龙芯 2K0500 编程参考手册

版本 1.0

苏州市天晟软件科技有限公司

www.loongide.com

2023 年 4 月

目录

前言

龙芯 2K 系列芯片是龙芯中科技术股份有限公司研发的 SoC 芯片，使用完全自主知识产权的 loongarch 架构；该芯片基于 LA264 内核、使用 loongarch64 指令集，片内集成了丰富的外围设备，芯片按照工业级标准生产，具有高性能、低功耗、完全自主可控的优势。芯片的详细技术参数请参考龙芯中科的官方用户手册。

LoongIDE 是专用于龙芯嵌入式芯片的集成开发环境，旨在为龙芯嵌入式芯片提供一个简单易用、稳定可靠、符合工业标准的嵌入式开发解决方案，帮助用户在龙芯嵌入式应用开发中缩短开发周期、简化开发难度，助力工控行业的国产化进程。LoongIDE 的使用请参考《龙芯嵌入式集成开发环境使用说明书》。

用户通过使用 LoongIDE 实现龙芯嵌入式芯片的“裸机/RTThread/uCOS/FreeRTOS/RTEMS”应用项目的**编程、编译和在线调试**，方便用户学习和掌握龙芯嵌入式芯片的开发流程，模拟和实现各种自动化、工业控制、数据采集、物联传感等应用场景，从而推动龙芯嵌入式芯片在工控行业的国产化应用。

本文档为 LoongIDE 提供的龙芯 LS2K0500 的设备驱动程序库提供编程参考，适用“裸机/RTThread/uCOS/FreeRTOS”四种编程环境。

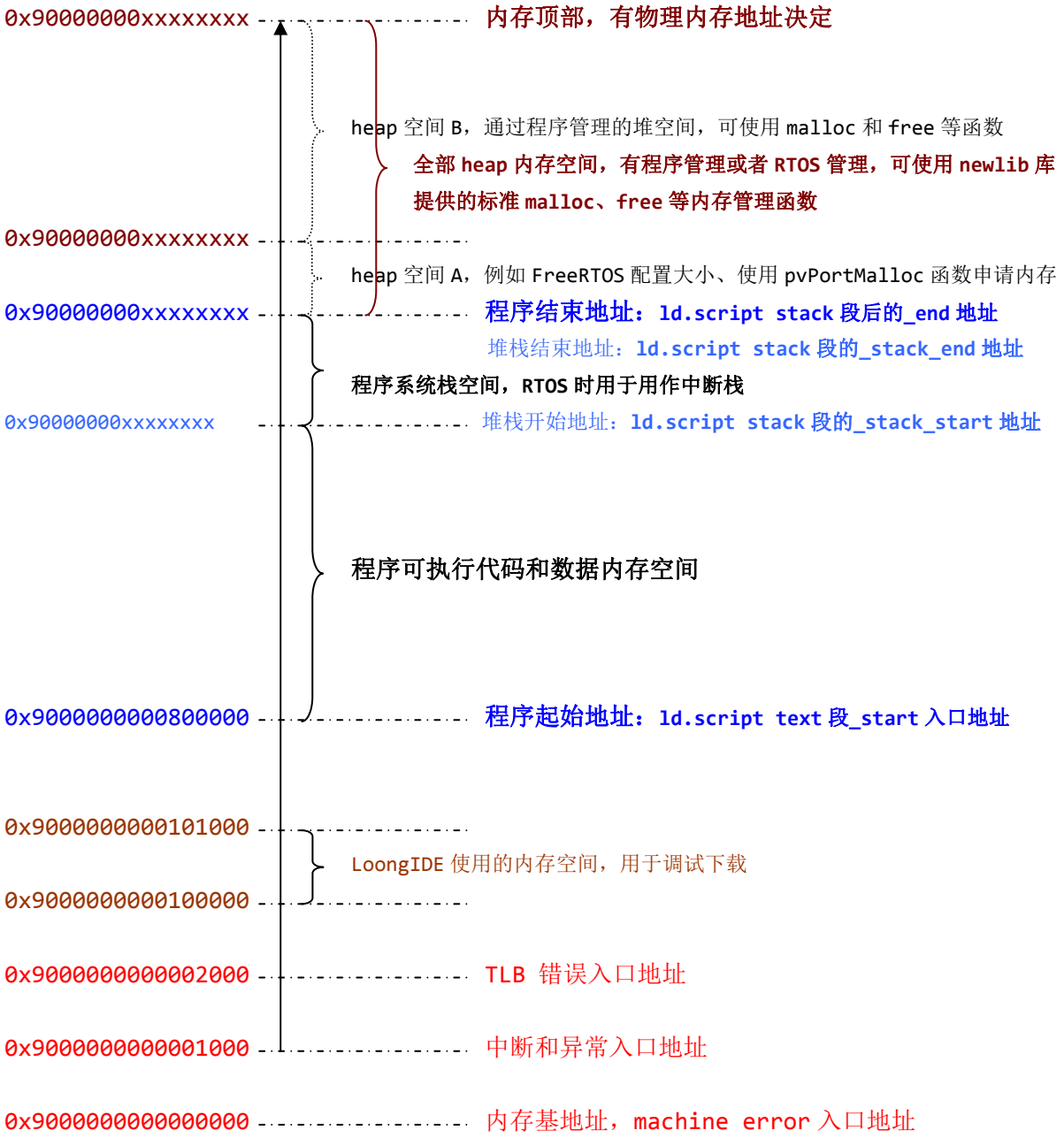
在使用 LoongIDE 进行编程前请阅读“帮助”文档

第一节 项目架构简介

1、内存分配

LoongIDE 创建的龙芯 2K0500 项目使用线性地址进行内存管理（MMU 工作在直接映射地址翻译模式，不使用 TLB），对应的内存地址范围是：0x9000000000000000 ~0x90000000xxxxxxxx；片上设备寄存器地址范围是：0x80000000xxxxxxxx。

使用 LoongIDE 创建的龙芯 2K0500 项目空间内存分配如下：



2、链接脚本

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。

链接脚本样例：

2.1 头部和代码段

```
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
    ...
```

2.2 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
    }
```

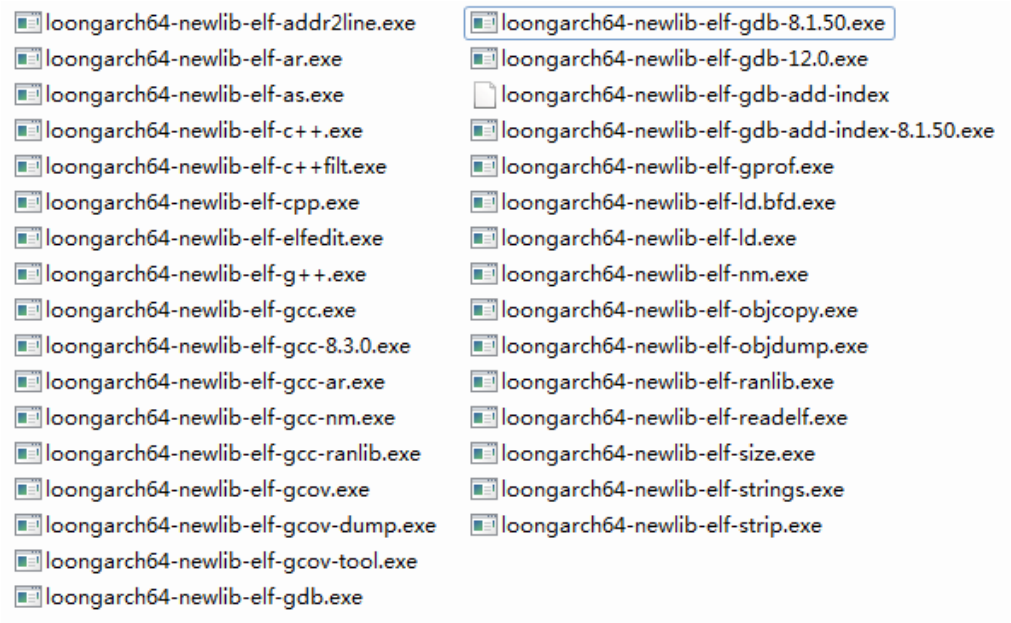
2.3 未初始化数据段和堆栈段

```
__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 = .);
```

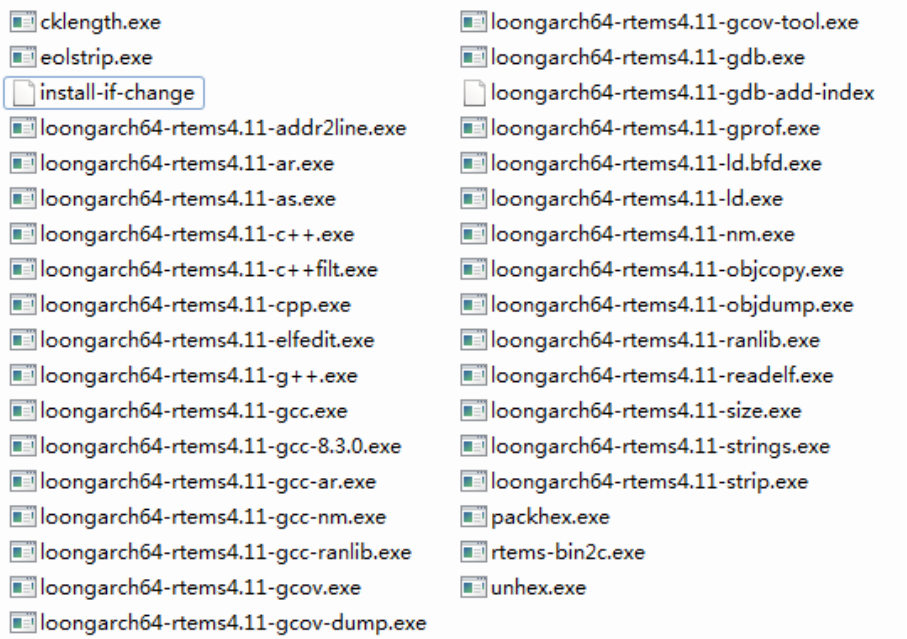
3、工具链

loongarch64 工具链使用龙芯中科官网（http://www.loongnix.cn）提供的 gcc 8.3.0 源码库、在 windows7-32 下使用 msys2 运行环境编译而成，并移植有 c 标准库 newlibc 2.2.0。

loongarch64 ELF 工具链：



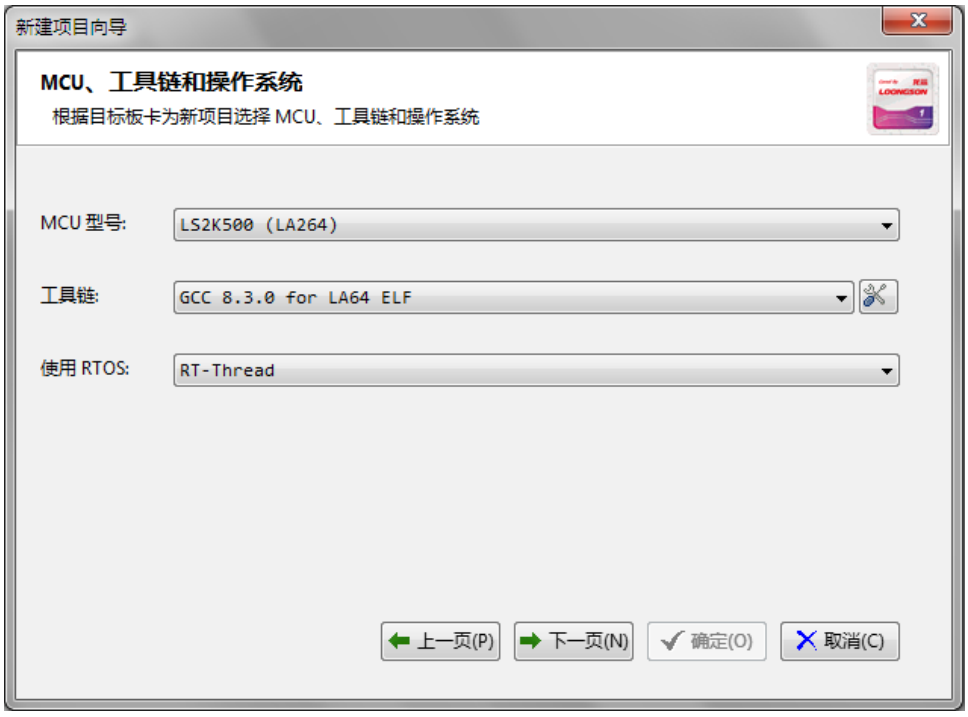
loongarch64 rtems4.11 工具链：



第二节 创建项目框架

1、项目向导

请参阅“帮助”文件“新建项目向导”章节。



注意：芯片型号的选择必须与开发板上的实际型号一致；
支持移植代码、第三方库的**静态库**选择和应用。

2、项目目录与文件

LoongIDE 为新建项目加入框架文件，如下图所示。

- RTThread：RTT 系统目录
  - components：RTT 系统组件
  - drv-glue：ls2k0500 驱动程序的 RTT 封装
  - include：RTT 系统头文件
  - port：ls2k0500 芯片的 RTT 移植文件，实现 tick、stack、cache、interrupt 等
  - src：RTT 内核文件

- ls2k500：芯片相关部分的实现
  - drivers：片上设备的**通用驱动程序等**
    - can // CAN 设备
    - console // 串口控制台
    - dc // 显示控制器
    - gmac // 网络控制器
    - gpio // 输入输出
    - hpet // 高精度定时器
    - i2c // IIC 控制器
    - include // 头文件
    - nand // NAND 控制器
    - pwm // PWM 控制器
    - rtc // 实时时钟
    - spi // SPI 控制器
    - uart // 串口
  - include：架构和芯片头文件
  - misc：内存管理、newlibc 适配模块

用户可创建基于 RTThread、FreeRTOS、uCOSII、RTEMS 和裸机编程的工程项目，除 RTEMS 使用已经编译好的静态库文件外，其余均可使用源代码或者静态库；因实现方式不同，生成的项目目录与文件略有区别。

第三节 配置 BSP

配置文件为 include/bsp.h，用户通过宏定义**选择需要使用的设备**。

例如：用户要使用 RTC 设备，必须打开宏定义：

```
#define BSP_USE_RTC     1
```

否则关闭宏定义：

```
#define BSP_USE_RTC     0
```

1、片上设备宏定义

宏定义	设备	相关模块	说明
#define BSP_USE_UART0 0	串口 0	ls2k_uart.c ls2k_uart.h	
#define BSP_USE_UART1 0	串口 1		
#define BSP_USE_UART2 1	串口 2		
#define BSP_USE_UART3 1	串口 3		
#define BSP_USE_UART4 0	串口 4		
#define BSP_USE_UART5 0	串口 5		
#define BSP_USE_UART6 0	串口 6		
#define BSP_USE_UART7 0	串口 7		
#define BSP_USE_UART8 0	串口 8		
#define BSP_USE_UART9 0	串口 9		
#define BSP_USE_SPI0 1	SPI 控制器 0	ls2k_spi_bus.c ls2k_spi_bus.h	
#define BSP_USE_SPI1 0	SPI 控制器 1		
#define BSP_USE_SPI2 0	SPI 控制器 2		
#define BSP_USE_SPI3 0	SPI 控制器 3		
#define BSP_USE_SPI4 0	SPI 控制器 4		
#define BSP_USE_SPI5 0	SPI 控制器 5		
#define BSP_USE_I2C0 0	I2C0 控制器 0	ls2k_i2c_bus.c ls2k_i2c_bus.h	
#define BSP_USE_I2C1 0	I2C1 控制器 1		
#define BSP_USE_I2C2 0	I2C2 控制器 2		
#define BSP_USE_I2C3 0	I2C2 控制器 3		
#define BSP_USE_I2C4 0	I2C2 控制器 4		
#define BSP_USE_I2C5 0	I2C2 控制器 5		
#define BSP_USE_NAND 0	NAND Flash 控制器	ls2k_nand.c ls2k_nand.h	
#define BSP_USE_RTC 0	RTC 设备	ls2k_rtc.c ls2k_rtc.h	
#define BSP_USE_HPET0 0	定时器设备 0	ls2k_hpet.c ls2k_hpet.h	
#define BSP_USE_HPET1 0	定时器设备 1		

#define BSP_USE_HPET2 0	定时器设备 2		
#define BSP_USE_HPET3 0	定时器设备 3		
#define BSP_USE_CAN0 0	CAN 控制器 0	ls2k_can.c ls2k_can.h	
#define BSP_USE_CAN1 0	CAN 控制器 1		
#define BSP_USE_CAN2 0	CAN 控制器 2		
#define BSP_USE_CAN3 0	CAN 控制器 3		
#define BSP_USE_GMAC0 1	GMAC 以太网控制器 0	ls2k_gmac.c ls2k_gmac.h	
#define BSP_USE_GMAC1 0	GMAC 以太网控制器 1		
#define BSP_USE_PWM0 0	PWM 控制器 0	ls2k_pwm.c ls2k_pwm.h	
#define BSP_USE_PWM1 0	PWM 控制器 1		
#define BSP_USE_PWM2 0	PWM 控制器 2		
#define BSP_USE_PWM3 0	PWM 控制器 3		
#define BSP_USE_PWM4 0	PWM 控制器 4		
#define BSP_USE_PWM5 0	PWM 控制器 5		
#define BSP_USE_PWM6 0	PWM 控制器 6		
#define BSP_USE_PWM7 0	PWM 控制器 7		
#define BSP_USE_PWM8 0	PWM 控制器 8		
#define BSP_USE_PWM9 0	PWM 控制器 9		
#define BSP_USE_PWM10 0	PWM 控制器 10		
#define BSP_USE_PWM11 0	PWM 控制器 11		
#define BSP_USE_PWM12 0	PWM 控制器 12		
#define BSP_USE_PWM13 0	PWM 控制器 13		
#define BSP_USE_PWM14 0	PWM 控制器 14		
#define BSP_USE_PWM15 0	PWM 控制器 15		
#define BSP_USE_DC 0	显示控制器	ls2k_dc.c ls2k_dc.h ls2k_fb_utils.c	

2、SPI0 总线上的从设备

要使用这些设备，必须先打开 SPI0 宏定义：

```
#define BSP_USE_SPI0     1
```

宏定义	设备	相关模块	说明
#if BSP_USE_SPI0 #define W25Q32_DRV 1 #endif	SPI Flash 芯片	w25q32.c w25q32.h	挂接在 spi0 上的设备

3、I2C0 总线上的从设备

要使用这些设备，必须打开 I2C0 宏定义：

```
#define BSP_USE_I2Cx     1
```

宏定义	设备	相关模块	说明
#ifdef BSP_USE_I2Cx 1 #define RC522_DRV 1 #endif	读卡器芯片	mfrc522.c mfrc522.h	挂接在 I2Cx 上的无线卡

4、其它

RTOS 配置：

宏定义	说明
#define BSP_USE_OS 1	是否使用 RTOS。驱动程序等通用于裸机的宏开关；参阅“编译参数”定义的 OS_RTTHREAD、OS_UCOS、OS_FREERTOS、OS_NONE 等宏定义。

外部函数：

定义	说明
extern void delay_us(int us);	延时微秒数
extern void delay_ms(int ms);	延时毫秒数；注意和 RTOS 延时函数的区别
extern void printk(const char *fmt, ...);	调试端口直接打印输出；功能和 rt_kprintf() 一致，有别于标准输出 printf()。

外部变量：

定义	说明
extern unsigned apb_frequency;	APB 总线时钟，用于 APB 设备的频率计算

第四节 配置 RTOS

当选用 RTOS 进行编程时，用户需要对操作系统进行配置。

- RT-Thread 的配置文件：RTThread/port/rtconfig.h
- FreeRTOS 的配置文件：FreeRTOS/port/FreeRTOSConfig.h
- uCOSII 的配置文件：uCOSIII/cfg/os_cfg.h 和 uCOSIII/cfg/os_cfg_app.h

用户根据所选 RTOS 进行裁剪。

如果使用静态库编程，不要修改配置文件，因为库文件编译时可能已经用到了该配置。

第五节 设备驱动程序

LoongIDE 实现的驱动程序在 ls2k500/drivers 目录下。

1、驱动模型

文件：ls2k500/drivers/include/ls2k_drv_io.h

中断句柄函数原型：

```
/*
 * 参数：   vector     中断编号
 *         arg        安装中断向量时传入的参数
 */
typedef void (* irq_handler_t)(int vector, void *arg);
```

宏定义：

```
#define PACK_DRV_OPS    1
```

用于选择是否**使用面向对象的方法封装**驱动程序。

1.1 通用驱动程序函数原型

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

#if (PACK_DRV_OPS)                          /* 驱动封装 */
typedef struct driver_ops
{
    driver_init_t    init_entry;            /* 设备初始化 */
    driver_open_t    open_entry;            /* 打开设备 */
    driver_close_t   close_entry;           /* 关闭设备 */
    driver_read_t    read_entry;            /* 读设备操作 */
    driver_write_t   write_entry;           /* 写设备操作 */
    driver_ioctl_t   ioctl_entry;           /* 设备控制 */
} driver_ops_t;
#endif
```

参数	用途
dev	待操作的目标设备，比如 devNAND/devUART3 etc
arg	各种类型的参数指针
buf	待读写数据缓冲区
size	待读写数据字节数
cmd	设备控制命令

对于芯片、开发板上设备，LoongIDE 实现了部分或者全部以上函数，分别用于该设备的**初始化、打开、关闭、读取、写入、控制**等操作。

一个设备在使用前需要执行 **initialize**，完成硬件初始化、创建 mutex、安装中断等的初始化工作；有些设备还需要执行 open 操作，才可以进行读写操作。

用户可以直接调用一个设备的驱动函数来对该设备进行操作。

如果设备是通过复用功能配置的，必需在 initialize 中执行初始化操作，在 LoongIDE 提供的标准驱动中可能没有实现该部分代码，用户需要根据板卡的实际硬件设计来进行初始化。

1.2 SPI、I2C 总线驱动函数原型

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

SPI 设备的驱动函原型和 I2C 一致：

```
typedef I2C_init_t          SPI_init_t;
typedef I2C_send_start_t    SPI_send_start_t;
typedef I2C_send_stop_t     SPI_send_stop_t;
typedef I2C_send_addr_t     SPI_send_addr_t;
typedef I2C_read_bytes_t    SPI_read_bytes_t;
typedef I2C_write_bytes_t   SPI_write_bytes_t;
typedef I2C_ioctl_t         SPI_ioctl_t;
```

```
#if (PACK_DRV_OPS)                          /* 驱动封装 */
typedef struct libi2c_ops
{
    I2C_init_t          init;               /* 初始化总线设备 */
    I2C_send_start_t    send_start;         /* 发送启动命令，独占总线 */
    I2C_send_stop_t     send_stop;          /* 发送结束命令，释放总线 */
    I2C_send_addr_t     send_addr;          /* 发送 I2C 设备地址 */
    I2C_read_bytes_t    read_bytes;         /* 从 I2C 设备读取数据 */
    I2C_write_bytes_t   write_bytes;        /* 向 I2C 设备写入数据 */
    I2C_ioctl_t         ioctl;              /* 发送控制命令 */
} libi2c_ops_t;
typedef libi2c_ops_t    libspi_ops_t;
#endif
```

参数	用途
bus	待操作的总线设备，busI2C0/busSPI0 etc
Addr	当操作 I2C 总线时，Addr 是 **I2C 芯片的 7 位 I2C 地址**
	当操作 SPI 总线时，Addr 是 **SPI 设备的片选序号**
rw	操作 I2C 总线时：1=读数据，0=写数据
bytes	待读写数据缓冲区
nbytes	待读写数据字节数
cmd	设备控制命令
arg	各种类型的参数指针

一根 SPI 总线、I2C 总线上，可能挂接有多个设备，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。

SPI、I2C 总线上挂接的芯片，在实现驱动程序时，通过调用总线驱动程序来实现。

芯片驱动调用总线驱动的一般顺序：

次序	总线驱动函数	芯片驱动实现的功能
1	**I2C_send_start**(bus, addr);	获取 I2C 总线的控制权
2	**I2C_ioctl**(bus, cmd, addr);	配置 I2C 设备的参数，以匹配芯片
3	**I2C_send_addr**(bus, addr, rw);	发送芯片的 I2C 地址
4	**I2C_write_bytes**(bus, buf, count);	执行读写操作，循环完成操作
	或者 **I2C_read_bytes**(bus, buf, count);	
5	**I2C_send_stop**(bus, addr);	释放 I2C 总线的控制权

SPI 总线挂接的芯片，驱动实现时的总线驱动调用顺序与上表相同。

用户编写新的 I2C/SPI 芯片驱动程序时，请参考 ls2k500/drivers/spi、ls2k500/drivers/i2c 目录下芯片驱动的实现方式。

2、串口设备

源代码：ls2k500/drivers/uart/ls2k_uart.c
头文件：ls2k500/drivers/include/ls2k_uart.h

串口设备是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_UART0    0
#define BSP_USE_UART1    0
#define BSP_USE_UART2    1                    // Console_Port 控制台串口
#define BSP_USE_UART3    1
#define BSP_USE_UART4    0
……
#define BSP_USE_UART9    0
```

串口设备的参数，在 ls2k_uart.c 中定义：

```
……

/* UART 2 */
#if BSP_USE_UART2
static NS16550_t ls2k_UART5 =
{
    .hwUART = (HW_NS16550_t *)PHYS_TO_UNCACHED(   // 串口寄存器基址
              UART2_BASE),
    .BaudRate    = 115200,                        // 默认速率
    .bFlowCtrl   = 0,
    .bInterrupt  = 1,                             // 是否使用中断
#if USE_EXTINT
    .irqVector   = EXTINTC0_UART2_IRQ,            // 扩展模式下的中断号
#else
    .irqVector   = INTC0_UART2_IRQ,               // 正常模式下的中断号
#endif
    .initialized = 0,                             // 是否初始化
    .opened       = 0,                            // 是否打开
    .dev_name    = "uart2",                       // 设备名称
};
void *devUART2 = &ls2k_UART2;
#endif
```

……

驱动程序 ls2k_uart.c 实现的函数：

函数
/* * 初始化串口 * 参数: dev 见上面定义的 UART 设备 * arg 类型 unsigned int, 串口波特率. 当该参数为 NULL 时, 串口设置为默认模式 115200,8N1 * * 返回: 0=成功 */ int NS16550_init(void *dev, void *arg);
/* * 打开串口. 如果串口配置为中断模式, 安装中断向量 * 参数: dev 见上面定义的 UART 设备 * arg 类型 struct termios *, 把串口配置为指定参数模式. 该参数可为 NULL. * * 返回: 0=成功 */ int NS16550_open(void *dev, void *arg);
/* * 关闭串口. 如果串口配置为中断模式, 移除中断向量 * 参数: dev 见上面定义的 UART 设备 * arg 总是 0 或 NULL. * * 返回: 0=成功 */ int NS16550_close(void *dev, void *arg);
/* * 从串口读数据(接收) * 参数: dev 见上面定义的 UART 设备 * buf 类型 char *, 用于存放读取数据的缓冲区 * size 类型 int, 待读取的字节数, 长度不能超过 buf 的容量 * arg 类型 int. * 如果串口工作在中断模式: * >0: 该值用作读操作的超时等待毫秒数 * =0: 读操作立即返回 * 如果串口工作在查询模式: * !=0: 读操作工作在阻塞模式, 直到读取 size 个字节才返回 * =0: 读操作立即返回 * * 返回: 读取的字节数 * * 说明: 串口工作在中断模式: 读操作总是读的内部数据接收缓冲区 * 串口工作在查询模式: 读操作直接对串口设备进行读 */ int NS16550_read(void *dev, void *buf, int size, void *arg);

```
/*
 * 向串口写数据(发送)
 * 参数:    dev     见上面定义的 UART 设备
 *          buf     类型 char *, 用于存放待发送数据的缓冲区
 *          size    类型 int, 待发送的字节数, 长度不超过 buf 的容量
 *          arg     总是 0 或 NULL
 *
 * 返回:    发送的字节数
 *
 * 说明:    串口工作在中断模式: 写操作总是写的内部数据发送缓冲区
 *          串口工作在查询模式: 写操作直接对串口设备进行写
 */
int NS16550_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向串口发送控制命令
 * 参数:    dev         见上面定义的 UART 设备
 *          cmd         IOCTL_NS16550_SET_MODE
 *          arg         类型 struct termios *, 把串口配置为指定参数模式.
 *
 * 返回:    0=成功
 */
int NS16550_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls2k_uart_drv_ops;
    #define ls2k_uart_init(uart, arg)     ls2k_uart_drv_ops->init_entry(uart, arg)
    #define ls2k_uart_open(uart, arg)     ls2k_uart_drv_ops->open_entry(uart, arg)
    #define ls2k_uart_close(uart, arg)    ls2k_uart_drv_ops->close_entry(uart, arg)
    #define ls2k_uart_read(uart, buf, size, arg) \
            ls2k_uart_drv_ops->read_entry(uart, buf, size, arg)
    #define ls2k_uart_write(uart, buf, size, arg) \
            ls2k_uart_drv_ops->write_entry(uart, buf, size, arg)
    #define ls2k_uart_ioctl(uart, cmd, arg) \
            ls2k_uart_drv_ops->ioctl_entry(uart, cmd, arg)
#else
    #define ls2k_uart_init(uart, arg)             NS16550_init(uart, arg)
    #define ls2k_uart_open(uart, arg)             NS16550_open(uart, arg)
    #define ls2k_uart_close(uart, arg)            NS16550_close(uart, arg)
    #define ls2k_uart_read(uart, buf, size, arg)  NS16550_read(uart, buf, size, arg)
    #define ls2k_uart_write(uart, buf, size, arg) NS16550_write(uart, buf, size, arg)
    #define ls2k_uart_ioctl(uart, cmd, arg)       NS16550_ioctl(uart, cmd, arg)
#endif
```

- 串口工作时使用中断模式或者查询模式，通过串口参数的定义域 bIntrrupt 来控制；
- 串口读写函数的 arg 参数为 0 时，读写立即返回；
- 串口读写函数的 arg 参数为非 0 时：
  在中断模式下，该值用作读等待超时毫秒数；
  在查询模式下，表示使用阻塞模式进行读操作；

控制台相关函数：

`extern void *ConsolePort;`	指定一个串口为控制台串口，用于 printk 函数的打印输出
`char Console_get_char(void *pUART);`	用于串口控制台读取一个字符
`void Console_output_char(void *pUART, char ch);`	用于串口控制台输出一个字符
`char *ls2k_uart_get_device_name(void *pUART);`	返回串口设备名称

注：如果一个串口是通过复用来实现的，需要在 NS16550_init 中增加相关初始化操作；
控制台串口使用查询模式；

3、SPI 设备

源代码：ls2k500/drivers/spi/ls2k_spi_bus.c
头文件：ls2k500/drivers/include/ls2k_spi_bus.h

SPI 设备是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_SPI0     1                                  仅 SPI0 和 SPI1 有 spi flash 功能
#define BSP_USE_SPI1     0
#define BSP_USE_SPI2     0
#define BSP_USE_SPI3     0
#define BSP_USE_SPI4     0
#define BSP_USE_SPI5     0
```

SPI 设备的参数，在 ls2k_spi_bus.c 中定义：

```
#if BSP_USE_SPI0
static SPI_bus_t ls2k_SPI0 =
{
    .hwSPI        = (HW_SPI_t *)PHYS_TO_UNCACHED(       /* 设备基地址 */
                    SPI0_BASE),
#if USE_EXTINT
    .irqNum       = EXTINTC2_SPI0_IRQ,                  /* 扩展模式下的中断号 */
#else
    .irqNum       = INTC1_SPI0_IRQ,                     /* 正常模式下的中断号 */
#endif
    .base_frq     = 0,                                  /* 总线频率，初始化填充 */
    .chipsel_nums = 4,                                  /* 片选总数 */
     .chipsel_high = 0,                                 /* 片选低有效 */
    .dummy_char   = 0,
    .initialized  = 0,                                  /* 是否初始化 */
    .dev_name     = "spi0",                             /* 设备名称 */
};
void *busSPI0 = &ls2k_SPI0;                             /* 总线名 */
#endif

……
```

驱动程序 ls2k_spi_bus.c 实现的函数：

函数
/* * 初始化SPI总线 * 参数: bus busSPI0……busSPI5 * * 返回: 0=成功 * * 说明: SPI总线在使用前，必须要先调用该初始化函数 */ int SPI_initialize(void *bus);
/* * 开始SPI总线操作. 本函数获取SPI总线的控制权 * 参数: bus busSPI0……busSPI5 * Addr 片选. 取值范围0~3, 表示将操作SPI总线上挂接的某个从设备 * * 返回: 0=成功 */ int SPI_send_start(void *bus, unsigned int Addr);
/* * 结束SPI总线操作. 本函数释放SPI总线的控制权 * 参数: bus busSPI0……busSPI5 * Addr 片选. 取值范围0~3, 表示将操作SPI总线上挂接的某个从设备 * * 返回: 0=成功 */ int SPI_send_stop(void *bus, unsigned int Addr);
/* * 读写SPI总线前发送片选信号 * 参数: bus busSPI0……busSPI5 * Addr 片选. 取值范围0~3, 表示将操作SPI总线上挂接的某个从设备 * rw 未使用 * * 返回: 0=成功 */ int SPI_send_addr(void *bus, unsigned int Addr, int rw);
/* * 从SPI从设备读取数据 * 参数: bus busSPI0……busSPI5 * buf 类型 unsigned char *, 用于存放读取数据的缓冲区 * len 类型 int, 待读取的字节数, 长度不能超过 buf 的容量 * * 返回: 本次读操作的字节数 */ int SPI_read_bytes(void *bus, unsigned char *rxbuf, int len);
/* * 向SPI从设备写入数据 * 参数: bus busSPI0……busSPI5 * buf 类型 unsigned char *, 用于存放待写数据的缓冲区 * len 类型 int, 待写的字节数, 长度不能超过 buf 的容量

```
 *
 * 返回:    本次写操作的字节数
 */
int SPI_write_bytes(void *bus, unsigned char *txbuf, int len);
```

```
/*
 * 向SPI总线发送控制命令
 * 参数:    bus     busSPI0……busSPI5
 *      ------------------------------------------------------------------------------
 *          cmd                             |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_SPI_I2C_SET_TFRMODE           |   类型: SPI_mode_t *
 *                                          |   用途: 设置SPI总线的通信模式
 *      ------------------------------------------------------------------------------
 *      IOCTL_FLASH_FAST_READ_ENABLE        |   NULL, 设置SPI控制器为 Flash快速读模式
 *      ------------------------------------------------------------------------------
 *      IOCTL_FLASH_FAST_READ_DISABLE       |   NULL, 取消SPI控制器的 Flash快速读模式
 *      ------------------------------------------------------------------------------
 *      IOCTL_FLASH_GET_FAST_READ_MODE      |   类型: unsigned int *
 *                                          |   用途: 读取SPI控制器是否处于 Flash快速读模式
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 *
 * 说明:    该函数调用的时机是: SPI设备已经初始化且空闲, 或者已经获取总线控制权
 */
int SPI_ioctl(void *bus, int cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern libspi_ops_t *spi_drv_ops;
    #define ls2k_spi_initialize(spi)              spi_drv_ops->init(spi)
    #define ls2k_spi_send_start(spi, addr)        spi_drv_ops->send_start(spi, addr)
    #define ls2k_spi_send_stop(spi, addr)         spi_drv_ops->send_stop(spi, addr)
    #define ls2k_spi_send_addr(spi, addr, rw)     spi_drv_ops->send_addr(spi, addr, rw)
    #define ls2k_spi_read_bytes(spi, buf, len)    spi_drv_ops->read_bytes(spi, buf, len)
    #define ls2k_spi_write_bytes(spi, buf, len)   spi_drv_ops->write_bytes(spi, buf, len)
    #define ls2k_spi_ioctl(spi, cmd, arg)         spi_drv_ops->ioctl(spi, cmd, arg)
#else
    #define ls2k_spi_initialize(spi)              SPI_initialize(spi)
    #define ls2k_spi_send_start(spi, addr)        SPI_send_start(spi, addr)
    #define ls2k_spi_send_stop(spi, addr)         SPI_send_stop(spi, addr)
    #define ls2k_spi_send_addr(spi, addr, rw)     SPI_send_addr(spi, addr, rw)
    #define ls2k_spi_read_bytes(spi, buf, len)    SPI_read_bytes(spi, buf, len)
    #define ls2k_spi_write_bytes(spi, buf, len)   SPI_write_bytes(spi, buf, len)
    #define ls2k_spi_ioctl(spi, cmd, arg)         SPI_ioctl(spi, cmd, arg)
#endif
```

实用函数

设置 SPI0_CS0/ SPI1_CS0 的 SPI FLASH 控制器的快速读模式函数：

函数
/* * 设置SPI控制器为 Flash快速读模式 * 参数: bus busSPI0/busSPI1 */ **int ls2k_spiflash_fastread_enable(void *pSPI);**
/* * 取消SPI控制器的 Flash快速读模式 * 参数: bus busSPI0/busSPI1 */ **int ls2k_spiflash_fastread_disable(void *pSPI);**

3.1 NorFlash 芯片 W25Q32

源代码：ls2k500/drivers/spi/w25q32/w25q32.c
头文件：ls2k500/drivers/include/spi/w25q32.h

W25Q32 是否使用，在 bsp.h 中配置宏定义：

```
#if BSP_USE_SPI0
#define W25Q32_DRV    1
#endif
```

W25Q32 连接在 **SPI0 的片选 0**，在 w25q32.c 中定义：

```
#define W25Q32_CS       0
```

W25Q32 的参数，在 w25q32.c 中定义：

芯片参数：

```
static W25Q32_param_t m_chipParam =
{
    .baudrate            = SPI_FLASH_BAUDRATE,          /* 最大速率 */
    .erase_before_program = true,
    .empty_state          = 0xff,
    .page_size            = SPI_FLASH_PAGE_SIZE,         /* 页大小 */
    .sector_size          = SPI_FLASH_BLOCK_SIZE,        /* 块大小 */
    .mem_size             = SPI_FLASH_CHIP_SIZE,         /* 总容量 */
};
```

通信参数：

```
static SPI_mode_t m_devMode =
{
    .baudrate = 10000000,                                /* 通信速率 10M */
    .bits_per_char = SPI_FLASH_BITSPERCHAR,              /* 通信字节的位数 */
    .lsb_first = false,                                  /* 低位先发送 */
    .clock_pha = true,                                   /* spi 时钟相位 */
    .clock_pol = true,                                   /* spi 时钟极性 */
    .clock_inv = true,                                   /* true: 片选低有效 */
    .clock_phs = false,                                  /* true: spi 接口模式，时钟
};                                                       与数据发送同步 */
```

驱动程序 w25q32.c 实现的函数：

函数
/* * 初始化W25Q32芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25Q32_init(void *dev, void *arg);
/* * 打开W25Q32芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25Q32_open(void *dev, void *arg);
/* * 关闭W25Q32芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25Q32_close(void *dev, void *arg);
/* * 从W25Q32芯片读数据 * 参数: dev busSPI0 * buf 类型: unsigned char *, 用于存放读取数据的缓冲区 * size 类型: int, 待读取的字节数, 长度不能超过 buf 的容量 * arg 类型: unsigned int *, 读flash的起始地址(W25Q32内部地址从0开始进行线性编址) * * 返回: 读取的字节数 */ int W25Q32_read(void *dev, void *buf, int size, void *arg);
/* * 向W25Q32芯片写数据 * 参数: dev busSPI0 * buf 类型: unsigned char *, 用于存放待写数据的缓冲区 * size 类型: int, 待写入的字节数, 长度不能超过 buf 的容量 * arg 类型: unsigned int *, 写flash的起始地址(W25Q32内部地址从0开始进行线性编址) * * 返回: 写入的字节数 * * 说明: 待写入的W25Q32块已经格式化 */ int W25Q32_write(void *dev, void *buf, int size, void *arg);

```
/*
 * 向总线/W25Q32芯片发送控制命令
 * 参数:    dev     busSPI0
 *
 *      ------------------------------------------------------------------------------
 *         cmd                          |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_FLASH_FAST_READ_ENABLE    |   NULL. 开启SPI总线的FLASH快速读模式
 *      ------------------------------------------------------------------------------
 *      IOCTL_FLASH_FAST_READ_DISABLE   |   NULL. 停止SPI总线的FLASH快速读模式
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_READ_ID            |   类型: unsigned int *
 *                                      |   用途: 读取W25Q32芯片的ID
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_ERASE_4K           |   类型: unsigned int
 *      IOCTL_W25Q32_SECTOR_ERASE       |   用途: 擦除该地址所在的4K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_ERASE_32K          |   类型: unsigned int
 *                                      |   用途: 擦除该地址所在的32K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_ERASE_64K          |   类型: unsigned int
 *                                      |   用途: 擦除该地址所在的64K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_BULK_ERASE         |   NULL, 擦除整块flash芯片
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25Q32_IS_BLANK           |   NULL, 检查是否为空
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int W25Q32_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *w25q32_drv_ops;
    #define ls2k_w25q32_init(spi, arg)    w25q32_drv_ops->init_entry(spi, arg)
    #define ls2k_w25q32_open(spi, arg)    w25q32_drv_ops->open_entry(spi, arg)
    #define ls2k_w25q32_close(spi, arg)   w25q32_drv_ops->close_entry(spi, arg)
    #define ls2k_w25q32_read(spi, buf, size, arg) \
            w25q32_drv_ops->read_entry(spi, buf, size, arg)
    #define ls2k_w25q32_write(spi, buf, size, arg) \
            w25q32_drv_ops->write_entry(spi, buf, size, arg)
    #define ls2k_w25q32_ioctl(spi, cmd, arg)  \
            w25q32_drv_ops->ioctl_entry(spi, cmd, arg)
#else
    #define ls2k_w25q32_init(spi, arg)            W25Q32_init(spi, arg)
    #define ls2k_w25q32_open(spi, arg)            W25Q32_open(spi, arg)
    #define ls2k_w25q32_close(spi, arg)           W25Q32_close(spi, arg)
    #define ls2k_w25q32_read(spi, buf, size, arg) W25Q32_read(spi, buf, size, arg)
    #define ls2k_w25q32_write(spi, buf, size, arg) W25Q32_write(spi, buf, size, arg)
```

```
    #define ls2k_w25q32_ioctl(spi, cmd, arg)       W25Q32_ioctl(spi, cmd, arg)
#endif
```

注：W25Q32 芯片内部写有 PMON，用作开发板的 bootloader，占用空间约 790K；
用户可以使用的地址空间必须在 bootloader 之上，例如 800K~4M。

ls2k_w25q32_read 和 ls2k_w25q32_write 函数的 **arg** 参数：

W25Q32 的片内 FLASH 地址空间从 0 到 4M 进行编址，入口时 arg 参数为 FLASH 地址空间内的读写起始偏移量（类型：unsigned int *）。

如果返回负值，本次操作失败。

ls2k_w25q32_ioctl 函数的 **arg** 参数：

根据不同的 cmd 有不同的数据类型，使用时请参照 ls2k_w25q32_ioctl 函数的实现。

4、I2C 设备

源代码：ls2k500/drivers/i2c/ls2k_i2c_bus.c
头文件：ls2k500/drivers/include/ls2k_i2c_bus.h

I2C 设备是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_I2C0    1
#define BSP_USE_I2C1    0
#define BSP_USE_I2C2    0
#define BSP_USE_I2C3    0
#define BSP_USE_I2C4    0
#define BSP_USE_I2C5    0
```

I2C 设备的参数，在 ls2k_i2c_bus.c 中定义：

```
#if BSP_USE_I2C0
static I2C_bus_t ls2k_I2C0 =
{
    .hwI2C      = (HW_I2C_t *)PHYS_TO_UNCACHED(      /* 设备基地址 */
                  I2C0_BASE),
#if USE_EXTINT
    .irqNum     = EXTINTC0_I2C0_IRQ,                 /* 扩展模式下的中断号 */
#else
    .irqNum     = INTC0_I2C0_IRQ,                    /* 正常模式下的中断号 */
#endif
    .base_frq   = 0,                                 /* 总线频率 */
    .baudrate   = 100000,                            /* 通信速率 */
    .dummy_char = 0,
    .initialized = 0,                                /* 是否初始化 */
    .dev_name   = "i2c0",                            /* 设备名称 */
};
void *busI2C0 = &ls2k_I2C0;
#endif
……
```

驱动程序 ls2k_i2c_bus.c 实现的函数：

函数
/* * 初始化I2C总线 * 参数: bus busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5 * * 返回: 0=成功 * * 说明: I2C总线在使用前，必须要先调用该初始化函数 */ int I2C_initialize(void *bus);
/* * 开始I2C总线操作. 本函数获取I2C总线的控制权 * 参数: bus busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5 * Addr 总线bus上挂接的某个I2C从设备的 7 位I2C地址 * * 返回: 0=成功 */ int I2C_send_start(void *bus, unsigned int Addr);
/* * 结束I2C总线操作. 本函数释放I2C总线的控制权 * 参数: bus busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5 * Addr 总线bus上挂接的某个I2C从设备的 7 位I2C地址 * * 返回: 0=成功 */ int I2C_send_stop(void *bus, unsigned int Addr);
/* * 读写I2C总线前发送读写请求命令 * 参数: bus busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5 * Addr 总线bus上挂接的某个I2C从设备的 7 位I2C地址 * rw 1: 读操作; 0: 写操作. * * 返回: 0=成功 */ int I2C_send_addr(void *bus, unsigned int Addr, int rw);
/* * 从I2C从设备读取数据 * 参数: bus busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5 * buf 类型 unsigned char *, 用于存放读取数据的缓冲区 * len 类型 int, 待读取的字节数, 长度不能超过 buf 的容量 * * 返回: 本次读操作的字节数 */ int I2C_read_bytes(void *bus, unsigned char *rxbuf, int len);

```
/*
 * 向I2C从设备写入数据
 * 参数:    bus     busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5
 *          buf     类型 unsigned char *, 用于存放待写数据的缓冲区
 *          len     类型 int, 待写的字节数, 长度不能超过 buf 的容量
 *
 * 返回:    本次写操作的字节数
 */
int I2C_write_bytes(void *bus, unsigned char *txbuf, int len);
```

```
/*
 * 向I2C总线发送控制命令
 * 参数:    bus     busI2C0/busI2C1/busI2C2/busI2C3/busI2C4/busI2C5
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_SPI_I2C_SET_TFRMODE   |   类型: unsigned int
 *                                  |   用途: 设置I2C总线的通信速率
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 *
 * 说明:    该函数调用的时机是: I2C设备已经初始化且空闲, 或者已经获取总线控制权
 */
int I2C_ioctl(void *bus, int cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern libi2c_ops_t *i2c_drv_ops;
    #define ls2k_i2c_initialize(i2c)              i2c_drv_ops->init(i2c)
    #define ls2k_i2c_send_start(i2c, addr)        i2c_drv_ops->send_start(i2c, addr)
    #define ls2k_i2c_send_stop(i2c, addr)         i2c_drv_ops->send_stop(i2c, addr)
    #define ls2k_i2c_send_addr(i2c, addr, rw)     i2c_drv_ops->send_addr(i2c, addr, rw)
    #define ls2k_i2c_read_bytes(i2c, buf, len)    i2c_drv_ops->read_bytes(i2c, buf, len)
    #define ls2k_i2c_write_bytes(i2c, buf, len)   i2c_drv_ops->write_bytes(i2c, buf, len)
    #define ls2k_i2c_ioctl(i2c, cmd, arg)         i2c_drv_ops->ioctl(i2c, cmd, arg)
#else
    #define ls2k_i2c_initialize(i2c)              I2C_initialize(i2c)
    #define ls2k_i2c_send_start(i2c, addr)        I2C_send_start(i2c, addr)
    #define ls2k_i2c_send_stop(i2c, addr)         I2C_send_stop(i2c, addr)
    #define ls2k_i2c_send_addr(i2c, addr, rw)     I2C_send_addr(i2c, addr, rw)
    #define ls2k_i2c_read_bytes(i2c, buf, len)    I2C_read_bytes(i2c, buf, len)
    #define ls2k_i2c_write_bytes(i2c, buf, len)   I2C_write_bytes(i2c, buf, len)
    #define ls2k_i2c_ioctl(i2c, cmd, arg)         I2C_ioctl(i2c, cmd, arg)
#endif
```

4.1 无线读卡器芯片 MFRC522

源代码：ls2k500/drivers/i2c/rc522/mfrc522.c

头文件：ls2k500/drivers/include/i2c/mfrc522.h

MFRC522 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_I2C0   1
#if BSP_USE_I2C0
#define RC522_DRV      1
#endif
```

MFRC522 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（mfrc522.c）：

```
#define MFRC522_ADDRESS         0x28            /* 7 位地址 */
#define MFRC522_BAUDRATE        100000          /* 100K */
```

没有按照标准驱动格式实现 mfrc522.c 的驱动程序。

rtthread 项目使用 mfrc522 的例子：

```
/*
 * 文件 main.c
 */
#include <stdio.h>
#include <rtthread.h>
#include "bsp.h"
#include "ls2k_i2c_bus.h"
#include "i2c/mfrc522.h"

/*
 * mfrc522 线程
 */

static rt_thread_t m_rc522_thread = NULL;

static void rc522_thread(void *arg)
{
    unsigned int tickcount;

    ls2k_i2c_initialize(busI2C0);          /* 初始化 i2c0 总线 */
    PCD_Init(busI2C0);                     /* 初始化 MFRC522 */

    for ( ; ; )
    {
        tickcount = rt_tick_get();
        rt_kprintf("rc522 thread tick count = %i\r\n", tickcount);
```

```
            /* Look for new cards
             */
            if (PICC_IsNewCardPresent(busI2C0))
            {
                printk("new card.\n");

                // Select one of the cards
                if (PICC_ReadCardSerial(busI2C0))
                {
                    MIFARE_Key_t mfKey;
                    unsigned char i, status, piccType;

                    // Print Card UID
                    printk("card id:");
                    for (i = 0; i < uid.size; i++)
                    {
                        printk(" %02X", uid.uidByte[i]);
                    }
                    printk("\n");

                    PICC_Dump(busI2C0, &uid);

                    // Print Card type
                    piccType = PICC_GetType(uid.sak);
                    printk("PICC Type: %s \n", PICC_GetTypeName(piccType));

                    memset((void *)&mfKey, 0xFF, sizeof(MIFARE_Key_t));
                    // Authenticate Card
                    status = PCD_Authenticate(busI2C0,
                                              0x61/*PICC_CMD_MF_AUTH_KEY_B*/,
                                              6,
                                              &mfKey,
                                              &uid);
                    if (0/*STATUS_OK*/ == status)
                    {
                        unsigned char rbuf[18], bufsize=18;

                        printk("auth ok.\n");
                        status = MIFARE_Read(busI2C0, 7, rbuf, &bufsize);
                        if (0/*STATUS_OK*/ == status)
                        {
                            printk("read ok.\n");
                            PICC_HaltA(busI2C0);
                        }
                    }
                }
            }

            rt_thread_delay(300);
        }
    }
```

```c
/*
 * 主程序
 */
int main(void)
{
    printk("Hello world!\r\n");
    printk("Welcome to Loongson 2K0500!\r\n\r\n");

    /*
     * MFRC522 Task initializing...
     */
    m_rc522_thread = rt_thread_create("rc522thread",
                                      rc522_thread,
                                      NULL,         // arg
                                      1024*4,       // statck size
                                      11,           // priority
                                      10);          // slice ticks

    if (m_rc522_thread == NULL)
    {
        rt_kprintf("create rc522 thread fail!\r\n");
    }
    else
    {
        rt_thread_startup(m_rc522_thread);
    }

    rt_kprintf("main() is exit!\r\n");

    /*
     * Finsh as another thread...
     */
    return 0;
}
```

4.2 ADC 芯片 ADS1015

源代码：ls2k500/drivers/i2c/ads1015/ads1015.c

头文件：ls2k500/drivers/include/i2c/ads1015.h

ADS1015 是否使用，在 bsp.h 中配置宏定义：

```
#define ADS1015_DRV    1
```

ADS1015 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（ads1015.c）：

```
#define ADS1015_ADDRESS         0x48            /* 7 位地址 */
#define ADS1015_BAUDRATE        100000          /* 100K */
```

驱动程序 ads1015.c 实现的函数：

函数

```
/*
 * 从ADS1015读取 2 字节ADC转换值
 * 参数:   bus       busI2C0
 *         buf       类型: uint16_t *
 *         arg       ADS1015_CHANNEL_D0: 差分输入 P=AIN0, N=AIN1
 *                   ADS1015_CHANNEL_D1: 差分输入 P=AIN0, N=AIN3
 *                   ADS1015_CHANNEL_D2: 差分输入 P=AIN1, N=AIN3
 *                   ADS1015_CHANNEL_D3: 差分输入 P=AIN2, N=AIN3
 *                   ADS1015_CHANNEL_S0: 单端输入 AIN0
 *                   ADS1015_CHANNEL_S1: 单端输入 AIN1
 *                   ADS1015_CHANNEL_S2: 单端输入 AIN2
 *                   ADS1015_CHANNEL_S3: 单端输入 AIN3
 *
 * 返回:   0=成功
 */
int ADS1015_read(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 控制ADS1015芯片
 *
 * 参数:   bus       busI2C0
 *   ------------------------------------------------------------------------------------
 *         cmd                             | arg
 *   ------------------------------------------------------------------------------------
 *    IOCTL_ADS1015_SET_CONV_CTRL          | uint16_t, set convert config register
 *    IOCTL_ADS1015_GET_CONV_CTRL          | uint16_t *, get convert config register
 *    IOCTL_ADS1015_SET_LOW_THRESH         | uint16_t, set low thresh register
 *    IOCTL_ADS1015_GET_LOW_THRESH         | uint16_t *, get low thresh register
 *    IOCTL_ADS1015_SET_HIGH_THRESH        | uint16_t, set high thresh register
 *    IOCTL_ADS1015_GET_HIGH_THRESH        | uint16_t *, get high thresh register
 *    IOCTL_ADS1015_DISP_CONFIG_REG        | NULL, display config register
 *   ------------------------------------------------------------------------------------
 *
 * 返回:   0=成功
 */
int ADS1015_ioctl(void *dev, int cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ads1015_drv_ops;
    #define ls2k_ads1015_read(iic, buf, size, arg) \
            ads1015_drv_ops->read_entry(iic, buf, size, arg)
    #define ls2k_ads1015_ioctl(iic, cmd, arg) \
            ads1015_drv_ops->ioctl_entry(iic, cmd, arg)
#else
    #define ls2k_ads1015_read(iic, buf, size, arg)     ADS1015_read(iic, buf, size, arg)
    #define ls2k_ads1015_ioctl(iic, cmd, arg)          ADS1015_ioctl(iic, cmd, arg)
#endif
```

ADS1015_read：读取当前 ADC 转换结果，buf 是 **unsigned short** 类型。

ADS1015_ioctl：设置转换模式。

实用函数：

函数	功能
/* * 参数: bus busI2C0 * channel see ADS1015_read()'s arg */ uint16_t get_ads1015_adc(void *bus, int channel);	读一个通道的 ADC 值

编程示例：

```
uint16_t val;
float vin;
val = ls2k_ads1015_adc(busI2C0, ADS1015_REG_CONFIG_MUX_SINGLE_0); // 读通道 0
vin = 0.002 * val;      /* 0.002 是电压值校正系数 */
printf("ADS1015_IN0 = 0x%04X, voltage=%6.3f\r\n\r\n", val, vin);
ADS1015_ioctl(busI2C0, ADS1015_DISP_CONFIG_REG, NULL);          // 显示
```

4.3 DAC 芯片 MCP4725

源代码：ls2k500/drivers/i2c/mcp4725/mcp4725.c
头文件：ls2k500/drivers/include/i2c/mcp4725.h

MCP4725 是否使用，在 bsp.h 中配置宏定义：

```
#define MCP4725_DRV    1
```

MCP4725 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（mcp4725.c）：

```
#define MCP4725_ADDRESS         0x60            /* 7 位地址 */
#define MCP4725_BAUDRATE        100000          /* 100K */
```

驱动程序 mcp4725.c 实现的函数：

函数

```
/*
 * 向MCP4725写 2 字节数值进行DAC转换
 *
 * 参数:   bus       busI2C0
 *         buf       uint16_t *, value to be convert out
 *         arg       NULL
 *
 * 返回:   0=成功
 */
int MCP4725_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 控制MCP4725芯片
 *
 * 参数:   bus       busI2C0
 *   ------------------------------------------------------------------------------------------
 *         cmd                           | arg
 *   ------------------------------------------------------------------------------------------
 *    IOCTL_MCP4725_WRTIE_EEPROM         | uint16_t, set default value to eeprom of mcp4725
 *                                       | to convert out after poweron
 *    IOCTL_MCP4725_READ_EEPROM          | uint16_t *, get default value of mcp4725's eeprom
 *    IOCTL_MCP4725_DISP_CONFIG_REG      | NULL, display all config register
 *    IOCTL_MCP4725_READ_DAC             | uint16_t *, get current value in convert register
 *   ------------------------------------------------------------------------------------------
 *
 * 返回:   0=成功
 */
int MCP4725_ioctl(void *dev, int cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *mcp4725_drv_ops;
    #define ls2k_mcp4725_write(iic, buf, size, arg) \
            mcp4725_drv_ops->write_entry(iic, buf, size, arg)
    #define ls2k_mcp4725_ioctl(iic, cmd, arg) \
            mcp4725_drv_ops->ioctl_entry(iic, cmd, arg)
#else
    #define ls2k_mcp4725_write(iic, buf, size, arg)    MCP4725_write(iic, buf, size, arg)
    #define ls2k_mcp4725_ioctl(iic, cmd, arg)          MCP4725_ioctl(iic, cmd, arg)
#endif
```

实用函数：

函数	功能
/* * 参数: bus busI2C0 * dacVal value to be convert out */ int set_mcp4725_dac(void *bus, uint16_t dacVal);	写 DAC 转换值

编程示例：

```
unsigned short dac = 0x800;
MCP4725_write(busI2C0, (unsigned char *)&dac, 2, NULL);
```

5、NAND 控制设备

源代码：ls2k500/drivers/nand/ls2k_nand.c

头文件：ls2k500/drivers/include/ls2k_nand.h

NAND 连接的 Flash 芯片参数在 ls2k_nand.h 中定义。

NAND 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_NAND    1
```

驱动程序 ls2k_nand.c 实现的函数：

函数
/* * 初始化NAND设备 * 参数: dev devNAND * arg NULL * * 返回: 0=成功 */ int NAND_initialize(void *dev, void *arg);
/* * 打开NAND设备 * 参数: dev devNAND * arg NULL * * 返回: 0=成功 */ int NAND_open(void *dev, void *arg);
/* * 关闭NAND设备 * 参数: dev devNAND * arg NULL * * 返回: 0=成功 */ int NAND_close(void *dev, void *arg);
/* * 从NAND Flash芯片读数据 * 参数: dev devNAND * buf 类型: char *, 用于存放读取数据的缓冲区 * size 类型: int, 待读取的字节数, 长度不能超过 buf 的容量 * arg 类型: NAND_PARAM_t *. * * 返回: 读取的字节数 * * 说明: 读取NAND FLash芯片的字节数取 16 的倍数. */ int NAND_read(void *dev, void *buf, int size, void *arg);

```
/*
 * 向NAND Flash芯片写数据
 * 参数:    dev     devNAND
 *          buf     类型: char *, 用于存放待写数据的缓冲区
 *          size    类型: int, 待写入的字节数, 长度不能超过 buf 的容量
 *          arg     类型: NAND_PARAM_t *.
 *
 * 返回:    写入的字节数
 *
 * 说明:    1. 写入前的NAND Flash块已经格式化;
 *          2. 建议对 NAND Flash芯片的写操作按照整页写入.
 */
int NAND_write(void *dev,void *buf, int size, void *arg);
```

```
/*
 * 向NAND Flash芯片发送控制命令
 * 参数:    dev     devNAND
 *
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_RESET            |   NULL, 复位Flash芯片
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_GET_ID           |   类型: unsigned int *
 *                                  |   用途: 读取Flash芯片 ID
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_ERASE_BLOCK      |   类型: unsigned int
 *                                  |   用途: 删除/格式化Flash芯片的一个块
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_ERASE_CHIP       |   NULL, 删除/格式化整个Flash芯片
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_PAGE_BLANK       |   类型: unsigned int
 *                                  |   用途: 检查是否Flash芯片的一个块是不是空的
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_MARK_BAD_BLOCK   |   类型: unsigned int
 *                                  |   用途: 标记Flash芯片的一个块为坏块
 *      ------------------------------------------------------------------------------
 *      IOCTL_NAND_IS_BAD_BLOCK     |   类型: unsigned int
 *                                  |   用途: 检查Flash芯片的一个块是否是坏块
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int NAND_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *nand_drv_ops;
    #define ls2k_nand_init(nand, arg)      nand_drv_ops->init_entry(nand, arg)
    #define ls2k_nand_open(nand, arg)      nand_drv_ops->open_entry(nand, arg)
    #define ls2k_nand_close(nand, arg)     nand_drv_ops->close_entry(nand, arg)
    #define ls2k_nand_read(nand, buf, size, arg) \
            nand_drv_ops->read_entry(nand, buf, size, arg)
```

```
    #define ls2k_nand_write(nand, buf, size, arg) \
            nand_drv_ops->write_entry(nand, buf, size, arg)
    #define ls2k_nand_ioctl(nand, cmd, arg) \
            nand_drv_ops->ioctl_entry(nand, cmd, arg)
#else
    #define ls2k_nand_init(nand, arg)              NAND_initialize(nand, arg)
    #define ls2k_nand_open(nand, arg)              NAND_open(nand, arg)
    #define ls2k_nand_close(nand, arg)             NAND_close(nand, arg)
    #define ls2k_nand_read(nand, buf, size, arg)   NAND_read(nand, buf, size, arg)
    #define ls2k_nand_write(nand, buf, size, arg)  NAND_write(nand, buf, size, arg)
    #define ls2k_nand_ioctl(nand, cmd, arg)        NAND_ioctl(nand, cmd, arg)
#endif
```

NAND 设备的参数，在初始化时配置；

NAND 用到的数据类型：

```
enum
{
    NAND_OP_MAIN  = 0x0001,        /* 操作 page 的 main 区域 */
    NAND_OP_SPARE = 0x0002,        /* 操作 page 的 spare 区域 */
};

typedef struct
{
    unsigned int pageNum;         // physcal page number
    unsigned int colAddr;         // address in page
    unsigned int opFlags;         // NAND_OP_MAIN / NAND_OP_SPARE
} NAND_PARAM_t;
```

NAND_read 和 NAND_write 的 arg 参数是 **NAND_PARAM_t *** 类型，指示读写的 Flash 位置。

在使用 NAND_read 和 NAND_write 之前，**必须执行** NAND_init 和 NAND_open 函数。

注：驱动程序仅支持 NAND0；用于 NAND1～NAND3 设备时需要修改设备参数和初始化代码。

NAND 驱动函数的使用，请参考 ls2k_yaffs.c。

6、显示控制器

源代码：ls2k500/drivers/dc/ls2k_dc.c

头文件：ls2k500/drivers/include/ls2k_dc.h

framebuffer 实用函数：ls2k500/drivers/dc/ls2k_fb_utils.c

framebuffer 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_DC    1
```

驱动程序 ls2k_dc.c 实现的函数：

函数
/* * 初始化DC设备 * 参数: dev devDC * arg NULL * * 返回: 0=成功 */ int DC_initialize(void *dev, void *arg);
/* * 打开DC设备 * 参数: dev devDC * arg NULL * * 返回: 0=成功 */ int DC_open(void *dev, void *arg);
/* * 关闭DC设备 * 参数: dev devDC * arg NULL * * 返回: 0=成功 */ int DC_close(void *dev, void *arg);
/* * 读显示缓冲区 * 参数: dev devDC * buf 类型: char *, 用于存放读取数据的缓冲区 * size 类型: int, 待读取的字节数, 长度不能超过 buf 的容量 * arg 类型: unsigned int, framebuffer显示缓冲区内地址偏移量. * * 返回: 读取的字节数 */ int DC_read(void *dev, void *buf, int size, void *arg);

```
/*
 * 写显示缓冲区
 * 参数:    dev     devDC
 *          buf     类型: char *, 用于存放待写数据的缓冲区
 *          size    类型: int, 待写入的字节数, 长度不能超过 buf 的容量
 *          arg     类型: unsigned int, framebuffer显示缓冲区内地址偏移量.
 *
 * 返回:    写入的字节数
 */
int DC_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向DC设备发送控制命令
 * 参数:    dev     devDC
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      FBIOGET_FSCREENINFO         |   类型: struct fb_fix_screeninfo *
 *                                  |   用途: 读取framebuffer固定显示信息
 *      ------------------------------------------------------------------------------
 *      FBIOGET_VSCREENINFO         |   类型: struct fb_var_screeninfo *
 *                                  |   用途: 读取framebuffer可视显示信息
 *      ------------------------------------------------------------------------------
 *      FBIOGETCMAP                 |   类型: struct fb_cmap *
 *                                  |   用途: 读取framebuffer调色板
 *      ------------------------------------------------------------------------------
 *      FBIOPUTCMAP                 |   类型: struct fb_cmap *
 *                                  |   用途: 设置framebuffer调色板
 *      ------------------------------------------------------------------------------
 *      IOCTRL_FB_CLEAR_BUFFER      |   类型: unsigned int
 *                                  |   用途: 用指定值填充显示缓冲区(颜色值)
 *      ------------------------------------------------------------------------------
 *      IOCTRL_LCD_POWERON          |   TODO 如果硬件设计有LCD电源控制电路, 补充实现
 *      ------------------------------------------------------------------------------
 *      IOCTRL_LCD_POWEROFF         |   TODO 如果硬件设计有LCD电源控制电路, 补充实现
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int DC_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *dc_drv_ops;
    #define ls2k_dc_init(dc, arg)              dc_drv_ops->init_entry(dc, arg)
    #define ls2k_dc_open(dc, arg)              dc_drv_ops->open_entry(dc, arg)
    #define ls2k_dc_close(dc, arg)             dc_drv_ops->close_entry(dc, arg)
    #define ls2k_dc_read(dc, buf, size, arg) \
            dc_drv_ops->read_entry(dc, buf, size, arg)
    #define ls2k_dc_write(dc, buf, size, arg) \
            dc_drv_ops->write_entry(dc, buf, size, arg)
    #define ls2k_dc_ioctl(dc, cmd, arg)        dc_drv_ops->ioctl_entry(dc, cmd, arg)
```

```
#else
    #define ls2k_dc_init(dc, arg)             DC_initialize(dc, arg)
    #define ls2k_dc_open(dc, arg)             DC_open(dc, arg)
    #define ls2k_dc_close(dc, arg)            DC_close(dc, arg)
    #define ls2k_dc_read(dc, buf, size, arg)  DC_read(dc, buf, size, arg)
    #define ls2k_dc_write(dc, buf, size, arg) DC_write(dc, buf, size, arg)
    #define ls2k_dc_ioctl(dc, cmd, arg)       DC_ioctl(dc, cmd, arg)
#endif
```

实用函数：

函数
/* * 显示控制器是否初始化 * 返回: 1=已初始化; 0=未初始化 */ int ls2k_dc_initialized(void);
/* * 显示控制器是否启动(open) * 返回: 1=已启动; 0=未启动 */ int ls2k_dc_started(void);

头文件 ls2k_dc.h 中的 LCD 工作模式宏定义：

```
#define LCD_480x272   "480x272-16@60"    /* Fit: 4 inch LCD */
#define LCD_480x800   "480x800-16@60"    /* Fit: 4.3 inch vertical LCD */
#define LCD_800x480   "800x480-16@60"    /* Fit: 7 inch LCD */
                                         /* 格式: X 分辨率*Y 分辨率-16 位@刷新频率 */
```

全局变量 LCD 工作模式：

```
extern char LCD_display_mode[];          /* LCD 工作模式 */
```

用户应用程序**必须定义**该全局变量，例如：

```
char LCD_display_mode[] = LCD_800x480;
```

ls2k_fb_utils.c 实现的 LCD 实用函数如下：

函数	功能
int fb_open(void);	初始化并打开 framebuffer 驱动
void fb_close(void);	关闭 framebuffer 驱动

int fb_get_pixelsx(void);	返回 LCD 的 X 分辨率
int fb_get_pixelsy(void);	返回 LCD 的 Y 分辨率
void fb_set_color(unsigned colidx, unsigned value);	设定颜色索引表 colidx 处的颜色
unsigned fb_get_color(unsigned colidx);	读取颜色索引表 colidx 处的颜色
void fb_set_bgcolor(unsigned colidx, unsigned value);	设置字符输出使用的背景色
void fb_set_fgcolor(unsigned colidx, unsigned value);	设置字符输出使用的前景色
void fb_cons_putc(char chr);	在 LCD 控制台输出一个字符
void fb_cons_puts(char *str);	在 LCD 控制台输出一个字符串
void fb_cons_clear(void);	执行 LCD 控制台清屏
void fb_textout(int x, int y, char *str);	在 LCD[x,y]处打印字符串
int fb_showbmp(char *bmpfilename, int x, int y);	在 LCD[x,y]处显示 bmp 图像
void fb_put_cross(int x, int y, unsigned colidx);	在 LCD[x,y]处画”十”符号
void fb_put_string(int x, int y, char *str, unsigned colidx);	在 LCD[x,y]处用指定颜色打印字符串
void fb_put_string_center(int x, int y, char *str, unsigned colidx);	在 LCD 上以[x,y]为中心用指定颜色打印字符串
void fb_drawpixel(int x, int y, unsigned colidx);	在 LCD[x,y]处用指定颜色画像素
void fb_drawpoint(int x, int y, int thickness, unsigned colidx);	在 LCD[x,y]处用指定颜色、宽度画点
void fb_drawline(int x1, int y1, int x2, int y2, unsigned colidx);	在 LCD[x1,y1]～[x2,y2]处用指定颜色画线
void fb_drawrect(int x1, int y1, int x2, int y2, unsigned colidx);	在 LCD[x1,y1]～[x2,y2]处用指定颜色画矩形框
void fb_fillrect(int x1, int y1, int x2, int y2, unsigned colidx);	在 LCD[x1,y1]～[x2,y2]处用指定颜色填充矩形框
void fb_copyrect(int x1, int y1, int x2, int y2, int px, int py);	把 LCD[x1,y1]～[x2,y2]处的图像相对于[x1,y1 移动到[px, py]的位置
void ls2k_draw_rgb565_pixel(int x, int y, unsigned int color);	**LVGL 驱动使用**

注：开发板硬件和驱动程序实现的是 RGB565 模式。

7、CAN 控制器

源代码：ls2k500/drivers/can/ls2k_can.c
头文件：ls2k500/drivers/include/ls2k_can.h

CAN 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_CAN0    1
#define BSP_USE_CAN1    1
#define BSP_USE_CAN2    0
#define BSP_USE_CAN3    0
```

CAN 设备参数定义在中 ls2k_can.c 定义：

```
#ifdef BSP_USE_CAN0
static CAN_t ls2k_CAN0 =
{
    .hwCAN       = (HW_CAN_t *)PHYS_TO_UNCACHED(        /* 寄存器基址 */
                   CAN0_BASE),
#if USE_EXTINT
    .irqNum      = EXTINTC0_CAN0_IRQ,                   // 扩展模式下的中断号
#else
    .irqNum      = INTC0_CAN0_1_IRQ,                    // 正常模式下的中断号
#endif
    .initialized = 0,                                   // 是否初始化
    .dev_name    = "can0",                              // 设备名称
};
void *devCAN0 = (void *)&ls2k_CAN0;
#endif

……
```

驱动程序 ls2k_can.c 实现的函数：

函数
/* * 初始化CAN设备 * 参数: dev devCAN0/devCAN1/devCAN2/devCAN3 * arg NULL * * 返回: 0=成功 * * 默认值: 内核模式: CAN_CORE_20B; 工作模式: CAN_STAND_MODE; 通信速率: CAN_SPEED_250K */ int CAN_init(void *dev, void *arg);
/* * 打开CAN设备 * 参数: dev devCAN0/devCAN1/devCAN2/devCAN3 * arg NULL * * 返回: 0=成功 */ int CAN_open(void *dev, void *arg);
/* * 关闭CAN设备 * 参数: dev devCAN0/devCAN1/devCAN2/devCAN3 * arg NULL * * 返回: 0=成功 */ int CAN_close(void *dev, void *arg);
/* * 从CAN设备读数据(接收) * 参数: dev devCAN0/devCAN1 * buf 类型: CANMsg_t *, 数组, 用于存放读取数据的缓冲区 * size 类型: int, 待读取的字节数, 长度不能超过 buf 的容量, sizeof(CANMsg_t)倍数 * arg NULL * * 返回: 读取的字节数 * * 说明: CAN使用中断接收, 接收到的数据存放在驱动内部缓冲区, 读操作总是从缓冲区读取. * 必须注意接收数据缓冲区溢出. */ int CAN_read(void *dev, void *buf, int size, void *arg);
/* * 向CAN设备写数据(发送) * 参数: dev devCAN0/devCAN1/devCAN2/devCAN3 * buf 类型: CANMsg_t *, 数组, 用于存放待写数据的缓冲区 * size 类型: int, 待写入的字节数, 长度不能超过 buf 的容量, sizeof(CANMsg_t)倍数 * arg NULL * * 返回: 写入的字节数 * * 说明: CAN使用中断发送, 待发送的数据直接发送, 或者存放在驱动内部缓冲区待中断发生时继续发送.

```
 *          必须注意发送数据缓冲区溢出.
 */
int CAN_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向CAN设备发送控制命令
 * 参数:    dev     devCAN0/devCAN1/devCAN2/devCAN3
 *
 *      ------------------------------------------------------------------------------
 *          cmd                         |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_START                 |   NULL. 启动CAN进入工作状态, 注意 ls2k_can_open()
 *                                      |   之后必须调用此命令, CAN硬件进入接收或者发送状态.
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_STOP                  |   NULL. 停止CAN的工作状态, 在ls2k_can_close()
 *                                      |   之前调用此命令, 结束CAN硬件的接收或者发送状态.
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_GET_STATS             |   类型: CAN_stats_t *
 *                                      |   用途: 读取CAN设备的统计信息
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_GET_STATUS            |   类型: unsigned int *
 *                                      |   用途: 读取CAN设备的当前状态, 见上面"Status"定义
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_SPEED             |   类型: unsigned int
 *                                      |   用途: 设置CAN设备的通信速率, 见"CAN_SPEED_x"定义
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_FILTER            |   类型: CAN_afilter_t *
 *                                      |   用途: 设置CAN设备的硬件过滤器
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_BUFLEN            |   类型: unsigned int
 *                                      |   用途: 更改CAN设备的内部缓存大小.
 *                                      |         低16位: 接收缓冲区个数; 高16位: 发送缓冲区个数.
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_CORE              |   类型: unsigned int
 *                                      |   用途: 设置CAN设备的内核模式, CAN_CORE_20A/CAN_CORE_20B
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_WORKMODE          |   类型: unsigned int
 *                                      |   用途: 设置CAN设备2.0B的工作模式,
 *                                      |         CAN_STAND_MODE/CAN_SLEEP_MODE/
 *                                      |         CAN_LISTEN_ONLY/CAN_SELF_RECEIVE
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_SET_TIMEOUT           |   类型: unsigned int
 *                                      |   用途: 设置CAN设备接收/发送的超时等待毫秒数
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int CAN_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *can_drv_ops;
    #define ls2k_can_init(can, arg)        can_drv_ops->init_entry(can, arg)
    #define ls2k_can_open(can, arg)        can_drv_ops->open_entry(can, arg)
    #define ls2k_can_close(can, arg)       can_drv_ops->close_entry(can, arg)
```

```
    #define ls2k_can_read(can, buf, size, arg) \
            can_drv_ops->read_entry(can, buf, size, arg)
    #define ls2k_can_write(can, buf, size, arg) \
            can_drv_ops->write_entry(can, buf, size, arg)
    #define ls2k_can_ioctl(can, cmd, arg) \
            can_drv_ops->ioctl_entry(can, cmd, arg)
#else
    #define ls2k_can_init(can, arg)              CAN_init(can, arg)
    #define ls2k_can_open(can, arg)              CAN_open(can, arg)
    #define ls2k_can_close(can, arg)             CAN_close(can, arg)
    #define ls2k_can_read(can, buf, size, arg)   CAN_read(can, buf, size, arg)
    #define ls2k_can_write(can, buf, size, arg)  CAN_write(can, buf, size, arg)
    #define ls2k_can_ioctl(can, cmd, arg)        CAN_ioctl(can, cmd, arg)
#endif
```

CAN 用到的数据类型：

```
    /* CAN 消息，用于读写 */
    typedef struct
    {
        unsigned int     id;             /* CAN message id */
        char             rtr;            /* RTR - Remote Transmission Request */
        char             extended;       /* whether extended message package */
        unsigned char    len;            /* length of data */
        unsigned char    data[8];        /* data for transfer */
    } CANMsg_t;

    /* CAN 速率参数，用于 ioctl */
    typedef struct
    {
        unsigned char btr0;
        unsigned char btr1;
        unsigned char samples;         /* =1: samples 3 times, otherwise once */
    } CAN_speed_t;

    /* CAN ID 和过滤，用于 ioctl */
    typedef struct
    {
        unsigned char code[4];
        unsigned char mask[4];
        int           afmode;          /* =1: single filter, otherwise twice */
    } CAN_afilter_t;
```

CAN_read 和 CAN_write 的 arg 参数是 CANMsg_t * 类型。

在使用 CAN_read 和 CAN_write 之前：

- 执行 CAN_init 和 CAN_open 函数
- 使用 CAN_ioctl 设置工作模式、速率等
- 使用 CAN_ioctl 发送 start 命令
- ……

CAN 发送例程：

```
#include "bsp.h"

#include <stdio.h>
#include <stdlib.h>

#include "ls2k_can.h"
#include "drv_os_priority.h"

#include "rtthread.h"

#define CAN1_STK_SIZE   (1024*2)

/* RT-Thread 优先级说明：
 *
 *     1.RT_THREAD_PRIORITY_MAX == 32
 *     2.允许同优先级、时间片
 *     3.数值越小，优先级越高
 */
#define CAN1_TASK_PRIO      17
#define CAN1_TASK_SLICE     10

#define S_LEN        64
static unsigned char info_buf[S_LEN+1];

#define SPRINTF             rt_sprintf
#define READ_CAN1(can, m)  rt_device_read(can, 0, (const void *)&m, sizeof(m))
#define SLEEP               rt_thread_sleep

static  rt_thread_t     can1_thread;

/*
 * can1 接收数据.
 */
int ls2k_can1_do_receive(void *can)
{
    int rd_cnt;
    CANMsg_t msg;

    rd_cnt = READ_CAN1(can, msg);
     if (rd_cnt > 0)
     {
        SPRINTF(info_buf, " %02x %02x %02x %02x %02x %02x %02x %02x",
                msg.data[0], msg.data[1], msg.data[2], msg.data[3],
```

```
                msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
        printk("CAN1 RX: %s\r\n", info_buf);
    }
}

/*
 * CAN1 任务
 */
static void can1_test_task(void *arg)
{
    rt_device_t pCAN1;

    pCAN1 = rt_device_find(ls2k_can_get_device_name(devCAN1));
    if (pCAN1 == NULL)
        return;

    rt_device_open(pCAN1, RT_DEVICE_FLAG_WRONLY);
    /* set mode: CAN_CORE_20B */
    rt_device_control(pCAN1, IOCTL_CAN_SET_CORE, (void *)CAN_CORE_20B);
    /* set speed: CAN_SPEED_500K */
    rt_device_control(pCAN1, IOCTL_CAN_SET_SPEED, (void *)CAN_SPEED_250K);
    rt_device_control(pCAN1, IOCTL_CAN_START, NULL);       /* start It */

    SLEEP(100);
    DBG_OUT("can1_test_task started.\r\n");

    for ( ; ; )
    {
        ls2k_can1_do_receive(pCAN1);
        /* abandon cpu time to run other task */
        SLEEP(100);       // task sleep 100 ms
    }

    rt_device_close(pCAN1);
}

/*
 * 创建 CAN1 任务
 */
int can1_test_start(void)
{
    ls2k_can_init(devCAN1, NULL);
    ls2k_can_open(devCAN1, NULL);
    can1_thread = rt_thread_create("can1thread",
                                   can1_test_task,
                                   NULL,                // arg
                                   CAN1_STK_SIZE*4,     // statck size
                                   CAN1_TASK_PRIO,      // priority
                                   CAN1_TASK_SLICE);    // slice ticks
    if (can1_thread == NULL)
    {
        printk("create can1 test thread fail!\r\n");
        return -1;
    }
    rt_thread_startup(can1_thread);
    return 0;
}
```

CAN 接收例程：

```
#include "bsp.h"

#include <stdio.h>
#include <stdlib.h>

#include "ls2k_can.h"
#include "drv_os_priority.h"

#include "rtthread.h"

/*
 * RT-Thread 优先级说明：
 *
 *     1.RT_THREAD_PRIORITY_MAX == 32
 *     2.允许同优先级、时间片
 *     3.数值越小，优先级越高
 */
#define CAN0_TASK_PRIO      17
#define CAN0_TASK_SLICE     10
#define CAN0_STK_SIZE   (1024*2)

#define S_LEN   64
static unsigned char info_buf[S_LEN+1];

#define SPRINTF             rt_sprintf
#define SLEEP               rt_thread_sleep
#define WRITE_CAN0(can, m)  rt_device_write(can, 0, (const void *)&m, sizeof(m))

static rt_thread_t  can0_thread;
static int          tx_count = 0;

/*
 * 每隔 0.1 秒发送一次数据.
 */
int ls2k_can0_do_transmit(void *can)
{
    int wr_cnt;
    CANMsg_t msg;

     msg.id = 2; // MSG_ID;
     msg.extended = 1;
     msg.rtr = 0;
    tx_count++;
     msg.data[0] = (unsigned char)(tx_count >> 24);
     msg.data[1] = (unsigned char)(tx_count >> 16);
     msg.data[2] = (unsigned char)(tx_count >> 8);
     msg.data[3] = (unsigned char)(tx_count >> 0);
    tx_count++;
     msg.data[4] = (unsigned char)(tx_count >> 24);
     msg.data[5] = (unsigned char)(tx_count >> 16);
     msg.data[6] = (unsigned char)(tx_count >> 8);
     msg.data[7] = (unsigned char)(tx_count >> 0);
     msg.len = 8;
```

```
    /*
     * Send Message
     */
    wr_cnt = WRITE_CAN0(can, msg);
    if (wr_cnt > 0)
    {
        SPRINTF(info_buf, "%02x %02x %02x %02x %02x %02x %02x %02x",
                msg.data[0], msg.data[1], msg.data[2], msg.data[3],
                msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
        printk("CAN0 TX: %s\r\n", info_buf);
    }

    return 0;
}


/*
 * CAN0 任务
 */
static void can0_test_task(void *arg)
{
    /*
     * CAN0 initialize.
     */
    rt_device_t pCAN0;

    pCAN0 = rt_device_find(ls2k_can_get_device_name(devCAN0));
    if (pCAN0 == NULL)
        return;

    rt_device_open(pCAN0, RT_DEVICE_FLAG_WRONLY);
    /* set mode: CAN_CORE_20B */
    rt_device_control(pCAN0, IOCTL_CAN_SET_CORE, (void *)CAN_CORE_20B);
    /* set speed: CAN_SPEED_500K */
    rt_device_control(pCAN0, IOCTL_CAN_SET_SPEED, (void *)CAN_SPEED_250K);
    rt_device_control(pCAN0, IOCTL_CAN_START, NULL);    /* start It */

    SLEEP(100);
    DBG_OUT("can0_test_task started.\r\n");

    for ( ; ; )
    {
        /*
         * CAN0 task code.
         */
         ls2k_can0_do_transmit(pCAN0);
         /* abandon cpu time to run other task */
        SLEEP(500);                    // task sleep 100 ms
    }

    rt_device_close(pCAN0);
}
```

```c
/*
 * 创建 CAN0 任务
 */
int can0_test_start(void)
{
    ls2k_can_init(devCAN0, NULL);
    ls2k_can_open(devCAN0, NULL);

    can0_thread = rt_thread_create("can0thread",
                                   can0_test_task,
                                   NULL,               // arg
                                   CAN0_STK_SIZE*4,    // statck size
                                   CAN0_TASK_PRIO,     // priority
                                   CAN0_TASK_SLICE);   // slice ticks

    if (can0_thread == NULL)
    {
         printk("create can0 test thread fail!\r\n");
         return -1;
     }

     rt_thread_startup(can0_thread);
     return 0;
}
```

8、网络控制器

源代码：ls2k500/drivers/gmac/ls2k_gmac.c
头文件：ls2k500/drivers/include/ls2k_gmac.h

GMAC 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_GMAC0    1
#define BSP_USE_GMAC1    0
```

GMAC 设备参数定义在中 ls2k_gmac.c 定义：

```
#if BSP_USE_GMAC0
static GMAC_t ls2k_GMAC0 =
{
    .hwGMAC      = (HW_GMAC_t *)PHYS_TO_UNCACHED(        /* GMAC 基址 */
                    GMAC0_BASE),
    .hwGDMA      = (HW_GDMA_t *)PHYS_TO_UNCACHED(        /* GDMA 基址 */
                    GDMA0_BASE),
#if USE_EXTINT
    .irqNum      = EXTINTC3_GMAC0_IRQ,                   // 扩展模式下的中断号
#else
    .irqNum      = INTC0_GMAC0_IRQ,                      // 正常模式下的中断号
#endif
    .unitNumber  = 0,                                    // 单元编号 0
    .descmode    = CHAINMODE,                            // 描述符模式
    .timeout     = 0,                                    // 超时
    .started     = 0,                                    // 是否启动
    .initialized = 0,                                    // 是否初始化
    .dev_name    = "gmac0",                              // 设备名称
    .acceptBroadcast = 0,                                // 是否接收广播数据包
    .autoNegotiation = 1,                                // 自动协商
    .autoNegoTimeout = 5000,                             // 自动协商超时毫秒数
};
void *devGMAC0 = (void *)&ls2k_GMAC0;
#endif

……
```

驱动程序 ls2k_gmac.c 实现的函数：

函数

```
/*
 * GMAC初始化
 * 参数:    dev     devGMAC0/devGMAC1
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int GMAC_initialize(void *dev, void *arg);
```

```
/*
 * 从GMAC读取接收到的网络数据
 * 参数:    dev     devGMAC0/devGMAC1
 *          buf     类型: char *, 用于存放读取数据的缓冲区
 *          size    类型: int, 待读取的字节数, 长度不能超过 buf 的容量
 *          arg     NULL
 *
 * 返回:    本次读取的字节数
 */
int GMAC_read(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向GMAC写入待发送的网络数据
 * 参数:    dev     devGMAC0/devGMAC1
 *          buf     类型: char *, 用于存放待发送数据的缓冲区
 *          size    类型: int, 待发送的字节数, 长度不能超过 buf 的容量
 *          arg     NULL
 *
 * 返回:    本次发送的字节数
 */
int GMAC_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * GMAC 控制命令
 * 参数:    dev     devGMAC0/devGMAC1
 *
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_START            |   NULL, 启动GMAC设备
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_STOP             |   NULL, 停止GMAC设备
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_RESET            |   NULL, 复位GMAC设备
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_SET_TIMEOUT      |   类型: unsigned int
 *                                  |   用途: 设置接收/发送的超时等待时间(ms)
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_IS_RUNNING       |   NULL, GMAC设备是否运行
 *      ------------------------------------------------------------------------------
 *      IOCTL_GMAC_SHOW_STATS       |   NULL, 打印GMAC设备统计信息
 *      ------------------------------------------------------------------------------
 *
```

```
 * 返回:    0=成功
 */
int GMAC_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *gmac_drv_ops;
    #define ls2k_gmac_init(gmac, arg) \
            gmac_drv_ops->init_entry(gmac, arg)
    #define ls2k_gmac_read(gmac, buf, size, arg) \
            gmac_drv_ops->read_entry(gmac, buf, size, arg)
    #define ls2k_gmac_write(gmac, buf, size, arg) \
            gmac_drv_ops->write_entry(gmac, buf, size, arg)
    #define ls2k_gmac_ioctl(gmac, cmd, arg) \
            gmac_drv_ops->ioctl_entry(gmac, cmd, arg)
#else
    #define ls2k_gmac_init(gmac, arg)             GMAC_initialize(gmac, arg)
    #define ls2k_gmac_read(gmac, buf, size, arg)  GMAC_read(gmac, buf, size, arg)
    #define ls2k_gmac_write(gmac, buf, size, arg) GMAC_write(gmac, buf, size, arg)
    #define ls2k_gmac_ioctl(gmac, cmd, arg)       GMAC_ioctl(gmac, cmd, arg)
#endif
```

LwIP 网络协议栈接口函数：

函数
/* * 等待GMAC接收到网络数据 * 参数: dev devGMAC0/devGMAC1 * bufptr 类型: unsigned char **, 返回GMAC驱动内部接收缓冲区地址 * * 返回: 0=成功 * * 说明: 1. RTOS下调用该函数时, 使用RTOS事件实现无限等待; * 2. 裸机下调用该函数时, 等待时间为IOCTL_GMAC_SET_TIMEOUT设置的超时毫秒数 */ int ls2k_gmac_wait_receive_packet(void *dev, unsigned char **bufptr);
/* * 等待GMAC空闲时发送数据 * 参数: dev devGMAC0/devGMAC1 * bufptr 类型: unsigned char **, 返回GMAC驱动内部接收缓冲区地址 * * 返回: 0=成功 * * 说明: 1. RTOS下调用该函数时, 使用RTOS事件实现无限等待; * 2. 裸机下调用该函数时, 等待时间为IOCTL_GMAC_SET_TIMEOUT设置的超时毫秒数 */ int ls2k_gmac_wait_transmit_idle(void *dev, unsigned char **bufptr);

用户接口：

```
#define ls2k_gmac_wait_rx_packet(gmac, pbuf)  ls2k_gmac_wait_receive_packet(gmac, pbuf)
#define ls2k_gmac_wait_tx_idle(gmac, pbuf)    ls2k_gmac_wait_transmit_idle(gmac, pbuf)
```

GMAC 驱动程序已适配 lwIP-1.4.1 和 lwIP-2.1.3，参见 ls2k_ethernetif.c

9、PWM 设备

源代码：ls2k500/drivers/pwm/ls2k_pwm.c
头文件：ls2k500/drivers/include/ls2k_pwm.h

PWM 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_PWM0    1
#define BSP_USE_PWM1    0
#define BSP_USE_PWM2    0
#define BSP_USE_PWM3    0
#define BSP_USE_PWM4    0
#define BSP_USE_PWM5    0
#define BSP_USE_PWM6    0
#define BSP_USE_PWM7    0
#define BSP_USE_PWM8    0
#define BSP_USE_PWM9    0
#define BSP_USE_PWM10   0
#define BSP_USE_PWM11   0
#define BSP_USE_PWM12   0
#define BSP_USE_PWM13   0
#define BSP_USE_PWM14   0
#define BSP_USE_PWM15   0
```

PWM 的工作模式：

```
#define PWM_SINGLE_PULSE    0x01    // 单次脉冲
#define PWM_SINGLE_TIMER    0x02    // 单次定时器
#define PWM_CONTINUE_PULSE  0x04    // 连续脉冲
#define PWM_CONTINUE_TIMER  0x08    // 连续定时器
```

PWM 定时器中断触发的回调函数类型：

```
/*
 * 参数:   dev      devPWM0/devPWM1/……/devPWM15
 *         stopit   如果给*stopit 赋非零值，该定时器将停止不再工作，否则定时器会自动重新载入hi_ns值，
 *                  等待下一次PWM计数到达阀值产生中断.
 */
typedef void (*pwmtimer_callback_t)(void *pwm, int *stopit);
```

PWM 使用的配置参数

```
typedef struct pwm_cfg
{
    unsigned int      hi_ns;       /* 高电平脉冲宽度(纳秒), 定时器模式仅用 hi_ns */
    unsigned int      lo_ns;       /* 低电平脉冲宽度(纳秒),定时器模式没用 lo_ns */
    int               mode;        /* 工作模式 */
    irq_handler_t     isr;         /* 用户自定义中断函数 */
    pwmtimer_callback_t cb;        /* 定时器中断回调函数 */
#if BSP_USE_OS
    void              *event;      /* 用户定义的 RTOS 事件变量 */
#endif
} pwm_cfg_t;
```

pwm_cfg_t 参数的说明：

①、hi_ns　当 PWM 工作在脉冲模式时，用于设置高电平脉冲宽度（纳秒）；

当 PWM 工作在定时器模式时，用于设置定时时间（纳秒）。

②、lo_ns　当 PWM 工作在脉冲模式时，用于设置低电平脉冲宽度（纳秒）；

当 PWM 工作在定时器模式时，忽略该参数。

③、mode　PWM 工作模式。

需要 PWM 工作在脉冲模式，设置为 PWM_SINGLE_PULSE 或者 PWM_CONTINUE_PULSE；

需要 PWM 工作在定时器模式，设置为 PWM_SINGLE_TIMER 或者 PWM_CONTINUE_TIMER。

④、isr　自定义 PWM 定时器中断处理函数。

当 isr!=NULL 时，调用 ls2k_pwm_open()将安装该中断；否则使用默认中断函数。

⑤、cb　定时器中断回调函数。

当 PWM 定时器使用默认中断、且发生中断时，将自动调用该回调函数，让用户实现自定义的定时操作。当 cb!=NULL 时，忽略 event 的设置。

⑥、event　定时器中断 RTOS 响应事件（调用者创建）。

当 PWM 定时器使用默认中断、且发生中断时，中断函数使用 RTOS event 发出 PWM_TIMER_EVENT 事件，用户代码接收到该事件并进行相关处理。

注意：

- 当 PWM 用作定时器时，定时时间的设置是否能实现正确计时（Dead Zone）。
- 当 PWM 的引脚被复用时，PWM 脉冲发生器不能从引脚输出脉冲，但仍可以用作定时器。

驱动程序 ls2k_pwm.c 实现的函数：

函数

```
/*
 * PWM初始化
 * 参数:    dev     devPWM0/devPWM1/……/devPWM15
 *          arg     总是  NULL
 *
 * 返回:    0=成功
 */
int PWM_initialize(void *dev, void *arg);
```

```
/*
 * 打开PWM设备
 * 参数:    dev     devPWM0/devPWM1/……/devPWM15
 *          arg     类型: pwm_cfg_t *, 用于设置PWM设备的工作模式并启动.
 *
 * 返回:    0=成功
 *
 * 说明:    如果PWM工作在脉冲模式, hi_ns是高电平纳秒数, lo_ns是低电平纳秒数.
 *          mode    PWM_SINGLE_PULSE:   产生单次脉冲
 *                  PWM_CONTINUE_PULSE: 产生连续脉冲
 *
 *          如果PWM工作在定时器模式, 定时器时间间隔使用hi_ns纳秒数, (忽略lo_ns). 当PWM计数到达时将触发PWM
 *          定时中断, 这时中断响应:
 *          1. 如果传入参数有用户自定义中断 isr(!=NULL), 则响应isr;
 *          2. 如果自定义中断 isr=NULL, 使用PWM默认中断, 该中断调用cb 回调函数让用户作出定时响应;
 *          3. 如果自定义中断 isr=NULL且cb=NULL, 如果有event参数, PWM默认中断将发出PWM_TIMER_EVENT事件.
 *
 *          mode    PWM_SINGLE_TIMER:   产生单次定时
 *                  PWM_CONTINUE_TIMER: 产生连续定时
 *
 */
int PWM_open(void *dev, void *arg);
```

```
/*
 * 关闭PWM定时器
 * 参数:    dev     devPWM0/devPWM1/……/devPWM15
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int PWM_close(void *dev, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *pwm_drv_ops;
    #define ls2k_pwm_init(pwm, arg)        pwm_drv_ops->init_entry(pwm, arg)
    #define ls2k_pwm_open(pwm, arg)        pwm_drv_ops->open_entry(pwm, arg)
    #define ls2k_pwm_close(pwm, arg)       pwm_drv_ops->close_entry(pwm, arg)
```

```
#else
    #define ls2k_pwm_init(pwm, arg)        PWM_initialize(pwm, arg)
    #define ls2k_pwm_open(pwm, arg)        PWM_open(pwm, arg)
    #define ls2k_pwm_close(pwm, arg)       PWM_close(pwm, arg)
#endif
```

PWM 实用函数：

函数
/* * 启动PWM设备产生脉冲 * 参数: dev devPWM0/devPWM1/……/devPWM15 * cfg 类型: pwm_cfg_t *, mode==PWM_SINGLE_PULSE/PWM_CONTINUE_PULSE * * 返回: 0=成功 */ int ls2k_pwm_pulse_start(void *pwm, pwm_cfg_t *cfg);
/* * 停止PWM设备产生脉冲 * 参数: dev devPWM0/devPWM1/……/devPWM15 * * 返回: 0=成功 */ int ls2k_pwm_pulse_stop(void *pwm);
/* * 启动PWM定时器 * 参数: dev devPWM0/devPWM1/……/devPWM15 * cfg 类型: pwm_cfg_t *, mode==PWM_SINGLE_TIMER/PWM_CONTINUE_TIMER * * 返回: 0=成功 */ int ls2k_pwm_timer_start(void *pwm, pwm_cfg_t *cfg);
/* * 停止PWM定时器 * 参数: dev devPWM0/devPWM1/……/devPWM15 * * 返回: 0=成功 */ int ls2k_pwm_timer_stop(void *pwm);

例程：

```
/*
 * pwm_test.c
 */

#include "bsp.h"

#if BSP_USE_PWM

#include <stdlib.h>
#include "ls2k_pwm.h"

// PWM 中断回调函数
void PWM2_TIM_callback(void *pwm, int *stopit)
{
    printk("2.");
    // *stopit = 1;
}

/*
 * PWM 测试函数
 */
void pwm_test(void)
{
    pwm_cfg_t cfg;

    memset((void *)&cfg, 0, sizeof(pwm_cfg_t));  // 这个结构必须清零

    cfg.mode = PWM_CONTINUE_TIMER;

    cfg.hi_ns = 200*1000*1000;
    cfg.cb = PWM2_TIM_callback;                  // 定时器中断回调函数

    ls2k_pwm_timer_start(devPWM2, &cfg);
}

#endif // #if BSP_USE_PWM
```

10、RTC 时钟设备

源代码：ls2k500/drivers/rtc/ls2k_rtc.c

头文件：ls2k500/drivers/include/ls2k_rtc.h

RTC 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_RTC    1
```

RTC 驱动把 RTC 设备分为 6 个虚拟子设备，可以各自独立操作：

```
/* Virtual Sub-Device */
#define DEVICE_RTCMATCH0
#define DEVICE_RTCMATCH1
#define DEVICE_RTCMATCH2
#define DEVICE_TOYMATCH0
#define DEVICE_TOYMATCH1
#define DEVICE_TOYMATCH2
```

RTC 定时器中断触发的回调函数类型：

```
/*
 * 参数:    device    RTC虚拟子设备产生的中断
 *          match     当前RTCMATCH或者TOYMATCH的寄存器值
 *          stop      如果给*stop 赋非零值，该定时器将停止不再工作，否则定时器会自动重新载入interval_ms值，
 *                    等待下一次定时器计时阀值产生中断.
 */
typedef void (*rtctimer_callback_t)(int device, unsigned match, int *stop);
```

RTC 驱动使用的配置参数：

```
typedef struct rtc_cfg
{
    int        interval_ms;            /* 定时器时间间隔（毫秒）*/
    struct tm *trig_datetime;          /* 用于 toymatch 的日期 */
    irq_handler_t       isr;           /* 用户自定义中断函数 */
    rtctimer_callback_t cb;            /* 定时器中断回调函数 */
#if BSP_USE_OS
    void               *event;         /* 用户定义的 RTOS 事件变量 */
#endif
} rtc_cfg_t;
```

rtc_cfg_t 参数的说明：

①、interval_ms　用于 rtcmatch；当 trig_datetime==NULL 时，该参数也用于 toymatch。

当中断发生后，该值会自动载入以等待下一次中断。

②、trig_datetime 仅用于 toymatch；当 toymatch 到达该日期时，触发中断。

使用该日期触发的中断仅触发一次。

③、isr　自定义定时器中断处理函数。

当 isr!=NULL 时，调用 ls2k_rtc_open()将安装该中断；否则使用默认中断函数。

④、cb　定时器中断回调函数。

当 rtc 定时器使用默认中断、且发生中断时，将自动调用该回调函数，让用户实现自定义的定时操作。当 cb!=NULL 时，忽略 event 的设置。

⑤、event　定时器中断 RTOS 响应事件（调用者创建）。

当 rtc 定时器使用默认中断、且发生中断时，中断函数使用 RTOS event 发出 RTC_TIMER_EVENT 事件，用户代码接收到该事件并进行相关处理。

驱动程序 ls2k_rtc.c 实现的函数：

函数

```
/*
 * RTC初始化
 * 参数:    dev     总是 NULL
 *          arg     类型: struct tm *. 如果该参数不是 NULL, 其值用于初始化RTC系统时间.
 *
 * 返回:    0=成功
 */
int RTC_initialize(void *dev, void *arg);
```

```
/*
 * 打开RTC定时器
 * 参数:    dev     要打开的RTC子设备 DEVICE_XXX
 *          arg     类型: rtc_cfg_t *, 用于设置RTC子设备的工作模式并启动
 *
 * 返回:    0=成功
 *
 * 说明:    如果使用的是RTC子设备, 必须设置参数rtc_cfg_t的interval_ms值, 当RTC计时到达interval_ms阀值
时,
 *          将触发RTC定时中断, 这时中断响应:
 *          1. 如果传入参数有用户自定义中断 isr(!=NULL), 则响应isr;
 *          2. 如果自定义中断 isr=NULL, 使用RTC默认中断, 该中断调用cb 回调函数让用户作出定时响应;
 *          3. 如果自定义中断 isr=NULL且cb=NULL, 如果有event参数, RTC默认中断将发出RTC_TIMER_EVENT事件.
 *
 *          如果使用的是TOY子设备, 并且设置有interval_ms参数(>1000), 用法和使用RTC子设备一样;
 *          当interval_ms==0且trig_datetime!=NULL时, 表示TOY子设备将在计时到达这个未来时间点时触发中断,
 *          中断处理流程和上面一致.
 *          使用trig_datetime触发的中断仅发生一次.
 *
 *          interval_ms用于间隔产生中断并且一直产生; trig_datetime用于到时产生中断仅产生一次.
 */
int RTC_open(void *dev, void *arg);
```

```
/*
 * 关闭RTC定时器
 * 参数:    dev     要关闭的RTC子设备 DEVICE_XXX
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int RTC_close(void *dev, void *arg);
```

```
/*
 * 读取当前RTC时钟
 * 参数:    dev     NULL
 *          buf     类型: struct tm *, 用于存放读取的时钟值
 *          size    类型: int, 大小=sizeof(struct tm)
 *          arg     NULL
 *
 * 返回:    读取的字节数, 正常为sizeof(struct tm)
 */
int RTC_read(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 设置RTC时钟
 * 参数:    dev     NULL
 *          buf     类型: struct tm *, 用于存放待写入的时钟值
 *          size    类型: int, 大小=sizeof(struct tm)
 *          arg     NULL
 *
 * 返回:    写入的字节数, 正常为sizeof(struct tm)
 */
int RTC_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 控制RTC时钟设备
 * 参数:    dev     NULL or DEVICE_XXX
 *
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_SET_SYS_DATETIME      |   类型: truct tm *
 *                                  |   用途: 设置RTC系统时间值
 *      ------------------------------------------------------------------------------
 *      IOCTL_GET_SYS_DATETIME      |   类型: struct tm *
 *                                  |   用途: 获取当前RTC系统时间值
 *      ------------------------------------------------------------------------------
 *      IOCTL_RTC_SET_TRIM          |   类型: unsigned int *
 *                                  |   用途: 设置RTC的32768HZ时钟脉冲分频值
 *      ------------------------------------------------------------------------------
 *      IOCTL_RTC_GET_TRIM          |   类型: unsigned int *
 *                                  |   用途: 获取RTC的32768HZ时钟脉冲分频值
 *      ------------------------------------------------------------------------------
 *      IOCTL_RTCMATCH_START        |   类型: rtc_cfg_t *, 启动RTC定时器
 *                                  |   dev==DEVICE_RTCMATCHx
 *      ------------------------------------------------------------------------------
 *      IOCTL_RTCMATCH_STOP         |   类型: NULL, 停止RTC定时器
 *                                  |   dev==DEVICE_RTCMATCHx
 *      ------------------------------------------------------------------------------
 *      IOCTL_TOY_SET_TRIM          |   类型: unsigned int *
 *                                  |   用途: 设置TOY的32768HZ时钟脉冲分频值
 *      ------------------------------------------------------------------------------
```

```
 *      IOCTL_TOY_GET_TRIM          |   类型: unsigned int *
 *                                  |   用途: 获取TOY的32768HZ时钟脉冲分频值
 *      ------------------------------------------------------------------------------
 *      IOCTL_TOYMATCH_START        |   类型: rtc_cfg_t *, 启动TOY定时器
 *                                  |   dev==DEVICE_TOYMATCHx
 *      ------------------------------------------------------------------------------
 *      IOCTL_TOYMATCH_STOP         |   类型: NULL, 停止TOY定时器
 *                                  |   dev==DEVICE_TOYMATCHx
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int RTC_ioctl(void *dev, int cmd, void *arg);
```

注意各函数的 dev、arg 参数类型。

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *rtc_drv_ops;
    #define ls2k_rtc_init(rtc, arg)            rtc_drv_ops->init_entry(rtc, arg)
    #define ls2k_rtc_open(rtc, arg)            rtc_drv_ops->open_entry(rtc, arg)
    #define ls2k_rtc_close(rtc, arg)           rtc_drv_ops->close_entry(rtc, arg)
    #define ls2k_rtc_read(rtc, buf, size, arg)  \
            rtc_drv_ops->read_entry(rtc, buf, size, arg)
    #define ls2k_rtc_write(rtc, buf, size, arg) \
            rtc_drv_ops->write_entry(rtc, buf, size, arg)
    #define ls2k_rtc_ioctl(rtc, cmd, arg)      rtc_drv_ops->ioctl_entry(rtc, cmd, arg)
#else
    #define ls2k_rtc_init(rtc, arg)             RTC_initialize(rtc, arg)
    #define ls2k_rtc_open(rtc, arg)             RTC_open(rtc, arg)
    #define ls2k_rtc_close(rtc, arg)            RTC_close(rtc, arg)
    #define ls2k_rtc_read(rtc, buf, size, arg)  RTC_read(rtc, buf, size, arg)
    #define ls2k_rtc_write(rtc, buf, size, arg) RTC_write(rtc, buf, size, arg)
    #define ls2k_rtc_ioctl(rtc, cmd, arg)       RTC_ioctl(rtc, cmd, arg)
#endif
```

RTC 实用函数：

函数

```
/*
 * 设置RTC时钟值, 参见ls2k_rtc_read()
 */
int ls2k_rtc_set_datetime(struct tm *dt);
```

```
/*
 * 获取当前RTC时间，参见ls2k_rtc_write()
 */
int ls2k_rtc_get_datetime(struct tm *dt);
```

```
/*
 * 开启定时器，参见ls2k_rtc_open()
 */
int ls2k_rtc_timer_start(unsigned device, rtc_cfg_t *cfg);
```

```
/*
 * 关闭定时器，参见ls2k_rtc_close()
 */
int ls2k_rtc_timer_stop(unsigned device);
```

```
/*
 * struct tm 日期格式转换为 toymatch
 */
void ls2k_tm_to_toymatch(struct tm *dt, unsigned int *match);
```

```
/*
 * toymatch 日期格式转换为 struct tm
 */
void ls2k_toymatch_to_tm(struct tm *dt, unsigned int match);
```

```
/*
 * 秒数转换为 toymatch
 */
unsigned int ls2k_seconds_to_toymatch(unsigned int seconds);
```

```
/*
 * toymatch 转换为秒数
 */
unsigned int ls2k_toymatch_to_seconds(unsigned int match);
```

```
/*
 * struct tm 日期标准化，+1900/-1900
 */
void normalize_tm(struct tm *tm, bool tm_format);
```

例程：

```
#include "bsp.h"

#if BSP_USE_RTC

#include <stdio.h>
#include "ls2k500.h"
#include "ls2k_rtc.h"

static void rtc0_isr(int vector, void *arg)
{
    int device, index;
    if (arg == NULL)
        return;

    device = (long)arg & 0xFF00;
    index  = (long)arg & 0x00FF;

    printk("isr from device=%i, index=%i\r\n", device, index - 1);
}

static void rtc0_callback(int device, unsigned match, int *stop)
{
    struct tm dt;

    switch (device & 0xFF00)
    {
        case LS2K_TOY:
            ls2k_toymatch_to_tm(&dt, match);
            normalize_tm(&dt, false);
            printk("isr = %i.%i.%i-%i:%i:%i <-\r\n",
                   dt.tm_year, dt.tm_mon, dt.tm_mday, dt.tm_hour, dt.tm_min, dt.tm_sec);
            break;

        case LS2K_RTC:
            printk("rtc%i<-\r\n", (device & 0xFF) - 1);
            break;
    }
}

void rtc_test(void)
{
    unsigned rtcread;
    struct tm dt;
    rtc_cfg_t cfg;

    // 2020.2.28-23:59:40
    dt.tm_year = 2022;  // - 1900;  /* 年份，其值等于实际年份减去 1900 */
    dt.tm_mon  = 7;     // - 1;     /* 月份（从一月开始，0 代表一月)- 取值区间为[0, 11] */
    dt.tm_mday = 1;
    dt.tm_hour = 23;
    dt.tm_min  = 59;
    dt.tm_sec  = 55;
```

```
    ls2k_rtc_init(NULL, &dt);

#if 1
    cfg.interval_ms   = 1000;   // 重复触发中断模式.
    cfg.trig_datetime = NULL;

#else
    dt.tm_mon  = 3;            // 仅触发一次的模式. trigger at 2021.3.1 0:0:5
    dt.tm_mday = 1;
    dt.tm_hour = 0;
    dt.tm_min  = 0;
    dt.tm_sec  = 5;
    cfg.interval_ms   = 0;
    cfg.trig_datetime = &dt;    /* only trigger once */

#endif

    cfg.cb  = rtc0_callback;
    cfg.isr = NULL;

    ls2k_rtc_timer_start(DEVICE_RTCMATCH0, &cfg);

}

#endif // #if BSP_USE_RTC
```

11、HPET 定时器设备

源代码：ls2k500/drivers/hpet/ls2k_hpet.c
头文件：ls2k500/drivers/include/ls2k_hpet.h

HPET 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_HPET0    1
#define BSP_USE_HPET1    0
#define BSP_USE_HPET2    0
#define BSP_USE_HPET3    0
```

每个 HPET 设备包含 3 个定时器子设备，可以各自独立操作：

```
/* HPET Sub-Device */
#define HPET_TIMER0            0x501
#define HPET_TIMER1            0x502
#define HPET_TIMER2            0x503
```

HPET 设备工作模式：

```
#define HPET_MODE_SINGLE        0x01    // 产生单次中断
#define HPET_MODE_CYCLE         0x02    // 产生周期中断
```

HPET 定时器中断触发的回调函数类型：

```
/*
 * 参数:    hpet     HPET设备
 *          timer    HPET定时器子设备TIMERn
 *          stop     如果给*stop 赋非零值，该定时器将停止不再工作，否则定时器会自动重新载入interval_ns值，
 *                   等待下一次定时器计时阀值产生中断.
 */
typedef void (*hpettimer_callback_t)(void *hpet, int timer, int *stop);
```

HPET 驱动使用的配置参数：

```
typedef struct hpet_cfg
{
    int timer;                      /* HPET 定时器子设备: HPET_TIMERn */

    /*
     * This parameter's unit is nanosecond(ns). After interrupt ocurred,
     * this value will auto loaded for next match interrupt.
     */
    int interval_ns;                /* 纳秒为单位进行定时 */
    int work_mode;                  /* 单次/周期, 边缘/电平 */
```

```
    /*
     * HEPT 的共享中断将回调，三个定时器可以各自响应
     */
    hpetimer_callback_t cb;         /* 优先级：中. called by match-isr */
#if BSP_USE_OS
    void *event;                    /* 优先级：低. RTOS event created by user */
#endif
} hpet_cfg_t;
```

hpet_cfg_t 参数的说明：

①、timer　　使用的 hpet 子设备编号

②、interval_ns　以纳秒为单位的定时器时间间隔；如果工作在周期模式，当中断发生后，该值会自动载入以等待下一次中断。

③、work_mode　　定时器工作模式，单次或者周期模式。

④、cb　　定时器中断回调函数。

当 hpet 定时器发生中断时，将自动调用该回调函数，让用户实现自定义的定时操作。当 cb!=NULL 时，忽略 event 的设置。

⑤、event　　定时器中断 RTOS 响应事件（调用者创建）。

当 hpet 定时器发生中断、且没有设置回调函数时，中断函数使用 RTOS event 发出 HPET_TIMER_EVENT 事件，用户代码接收到该事件并进行相关处理。

TODO：细分 HPET_TIMER_EVENT 事件

驱动程序 ls2k_hpet.c 实现的函数：

函数
/* * HPET初始化 * 参数:　dev　devHPET0/devHPET1/devHPET2/devHPET3 *　　arg　NULL * * 返回:　0=成功 */ int HPET_initialize(void *dev, void *arg);
/* * 打开HPET定时器 * 参数:　dev　devHPET0/devHPET1/devHPET2/devHPET3 *　　arg　类型: hpet_cfg_t *, 用于设置HPET定时器子设备的工作模式并启动 * * 返回:　0=成功 * * 说明:　必须设置参数hpet_cfg_t的interval_ns值, 当HPET计时到达interval_ns阀值时, 将触发HPET定时中断, *　　这时中断响应: *　　1. HPET 设备使用共享中断; *　　2. 如果cb != NULL, 该中断调用cb 回调函数让用户作出定时响应; *　　3. 如果cb == NULL, 但有event参数, HPET中断将发出HPET_TIMER_EVENT事件. */ int HPET_open(void *dev, void *arg);

```
/*
 * 关闭HPET定时器
 * 参数:    dev     HPET 设备
 *          arg     HPET 定时器子设备: HPET_TIMERn
 *
 * 返回:    0=成功
 */
int HPET_close(void *dev, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *hpet_drv_ops;
    #define ls2k_hpet_init(hpet, arg)          hpet_drv_ops->init_entry(hpet, arg)
    #define ls2k_hpet_open(hpet, arg)          hpet_drv_ops->open_entry(hpet, arg)
    #define ls2k_hpet_close(hpet, arg)         hpet_drv_ops->close_entry(hpet, arg)
#else
    #define ls2k_hpet_init(hpet, arg)          HPET_initialize(hpet, arg)
    #define ls2k_hpet_open(hpet, arg)          HPET_open(hpet, arg)
    #define ls2k_hpet_close(hpet, arg)         HPET_close(hpet, arg)
#endif
```

RTC 实用函数：

函数

```
/*
 * 启动HPET定时器
 * 参数:    dev     devHPET0~devHPET3
 *          timer   HPET_TIMER0~HPET_TIMER2
 *          cfg     类型: hpet_cfg_t *
 *
 * 返回:    0=成功
 */
int ls2k_hpet_timer_start(void *hpet, int timer, hpet_cfg_t *cfg);
```

```
/*
 * 停止HPET定时器
 * 参数:    dev     devHPET0~devHPET3
 *          timer   HPET_TIMER0~HPET_TIMER2
 *
 * 返回:    0=成功
 */
int ls2k_hpet_timer_stop(void *hpet, int timer);
```

例程：

```
/*
 * hpet_test.c
 */

#include "bsp.h"
#include <stdlib.h>
#include "ls2k_hpet.h"

static int t0 = 0;
static int t1 = 0;
static int t2 = 0;

static void hpet_tm_callback(void *hpet, int timer, int *stop)
{
    switch (timer)
    {
        case HPET_TIMER0:
            printk("T0");
            if (t0++ > 10)
                *stop = 1;
            break;
        case HPET_TIMER1:
            printk("T1");
            if (t1++ > 20)
                *stop = 1;
            break;
        case HPET_TIMER2:
            printk("T2");
            if (t2++ > 30)
                *stop = 1;
            break;
        default:
            printk("+");
            break;
    }
}

void hpet_test(void)
{
    hpet_cfg_t cfg = { 0 };

    cfg.interval_ns = 1000*1000*1000;
    cfg.work_mode   = HPET_MODE_CYCLE;     // HPET_MODE_SINGLE;
    cfg.cb          = hpet_tm_callback;

    ls2k_hpet_timer_start(devHPET1, HPET_TIMER1, &cfg);
    cfg.interval_ns = 300*1000*1000;
    ls2k_hpet_timer_start(devHPET1, HPET_TIMER2, &cfg);
}
```

12、GPIO 端口

源代码：ls2k500/drivers/gpio/ls2k_gpio.c

头文件：ls2k500/drivers/include/ls2k_gpio.h

GPIO 方向：

```
#define DIR_IN      1               /* 定义 gpio 为输入 */
#define DIR_OUT     0               /* 定义 gpio 为输出 */
```

GPIO 引脚复用：

```
#define GPIO_AS_GPIO    0           /* 引脚复用为 GPIO */
#define GPIO_AS_MUX1    1           /* 引脚复用 1  */
#define GPIO_AS_MUX2    2           /* 引脚复用 2  */
#define GPIO_AS_MUX3    3           /* 引脚复用 3  */
#define GPIO_AS_MUX4    4           /* 引脚复用 4  */
#define GPIO_AS_PAD     5           /* 引脚主功能 */
```

驱动程序 ls2k_gpio.c 实现的函数：

函数
/* * GPIO 端口复用 * 参数: gpionum gpio端口序号 * mux 复用编号 * */ void gpio_mux(int gpionum, int mux);
/* * 使能GPIO端口 * 参数: gpionum gpio端口序号 * dir_in gpio方向. ~0: 输入, 0: 输出 */ void gpio_enable(int gpionum, int dir_in);
/* * 读GPIO端口, 该GPIO被设置为输入模式 * 参数: gpionum gpio端口序号 * 返回: 0或者1 */ int gpio_read(int gpionum);
/* * 写GPIO端口, 该GPIO被设置为输出模式 * 参数: gpionum gpio端口序号 * val 0或者1 */ void gpio_write(int gpionum, int val);

```
/*
 * 关闭GPIO功能，端口恢复默认设置
 * 参数:    gpionum   gpio端口序号
 */
void gpio_disable(int gpionum);
```

GPIO 中断触发模式的宏定义：

```
#define INT_TRIG_EDGE_UP        0x01        /* 上升沿触发 gpio 中断 */
#define INT_TRIG_EDGE_DOWN      0x02        /* 下降沿触发 gpio 中断 */
#define INT_TRIG_LEVEL_HIGH     0x04        /* 高电平触发 gpio 中断 */
#define INT_TRIG_LEVEL_LOW      0x08        /* 低电平触发 gpio 中断 */
```

GPIO 中断处理函数（实现在 irq.c 中）：

函数

```
/*
 * 使能GPIO中断
 * 参数:    gpio    gpio端口序号
 */
extern int ls2k_gpio_interrupt_enable(int gpionum, int trigger_mode);
```

```
/*
 * 禁止GPIO中断
 * 参数:    gpio    gpio端口序号
 */
extern int ls2k_gpio_interrupt_disable(int gpionum);
```

```
/*
 * 安装GPIO中断向量
 * 参数:    gpio            gpio端口序号
 *          isr             中断向量，类型同 irq_handler_t
 *          arg             用户自定义参数，该参数供中断向量引用
 */
extern int ls2k_gpio_isr_install(int gpionum, void (*isr)(int, void *), void *arg);
```

```
/*
 * 取消已安装GPIO中断向量
 * 参数:    gpio    gpio端口序号
 */
extern int ls2k_remove_gpio_isr_remove(int gpionum);
```

第六节 其它宏定义与函数

1、内存/寄存器读写操作

头文件：ls2k500/include/ls2k500.h

宏定义	返回值 / Val	功能
READ_REG8(addr)	unsigned char 读写 1 个字节；	返回地址 addr 处的字节值
WRITE_REG8(addr, v)		写一个字节到地址 addr 处
OR_REG8(addr, v)		对地址 addr 处的字节做或运算
AND_REG8(addr, v)		对地址 addr 处的字节做与运算
READ_REG16(addr)	unsigned short 读写 2 个字节； 地址对齐 2 字节	返回地址 addr 处的字值
WRITE_REG16(addr, v)		写一个字到地址 addr 处
OR_REG16(addr, v)		对地址 addr 处的字做或运算
AND_REG16(addr, v)		对地址 addr 处的字做与运算
READ_REG32(addr)	unsigned int 读写 4 个字节； 地址对齐 4 字节	返回地址 addr 处的双字值
WRITE_REG32(addr, v)		写一个双字到地址 addr 处
OR_REG32(addr, v)		对地址 addr 处的双字做或运算
AND_REG32(addr, v)		对地址 addr 处的双字做与运算
READ_REG64(addr)	unsigned long int 读写 8 个字节； 地址对齐 8 字节	返回地址 addr 处的四字值
WRITE_REG64(addr, v)		写一个四字到地址 addr 处
OR_REG64(addr, v)		对地址 addr 处的四字做或运算
AND_REG64(addr, v)		对地址 addr 处的四字做与运算

参数：addr 是一个 **unsigned long int** 型**合法内存地址**

2、中断相关操作

引用定义在 ls2k500.h；源文件：irq.c/irq_c.c

```
/*
 * 中断函数
 */
extern void ls2k_install_irq_handler(int vector, irq_handler_t isr, void *arg);
extern void ls2k_remove_irq_handler(int vector);

/*
```

```
 * 中断映射
 */
#define INT_ROUTE_IP2                   0x10        /* 中断路由到 IP2 */
#define INT_ROUTE_IP3                   0x20        /* 中断路由到 IP3 */
#define INT_ROUTE_IP4                   0x40        /* 中断路由到 IP4 */
#define INT_ROUTE_IP5                   0x80        /* 中断路由到 IP5 */
extern void ls2k_set_irq_routeip(int vector, int route_ip);

/*
 * 中断触发
 */
#define INT_TRIGGER_LEVEL               0x04        /* 电平触发中断 */
#define INT_TRIGGER_PULSE               0x08        /* 脉冲触发中断 */
extern void ls2k_set_irq_triggermode(int vector, int mode);

extern void ls2k_interrupt_enable(int vector);     /* 根据中断向量使能中断 */
extern void ls2k_interrupt_disable(int vector);    /* 根据中断向量禁止中断 */

extern int assert_sw_irq(unsigned int irqnum);     /* Generate a software interrupt */
extern int negate_sw_irq(unsigned int irqnum);     /* Clear a software interrupt */
```

3、cache 操作函数

引用定义在 ls2k500.h；实现在汇编文件：cache.S

```
extern void flush_cache(void);
extern void flush_cache_nowrite(void);
extern void clean_cache(unsigned long kva, unsigned int n);

extern void flush_dcache(void);
extern void clean_dcache(unsigned long kva, unsigned int n);
extern void clean_dcache_indexed(unsigned long kva, unsigned int n);
extern void clean_dcache_nowrite(unsigned long kva, unsigned int n);
extern void clean_dcache_nowrite_indexed(unsigned long kva, unsigned int n);

extern void clean_icache(unsigned long kva, unsigned int n);
extern void clean_icache_indexed(unsigned long kva, unsigned int n);

extern void clean_scache(unsigned long kva, unsigned int n);
extern void clean_scache_indexed(unsigned long kva, unsigned int n);
extern void clean_scache_nowrite(unsigned long kva, unsigned int n);
extern void clean_scache_nowrite_indexed(unsigned long kva, unsigned int n);
```

4、芯片运行频率

全局变量，实现在文件 bsp_start.c 中（**未对外声明**）

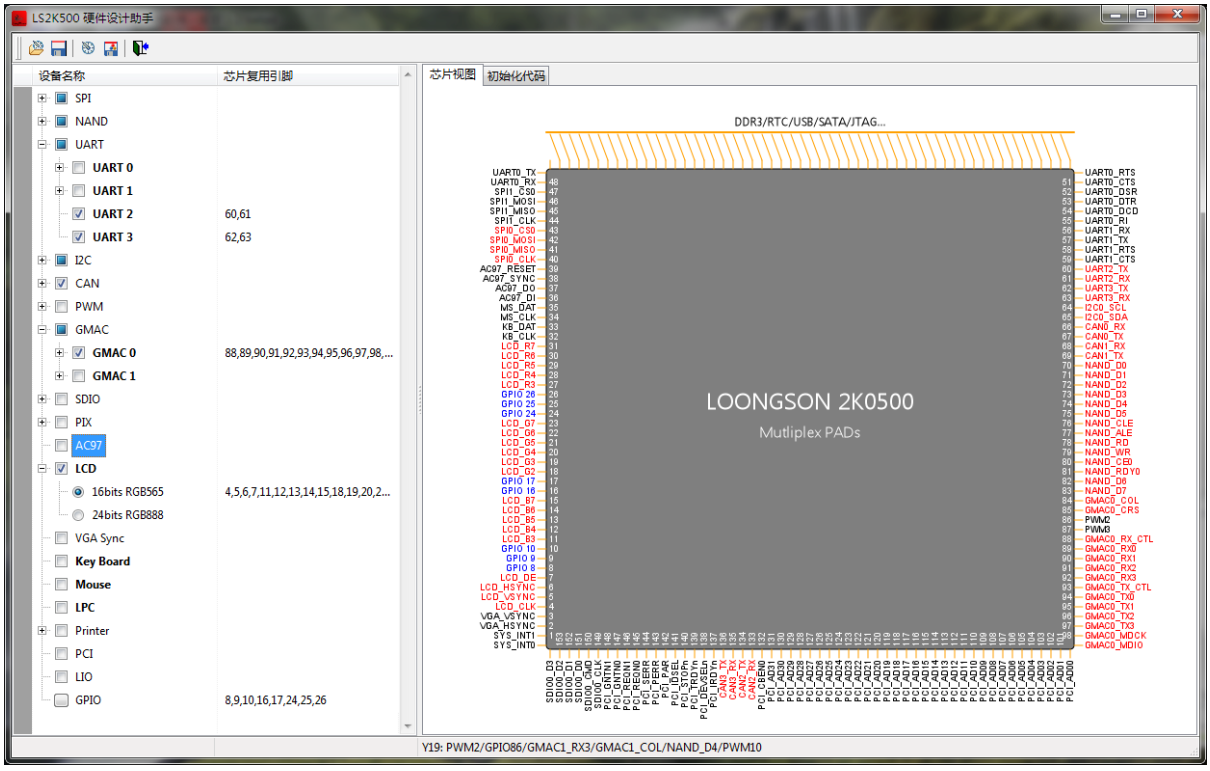
变量引用格式	功能
`extern unsigned int cpu_frequency;`	芯片主频
`extern unsigned int ddr_frequency;`	内存主频
`extern unsigned int hda_frequency;`	
`extern unsigned int net_frequency;`	
`extern unsigned int gmac_frequency;`	GMAC 主频
`extern unsigned int sb_frequency;`	
`extern unsigned int gpu_frequency;`	
`extern unsigned int pix0_frequency;`	
`extern unsigned int pix1_frequency;`	
`extern unsigned int lsu_frequency;`	
`extern unsigned int print_frequency;`	
`extern unsigned int apb_frequency;`	APB 总线频率
`extern unsigned int usb_frequency;`	
`extern unsigned int sata_frequency;`	

注：外接晶振频率固定 100MHZ。

第七节 硬件初始化助手

“硬件初始化助手”让用户通过可视化窗口设置硬件设计、自动生成硬件初始化代码。

1、LS2K0500 配置窗口



LS2K0500 视图显示的 PAD 是可复用的 PAD 和设备模块。

2、LS2K500 初始化代码（demo）

以下代码有上图配置自动生成。

```
/*
 * LS2K500 hardware initialize code, auto generated by hardware assisant.
 * Created: 2023-04-27 20:49:01
 */

#include "ls2k500.h"
#include "ls2k500_gpio.h"

/*
 * Dynamic Power Manager Register at 0x1ff60000
 */
static void ls2k_set_dpm_register(void)
{
    unsigned int regVal = 0;

    WRITE_REG32(0x1ff60000, 0x1FFC);     // Enable all but DDR & NODE

    regVal |= 0x3 << 24;                 // 25:24 Power-off PRINT
    regVal |= 0x3 << 20;                 // 21:20 Power-off USB3
    regVal |= 0x3 << 18;                 // 19:18 Power-off USB
    regVal |= 0x3 << 16;                 // 17:16 Power-off SATA
    regVal |= 0x3 << 14;                 // 15:14 Power-off GMAC1
    regVal |= 0x3 << 6;                  // 7:6   Power-off PCIE
    regVal |= 0x3 << 4;                  // 5:4   Power-off PCI

    WRITE_REG32(0x1ff60008, regVal);
}

/*
 * General Config Register at 0x1fe10100
 */
static void ls2k_set_config_register0(void)
{
    // do nothing
}

/*
 * GPIO Muxed Pins
 */
static void ls2k_set_muxed_pins(void)
{
    gpio_mux(4, 0);                  // [4]: LCD_CLK as LCD_CLK
    gpio_mux(5, 0);                  // [5]: LCD_VSYNC as LCD_VSYNC
    gpio_mux(6, 0);                  // [6]: LCD_HSYNC as LCD_HSYNC
    gpio_mux(7, 0);                  // [7]: LCD_DE as LCD_DE
    gpio_mux(11, 0);                 // [11]: LCD_B3 as LCD_B3
    gpio_mux(12, 0);                 // [12]: LCD_B4 as LCD_B4
    gpio_mux(13, 0);                 // [13]: LCD_B5 as LCD_B5
    gpio_mux(14, 0);                 // [14]: LCD_B6 as LCD_B6
    gpio_mux(15, 0);                 // [15]: LCD_B7 as LCD_B7
    gpio_mux(18, 0);                 // [18]: LCD_G2 as LCD_G2
```

```
    gpio_mux(19, 0);                // [19]: LCD_G3 as LCD_G3
    gpio_mux(20, 0);                // [20]: LCD_G4 as LCD_G4
    gpio_mux(21, 0);                // [21]: LCD_G5 as LCD_G5
    gpio_mux(22, 0);                // [22]: LCD_G6 as LCD_G6
    gpio_mux(23, 0);                // [23]: LCD_G7 as LCD_G7
    gpio_mux(27, 0);                // [27]: LCD_R3 as LCD_R3
    gpio_mux(28, 0);                // [28]: LCD_R4 as LCD_R4
    gpio_mux(29, 0);                // [29]: LCD_R5 as LCD_R5
    gpio_mux(30, 0);                // [30]: LCD_R6 as LCD_R6
    gpio_mux(31, 0);                // [31]: LCD_R7 as LCD_R7
    gpio_mux(40, 0);                // [40]: SPI0_CLK as SPI0_CLK
    gpio_mux(41, 0);                // [41]: SPI0_MISO as SPI0_MISO
    gpio_mux(42, 0);                // [42]: SPI0_MOSI as SPI0_MOSI
    gpio_mux(43, 0);                // [43]: SPI0_CS0 as SPI0_CS0
    gpio_mux(60, 0);                // [60]: UART2_TX as UART2_TX
    gpio_mux(61, 0);                // [61]: UART2_RX as UART2_RX
    gpio_mux(62, 0);                // [62]: UART3_TX as UART3_TX
    gpio_mux(63, 0);                // [63]: UART3_RX as UART3_RX
    gpio_mux(64, 0);                // [64]: I2C0_SCL as I2C0_SCL
    gpio_mux(65, 0);                // [65]: I2C0_SDA as I2C0_SDA
    gpio_mux(66, 0);                // [66]: CAN0_RX as CAN0_RX
    gpio_mux(67, 0);                // [67]: CAN0_TX as CAN0_TX
    gpio_mux(68, 0);                // [68]: CAN1_RX as CAN1_RX
    gpio_mux(69, 0);                // [69]: CAN1_TX as CAN1_TX
    gpio_mux(70, 1);                // [70]: LPC_AD0 as NAND_D0
    gpio_mux(71, 1);                // [71]: LPC_AD1 as NAND_D1
    gpio_mux(72, 1);                // [72]: LPC_AD2 as NAND_D2
    gpio_mux(73, 1);                // [73]: LPC_AD3 as NAND_D3
    gpio_mux(74, 1);                // [74]: LPC_FRAME as NAND_D4
    gpio_mux(75, 1);                // [75]: LPC_SERIRQ as NAND_D5
    gpio_mux(76, 0);                // [76]: NAND_CLE as NAND_CLE
    gpio_mux(77, 0);                // [77]: NAND_ALE as NAND_ALE
    gpio_mux(78, 0);                // [78]: NAND_RD as NAND_RD
    gpio_mux(79, 0);                // [79]: NAND_WR as NAND_WR
    gpio_mux(80, 0);                // [80]: NAND_CE0 as NAND_CE0
    gpio_mux(81, 0);                // [81]: NAND_RDY0 as NAND_RDY0
    gpio_mux(82, 0);                // [82]: NAND_D6 as NAND_D6
    gpio_mux(83, 0);                // [83]: NAND_D7 as NAND_D7
    gpio_mux(84, 2);                // [84]: PWM0 as GMAC0_COL
    gpio_mux(85, 2);                // [85]: PWM1 as GMAC0_CRS
    gpio_mux(88, 0);                // [88]: GMAC0_RX_CTL as GMAC0_RX_CTL
    gpio_mux(89, 0);                // [89]: GMAC0_RX0 as GMAC0_RX0
    gpio_mux(90, 0);                // [90]: GMAC0_RX1 as GMAC0_RX1
    gpio_mux(91, 0);                // [91]: GMAC0_RX2 as GMAC0_RX2
    gpio_mux(92, 0);                // [92]: GMAC0_RX3 as GMAC0_RX3
    gpio_mux(93, 0);                // [93]: GMAC0_TX_CTL as GMAC0_TX_CTL
    gpio_mux(94, 0);                // [94]: GMAC0_TX0 as GMAC0_TX0
    gpio_mux(95, 0);                // [95]: GMAC0_TX1 as GMAC0_TX1
    gpio_mux(96, 0);                // [96]: GMAC0_TX2 as GMAC0_TX2
    gpio_mux(97, 0);                // [97]: GMAC0_TX3 as GMAC0_TX3
    gpio_mux(98, 0);                // [98]: GMAC0_MDCK as GMAC0_MDCK
    gpio_mux(99, 0);                // [99]: GMAC0_MDIO as GMAC0_MDIO
    gpio_mux(133, 4);               // [133]: PCI_CBEN1 as CAN2_RX
    gpio_mux(134, 4);               // [134]: PCI_CBEN2 as CAN2_TX
    gpio_mux(135, 4);               // [135]: PCI_CBEN3 as CAN3_RX
    gpio_mux(136, 4);               // [136]: PCI_FRAMEN as CAN3_TX
}
```

```
/*
 * set gpio pin usage, TODO Should modify DIRECTION!
 */
static void ls2k_set_gpio_pins(void)
{
    gpio_enable(8, DIR_IN);     // Pin: LCD_B0
    gpio_enable(9, DIR_IN);     // Pin: LCD_B1
    gpio_enable(10, DIR_IN);    // Pin: LCD_B2
    gpio_enable(16, DIR_IN);    // Pin: LCD_G0
    gpio_enable(17, DIR_IN);    // Pin: LCD_G1
    gpio_enable(24, DIR_IN);    // Pin: LCD_R0
    gpio_enable(25, DIR_IN);    // Pin: LCD_R1
    gpio_enable(26, DIR_IN);    // Pin: LCD_R2
}

/*
 * LS2K500 Initialize function
 */
void ls2k_pins_initialize(void)
{
    ls2k_set_dpm_register();
    ls2k_set_config_register0();
    ls2k_set_muxed_pins();
    ls2k_set_gpio_pins();
}
```

版权声明