSP_USE_UART0       1       // 使用
#define BSP_USE_UART1       0
#define BSP_USE_UART2       0
#define BSP_USE_UART3       0
#define BSP_USE_UART4       1       // 使用
#define BSP_USE_UART5       0
#define BSP_USE_UART6       1       // 使用
#define BSP_USE_UART7       0
#define BSP_USE_UART8       0
#define BSP_USE_UART9       0
```

本项目使用：

uart0：用作串口控制台，打印应用程序的输出信息，实现监控。

uart4：用作本项目的发送串口。

uart6：用作本项目的接收串口。

③、文件系统和裁剪

```
#define BSP_USE_FS          1       // 使用文件系统
#if BSP_USE_FS
#define BSP_USE_USB         1       // 使用 USB 协议栈
#define BSP_USE_EMMC        1       // 使用 EMMC 存储系统
#endif
```

使用文件系统，让项目使用 posix 编程。

④、shell 命令行

```
#define BSP_USE_SHELL       1       // 使用
```

shell 实现的功能：

- 查看项目的运行信息；
- 命令行实现文件相关的操作。

2、src 目录

①、bsp_start_hook.c

全局变量：

```
/* DC 显示模式
 * 格式：X 分辨率 x Y 分辨率-RGB 模式@刷新率
 */
char LCD_display_mode[] = "1024x600-16@60";
```

注： ls2k_dc.c 只实现了 **RGB565** 模式；
新增不同分辨率的 LCD 屏幕可能需要修改参数表 **vga_modes[]**。

```
/* 串口控制台端口
 * 项目 printk()、printf()函数把数据输出到这个串口
 */
void *ConsolePort = NULL;

/* 在 heap 初始化完成、控制台初始化前执行
* 本函数完成：ConsolePort 的初始化；文件系统的初始化；完成设备文件的注册
*/
int bsp_start_hook1(void)

/* 在跳转 main() 之前执行
 * 本函数完成：块设备如 emmc、usb、sata、nand 等的初始化和文件系统挂载
 */
int bsp_start_hook2(void)
```

②、ls2k_devices_init_hook.c

设备初始化时调用的钩子函数。

例如： int **ls2k_uart_init_hook**(const void *dev)
当应用程序执行 **ls2k_uart_init**(devUART4, NULL) 对 **uart4** 进行初始化，此时调用 **ls2k_uart_init_hook**(devUART4)，执行与 **uart4** 相关的操作，如**设置引脚复用**。

注：设置引脚复用是钩子函数的常用功能。

③、ls2k_devices_register_fs.c

把设备驱动程序注册到文件系统，以便 posix 函数使用。

实现函数：void **register_all_devices**(void)；上述 int **bsp_start_hook1**(void)调用此函数。

④、bsp_config_nand_for_yaffs2.c

仅 LS2K0500、LS2K1000LA 有 NAND 设备。

根据开发板上的 NAND-Flash 芯片的实际参数来配置 yaffs2 文件系统。

3、ld.script 链接脚本

链接脚本用于将编译后的目标文件(**.o** 文件)和库文件(**.a** 文件)组合成可执行文件(**.elf** 文件)。

LoongIDE 链接脚本使用固定文件名 **ld.script** 或者 **linkcmds**，在用户创建项目时自动加入；可以根据项目需要修改脚本文件。

①、头部和代码段

物理内存大小：小于等于开发板的实际内存；

可执行文件内存起始地址：原则上不与 Bootloader 的运行地址冲突。

```
OUTPUT_FORMAT("elf64-loongarch", "elf64-loongarch", "elf64-loongarch")   elf 文件格式

OUTPUT_ARCH("loongarch")  elf芯片架构

_RamSize = DEFINED(_RamSize) ? _RamSize : 256M;   物理内存大小
_StackSize = DEFINED(_StackSize) ? _StackSize : 0x4000; /* 16k */    系统堆栈大小

ENTRY(_start)      入口函数，在start.S中定义

SECTIONS
{
    . = 0x9000000000800000;    可执行文件链接的内存起始地址
    .text :  代码段
    {
        _ftext = . ;
        *(.start)      start段，在start.S中定义，强制start.S链接在可执行文件的起始位置
        *(.text)
        *(.rodata)
        *(.rodata1)
        *(.reginfo)
        *(.init)
        *(.stub)
        /* .gnu.warning sections are handled specially by elf32.em. */
        *(.gnu.warning)
    } = 0
                                编程时可以把 _RamSize _StackSize _ftext 等当作常量引用
    ...
```

②、已初始化数据段

```
    .data :  数据段，存放已初始化的全局变量
    {
        _fdata = . ;
        *(.data)
        . = ALIGN(32);
        *(.data.align32)
        . = ALIGN(64);
        *(.data.align64)
        . = ALIGN(128);
        *(.data.align128)
        . = ALIGN(4096);
        *(.data.align4096)
         CONSTRUCTORS
    }
```

③、未初始化数据段和堆栈段

```
__bss_start = . ;    定义BSS段开始地址
_fbss = . ;

.sbss :   小的BSS段，用于存放未初始化的“近”数据
{
    *(.sbss)
    *(.scommon)
}

.bss :    数据段，用于存放未初始化的全局变量
{
    *(.dynbss)
    *(.bss)
    . = ALIGN(32);
    *(.bss.align32)
    . = ALIGN(64);
    *(.bss.align64)
    . = ALIGN(128);
    *(.bss.align128)
    . = ALIGN(4096);
    *(.bss.align4096)
    *(COMMON)
}
PROVIDE (__bss_end = .);      定义BSS段结束地址

.stack :   堆栈段
{
    . = ALIGN(32);
    __stack_start = .;       定义堆栈开始地址
    . += _StackSize;         /* system stack */
    __stack_end = .;         定义堆栈结束地址
}

_end = . ;    定义程序结束地址，之后是程序可使用的 heap 空间
PROVIDE (end = .);
```

④、内置变量

链接脚本中定义的 `__stack_start = .;` `__stack_end = .;` 等，在应用程序中可以当作指针地址来使用：`__stack_start` 是系统堆栈的起始地址、`__stack_end` 是系统堆栈的结束地址。

第三节 程序实现

1、通信协议设计

定义以下通信协议来实现两个 **uart** 之间的发送和接收。
字节序：

	功能描述	字节长度	说明
1	head	**2** bytes	起始位
2	length	**2** bytes	报文长度
3	content	**n** bytes	
4	crc	**2** bytes	校验码

1. head: 总是 **0xbeaf**
2. length: content 的长度，实际接收长度需要 length 加上 crc 的 2 字节
3. content: 报文内容
4. crc: length+content 的 uint8_t 类型做加法运算结果，保存到 uint16_t

2、发送代码

在项目根目录下新建源代码文件：**uart_tx_test.c**

源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

#include "bsp.h"
#include "ls2k_uart.h"

#include "osal.h"

extern int ioctl(int fd, unsigned cmd, ...);
extern unsigned short caclulate_crc(unsigned char *buf, int len);

/*
 *    发送间隔:
 *          115200 8N1 时, 1ms 最多传输 12 字节
 *          缓冲区最大长度 206 字节的传输耗时 18 ms
 */
#define TX_INTERVAL_MS          250 // 100 // 每隔 250 毫秒发送一次

/* 发送串口：打开的文件句柄 */
static int fd_TxU = -1;
```

```
//-------------------------------------------------------------------------

/* 发送字符串及总长度 */
#define STR_DOG         "The quick brown fox jumps over a lazy dog on ground."
#define STR_DOGLEN      52

/* 协议定义 */
#define PKG_HEAD_LEN    2                   // 识别码 2 字节
#define PKG_LEN_LEN     2                   // 报文长度 2 字节
#define PKG_DATA_MAX    STR_DOGLEN          // 报文总长限制
#define PKG_CRC_LEN     2                   // 校验码 2 字节

#define TX_BUF_LEN      (PKG_DATA_MAX+6) // 缓冲区长度

//-------------------------------------------------------------------------
// 数据发送函数
//-------------------------------------------------------------------------
static void uart_do_send_1_pkg(void *arg)
{
    unsigned char tx_buf[TX_BUF_LEN];
    int totalbytes = 0, pkglen;
    unsigned short calc_crc;
    unsigned char *p = tx_buf;

    /* 使用随机数获取本次发送长度 */
    pkglen = rand();
    pkglen %= PKG_DATA_MAX;
    if (pkglen <= 0)
        pkglen = STR_DOGLEN;

     /* 向缓冲区写入 uint16_t 的数据，地址可能是非对齐的，所以使用字节操作 */

    /* head 写入识别 */
    *p++ = 0xbe;
    *p++ = 0xaf;

    /* length 写入报文长度 */
    *p++ = pkglen & 0xFF;
    *p++ = (pkglen >> 8) & 0xFF;

    /* content 写入报文内容 */
    strncpy((char *)p, STR_DOG, (size_t)pkglen);
    p += pkglen;

    /* crc 校验码计算并写入 */
    calc_crc = caclulate_crc(tx_buf + 2, pkglen + 2);
    *p++ = calc_crc & 0xFF;
    *p++ = (calc_crc >> 8) & 0xFF;

    /* 使用 posix 函数 write() 发送报文 */
    totalbytes = pkglen + 6;
    if (write(fd_TxU, tx_buf, totalbytes) == totalbytes)
        printf("uart send %i bytes success.\r\n", totalbytes);
    else
        printf("uart send fail!\r\n");
}
```

```
//-------------------------------------------------------------------------
// UART 发送任务
//-------------------------------------------------------------------------
static void uart_send_task(void *arg)
{
    printk("uart send task started...\r\n");

    for ( ; ; )          /* 这里用了死循环，循环体内必须要有 PesudoOS 的阻塞函数执行 */
    {
        /* 执行一次数据包发送 */
        uart_do_send_1_pkg(NULL);

        /* 任务休眠 250 毫秒 */
        osal_task_sleep(TX_INTERVAL_MS);          /* 这个函数引起阻塞 */
    }

    close(fd_TxU);    /* 关闭文件句柄，执行不到这里... */
}

//-------------------------------------------------------------------------
// 启动 UART 发送
//-------------------------------------------------------------------------
#define UART_STK_SIZE          1024       // 堆栈大小
#define UART_TASK_PRIO         16         // 堆栈优先级  RTOS 有效
#define UART_TASK_SLICE        10         // 时间片      RTThread 有效

static osal_task_t uart_tx_task = NULL;      // task 句柄

void start_uart_tx(char *dev_uart_name)
{
    if (!dev_uart_name)
        return;

    /* 使用 posix 函数打开串口 */
    fd_TxU = open(dev_uart_name, O_RDWR, NULL);              /* 默认 115200,8N1 */
    if (fd_TxU < 0)
    {
        printf("open device %s fail!\r\n", dev_uart_name);
        return;
    }

    /* 设置串口使用 DMA 发送 */
    ioctl(fd_TxU, IOCTL_UART_SET_RXTX_MODE, (void *)UART_TX_DMA);

    /* 使用 osal 创建 UART 发送任务 */
    uart_tx_task = osal_task_create("UARTtx",
                                    UART_STK_SIZE * sizeof(size_t),
                                    UART_TASK_PRIO, UART_TASK_SLICE,
                                    uart_send_task, NULL);

    if (uart_tx_task == NULL)
    {
        close(fd_TxU);
        printf("create uart tx test task fail!\r\n");
    }
}
```

3、接收代码

在项目根目录下新建源代码文件：**uart_rx_test.c**

源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

#include "bsp.h"
#include "ls2k_uart.h"

#include "osal.h"

extern int ioctl(int fd, unsigned cmd, ...);

/*
 * 接收间隔:
 *   115200 8N1 时, 1ms 最多传输 12 字节
 *   缓冲区最大长度 206 字节的传输耗时 18 ms
 */
#define RX_INTERVAL_MS          100 // 250 //

/* 接收串口文件句柄 */
static int fd_RxU = -1;

//-------------------------------------------------------------------------
// 校验码计算函数
//-------------------------------------------------------------------------
unsigned short caclulate_crc(unsigned char *buf, int len)
{
    unsigned short crc = 0;
    for (int i=0; i<len; i++)
        crc += buf[i];
    return crc;
}

//-------------------------------------------------------------------------
// 使用状态机表示当前接收处于的阶段
//-------------------------------------------------------------------------

#define RX_STATE_HEAD1      0x01    /* 接收到识别码第一个字节 */
#define RX_STATE_HEAD2      0x02    /* 接收到识别码第二个字节*/
#define RX_STATE_LEN1       0x04    /* 接收到报文长度第一个字节*/
#define RX_STATE_LEN2       0x08    /* 接收到报文长度第二个字节*/
#define RX_STATE_LCONTENT   0x10    /* 接收内容中，到长度为止 */
#define RX_STATE_CRC1       0x20    /* 接收到校验码第一个字节*/
#define RX_STATE_CRC2       0x40    /* 接收到校验码第二个字节*/
```

```c
/* 协议定义，同发送 */
#define PKG_HEAD_LEN    2                   // 识别码 2 字节
#define PKG_LEN_LEN     2                   // 报文长度 2 字节
#define PKG_DATA_MAX    STR_DOGLEN          // 报文总长限制
#define PKG_CRC_LEN     2                   // 校验码 2 字节

#define TX_BUF_LEN      (PKG_DATA_MAX+6) // 缓冲区长度

//-------------------------------------------------------------------------
// 一次接收数据的函数
//-------------------------------------------------------------------------

static int rx_timeout = 10;       // 25; //    /* 接收超时 */

/* 宏定义：接收一个字节并赋值接收状态 */
#define INC_RX(st)    do { p++; totalbytes++; state = st; } while (0)

/* 宏定义：复位接收并设置为初始状态 */
#define RESET_RX     do { p = rx_buf; totalbytes = 0; state = RX_STATE_HEAD1; } while (0)

static void uart_do_receive_1_pkg(void *arg)
{
    unsigned char rx_buf[RX_BUF_LEN];
    int totalbytes = 0, thisbytes = 0, pkglen;
    int tmo = rx_timeout, state = RX_STATE_HEAD1;
    unsigned short calc_crc, recv_crc = 0;
    unsigned char *p = rx_buf;

    for (;;)
    {
        /* 使用 posix read()函数，每次读一个字节 */
        thisbytes = read(fd_RxU, p, 1);
        if ((thisbytes == 0) || (*p == 0xFF))
        {
            if (rx_timeout > 0)             /* 如果使用了接收超时，判断是否超时 */
            {
                tmo--;
                if (tmo <= 0)               /* 如果超时，退出本次接收 */
                {
                    printf("uart rx timeout!\r\n");
                    return;
                }
            }

            osal_task_sleep(10);            /* 没有接收到数据，延时 10ms */
            continue;
        }

        /* 接收到了一个字符，根据状态继续接收 */
        switch (state)
        {
            case RX_STATE_HEAD1:
                if (*p == 0xbe)             /* 是否识别码第一个字节 */
                    INC_RX(RX_STATE_HEAD2);
                break;

            case RX_STATE_HEAD2:
                if (*p == 0xaf)             /* 是否识别码第二个字节 */
```

```
                INC_RX(RX_STATE_LEN1);
            else
                RESET_RX;           /* 重新开始接收 */
            break;

        case RX_STATE_LEN1:
            INC_RX(RX_STATE_LEN2);
            break;

        case RX_STATE_LEN2:
            INC_RX(RX_STATE_LCONTENT);
            pkglen = rx_buf[2] | (rx_buf[3] << 8);    // 报文长度
            if (pkglen > PKG_DATA_MAX)
                RESET_RX;           /* 太长，重新开始接收 */
            break;

        case RX_STATE_LCONTENT:
            INC_RX(RX_STATE_LCONTENT);
            if (totalbytes - 4 == pkglen)         /* 没有全部接收，状态不变 */
                state = RX_STATE_CRC1;
            break;

        case RX_STATE_CRC1:
            INC_RX(RX_STATE_CRC2);
            break;

        case RX_STATE_CRC2:
            INC_RX(RX_STATE_CRC2);
            /* 接收到的校验码 */
            recv_crc = rx_buf[totalbytes - 2] | (rx_buf[totalbytes - 1] << 8);
            break;

        default:
            break;
    }

    /**
     * 验证 crc，如果正确退出本次接收，否则继续
     */
    If (state == RX_STATE_CRC2)
    {
        calc_crc = caclulate_crc(rx_buf + 2, pkglen + 2);
        if (recv_crc == calc_crc)
        {
            rx_buf[pkglen + 4] = '\0';
            printf("RX[%i]: %s\r\n", pkglen, rx_buf + 4);
        }
        else
        {
            printf("rx crc error!\r\n");
        }

        break;      /* 退出 */
    }
  }
}
```

```c
//-------------------------------------------------------------------------
// UART 接收任务
//-------------------------------------------------------------------------
static void uart_receive_task(void *arg)
{
    printk("uart receive task started...\r\n");

    for ( ; ; )          /* 这里用了死循环，循环体内必须要有 PesudoOS 的阻塞函数执行 */
    {
        /* 执行一次接收 */
        uart_do_receive_1_pkg(NULL);

        /* abandon cpu time to run other task */
        osal_task_sleep(RX_INTERVAL_MS);          /* 这个函数引起阻塞 */
    }

    close(fd_RxU);   /* 关闭文件句柄，执行不到这里... */
}

//-------------------------------------------------------------------------
// 启动 UART 接收
//-------------------------------------------------------------------------
#define UART_STK_SIZE          1024       // 堆栈大小
#define UART_TASK_PRIO         16         // 堆栈优先级   RTOS 有效
#define UART_TASK_SLICE        10         // 时间片       RTThread 有效

static osal_task_t uart_rx_task = NULL;      // task 句柄

void start_uart_rx(char *dev_uart_name)
{
    if (!dev_uart_name)
        return;

    /* 使用 posix 函数打开串口 */
    fd_RxU = open(dev_uart_name, O_RDWR, NULL);             /* 默认 115200,8N1 */
    if (fd_RxU < 0)
    {
        printf("open device %s fail!\r\n", dev_uart_name);
        return;
    }

    /* 设置串口使用中断接收，确保不丢数据 */
    ioctl(fd_RxU, IOCTL_UART_SET_RXTX_MODE, (void *)UART_RX_INT);

    /* 使用 osal 创建 UART 接收任务 */
    uart_rx_task = osal_task_create("UARTrx",
                                    UART_STK_SIZE * sizeof(size_t),
                                    UART_TASK_PRIO, UART_TASK_SLICE,
                                    uart_receive_task, NULL);

    if (uart_rx_task == NULL)
    {
        close(fd_RxU);
        printf("create uart rx test task fail!\r\n");
     }
}
```

4、加入到 main()

修改 main.c 程序

```
/*
 * 主程序
 */
int main(void)
{
    printf("Hello world!\r\n");
    printf("Welcome to Loongson 2K300!\r\n");

    /**
     * 接收串口：uart6
     * 发送串口：uart4
     * 硬件上把两个串口的 rx、tx 交叉相连
     * 加入以下串口收发的代码
     */
    #if BSP_USE_UART4 && BSP_USE_UART6
    {
        void start_uart_rx(char *dev_uart_name);
        void start_uart_tx(char *dev_uart_name);

        start_uart_rx("/dev/uart6");            /* 注册的 I/O 设备名称 */
        start_uart_tx("/dev/uart4");            /* 注册的 I/O 设备名称 */
    }
    #endif

    // ..... 其它任务代码

    //-----------------------------------------------------
    // Bare-Metal Main Loop
    //-----------------------------------------------------
    for (;;)
    {
        pesudoos_run(0);      /* 运行 pesudoos 主循环，参数 0：执行一次返回 */

        /*
         * If you do more work here, don't use any pesudo-os functions.
         */
    }

    return 0;
}
```

第四节 编译调试

TBD。

第三章 提高篇

第一节 全局搜索路径与宏定义

在使用 gcc 编译 C 或 C++程序时，头文件的搜索路径是一个重要的概念。gcc 会按照一定的顺序搜索头文件，以确保能够找到所需的文件。以下是 gcc 搜索头文件的几种主要路径：

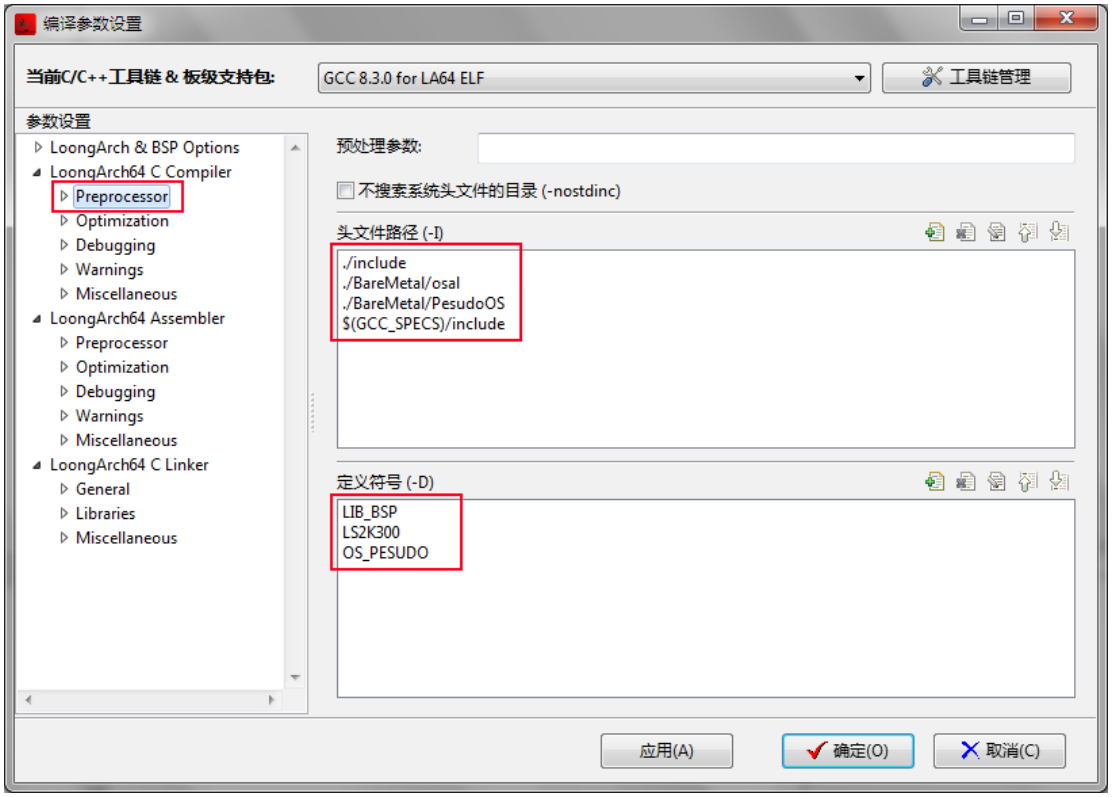
对于双引号 #include "header.h" 和尖括号 #include <header.h> 包含的头文件，gcc 的搜索路径有所不同。

双引号包含的头文件搜索顺序为：

- 当前文件所在的目录；
- 使用 **-I** 参数指定的路径；
- 环境变量 C_INCLUDE_PATH 或 CPLUS_INCLUDE_PATH 包含的路径；
- 内定路径（编译器内置的搜索路径，用户无法更改）

尖括号包含的头文件则不搜索当前文件所在的目录，直接从 **-I** 参数指定的路径开始。

项目打开时，使用菜单“**项目→编译选项**”或者快捷键“**F2**”打开“编译参数设置”窗口。



选中左侧参数树的“**C Compier→Preprocessor**”节点，窗口右侧显示用户设置的“**头文件搜索路径**”和“**定义符号**”，供 **loongarch64-newlib-elf-gcc.exe** 执行时使用。

此处设置的“搜索路径”和“宏定义”，**全局有效**。

1、头文件搜索路径

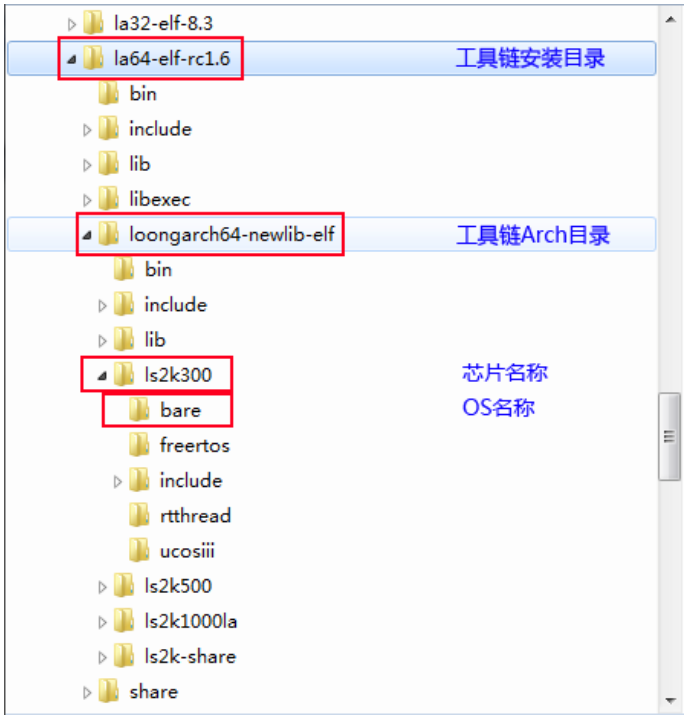
项目的目录：

例如：　./include　　// 需要头文件 bsp.h

　　./BareMetal/osal　　// 需要头文件 osal.h

　　./：表示相对于项目的根目录的路径

工具链下用户库文件的目录：



如果安装根目录：**d:/loongide2/la64-tool**

$(GCC_BASE)：　表示绝对路径：**d:/loongide2/la64-tool/loongarch64-newlib-elf**

$(CPU)：　　表示芯片名称：ls2k300、ls2k500、ls2k1000la

$(OS)：　　表示操作系统：bare、freertos、rtthread、ucosiii

$(GCC_SPECS)：等于：$(GCC_BASE)/$(CPU)

例如：　搜索路径$(GCC_SPECS)/include 的绝对路径是：

　　d:/loongide2/la64-tool/loongarch64-newlib-elf/ls2k300/include

此处设置的搜索路径，在项目编译时都是以 -I 参数供 gcc 搜索使用。

2、宏定义

芯片相关：　**LS2K300、LS2K500、LS2K1000LA**
OS 相关：　**OS_PESUDO、OS_RTTHREAD、OS_FREERTOS、OS_UCOS**

全局宏定义的作用是让同一套源代码适用不同芯片和操作系统。

在用户代码中使用：

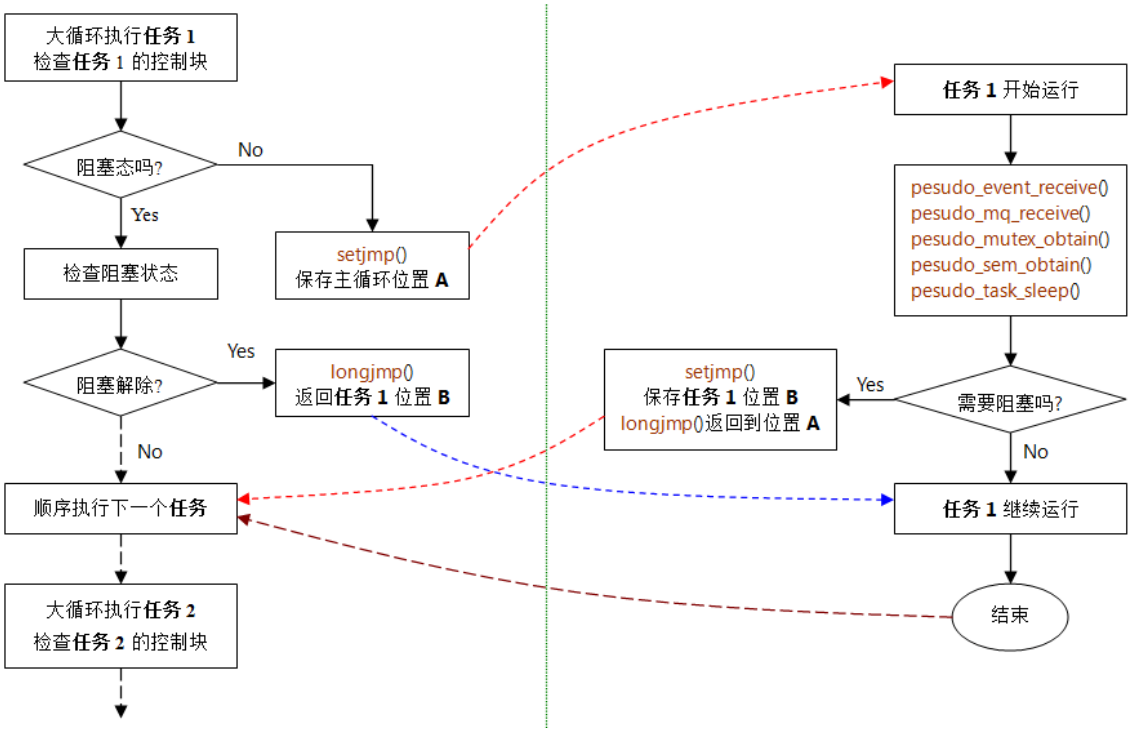
```
#if defined(LS2K300)
#include “ls2k300.h”
#elif defined(LS2K500)
#include “ls2k500.h”
#elif defined(LS2K1000LA)
#include “ls2k1000.h”
#else
#error “No Loongson SoC defined”
#endif
```

此处设置的宏定义，在项目编译时都是以 -D 参数供 gcc 搜索使用。

第二节 PesudoOS 编程

PesodoOS 将裸机大循环中执行的**函数作为任务**来顺序调用，PesodoOS 对每个**任务函数**分配一个**控制块**来判断任务函数的**状态**，以实现每个任务函数能够顺序执行。

PesudoOS 使用标准函数 **setjmp**()和 **longjmp**()来实现，其工作原理如下：



1、裸机主循环

项目中 main.c 中的 **main**() 函数实现裸机大循环：

```
//-------------------------------------------------
// Bare-Metal Main Loop
//-------------------------------------------------
for (;;)
{
    pesudoos_run(0);

    /*
     * If you do more work here, don't use any pesudo-os functions.
     */
}
```

函数定义：void **pesudoos_run**(int deadloop);

参数： deadloop 为 0 时， **pesudoos_run** 轮询一次任务控制块并执行需要执行的函数，然后退出，循环体内的其它函数得到一次执行机会；

deadloop 为 1 时， **pesudoos_run** 是个死循环，循环体内的其它函数没有运行机会。

2、任务函数是死循环

任务函数原型：`typedef void (*pesudo_task_entry_t)(void *arg);`

例如下面的 can0 发送函数（任务）是个死循环：

```
static void can0_transmit_task(void *arg)
{
    printk("can0 transmit task started...\r\n");

    for ( ; ; )
    {
        can0_do_transmit(NULL, NULL);

        /* abandon cpu time to run other task */
        osal_task_sleep(TX_DELAY_MS);
    }

    /* never go here... */
    close(fd_can0);
}
```

必须确保循环体内有以下函数的调用：

任务休眠	**osal_task_sleep**(); **osal_task_sleep_until**();	参数不为 0，一定阻塞而退出
延时	**osal_msleep**();	如果在任务中，调用 **osal_task_sleep()**函数，否则调用 **delay_ms**()函数
接收事件 获取信号量 获取互斥量 接收消息	**osal_event_receive**(); **osal_sem_obtain**(); **osal_mutex_obtain**(); **osal_mq_receive**();	如果操作不能引起阻塞、且整个函数执行过程中没有其它造成阻塞的调用，将造成死**循环一直执行、其它任务得不到运行机会**

3、延时问题

PesudoOS 编程中使用的**延时函数不是精确的：**

osal_task_sleep();	
osal_task_sleep_until();	
osal_msleep();	当调用 **delay_ms**()时是精确的

因为是顺序执行、而每个任务函数的执行时间可能是不定长的。

所以编程时需要对每个任务的执行时间进行预判（控制块有使用 ticks 的记录）、从而实现整个项目的优化。

第三节 驱动程序和 posix 接口

LoongIDE 1.x 版本是使用调用驱动程序的方式来使用设备；2.0 版本也可以直接使用驱动程序来进行程序设计。

以调用 CAN 设备为例来对比这两种方式：

	直接调用驱动	**posix 函数**	
头文件	#include "ls2k_can.h"	#include <fcntl.h> #include <unistd.h> #include "ls2k_can.h"	
变量	CANMsg_t *buf; int size, cmd, arg;	CANMsg_t *buf; int size, cmd, arg; int fd;	
初始化	**ls2k_can_init**(devCAN0, 0)	devCAN0 注册到文件系统时执行初始化	返回 0：成功
打开	**ls2k_can_open**(devCAN0, 0)	fd=**open**("/dev/can0", O_RDWR)	返回 0：成功
关闭	**ls2k_can_close**(devCAN0, 0)	**close**(fd);	返回 0：成功
读	**ls2k_can_read**(devCAN0, buf, size, 0)	**read**(fd, buf, size);	返回：读到字节数
写	**ls2k_can_write**(devCAN0, buf, size, 0)	**write**(fd, buf, size);	返回：写入字节数
控制	**ls2k_can_ioctl**(devCAN0, cmd, arg)	**ioctl**(fd, cmd, arg)	返回<0：不成功

posix 接口函数通过调用驱动程序对应函数来实现，中间经过了 I/O 设备文件系统的封装层，使 cpu、内存资源的开销略有增加，但带来的好处是源代码更具有移植性和可读性。

使用时需要注意以下两个带有“**可变参数**”的函数：

```
int open(const char *file, int flags, ...);      // 打开 I/O 设备不需要第三个参数
int ioctl(int fd, unsigned cmd, ...);            // 控制 I/O 设备需要第三个参数，严格按照驱动要求的类型
```

项目注册了哪些 I/O 设备，参阅项目“src/ ls2k_devices_register_fs.c”文件；

I/O 设备在文件系统中的名称，一是通过 shell 查看，二是查阅驱动源代码。

注：modbus 使用直接调用驱动程序的方式来实现。

第四节 目录和文件操作

LoongIDE 实现的文件系统，把 fat、ext、yaffs 这些不同格式、不同读写函数的文件系统统一到 posix 读写，从而简化和规范用户代码的编写。

1、access()函数

头文件： #include <sys/unistd.h>
函数原型： int **access**(const char *__path, int __amode);
作用： 用来检测文件系统中 __path 是否存在（没有使用__amode 参数）。

使用举例：
①、U 盘是动态插入的，如果编程需要读写 U 盘，可以使用 access()函数来判断磁盘是否存在

```
if (access("/usbd0", 0) == 0)
{
    // 检测到 U 盘存在，执行代码
}
```

②、简单判断目录或者文件是否存在

```
if (access("/hda/a1", 0) == 0)
{
    // 检测到 SATA 盘存在 a1，执行代码
}
```

但不能区分 a1 是文件还是目录。

2、stat()函数

头文件： #include <sys/stat.h>
函数原型： int **stat**(const char *__path, struct stat *__sbuf);
作用： 获取文件系统中 __path 的状态信息（写入*__sbuf）；返回 0：成功。

以下代码检查磁盘上目录、文件是否存在。

```
#include <sys/stat.h>

/* 检查目录是否存在，存在返回 1 */
int directory_exists(const char *pathname)
{
    struct stat st;
    if (stat(pathname, &st) == 0)
        return (st.st_mode & S_IFDIR) ? 1 : 0;
    return 0;
}
```

```
/* 检查文件是否存在，存在返回 1 */
int file_exists(const char *filename)
{
    struct stat st;
    if (stat(filename, &st) == 0)
        return (st.st_mode & S_IFREG) ? 1 : 0;
    return 0;
}
```

注：类似的函数 int fstat(int fd, struct stat *__sbuf)；是通过打开的文件句柄获取状态信息。

3、open()函数新建文件

open()、**close**()、**read**()、**write**()函数，可以操作 I/O 文件和磁盘文件。

使用 **open**()打开文件、**flags** 参数含有 O_CREAT 选项时（含义是如果文件不存在就新建），这时需要设置第三个参数（类型 mode_t，一般用八进制常数 0777）表示新建文件的“读写权限”。

例：fd = **open**(“/hda/dog.txt”, O_RDWR | O_CREAT, 0777);　　/* 需要给新建文件加上文件权限 */

O_RDWR、O_CREAT 等定义在 <fcntl.h>头文件中。

4、fopen()等函数

fopen()、**fread**()、**fwrite**()、**fclose**()是另一组磁盘文件创建、读、写函数；

```
FILE *f2;

/* 打开(创建)文件 */
f2 = fopen(filename, "w+");

/* 写文件 */
fwrite(wrbuf, 1, (size_t)n_wr, f2);

/* 重置读写位置 */
fseek(f2, 0, SEEK_SET);

/* 读文件 */
fread((void *)rdbuf, 1, 32, f2);

/* 关闭文件 */
fclose(f2);

/* 删除文件 */
unlink(filename);
```

5、目录操作函数

显示一个目录内容的函数：

```
//-------------------------------------------------------------------------
// list directory contents
//-------------------------------------------------------------------------

void list_dir(const char *path)
{
    DIR *dir;
    struct dirent *de;

    if (!directory_exists(path))
    {
        printf("Directory %s not exists\r\n", path);
        return;
    }

    printf("Contents of %s\r\n", path);

    dir = opendir(path);                    // 打开目录

    if (dir)
    {
        de = readdir(dir);                  // 读目录

        while (de)
        {
            if (de->d_attrs & S_IFDIR)
                printf("    <dir>     %s\r\n", de->d_name);
            else
                printf("              %s\r\n", de->d_name);

            de = readdir(dir);
        }

        printf("\r\n");
        closedir(dir);                      // 关闭目录

    }
    else
    {
        printf("    <empty>\r\n");
    }
}
```

创建目录：使用函数 int **mkdir**(const char *path, mode_t mode);

第五节 shell 命令行

1、命令列表

LS2K SDK 内置了以下 shell 命令：

命令	参数	功能
help	“ls”、”cd”、”copy”、”play”、”mkfs”	帮助
clear		清除控制台内容
set	“codec i2c0”、 “codec i2c3”	设置...
ls		列出内容
cd		改变文件系统当前目录
copy		复制文件
del	“文件名”	删除文件
rename	“原文件名 新文件名”	文件改名
mkdir	“目录名”	新建文件夹
rmdir	“目录名”	删除文件夹
mkfs	“磁盘名 文件系统格式 force”	在磁盘上创建文件系统， 参数：磁盘名：/mmc0、/usbd0 等 文件格式：fat32、ext4 force：强制执行
mount	“磁盘名”	挂载文件系统
unmount	“磁盘名”	卸载文件系统
task		列出所有线程
suspend	“taskID” taskID 有 task 命令列出	挂起线程
resume	“taskID” taskID 有 task 命令列出	恢复线程
stack		查看线程堆栈
play	“文件名.mp3”	播放 mp3 音频文件

所有使用的“文件名”、“目录名”，自动检查是否使用相对路径。

TBD：用户自定义命令的实现

2、ls 命令

ls 命令用于查看当前运行项目的信息。

参数	功能
(none)	列出**当前路径**下的内容，例如： 当前路径 "/"：　　输入：ls 回车 显示：/dev /mmc0 当前路径 "/dev"：　　输入：ls 回车 显示：/dev/uart0 /dev/uart4 ..... 当前路径 "/usbd0"：　输入：ls 回车 显示：/usbd0/下所有内容
"disk"	列出当前系统挂载了哪些磁盘
"dev"	列出当前系统挂载了哪些 I/O 设备
"fs"	列出当前系统挂载了哪些文件系统
"目录名"	列出指定目录下的所有内容
"-d 目录名"	列出指定目录下的所有目录
"-f 目录名"	列出指定目录下的所有文件
"task"	列出所有任务

3、cd 命令

cd 命令用于改变文件系统的当前目录，对应 posix 函数 **chdir**()和 **getcwd**()的目录。

参数	功能
..	根据当前路径返回上级目录，如果已经是系统根目录**"/"**，不变
"/diskname"	把指定磁盘根目录设置为当前路径
"/diskname/dir1"	把指定磁盘目录设置为当前路径
"dir2"	相对路径，把当前路径下的 **dir2** 设置为当前路径

4、copy 命令

copy 命令用于在磁盘目录间复制文件。

第一个参数：源文件名
第二个参数：目标文件名

文件名可使用绝对路径或者相对路径。

第六节 应用程序发布

应用程序在调试完成后，需要部署到板卡上、让 Bootloader 自动载入运行，实现既定板卡功能，这是项目开发的最终目的。LoongIDE 同时支持 PMON 和 U-Boot 下的应用程序发布，前提是 PMON 或者 U-Boot 支持所使用的读写命令。

与应用程序部署相关的配置在“开发板管理”窗口进行设置。

1、PMON

选择引导程序为 PMON，参数如下：

①、部署到设备上

常用设备有：/dev/mtd0、/dev/mtd1、/dev/disk/emmc0 etc

在 pmon 下使用命令 mtdparts 查看存在哪些 mtd 设备。

对于/dev/disk/xxx 这些设备，一般建有文件系统，**把应用程序直接部署到 disk 上会破坏文件系统**，这种情况使用“部署在文件系统上”来实现。

②、部署在文件系统上

板卡上存在 disk 且建有文件系统，此时把应用程序部署在文件系统上。

例如： LS2K300 的 PMON 只有 emmc0 上的 ext4 文件系统：

使用设备： /dev/fs/ext4@emmc0

LS2K1000LA 龙芯派的 PMON 只有 wd0 上的 ext4 文件系统：

使用设备： /dev/fs/ext4@wd0

③、环境变量

两个参数：在 SPI-Flash 的位置和大小。必须根据 PMON 来正确设置。

loongide 使用这两个参数设置 “set al”、“set al1” 命令。

注：PMON 自动载入的 loongarch64 可执行文件是经过 strip 的 elf 格式文件。

2、U-Boot

选择引导程序为 U-Boot，参数如下：

①、部署到设备上

- spi-flash 设备

  设备命令：sf 0　　0：spi-flash 的设备代码，第 0 个

  起始位置：0x20000　2M 位置

  表示：把应用程序部署在**第 0 个 spi-flash 的 2M 位置处**。

- mtd 设备

  设备命令：mtd xxx　　xxx：mtd 设备名称

  起始位置：无

  表示：把应用程序部署在 **mtd 设备 xxx 上面**。

- mmc 设备

  设备命令：mmc 0　　0：mmc 的设备代码，第 0 个

  起始位置：0x100000　16M 位置

  表示把应用程序部署在**第 0 个 mmc 的 16M 位置处**。

  注：如果 mmc 上建有文件系统，在 mmc 上部署必须确保存储空间在文件系统之外。

②、部署在文件系统上

板卡上存在 disk 且建有文件系统，此时把应用程序部署在文件系统上。

例如： LS2K300 的 U-Boot 只有 emmc0 上的 ext4 文件系统：

设备命令： ext4load emmc 0

③、环境变量

两个参数：在 SPI-Flash 的位置和大小。必须根据 U-Boot 来正确设置。

loongide 使用这两个参数设置 “bootcmd”命令。

注：U-Boot 自动载入的 loongarch64 可执行文件是 bin 格式文件。

部署应用程序必须让开发板处于 bootloader 命令行状态。

苏州市天晟软件科技有限公司

2025 年 3 月 18 日