龙芯 1B200/1C300 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

2021 年 5 月

目录

前言

龙芯 1 系列芯片（以下简称龙芯 1x）是龙芯中科技术股份有限公司研发的 SoC 芯片，具有完全意义上的自主知识产权。该芯片使用 LS232 内核，全兼容 MIPS32 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 1x 处理器用户手册》。

LoongIDE 是专用于龙芯 1x 芯片的集成开发环境，旨在为龙芯 1x 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 1x 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 1x 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 1x 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 1x 芯片在工控行业的国产化应用。

本文档为 LoongIDE 提供的龙芯 1x 的设备驱动程序库 ls1x-drv 提供编程参考。

ls1x-drv 适用 LS1B200/LS1C300B 两款芯片，同时适用“裸机/RTThread/uCOS/FreeRTOS”四种编程环境。

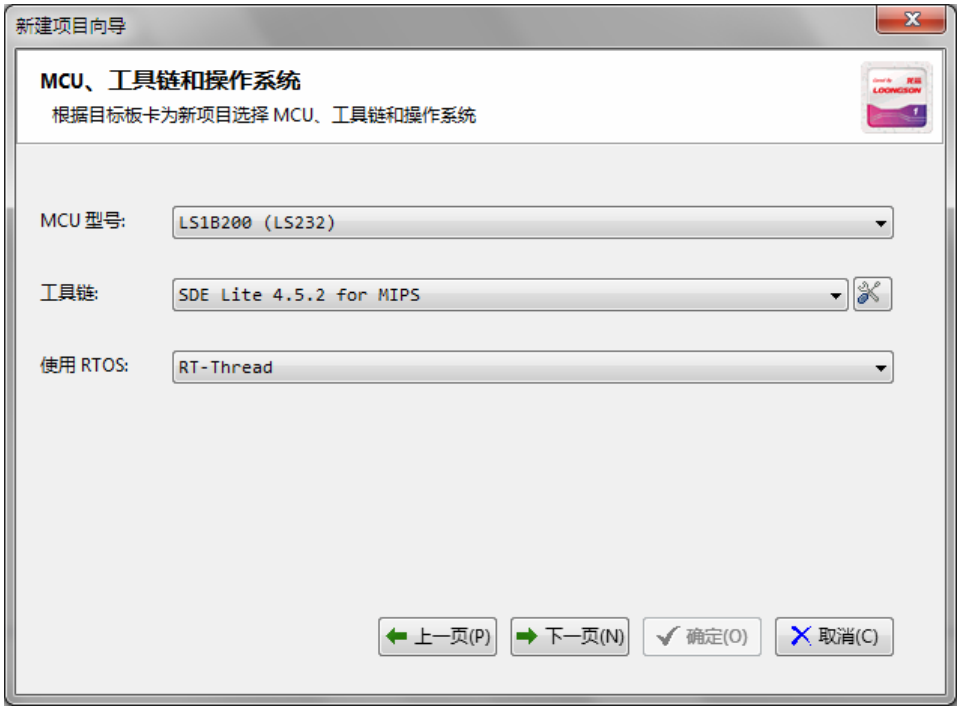
LoongIDE 软件是“龙芯嵌入式产业人才培养基地”认证产品。

在使用 LoongIDE 进行编程前请阅读“帮助”文档

第一节 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。



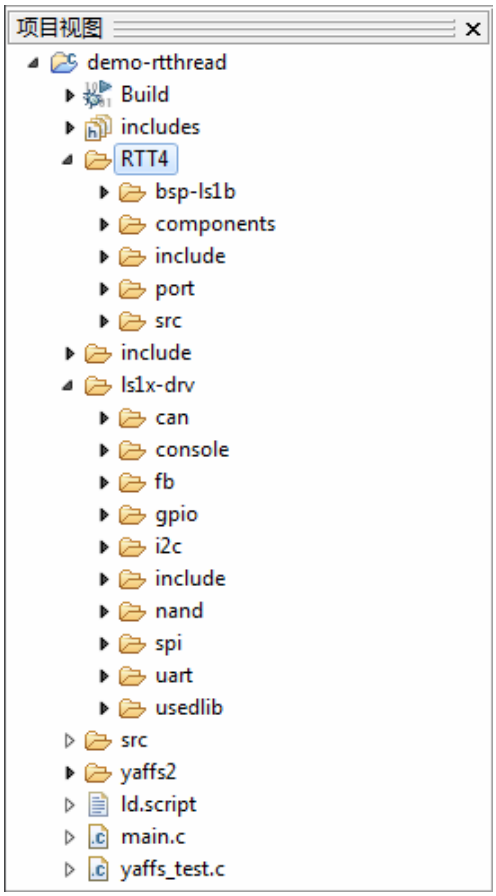
注意：芯片型号的选择必须与开发板上的实际型号一致；

以下文档以龙芯 1B 芯片为主进行介绍；龙芯 1C 芯片的差异请参阅源代码。

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。

- RTT4：RT-Thread 系统目录
  - bsp-ls1b：1B 驱动程序的 RTT 实现
  - components：RTT 系统组件
  - include：RTT 系统头文件
  - port：1B 芯片的 RTT 移植文件，实现 tick、stack、cache、interrupt 等处理
  - src：RTT 核心文件
- ls1x-drv：开发板设备的**通用驱动程序**
  - can：
  - console
  - fb
  - gpio
  - i2c
  - include
  - nand
  - spi
  - uart



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，其余均提供全部源代码，因实现方式不同，生成的项目目录与文件略有区别。

第二节 配置 BSP

配置文件为 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
```

否则关闭宏定义：

```
//#define BSP_USE_RTC
```

1、片上设备使用列表

宏定义	设备	相关模块	说明
//#define BSP_USE_SPI0	SPI 控制器 0	ls1x_spi_bus.h ls1x_spi_bus.c	
//#define BSP_USE_SPI1	SPI 控制器 1		
//#define BSP_USE_I2C0	I2C0 控制器 0	ls1x_i2c_bus.h ls1x_i2c_bus.c	
//#define BSP_USE_I2C1	I2C1 控制器 1		
//#define BSP_USE_I2C2	I2C2 控制器 2		
//#define BSP_USE_UART2	串口 2	ns16550.h ns16550.c	
#define BSP_USE_UART3	串口 3		
//#define BSP_USE_UART4	串口 4		
#define BSP_USE_UART5	串口 5		**控制台串口**
//#define BSP_USE_UART0	串口 0		串口 0 分为 4 个 2 线串口使用
//#define BSP_USE_UART01	串口 01		
//#define BSP_USE_UART02	串口 02		
//#define BSP_USE_UART03	串口 03		
//#define BSP_USE_UART1	串口 1		串口 1 分为 4 个 2 线串口使用
//#define BSP_USE_UART11	串口 11		
//#define BSP_USE_UART12	串口 12		
//#define BSP_USE_UART13	串口 13		
//#define BSP_USE_CAN0	CAN 控制器 0	ls1x_can.h ls1x_can.c	
//#define BSP_USE_CAN1	CAN 控制器 1		
//#define BSP_USE_NAND	NAND Flash 控制器	ls1x_nand.h ls1x_nand.c	
//#define BSP_USE_FB	LCD 控制器	ls1x_fb.h ls1x_fb.c	
//#define BSP_USE_GMAC0	GMAC 以太网控制器 0	ls1x_gmac.h ls1x_gmac.c	
//#define BSP_USE_GMAC1	GMAC 以太网控制器 1		
//#define BSP_USE_PWM0	PWM 控制器 0	ls1x_pwm.h	

//#define BSP_USE_PWM1	PWM 控制器 1	ls1x_pwm.c	
//#define BSP_USE_PWM2	PWM 控制器 2		
//#define BSP_USE_PWM3	PWM 控制器 3		
//#define BSP_USE_RTC	RTC 设备	ls1x_rtc.h ls1x_rtc.c	
//#define BSP_USE_AC97	AC97 控制器	ls1x_ac97.h ls1x_ac97.c	
//#define BSP_USE_WATCHDOG	看门狗电路	ls1x_watchdog.h ls1x_watchdog.c	

2、SPI0 总线上的从设备

要使用这些设备，必须打开 SPI0 宏定义：

```
#define BSP_USE_SPI0
```

宏定义	设备	相关模块	说明
#define W25X40_DRV	SPI Flash 芯片	w25x40.h w25x40.c	
#ifdef BSP_USE_FB #define XPT2046_DRV #endif	触摸控制芯片	xpt2046.h xpt2046.c	使用 LCD 时

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_I2C0
```

宏定义	设备	相关模块	说明
#ifdef BSP_USE_FB #define GP7101_DRV #endif	LCD 亮度控制芯片	gp7101.h gp7101.c	使用 LCD 时
#define PCA9557_DRV	GPIO 芯片	pca9557.h pca9557.c	
#define ADS1015_DRV	4 路 12bit ADC 芯片	ads1015.h ads1015.c	
#define MCP4725_DRV	1 路 12bit DAC 芯片	mcp4725.h mcp4725.c	
#define RX8010_DRV	RTC 芯片	rx8010.h rx8010.c	

4、其它关键配置

宏定义	说明
`#define BSP_USE_OS (!defined(OS_NONE))`	系统是否使用 RTOS
`#define BSP_USE_LWMEM (!defined(OS_RTTHREAD))`	是否使用堆、动态内存分配
`#define CPU_XTAL_FREQUENCY 24000000`	**芯片主晶振，必须与硬件一致**

第三节 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：rtconfig.h
- FreeRTOS 的配置文件：FreeRTOSConfig.h
- uCOSII 的配置文件：os_cfg.h

用户根据所选 RTOS 进行裁剪。

第四节 设备驱动程序

LoongIDE 实现的驱动程序在 ls1x-drv 目录下。

1、驱动模型

文件：ls1x-drv/include/ls1x-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)
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)
typedef struct libi2c_ops
{
    I2C_init_t          init;
    I2C_send_start_t    send_start;
    I2C_send_stop_t     send_stop;
    I2C_send_addr_t     send_addr;
    I2C_read_bytes_t    read_bytes;
    I2C_write_bytes_t   write_bytes;
    I2C_ioctl_t         ioctl;
} 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，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 ls1x-drv/spi、ls1x-drv/i2c 目录下芯片驱动的实现方式。

2、串口设备

源代码：ls1x-drv/uart/ns16550.c

头文件：ls1x-drv/include/ns16550.h

串口设备是否使用，在 bsp.h 中配置宏定义：

```
//#define BSP_USE_UART2
#define BSP_USE_UART3
//#define BSP_USE_UART4
#define BSP_USE_UART5                           // Console_Port 控制台串口
//#define BSP_USE_UART0
……
```

串口设备的参数，在 ns16550.c 中定义：

```
/* UART 5 */
#ifdef BSP_USE_UART5
static NS16550_t ls1b_UART5 =
{
    .BusClock  = 0,                             // 总线频率，初始化时填充
    .BaudRate  = 115200,                        // 默认速率
    .CtrlPort  = LS1B_UART5_BASE,               // 串口寄存器基址
    .DataPort  = LS1B_UART5_BASE,
    .bFlowCtrl = false,
    .ModemCtrl = 0,
    .bIntrrupt = true,                          // 是否使用中断
    .IntrNum   = LS1B_UART5_IRQ,                // 系统中断号
    .IntrCtrl  = LS1B_INTC0_BASE,               // 中断寄存器
    .IntrMask  = INTC0_UART5_BIT,               // 中断屏蔽位
    .dev_name  = "uart5",                       // 设备名称
};
void *devUART5 = &ls1b_UART5;
#endif
……
```

驱动程序 ns16550.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 0 或 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 0 或 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 *ls1x_uart_drv_ops;
    #define ls1x_uart_init(uart, arg)     ls1x_uart_drv_ops->init_entry(uart, arg)
    #define ls1x_uart_open(uart, arg)     ls1x_uart_drv_ops->open_entry(uart, arg)
    #define ls1x_uart_close(uart, arg)    ls1x_uart_drv_ops->close_entry(uart, arg)
    #define ls1x_uart_read(uart, buf, size, arg) \
            ls1x_uart_drv_ops->read_entry(uart, buf, size, arg)
    #define ls1x_uart_write(uart, buf, size, arg) \
            ls1x_uart_drv_ops->write_entry(uart, buf, size, arg)
    #define ls1x_uart_ioctl(uart, cmd, arg) \
            ls1x_uart_drv_ops->ioctl_entry(uart, cmd, arg)
#else
    #define ls1x_uart_init(uart, arg)             NS16550_init(uart, arg)
    #define ls1x_uart_open(uart, arg)             NS16550_open(uart, arg)
    #define ls1x_uart_close(uart, arg)            NS16550_close(uart, arg)
    #define ls1x_uart_read(uart, buf, size, arg)  NS16550_read(uart, buf, size, arg)
    #define ls1x_uart_write(uart, buf, size, arg) NS16550_write(uart, buf, size, arg)
    #define ls1x_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：

`#define ConsolePort devUARTx`	指定一个串口为串口控制台，用于 printf/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 *NS16550_get_dev_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：ls1x-drv/spi/ls1x_spi_bus.c
头文件：ls1x-drv/include/ls1x_spi_bus.h

SPI 设备是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_SPI0
//#define BSP_USE_SPI1
```

SPI 设备的参数，在 ls1x_spi_bus.c 中定义：

```
#ifdef BSP_USE_SPI0
static LS1x_SPI_bus_t ls1x_SPI0 =
{
    .hwSPI       = (struct LS1x_SPI_regs *)LS1x_SPI0_BASE,    /* 设备基地址 */
    .base_frq    = 0,                                         /* 总线频率 */
    .chipsel_nums = 4,                                        /* 片选总数 */
    .chipsel_high = 0,                                        /* 片选低有效 */
    .dummy_char   = 0,
    .irqNum      = LS1x_SPI0_IRQ,                             /* 中断号 */
    .int_ctrlr   = LS1x_INTC0_BASE,                           /* 中断控制寄存器 */
    .int_mask    = INTC0_SPI0_BIT,                            /* 中断屏蔽位 */
```

```
    .spi_mutex    = 0,                                    // 设备锁
    .initialized  = 0,                                    // 是否初始化
    .dev_name     = "spi0",                               // 设备名称
};
LS1x_SPI_bus_t *busSPI0 = &ls1x_SPI0;
#endif
……
```

驱动程序 ls1x_spi_bus.c 实现的函数：

函数
/* * 初始化SPI总线 * 参数: bus busSPI0 或者 busSPI1 * * 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 LS1x_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 或者 busSPI1 * 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 LS1x_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 或者 busSPI1 * 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 LS1x_SPI_send_stop(void *bus, unsigned int Addr);
/* * 读写SPI总线前发送片选信号 * 参数: bus busSPI0 或者 busSPI1 * 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 LS1x_SPI_send_addr(void *bus, unsigned int Addr, int rw);
/* * 从SPI从设备读取数据 * 参数: bus busSPI0 或者 busSPI1 * 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 LS1x_SPI_read_bytes(void *bus, unsigned char *rxbuf, int len);

```
/*
 * 向SPI从设备写入数据
 * 参数:    bus     busSPI0 或者 busSPI1
 *          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 LS1x_SPI_write_bytes(void *bus, unsigned char *txbuf, int len);
```

```
/*
 * 向SPI总线发送控制命令
 * 参数:    bus     busSPI0 或者 busSPI1
 *      ------------------------------------------------------------------------------
 *          cmd                             |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_SPI_I2C_SET_TFRMODE           |   类型: LS1x_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 LS1x_SPI_ioctl(void *bus, int cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    #define ls1x_spi_initialize(spi)              spi->ops->init(spi)
    #define ls1x_spi_send_start(spi, addr)        spi->ops->send_start(spi, addr)
    #define ls1x_spi_send_stop(spi, addr)         spi->ops->send_stop(spi, addr)
    #define ls1x_spi_send_addr(spi, addr, rw)     spi->ops->send_addr(spi, addr, rw)
    #define ls1x_spi_read_bytes(spi, buf, len)    spi->ops->read_bytes(spi, buf, len)
    #define ls1x_spi_write_bytes(spi, buf, len)   spi->ops->write_bytes(spi, buf, len)
    #define ls1x_spi_ioctl(spi, cmd, arg)         spi->ops->ioctl(spi, cmd, arg)
#else
    #define ls1x_spi_initialize(spi)              LS1x_SPI_initialize(spi)
    #define ls1x_spi_send_start(spi, addr)        LS1x_SPI_send_start(spi, addr)
    #define ls1x_spi_send_stop(spi, addr)         LS1x_SPI_send_stop(spi, addr)
    #define ls1x_spi_send_addr(spi, addr, rw)     LS1x_SPI_send_addr(spi, addr, rw)
    #define ls1x_spi_read_bytes(spi, buf, len)    LS1x_SPI_read_bytes(spi, buf, len)
    #define ls1x_spi_write_bytes(spi, buf, len)   LS1x_SPI_write_bytes(spi, buf, len)
    #define ls1x_spi_ioctl(spi, cmd, arg)         LS1x_SPI_ioctl(spi, cmd, arg)
#endif
```

实用函数

设置 SPIx_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 或者 busSPI1 */ **int ls1x_enable_spiflash_fastread(LS1x_SPI_bus_t *pSPI);**
/* * 取消SPI控制器的 Flash快速读模式 * 参数: bus busSPI0 或者 busSPI1 */ **int ls1x_disable_spiflash_fastread(LS1x_SPI_bus_t *pSPI);**

3.1 NorFlash 芯片 W25X40

源代码：ls1x-drv/spi/w25x40/w25x40.c

头文件：ls1x-drv/include/spi/w25x40.h

W25X40 是否使用，在 bsp.h 中配置宏定义：

```
#define W25X40_DRV
```

W25X40 连接在 **SPI0 的片选 0**，在 w25x40.c 中定义：

```
#define W25X40_CS       0
```

W25X40 的参数，在 w25x40.c 中定义：

芯片参数：

```
static w25x40_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 LS1x_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 w25x40.c 实现的函数：

函数
/* * 初始化W25X40芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25X40_init(void *dev, void *arg);
/* * 打开W25X40芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25X40_open(void *dev, void *arg);
/* * 关闭W25X40芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int W25X40_close(void *dev, void *arg);
/* * 从W25X40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(W25X40内部地址从0开始进行线性编址) * * 返回: 读取的字节数 */ int W25X40_read(void *dev, void *buf, int size, void *arg);

```
/*
 * 向W25X40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(W25X40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W25X40块已经格式化
 */
int W25X40_write(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向总线/W25X40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_W25X40_READ_ID            |   类型: unsigned int *
 *                                      |   用途: 读取W25X40芯片的ID
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25X40_ERASE_4K           |   类型: unsigned int
 *      IOCTL_W25X40_SECTOR_ERASE       |   用途: 擦除该地址所在的4K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25X40_ERASE_32K          |   类型: unsigned int
 *                                      |   用途: 擦除该地址所在的32K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25X40_ERASE_64K          |   类型: unsigned int
 *                                      |   用途: 擦除该地址所在的64K块
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25X40_BULK_ERASE         |   NULL, 擦除整块flash芯片
 *      ------------------------------------------------------------------------------
 *      IOCTL_W25X40_IS_BLANK           |   NULL, 检查是否为空
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int W25X40_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_w25x40_drv_ops;
    #define ls1x_w25x40_init(spi, arg)    ls1x_w25x40_drv_ops->init_entry(spi, arg)
    #define ls1x_w25x40_open(spi, arg)    ls1x_w25x40_drv_ops->open_entry(spi, arg)
    #define ls1x_w25x40_close(spi, arg)   ls1x_w25x40_drv_ops->close_entry(spi, arg)
    #define ls1x_w25x40_read(spi, buf, size, arg) \
            ls1x_w25x40_drv_ops->read_entry(spi, buf, size, arg)
    #define ls1x_w25x40_write(spi, buf, size, arg) \
            ls1x_w25x40_drv_ops->write_entry(spi, buf, size, arg)
    #define ls1x_w25x40_ioctl(spi, cmd, arg)  \
            ls1x_w25x40_drv_ops->ioctl_entry(spi, cmd, arg)
```

```
#else
    #define ls1x_w25x40_init(spi, arg)             W25X40_init(spi, arg)
    #define ls1x_w25x40_open(spi, arg)             W25X40_open(spi, arg)
    #define ls1x_w25x40_close(spi, arg)            W25X40_close(spi, arg)
    #define ls1x_w25x40_read(spi, buf, size, arg)  W25X40_read(spi, buf, size, arg)
    #define ls1x_w25x40_write(spi, buf, size, arg) W25X40_write(spi, buf, size, arg)
    #define ls1x_w25x40_ioctl(spi, cmd, arg)       W25X40_ioctl(spi, cmd, arg)
#endif
```

注：W25X40 芯片内部写有 PMON，用作开发板的 bootloader，占用空间约 268K；
用户可以使用的地址空间必须在 bootloader 之上，例如 300K~512K。

W25X40_read 和 W25X40_write 函数的 **arg** 参数：

W25X40 的片内 FLASH 地址空间从 0 到 512K 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。

W25X40_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 W25X40_ioctl 函数的实现。

以下函数用于保存触摸屏校正值：

```
#define TOUCHSCREEN_CAL_POS   (501*1024)   // 数据保存地址
```

函数
/* * 把触摸屏校正数据保存到W25X40的地址(501*1024)处 */ int save_touchscreen_calibrate_values_to_w25x40(int *calibrate_coords, int len);
/* * 读出上面函数保存的触摸屏校正数据 */ int load_touchscreen_calibrate_values_from_w25x40(int *calibrate_coords, int len);

3.2 触摸屏芯片 XPT2046

源代码：ls1x-drv/spi/xpt2046/xpt1046.c

头文件：ls1x-drv/include/spi/xpt2046.h

XPT2046 是否使用，在 bsp.h 中配置宏定义：

```
#define XPT2046_DRV
```

XPT2046 连接在 **SPI0 的片选 1**，在 xpt1046.c 中定义：

```
#define XPT2046_CS      1
```

XPT2046 的通信参数，在 XPT2046_read 函数中定义：

通信参数：

```
LS1x_SPI_mode_t tfr_mode =
{
    baudrate:       2000000,        /* 通信速率 2M */
    bits_per_char:  8,              /* 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 xpt1046.c 实现的函数：

函数
/* * 初始化XPT2046芯片 * 参数: dev busSPI0 * arg NULL * * 返回: 0=成功 */ int XPT2046_initialize(void *dev, void *arg);
/* * 从XPT2046芯片读数据(触摸点) * 参数: dev busSPI0 * buf 类型: ts_message_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(ts_message_t)倍数 * arg NULL * * 返回: 读取的字节数 */ int W25X40_read(void *dev, void *buf, int size, void *arg);

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_xpt2046_drv_ops;
    #define ls1x_xpt2046_init(spi, arg)          ls1x_xpt2046_drv_ops->init_entry(spi, arg)
    #define ls1x_xpt2046_read(spi, buf, size, arg) \
            ls1x_xpt2046_drv_ops->read_entry(spi, buf, size, arg)
#else
    #define ls1x_xpt2046_init(spi, arg)           XPT2046_initialize(spi, arg)
    #define ls1x_xpt2046_read(spi, buf, size, arg  XPT2046_read(spi, buf, size, arg)
#endif
```

注：XPT2046 芯片设计在 LCD 转接板上，通过 40Pins FPC 接口和主板连接；
XPT2046 的 PENIRQ 信号使用 GPIO54(UART2_RX)连接芯片：

```
#define XPT2046_USE_GPIO_INT      0          /* 触摸屏 GPIO 中断支持 */
#define XPT2046_USE_GPIO_NUM      54         /* 触摸屏 GPIO 端口 */
```

3.3 触摸屏实用程序

源代码：ls1x-drv/spi/xpt2046/touch_utils.c
头文件：ls1x-drv/include/spi/xpt2046.h

触摸信号使用消息发送的宏定义：

```
#define TOUCHSCREEN_USE_MESSAGE   0
```

触摸屏回调函数原型：

```
/*
 * 触摸事件回调函数
 * 参数:    x, y    触摸坐标
 */
typedef void (*touch_callback_t)(int x, int y);
```

实用函数：

函数
/* * 触摸屏校正函数 */ int do_touchscreen_calibrate(void);
/* * 在RTOS编程下，启动触摸屏任务 * 参数: cb   类型: touch_callback_t，发生触摸事件时该回调函数供用户处理 */ int start_touchscreen_task(touch_callback_t cb);

```
/*
 * 在RTOS编程下，停止触摸屏任务
 */
int stop_touchscreen_task(void);
```

```
/*
 * 在裸机编程下，获取发生触摸事件的坐标
 */
extern int bare_get_touch_point(int *x, int *y);
```

4、I2C 设备

源代码：ls1x-drv/i2c/ls1x_i2c_bus.c

头文件：ls1x-drv/include/ls1x_i2c_bus.h

I2C 设备是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_I2C0
//#define BSP_USE_I2C1
//#define BSP_USE_I2C2
```

I2C 设备的参数，在 ls1x_i2c_bus.c 中定义：

```
#ifdef BSP_USE_I2C0
static LS1x_I2C_bus_t ls1x_I2C0 =
{
    .hwI2C       = (struct LS1x_I2C_regs *)LS1x_I2C0_BASE,    /* 设备基地址 */
    .base_frq    = 0,                                         /* 总线频率 */
    .baudrate    = 100000,                                    /* 通信速率 */
    .dummy_char  = 0,
    .i2c_mutex   = 0,                                         // 设备锁
    .initialized = 0,                                         // 是否初始化
    .dev_name    = "i2c0",                                    // 设备名称
};
LS1x_I2C_bus_t *busI2C0 = &ls1x_I2C0;
#endif
……
```

驱动程序 ls1x_i2c_bus.c 实现的函数：

函数
/* * 初始化I2C总线 * 参数: bus busI2C0/busI2C1/busI2C2 * * 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 LS1x_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 * 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 LS1x_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 * 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 LS1x_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 * 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 LS1x_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 * 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 LS1x_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 * 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 LS1x_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
 *      ------------------------------------------------------------------------------
 *          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 LS1x_I2C_ioctl(void *bus, int cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    #define ls1x_i2c_initialize(i2c)              i2c->ops->init(i2c)
    #define ls1x_i2c_send_start(i2c, addr)        i2c->ops->send_start(i2c, addr)
    #define ls1x_i2c_send_stop(i2c, addr)         i2c->ops->send_stop(i2c, addr)
    #define ls1x_i2c_send_addr(i2c, addr, rw)     i2c->ops->send_addr(i2c, addr, rw)
    #define ls1x_i2c_read_bytes(i2c, buf, len)    i2c->ops->read_bytes(i2c, buf, len)
    #define ls1x_i2c_write_bytes(i2c, buf, len)   i2c->ops->write_bytes(i2c, buf, len)
    #define ls1x_i2c_ioctl(i2c, cmd, arg)         i2c->ops->ioctl(i2c, cmd, arg)
#else
    #define ls1x_i2c_initialize(i2c)              LS1x_I2C_initialize(i2c)
    #define ls1x_i2c_send_start(i2c, addr)        LS1x_I2C_send_start(i2c, addr)
    #define ls1x_i2c_send_stop(i2c, addr)         LS1x_I2C_send_stop(i2c, addr)
    #define ls1x_i2c_send_addr(i2c, addr, rw)     LS1x_I2C_send_addr(i2c, addr, rw)
    #define ls1x_i2c_read_bytes(i2c, buf, len)    LS1x_I2C_read_bytes(i2c, buf, len)
    #define ls1x_i2c_write_bytes(i2c, buf, len)   LS1x_I2C_write_bytes(i2c, buf, len)
    #define ls1x_i2c_ioctl(i2c, cmd, arg)         LS1x_I2C_ioctl(i2c, cmd, arg)
#endif
```

4.1 GPIO 芯片 PCA9557

源代码：ls1x-drv/i2c/pca9557/pca9557.c

头文件：ls1x-drv/include/i2c/pca9557.h

PCA9557 是否使用，在 bsp.h 中配置宏定义：

```
#define PCA9557_DRV
```

PCA9557 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（pca9557.c）：

```
#define PCA9557_ADDRESS         0x1C            /* 7 位地址 */
#define PCA9557_BAUDRATE        100000          /* 100K */
```

驱动程序 pca9557.c 实现的函数：

函数
/* * 初始化PCA9557芯片，根据PCA9557的硬件设计初始化GPIO输入/输出端口 * 参数: bus   busI2C0 *   arg   NULL * * 返回: 0=成功 */ int PCA9557_init(void *dev, void *arg);
/* * 从PCA9557芯片读 1 字节输入端口数据 * 参数: bus   busI2C0 *   buf   类型: unsigned char * *   size   1 *   arg   NULL * * 返回: 0=成功 */ int PCA9557_read(void *dev, void *buf, int size, void *arg);
/* * 向PCA9557芯片输出端口写 1 字节数据 * 参数: bus   busI2C0 *   buf   类型: unsigned char * *   size   1 *   arg   NULL * * 返回: 0=成功 */ int PCA9557_write(void *dev, void *buf, int size, void *arg);

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_pca9557_drv_ops;
    #define ls1x_pca9557_init(iic, arg)       ls1x_pca9557_drv_ops->init_entry(iic, arg)
    #define ls1x_pca9557_read(iic, buf, size, arg) \
            ls1x_pca9557_drv_ops->read_entry(iic, buf, size, arg)
    #define ls1x_pca9557_write(iic, buf, size, arg) \
            ls1x_pca9557_drv_ops->write_entry(iic, buf, size, arg)
#else
    #define ls1x_pca9557_init(iic, arg)               PCA9557_init(iic, arg)
    #define ls1x_pca9557_read(iic, buf, size, arg)    PCA9557_read(iic, buf, size, arg)
    #define ls1x_pca9557_write(iic, buf, size, arg)   PCA9557_write(iic, buf, size, arg)
#endif
```

注：PCA9557 用作 GPIO 扩展，设计和使用本芯片的目的是不占用 1B 芯片的 GPIO 资源。

实用函数：

函数	功能
`bool get_ads1015_ready(void *bus);`	读取 ads1015 芯片 AD 转换状态 – 1B 开发板
`bool get_rx8010_irq1(void *bus);`	读取 rx8010 芯片的定时中断 1 状态
`bool get_rx8010_irq2(void *bus);`	读取 rx8010 芯片的定时中断 2 状态
`bool get_usb_otg_id(void *bus)`	– 1C 开发板
`bool get_touch_down(void *bus)`	– 1C 开发板 触摸屏触摸信号
`bool set_usb_otg_vbus(void *bus, bool en)`	– 1C 开发板
`bool set_camera_reset(void *bus, bool en)`	– 1C 开发板
`bool set_camera_powerdown(void *bus, bool en)`	– 1C 开发板

4.2 PWM 芯片 GP7101

源代码：ls1x-drv/i2c/gp7101/gp7101.c
头文件：ls1x-drv/include/i2c/gp7101.h

GP7101 是否使用，在 bsp.h 中配置宏定义：

```
#define GP7101_DRV
```

GP7101 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（gp7101.c）：

```
#define GP7101_ADDRESS          0x58            /* 7 位地址 */
#define GP7101_BAUDRATE         100000          /* 100K */
```

驱动程序 gp7101.c 实现的函数：

函数	功能
`int GP7101_write(void *dev, void *buf, int size, void *arg);`	写数据

注：GP7101 芯片输出 100Hz 的 PWM 信号，用于触摸屏亮度控制。

实用函数：

函数	功能
`int set_lcd_brightness(void *bus, int brightpercent);`	LCD 亮度控制 参数：brightpercent：1～100

4.3 ADC 芯片 ADS1015

源代码：ls1x-drv/i2c/ads1015/ ads1015.c

头文件：ls1x-drv/include/i2c/ ads1015.h

ADS1015 是否使用，在 bsp.h 中配置宏定义：

```
#define ADS1015_DRV
```

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 *ls1x_ads1015_drv_ops;
    #define ls1x_ads1015_read(iic, buf, size, arg) \
            ls1x_ads1015_drv_ops->read_entry(iic, buf, size, arg)
    #define ls1x_ads1015_ioctl(iic, cmd, arg) \
            ls1x_ads1015_drv_ops->ioctl_entry(iic, cmd, arg)
#else
    #define ls1x_ads1015_read(iic, buf, size, arg)     ADS1015_read(iic, buf, size, arg)
    #define ls1x_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 = ls1x_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);
ls1x_ads1015_ioctl(busI2C0, ADS1015_DISP_CONFIG_REG, NULL);          // 显示
```

4.4 DAC 芯片 MCP4725

源代码：ls1x-drv/i2c/mcp4725/mcp4725.c

头文件：ls1x-drv/include/i2c/mcp4725.h

MCP4725 是否使用，在 bsp.h 中配置宏定义：

```
#define MCP4725_DRV
```

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 *ls1x_mcp4725_drv_ops;
    #define ls1x_mcp4725_write(iic, buf, size, arg) \
            ls1x_mcp4725_drv_ops->write_entry(iic, buf, size, arg)
    #define ls1x_mcp4725_ioctl(iic, cmd, arg) \
            ls1x_mcp4725_drv_ops->ioctl_entry(iic, cmd, arg)
#else
    #define ls1x_mcp4725_write(iic, buf, size, arg)    MCP4725_write(iic, buf, size, arg)
    #define ls1x_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);
```

4.5 RTC 芯片 RX8010SJ

源代码：ls1x-drv/i2c/rx8010/ rx8010.c

头文件：ls1x-drv/include/i2c/ rx8010.h

RX8010 是否使用，在 bsp.h 中配置宏定义：

```
#define RX8010_DRV
```

RX8010 挂接在 **I2C0** 上，I2C 地址和通信速率定义如下（rx8010.c）：

```
#define RX8010_ADDRESS        0x32            /* 7 位地址 */
#define RX8010_BAUDRATE       100000          /* 100K */
```

驱动程序 rx8010.c 实现的函数：

函数	功能
int RX8010_initialize(void *dev, void *arg);	初始化
int RX8010_read(void *dev, void *buf, int size, void *arg);	读日期
int RX8010_write(void *dev, void *buf, int size, void *arg);	写日期
int RX8010_ioctl(void *dev, int cmd, void *arg);	

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_rx8010_drv_ops;
    #define ls1x_rx8010_init(iic, arg)    ls1x_rx8010_drv_ops->init_entry(iic, arg)
    #define ls1x_rx8010_read(iic, buf, size, arg) \
            ls1x_rx8010_drv_ops->read_entry(iic, buf, size, arg)
```

```
    #define ls1x_rx8010_write(iic, buf, size, arg) \
            ls1x_rx8010_drv_ops->write_entry(iic, buf, size, arg)
    #define ls1x_rx8010_ioctl(iic, cmd, arg) \
            ls1x_rx8010_drv_ops->ioctl_entry(iic, cmd, arg)
#else
    #define ls1x_rx8010_init(iic, arg)             RX8010_initialize(iic, arg)
    #define ls1x_rx8010_read(iic, buf, size, arg)  RX8010_read(iic, buf, size, arg)
    #define ls1x_rx8010_write(iic, buf, size, arg) RX8010_write(iic, buf, size, arg)
    #define ls1x_rx8010_ioctl(iic, cmd, arg)       RX8010_ioctl(iic, cmd, arg)
#endif
```

RX8010_read 和 RX8010_write 函数的 buf 参数类型：**struct tm *** 。

实用函数：

函数
/* * 向 RX8010 写入系统日期 */ **int RX8010_set_time**(**void** *bus, **struct tm** *dt);
/* * 从 RX8010 读取系统日期 */ **int RX8010_get_time**(**void** *bus, **struct tm** *dt);

5、NAND 控制设备

源代码：ls1x-drv/nand/ls1x_nand.c

头文件：ls1x-drv/include/ls1x_nand.h

NAND 连接的 Flash 芯片参数：ls1x-drv/nand/nand_k9f1g08.h

NAND 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_NAND
```

驱动程序 ls1x_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 LS1x_NAND_initialize(void *dev, void *arg);
/* * 打开NAND设备 * 参数: dev devNAND * arg NULL * * 返回: 0=成功 */ int LS1x_NAND_open(void *dev, void *arg);
/* * 关闭NAND设备 * 参数: dev devNAND * arg NULL * * 返回: 0=成功 */ int LS1x_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 LS1x_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 LS1x_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 LS1x_NAND_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_nand_drv_ops;
    #define ls1x_nand_init(nand, arg)      ls1x_nand_drv_ops->init_entry(nand, arg)
    #define ls1x_nand_open(nand, arg)      ls1x_nand_drv_ops->open_entry(nand, arg)
    #define ls1x_nand_close(nand, arg)     ls1x_nand_drv_ops->close_entry(nand, arg)
    #define ls1x_nand_read(nand, buf, size, arg) \
            ls1x_nand_drv_ops->read_entry(nand, buf, size, arg)
    #define ls1x_nand_write(nand, buf, size, arg) \
            ls1x_nand_drv_ops->write_entry(nand, buf, size, arg)
```

```
    #define ls1x_nand_ioctl(nand, cmd, arg) \
            ls1x_nand_drv_ops->ioctl_entry(nand, cmd, arg)
#else
    #define ls1x_nand_init(nand, arg)             LS1x_NAND_initialize(nand, arg)
    #define ls1x_nand_open(nand, arg)             LS1x_NAND_open(nand, arg)
    #define ls1x_nand_close(nand, arg)            LS1x_NAND_close(nand, arg)
    #define ls1x_nand_read(nand, buf, size, arg)  LS1x_NAND_read(nand, buf, size, arg)
    #define ls1x_nand_write(nand, buf, size, arg) LS1x_NAND_write(nand, buf, size, arg)
    #define ls1x_nand_ioctl(nand, cmd, arg)       LS1x_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 ls1x_yaffs.c。

6、显示控制器

源代码：ls1x-drv/fb/ls1x_fb.c

头文件：ls1x-drv/include/ls1x_fb.h

framebuffer 实用函数：ls1x-drv/fb/ls1x_fb_utils.c

framebuffer 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_FB
```

驱动程序 ls1x_fb.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 LS1x_DC_init(void *dev, void *arg);
/* * 打开DC设备 * 参数: dev devDC * arg NULL * * 返回: 0=成功 */ int LS1x_DC_open(void *dev, void *arg);
/* * 关闭DC设备 * 参数: dev devDC * arg NULL * * 返回: 0=成功 */ int LS1x_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 LS1x_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 LS1x_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 LS1x_DC_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数:

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_dc_drv_ops;
    #define ls1x_dc_init(dc, arg)              ls1x_dc_drv_ops->init_entry(dc, arg)
    #define ls1x_dc_open(dc, arg)              ls1x_dc_drv_ops->open_entry(dc, arg)
    #define ls1x_dc_close(dc, arg)             ls1x_dc_drv_ops->close_entry(dc, arg)
    #define ls1x_dc_read(dc, buf, size, arg) \
            ls1x_dc_drv_ops->read_entry(dc, buf, size, arg)
    #define ls1x_dc_write(dc, buf, size, arg) \
            ls1x_dc_drv_ops->write_entry(dc, buf, size, arg)
    #define ls1x_dc_ioctl(dc, cmd, arg)        ls1x_dc_drv_ops->ioctl_entry(dc, cmd, arg)
#else
    #define ls1x_dc_init(dc, arg)              LS1x_DC_initialize(dc, arg)
```

```
    #define ls1x_dc_open(dc, arg)             LS1x_DC_open(dc, arg)
    #define ls1x_dc_close(dc, arg)            LS1x_DC_close(dc, arg)
    #define ls1x_dc_read(dc, buf, size, arg)  LS1x_DC_read(dc, buf, size, arg)
    #define ls1x_dc_write(dc, buf, size, arg) LS1x_DC_write(dc, buf, size, arg)
    #define ls1x_dc_ioctl(dc, cmd, arg)       LS1x_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 ls1x_dc_initialized(void);`

`/*`
` * 显示控制器是否启动(open)`
` * 返回:  1=已启动; 0=未启动`
` */`
`int ls1x_dc_started(void);`

头文件 ls1x_fb.h 中的 LCD 工作模式宏定义：

```
#define LCD_480x272   "480x272-16@60"    /* Fit: 4 inch 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：

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

ls1x_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”+”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

FrameBuffer 函数的使用，参考 simple_gui.c。

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：ls1x-drv/can/ls1x_can.c
头文件：ls1x-drv/include/ls1x_can.h

CAN 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_CAN0
//#define BSP_USE_CAN1
```

CAN 设备参数定义在中 ls1x_can.c 定义：

```
#ifdef BSP_USE_CAN0
static CAN_t ls1x_CAN0 =
{
    .hwCAN     = (LS1x_CAN_regs_t *)LS1x_CAN0_BASE,          // 寄存器基址
    .irqNum    = LS1x_CAN0_IRQ,                              // 中断号
    .int_ctrlr = LS1x_INTC0_BASE,                            // 中断控制寄存器
    .int_mask  = INTC0_CAN0_BIT,                             // 中断屏蔽位

    .rxfifo      = NULL,                                     // 接收 FIFO
    .txfifo      = NULL,                                     // 发送 FISO
    .initialized = 0,                                        // 是否初始化
    .dev_name    = "can0",                                   // 设备名称
};
void *devCAN0 = (void *)&ls1x_CAN0;
#endif

……
```

驱动程序 ls1x_can.c 实现的函数：

函数
/* * 初始化CAN设备 * 参数: dev devCAN0/devCAN1 * 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 LS1x_CAN_init(void *dev, void *arg);

```
/*
 * 打开CAN设备
 * 参数:    dev     devCAN0/devCAN1
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int LS1x_CAN_open(void *dev, void *arg);
```

```
/*
 * 关闭CAN设备
 * 参数:    dev     devCAN0/devCAN1
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int LS1x_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 LS1x_CAN_read(void *dev, void *buf, int size, void *arg);
```

```
/*
 * 向CAN设备写数据(发送)
 * 参数:    dev     devCAN0/devCAN1
 *          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 LS1x_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
 *
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_CAN_START             |   NULL. 启动CAN进入工作状态, 注意 ls1x_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ls1x_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 LS1x_CAN_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_can_drv_ops;
    #define ls1x_can_init(can, arg)       ls1x_can_drv_ops->init_entry(can, arg)
    #define ls1x_can_open(can, arg)       ls1x_can_drv_ops->open_entry(can, arg)
    #define ls1x_can_close(can, arg)      ls1x_can_drv_ops->close_entry(can, arg)
    #define ls1x_can_read(can, buf, size, arg) \
            ls1x_can_drv_ops->read_entry(can, buf, size, arg)
    #define ls1x_can_write(can, buf, size, arg) \
            ls1x_can_drv_ops->write_entry(can, buf, size, arg)
    #define ls1x_can_ioctl(can, cmd, arg) \
            ls1x_can_drv_ops->ioctl_entry(can, cmd, arg)
#else
    #define ls1x_can_init(can, arg)               LS1x_CAN_init(can, arg)
    #define ls1x_can_open(can, arg)               LS1x_CAN_open(can, arg)
    #define ls1x_can_close(can, arg)              LS1x_CAN_close(can, arg)
    #define ls1x_can_read(can, buf, size, arg)    LS1x_CAN_read(can, buf, size, arg)
    #define ls1x_can_write(can, buf, size, arg)   LS1x_CAN_write(can, buf, size, arg)
    #define ls1x_can_ioctl(can, cmd, arg)         LS1x_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
- ……

8、网络控制器

源代码：ls1x-drv/gmac/ls1x_gmac.c
头文件：ls1x-drv/include/ls1x_gmac.h

GMAC 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_GMAC0
//#define BSP_USE_GMAC1
```

GMAC 设备参数定义在中 ls1x_gmac.c 定义：

```
#if defined(BSP_USE_GMAC0)
static GMAC_t ls1x_GMAC0 =
{
    .hwGMAC      = (LS1x_gmac_regs_t *)LS1B_GMAC0_BASE,      // GMAC 基址
    .hwGDMA      = (LS1x_gdma_regs_t *)LS1x_GDMA0_BASE,      // GDMA 基址
    .int_ctrlr   = LS1B_INTC1_BASE,                          // 中断控制寄存器
    .int_mask    = INTC1_GMAC0_BIT,                          // 中断屏蔽位
    .vector      = LS1B_GMAC0_IRQ,                           // 中断号
     .descmode    = CHAINMODE,                               // DMA 描述符模式
    .dev_name    = "gmac0",

    ......
};
void *devGMAC0 = (void *)&ls1x_GMAC0;
#endif

......
```

驱动程序 ls1x_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 LS1x_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 LS1x_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 LS1x_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 LS1x_GMAC_ioctl(void *dev, unsigned cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_gmac_drv_ops;
    #define ls1x_gmac_init(gmac, arg) \
            ls1x_gmac_drv_ops->init_entry(gmac, arg)
    #define ls1x_gmac_read(gmac, buf, size, arg) \
            ls1x_gmac_drv_ops->read_entry(gmac, buf, size, arg)
    #define ls1x_gmac_write(gmac, buf, size, arg) \
```

```
            ls1x_gmac_drv_ops->write_entry(gmac, buf, size, arg)
    #define ls1x_gmac_ioctl(gmac, cmd, arg) \
            ls1x_gmac_drv_ops->ioctl_entry(gmac, cmd, arg)
#else
    #define ls1x_gmac_init(gmac, arg)              LS1x_GMAC_initialize(gmac, arg)
    #define ls1x_gmac_read(gmac, buf, size, arg)   LS1x_GMAC_read(gmac, buf, size, arg)
    #define ls1x_gmac_write(gmac, buf, size, arg)  LS1x_GMAC_write(gmac, buf, size, arg)
    #define ls1x_gmac_ioctl(gmac, cmd, arg)        LS1x_GMAC_ioctl(gmac, cmd, arg)
#endif
```

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 LS1x_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 LS1x_GMAC_wait_transmit_idle(void *dev, unsigned char **bufptr);

用户接口：

```
#define ls1x_gmac_wait_rx_packet(gmac, pbuf)  LS1x_GMAC_wait_receive_packet(gmac, pbuf)
#define ls1x_gmac_wait_tx_idle(gmac, pbuf)    LS1x_GMAC_wait_transmit_idle(gmac, pbuf)
```

GMAC 驱动程序已适配 lwIP-1.4.1，参见 ls1x_ethernetif.c

9、PWM 设备

源代码：ls1x-drv/pwm/ls1x_pwm.c

头文件：ls1x-drv/include/ls1x_pwm.h

PWM 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_PWM0
//#define BSP_USE_PWM1
//#define BSP_USE_PWM2
//#define BSP_USE_PWM3
```

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/devPWM2/devPWM3
 *         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 open 函数的 arg 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 ls1x_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）。
- 龙芯 1B 芯片的 GMAC0 复用 PWM0 和 PWM1；GMAC1 复用 PWM2 和 PWM3；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 ls1x_pwm.c 实现的函数：

函数
/* * PWM初始化 * 参数:　dev　　devPWM0/devPWM1/devPWM2/devPWM3 *　　arg　　总是 NULL * * 返回:　0=成功 */ int LS1x_PWM_initialize(void *dev, void *arg);
/* * 打开PWM设备 * 参数:　dev　　devPWM0/devPWM1/devPWM2/devPWM3 *　　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 LS1x_PWM_open(void *dev, void *arg);
```

```
/*
 * 关闭PWM定时器
 * 参数:    dev     devPWM0/devPWM1/devPWM2/devPWM3
 *          arg     NULL
 *
 * 返回:    0=成功
 */
int LS1x_PWM_close(void *dev, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_pwm_drv_ops;
    #define ls1x_pwm_init(pwm, arg)       ls1x_pwm_drv_ops->init_entry(pwm, arg)
    #define ls1x_pwm_open(pwm, arg)       ls1x_pwm_drv_ops->open_entry(pwm, arg)
    #define ls1x_pwm_close(pwm, arg)  ls1x_pwm_drv_ops->close_entry(pwm, arg)
#else
    #define ls1x_pwm_init(pwm, arg)       LS1x_PWM_initialize(pwm, arg)
    #define ls1x_pwm_open(pwm, arg)       LS1x_PWM_open(pwm, arg)
    #define ls1x_pwm_close(pwm, arg)      LS1x_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/devPWM2/devPWM3
 *          cfg     类型: pwm_cfg_t *, mode==PWM_SINGLE_PULSE/PWM_CONTINUE_PULSE
 *
 * 返回:    0=成功
 */
int ls1x_pwm_pulse_start(void *pwm, pwm_cfg_t *cfg);
```

```
/*
 * 停止PWM设备产生脉冲
 * 参数:    dev     devPWM0/devPWM1/devPWM2/devPWM3
 *
 * 返回:    0=成功
 */
int ls1x_pwm_pulse_stop(void *pwm);
```

```
/*
 * 启动PWM定时器
 * 参数:    dev     devPWM0/devPWM1/devPWM2/devPWM3
 *          cfg     类型: pwm_cfg_t *, mode==PWM_SINGLE_TIMER/PWM_CONTINUE_TIMER
 *
 * 返回:    0=成功
 */
int ls1x_pwm_timer_start(void *pwm, pwm_cfg_t *cfg);
```

```
/*
 * 停止PWM定时器
 * 参数:    dev     devPWM0/devPWM1/devPWM2/devPWM3
 *
 * 返回:    0=成功
 */
int ls1x_pwm_timer_stop(void *pwm);
```

10、实时时钟设备

源代码：ls1x-drv/rtc/ls1x_rtc.c

头文件：ls1x-drv/include/ls1x_rtc.h

RTC 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_RTC
```

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 驱动的自定义 arg 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 ls1x_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 ls1x_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 LS1x_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 LS1x_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 LS1x_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 LS1x_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 LS1x_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 LS1x_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 *ls1x_rtc_drv_ops;
    #define ls1x_rtc_init(rtc, arg)            ls1x_rtc_drv_ops->init_entry(rtc, arg)
    #define ls1x_rtc_open(rtc, arg)            ls1x_rtc_drv_ops->open_entry(rtc, arg)
    #define ls1x_rtc_close(rtc, arg)           ls1x_rtc_drv_ops->close_entry(rtc, arg)
    #define ls1x_rtc_read(rtc, buf, size, arg)  \
            ls1x_rtc_drv_ops->read_entry(rtc, buf, size, arg)
    #define ls1x_rtc_write(rtc, buf, size, arg) \
            ls1x_rtc_drv_ops->write_entry(rtc, buf, size, arg)
    #define ls1x_rtc_ioctl(rtc, cmd, arg)      ls1x_rtc_drv_ops->ioctl_entry(rtc, cmd, arg)
#else
    #define ls1x_rtc_init(rtc, arg)             LS1x_RTC_initialize(rtc, arg)
    #define ls1x_rtc_open(rtc, arg)             LS1x_RTC_open(rtc, arg)
    #define ls1x_rtc_close(rtc, arg)            LS1x_RTC_close(rtc, arg)
    #define ls1x_rtc_read(rtc, buf, size, arg)  LS1x_RTC_read(rtc, buf, size, arg)
    #define ls1x_rtc_write(rtc, buf, size, arg) LS1x_RTC_write(rtc, buf, size, arg)
    #define ls1x_rtc_ioctl(rtc, cmd, arg)       LS1x_RTC_ioctl(rtc, cmd, arg)
#endif
```

RTC 实用函数：

函数
/* * 设置RTC时钟值，参见ls1x_rtc_read() */ int ls1x_rtc_set_datetime(struct tm *dt);
/* * 获取当前RTC时间，参见ls1x_rtc_write() */ int ls1x_rtc_get_datetime(struct tm *dt);
/* * 开启定时器，参见ls1x_rtc_open() */ int ls1x_rtc_timer_start(unsigned device, rtc_cfg_t *cfg);
/* * 关闭定时器，参见ls1x_rtc_close() */ int ls1x_rtc_timer_stop(unsigned device);
/* * struct tm 日期格式转换为 toymatch */ void ls1x_tm_to_toymatch(struct tm *dt, unsigned int *match);
/* * toymatch 日期格式转换为 struct tm */ void ls1x_toymatch_to_tm(struct tm *dt, unsigned int match);
/* * 秒数转换为 toymatch */ unsigned int ls1x_seconds_to_toymatch(unsigned int seconds);
/* * toymatch 转换为秒数 */ unsigned int ls1x_toymatch_to_seconds(unsigned int match);
/* * struct tm 日期标准化，+1900/-1900 */ void normalize_tm(struct tm *tm, bool tm_format);

11、AC97 声音设备

源代码：ls1x-drv/ac97/ls1x_ac97.c

头文件：ls1x-drv/include/ls1x_ac97.h

AC97 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_AC97
```

AC97 open 的 arg 参数：

```
typedef struct ac97_param
{
    unsigned char *buffer;          /* raw music data buffer */
    int buf_length;                 /* buffer length */
    int num_channels;               /* 1=mono;   2=stereo */
    int sample_bits;                /* 8=1byte;  16=2bytes */
    int flag;                       /* 1=record; 2=play */
} ac97_param_t;
```

ac97_param_t 参数的说明：

①、buffer　　用于录音或者播放的数据缓冲区；

②、buf_length　　数据缓冲区 buffer 的长度；

③、num_channels　　声道数；1：单声道；2：立体声；（仅支持单声道录音）

④、sample_bits　　采样位数；8 位采样或者 16 位采样；（仅支持 16 位采样）

⑤、flag　　open 执行操作：1=录音；2=播放。

驱动程序 ls1x_ac97.c 实现的函数：

函数
/* * 初始化AC97设备 * 参数:　dev　NULL *　　arg　NULL * * 返回:　0=成功 */ int LS1x_AC97_initialize(void *dev, void *arg);
/* * 打开AC97设备，并开始工作 * 参数:　dev　NULL *　　arg　类型: ac97_param_t *, 请阅读ac97_param_t域变量说明 * * 返回:　0=成功 */ int LS1x_AC97_open(void *dev, void *arg);

```
/*
 * 关闭AC97设备，并停止工作
 * 参数:    dev     NULL
 *          arg     类型: unsigned int, 该值非零时强制停止AC97
 *
 * 返回:    0=成功
 */
int LS1x_AC97_close(void *dev, void *arg);
```

```
/*
 * 向CAN设备发送控制命令
 * 参数:    dev     devCAN0/devCAN1
 *
 *      ------------------------------------------------------------------------------
 *          cmd                     |   arg
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_POWER            |   NULL. 打开AC97 外围芯片电源
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_RESET            |   NULL. 复位AC97及 外围芯片
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_SET_VOLUME       |   类型: unsigned int, 参见unpack_vol_arg()
 *                                  |   用途: 设置AC97的CODEC芯片音量
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_RECORD_GAIN      |   类型: unsigned int (& 0x1F1F)
 *                                  |   用途: 设置AC97的CODEC芯片录音增益
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_RECORD_SEL       |   类型: unsigned int (& 0x0707)
 *                                  |   用途: 设置AC97的CODEC芯片录音源
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_SET_REGISTER     |   类型: unsigned int, 参见unpack_reg_arg()
 *                                  |   用途: 设置AC97的CODEC寄存器值
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_GET_REGISTER     |   类型: unsigned int *, 参见unpack_reg_arg()
 *                                  |   用途: 读取AC97的CODEC寄存器值
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_START            |   NULL. 启动AC97开始工作
 *      ------------------------------------------------------------------------------
 *      IOCTL_AC97_STOP             |   NULL. 停止AC97工作
 *      ------------------------------------------------------------------------------
 *
 * 返回:    0=成功
 */
int LS1x_AC97_ioctl(void *dev, int cmd, void *arg);
```

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_ac97_drv_ops;
    #define ls1x_ac97_init(ac97, arg)          ls1x_ac97_drv_ops->init_entry(ac97, arg)
    #define ls1x_ac97_open(ac97, arg)          ls1x_ac97_drv_ops->open_entry(ac97, arg)
    #define ls1x_ac97_close(ac97, arg)         ls1x_ac97_drv_ops->close_entry(ac97, arg)
    #define ls1x_ac97_ioctl(ac97, cmd, arg) \
            ls1x_ac97_drv_ops->ioctl_entry(ac97, cmd, arg)
#else
```

```
    #define ls1x_ac97_init(ac97, arg)          LS1x_AC97_initialize(ac97, arg)
    #define ls1x_ac97_open(ac97, arg)          LS1x_AC97_open(ac97, arg)
    #define ls1x_ac97_close(ac97, arg)         LS1x_AC97_close(ac97, arg)
    #define ls1x_ac97_ioctl(ac97, cmd, arg)    LS1x_AC97_ioctl(ac97, cmd, arg)
#endif
```

AC97 实用函数：

函数
/* * 开始AC97放音 */ int ls1x_ac97_play(const ac97_param_t *param);
/* * 开始AC97录音 */ int ls1x_ac97_record(const ac97_param_t *param);
/* * AC97设备是否忙. 用户调用上面两个函数前必须判断AC97是否正在工作. */ int ls1x_ac97_busy(void);

说明：AC97 的驱动使用 DMA 和中断来实现，使用的采样速率、采样位数、声道数由外围 CODEC 芯片决定。当使用不同的 CODEC（目前是 ALC655）时，注意调整源代码中的相关参数。

12、GPIO 端口

源代码：ls1x-drv/gpio/ls1x_gpio.c

头文件：xxx/include/ls1x_gpio.h

ls1x_gpio.h 存放位置在裸机或 RTOS 的内核移植代码的 include 目录下。

GPIO 主要宏定义：

```
#define DIR_IN      1           /* 定义 gpio 为输入 */
#define DIR_OUT     0           /* 定义 gpio 为输出 */
#define GPIO_COUNT  62          /* 定义 gpio 端口总数 */
```

实现的 GPIO 内联函数：

函数
/* * 使能GPIO端口 * 参数: ioNum gpio端口序号 * dir gpio方向. DIR_IN: 输入, DIR_OUT: 输出 */ **static inline void** gpio_enable(**int** ioNum, **int** dir);
/* * 读GPIO端口, 该GPIO被设置为输入模式 * 参数: ioNum gpio端口序号 * 返回: 0或者1 */ **static inline int** gpio_read(**int** ioNum);
/* * 写GPIO端口, 该GPIO被设置为输出模式 * 参数: ioNum gpio端口序号 * val 0或者1 */ **static inline void** gpio_write(**int** ioNum, **int** val);
/* * 关闭GPIO功能, 端口恢复默认设置 * 参数: ioNum gpio端口序号 */ **static inline void gpio_disable**(**int** ioNum);

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：

函数
/* * 使能GPIO中断 * 参数: gpio gpio端口序号 */ int ls1x_enable_gpio_interrupt(int gpio);
/* * 禁止GPIO中断 * 参数: gpio gpio端口序号 */ int ls1x_disable_gpio_interrupt(int gpio);
/* * 安装GPIO中断向量 * 参数: gpio gpio端口序号 * trigger_mode 中断触发模式，见上定义 * 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 */ int ls1x_install_gpio_isr(int gpio, int trigger_mode,   void (*isr)(int, void *), void *arg);
/* * 取消已安装GPIO中断向量 * 参数: gpio gpio端口序号 */ int ls1x_remove_gpio_isr(int gpio);

13、看门狗

源代码：ls1x-drv/watchdog/ls1x_watchdog.c

头文件：ls1x-drv/watchdog/ls1x_watchdog.h

WatchDog 是否使用，在 bsp.h 中配置宏定义：

```
#define BSP_USE_WATCHDOG
```

驱动程序 ls1x_watchdog.c 实现的函数：

函数
/* * 开启看门狗 * 参数: dev NULL. 芯片只有一个dog设备 * arg 类型: unsigned int *. 毫秒数, dog计数到达这个数值时系统复位 * * 返回: 0=成功 */ int LS1x_DOG_open(void *dev, void *arg);
/* * 关闭看门狗 * 参数: dev NULL. 芯片只有一个dog设备 * arg NULL * * 返回: 总是 0 */ int LS1x_DOG_close(void *dev, void *arg);
/* * 向看门狗写毫秒数 * 参数: dev NULL. 芯片只有一个dog设备 * buf 类型: unsigned int *. 毫秒数, dog计数到达这个数值时系统复位 * size =4, sizeof(unsigned int) * arg NULL * * 返回: 4=成功 */ int LS1x_DOG_write(void *dev, void *buf, int size, void *arg);

用户接口函数：

```
#if (PACK_DRV_OPS)
    extern driver_ops_t *ls1x_dog_drv_ops;
    #define ls1x_dog_open(dog, arg)            ls1x_dog_drv_ops->open_entry(dog, arg)
    #define ls1x_dog_close(dog, arg)           ls1x_dog_drv_ops->close_entry(dog, arg)
    #define ls1x_dog_write(dog, buf, size, arg) \
            ls1x_dog_drv_ops->write_entry(dog, buf, size, arg)
#else
    #define ls1x_dog_open(dog, arg)                LS1x_DOG_open(dog, arg)
```

```
    #define ls1x_dog_close(dog, arg)              LS1x_DOG_close(dog, buf, size, arg)
    #define ls1x_dog_write(dog, buf, size, arg)   LS1x_DOG_write(dog, buf, size, arg)
#endif
```

Ls1x_dog_open 的 arg 参数和 ls1x_dog_write 的 buf 参数类型为：unsigned int *；启动看门狗和喂狗的**毫秒数**。

实用函数：

函数
/* * 开启看门狗 * 参数: ms 类型: unsigned int. 毫秒数, dog计数到达这个数值时系统复位 * * 返回: 0=成功 */ int ls1x_watchdog_start(unsigned int ms);
/* * 喂狗 * 参数: ms 类型: unsigned int. 毫秒数, dog计数到达这个数值时系统复位 * * 返回: 4=成功 */ int ls1x_watchdog_feed(unsigned int ms);
/* * 关闭看门狗 * 参数: (none) * * 返回: 总是 0 */ int ls1x_watchdog_stop(void);

第五节 其它宏定义与函数

1、内存/寄存器读写操作

头文件：ls1b.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, Val)		写一个字节到地址 Addr 处
OR_REG8(Addr, Val)		对地址 Addr 处的字节做或运算
AND_REG8(Addr, Val)		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, Val)		写一个字到地址 Addr 处
OR_REG16(Addr, Val)		对地址 Addr 处的字做或运算
AND_REG16(Addr, Val)		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, Val)		写一个双字到地址 Addr 处
OR_REG32(Addr, Val)		对地址 Addr 处的双字做或运算
AND_REG32(Addr, Val)		对地址 Addr 处的双字做与运算

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

2、芯片运行频率

头文件：ls1b.h

宏定义	返回值	功能
LS1B_PLL_FREQUENCY(xtal_freq)	**unsigned int**	返回 PLL 频率值
LS1B_CPU_FREQUENCY(xtal_freq)		返回 CPU 频率值
LS1B_DDR_FREQUENCY(xtal_freq)		返回内存频率值
LS1B_DC_FREQUENCY(xtal_freq)		返回显示控制器频率值
LS1B_BUS_FREQUENCY(xtal_freq)		返回总线频率值

参数：**xtal_freq** 是龙芯 1B **外接晶振**频率 **CPU_XTAL_FREQUENCY**，在 bsp.h 中定义。

3、cache 操作函数

汇编文件：cache.S

函数	功能
void flush_cache(void)	回写并刷新全部 cache
void flush_cache_nowrite(void)	作废并刷新全部 cache
void clean_cache(unsigned kva, size_t n)	回写并作废指定地址、指定长度的 cache
void flush_dcache(void)	回写并刷新全部数据 cache
void clean_dcache(unsigned kva, size_t n)	回写并作废指定地址、指定长度的数据 cache
void clean_dcache_nowrite(unsigned kva, size_t n)	作废指定地址、指定长度的数据 cache
void flush_icache(void)	刷新全部指令 cache
void clean_icache(unsigned kva, size_t n)	作废指定地址、指定长度的指令 cache

4、中断相关操作

源文件：irq.c/interrupt.c

函数	功能
void ls1x_install_irq_handler(int vector, void (*isr)(int, void *), void *arg)	安装中断句柄
void ls1x_remove_irq_handler(int vector)	移除中断句柄，替换为默认中断
int assert_sw_irq(unsigned int irqnum)	插入一个软件中断(SW0/SW1)
int negate_sw_irq(unsigned int irqnum)	撤销软件中断

头文件：mips.h

宏定义	功能
mips_interrupt_disable()	禁止芯片全局中断
mips_interrupt_enable()	允许芯片全局中断

头文件：ls1b.h

宏定义	功能
LS1B_INTC_ISR(base)	中断控制状态寄存器
LS1B_INTC_IEN(base)	中断控制使能寄存器
LS1B_INTC_SET(base)	中断置位寄存器
LS1B_INTC_CLR(base)	中断清空寄存器
LS1B_INTC_POL(base)	电平触发中断配置寄存器
LS1B_INTC_EDGE(base)	边沿触发中断配置寄存器

Base 是一个中断控制寄存器的基地址。

5、内存操作函数

源文件：lwmem.c

函数	功能	别名
void lwmem_initialize(**unsigned int** size)	初始化堆	
void *__malloc(**size_t** size)	申请内存块	**MALLOC**
void *__calloc(**size_t** nitems, **size_t** size)	申请内存块并清零	**CALLOC**
void *__realloc(**void** *ptr, **size_t** size)	重新申请内存	**REALLOC**
void __free(**void** *ptr)	释放内存	**FREE**
void *aligned_malloc(**size_t** size, **unsigned int** align)	申请 align 对齐的内存	
void aligned_free(**void** *addr)	释放申请的对齐内存	

说明：

- 使用堆进行动态内存分配前，必须调用 lwmen_initialize 进行初始化；
- 使用 aligned_mallo 申请的地址对齐内存，必须使用 aligned_free 来释放。

6、延时函数

源文件：tick.c/port_timer.c

函数	功能
`void delay_us(unsigned int us)`	延时微秒 us
`void delay_ms(unsigned int ms)`	延时毫秒 ms
`unsigned int get_clock_ticks(void)`	返回启动后的 tick 数

说明：当项目使用 RTOS 并已经启动，调用 delay_ms 进行延时将自动调用 RTOS 的任务延时函数。这里有个假设：RTOS 使用的 tick 是 1ms；如果 RTOS 的 tick 不是 1ms，**需要转换**。

7、打印函数

源文件：print.c

函数
`int printf(const char* format, ...)`
`int sprintf(char* buffer, const char* format, ...)`
`int snprintf(char* buffer, size_t count, const char* format, ...)`
`int vsnprintf(char* buffer, size_t count, const char* format, va_list va)`
`int vprintf(const char* format, va_list va)`
`int vsprintf(char* buffer, const char* format, va_list va)`
`int fctprintf(void (*out)( char character, char* arg),` `void* arg, const char* format, ...)`
`int printk(const char* format, ...)`

8、libc 库函数

源文件：libc.c

函数
char *strcpy(char *s1, const char *s2)
char *strncpy(char *s1, const char *s2, size_t n)
size_t strlen(const char *s)
size_t strnlen(const char *s, size_t n)
int strcmp(const char *s1, const char *s2)
int strncmp(const char *s1, const char *s2, size_t n)
char *strchr(const char *s, int c)
char *strcat(char *__restrict__ s1, const char *__restrict__ s2)
long strtol(const char *str, char **endptr, int base)
int atoi(const char *s)
int strncasecmp(const char *s1, const char *s2, register size_t n)
int strcasecmp(const char *s1, const char *s2)
void *memchr(const void *s, int c, size_t n)
void *memmove(void *s1, const void *s2, size_t n)
int memcmp(const void *s1, const void *s2, size_t n)
void *memset(void *s, int c, size_t n)
void *memcpy(void *__restrict__ s1, const void *__restrict__ s2, size_t n)

说明：标准 c 库函数，请查阅 c 手册。

如果要使用没有实现的函数，可以尝试在项目的链接参数中加入 c 库引用，或者自行实现该函数。

版权声明

本声明仅限于 LoongIDE 安装程序提供的所有源代码程序。

SPDX-License-Identifier: Apache-2.0