

外围设备驱动 操作指南

文档版本 V2.2

发布日期 2025-12-17

概述

本文档介绍如何操作 Linux 外围设备驱动，主要是指导使用 ETH、USB2.0 DRD 和 SD/MMC/eMMC 卡等驱动模块的相关人员，通过一定的步骤和方法对这些外围设备进行控制，主要包括操作准备、操作过程、操作中需要注意的问题以及操作示例。

为了便于说明，本文定义了以下术语：

术语	说明
<platform>	芯片平台代号，代表某个系列的芯片，例如 xmfalcon、xmorca
<soc>	芯片名，代表某个具体的芯片，例如 xm7606v13、xm7206v11a
<toolchain>	代表工具链，例如 arm-gcc12.2.0-linux、arm-gcc12.2.0-linux-uclibceabi

关于<platform>：

<platform>	说明	对应的<soc>
xmfalcon	xm7606v1x 系列芯片平台代号	xm7606v13 xm7605v13 xm7605v12 xm7605v11 xm7506v13
xmorca	xm7206v1x 系列芯片平台代号	xm7206v11a xm7206v10b xm7206v10

关于<toolchain>：

<toolchain>	说明
arm-gcc12.2.0-linux	GCC12.2, ARM32 工具链, 配套 C 库为 Glibc-2.37, 仅适用于 xmfalcon 平台
arm-gcc12.2.0-linux-ucLibcabi	GCC12.2, ARM32 工具链, 配套 C 库为 uClibc-ng-1.0.48, 仅适用于 xmorca 平台

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本

读者对象

本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

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

符号	说明
 须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修订记录

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

修订日期	版本	修订说明
2024-12-21	V1.0	DI 版本发布。
2025-03-04	V1.1	修正部分描述错误。
2025-05-21	V1.2	第二节“USB 操作指南”增加 7206V100 系列芯片的说明、7606V13 系列芯片多控制器的配置和确认说明。
2025-05-28	V2.0	支持 xm7206v1x 系列芯片，考虑与 xm7606v1x 兼容，增加了一些术语定义便于说明。
2025-08-20	V2.1	增加 2.3.8、2.3.9、2.3.10 关于 USB 虚拟键盘、USB 虚拟 HID、USB 动态切换模式的描述。
		PWM 操作指南增加如何反转波形的说明。
		修正 10.3.2 步骤 2 关于 PWM 极性反转配置的说明。
2025-12-17	V2.2	增加 2.2.3 关于 4.9 内核下的 media spport 配置路径。

目 录

1 ETH 操作指南	1
1.1 操作示例	1
1.2 IPv6 说明	2
1.3 PHY 地址配置	3
2 USB 操作指南	4
2.1 操作准备	4
2.1.1 USB 控制器的配置和确认	4
2.2 操作过程	4
2.2.1 Uboot 下 USB Host 操作过程	4
2.2.2 内核下 USB Host 操作过程	5
2.2.3 内核下 USB Device 操作过程	7
2.3 操作示例	11
2.3.1 Uboot 下 U 盘操作示例	11
2.3.2 内核下 U 盘操作示例	13
2.3.3 内核下键盘操作示例	14
2.3.4 内核下鼠标操作示例	14
2.3.5 内核下虚拟 U 盘操作示例	15
2.3.6 内核下虚拟网口操作示例	16
2.3.7 内核下虚拟串口操作示例	18
2.3.8 内核下虚拟键盘操作示例	19

2.3.9 内核下虚拟 HID 操作示例	20
2.3.10 USB 动态模式切换操作示例	21
2.3.11 内核下摄像头操作示例	22
2.4 操作中需要注意的问题	22
3 SD/EMMC/SDIO 操作指南	24
3.1 SD/EMMC 操作准备	24
3.2 SD/EMMC 操作过程	24
3.3 SD/EMMC 操作示例	25
3.4 SD/EMMC 操作中需要注意的问题	26
3.5 SDIO 操作准备	27
3.6 DTS 配置说明	27
3.7 内核配置说明	28
4 I2C 操作指南	29
4.1 操作准备	29
4.2 操作过程	29
4.3 接口速率设置说明	29
4.4 操作示例	30
4.4.1 用户态 I2C 读写程序示例:	30
4.4.2 内核态 I2C 读写程序示例:	33
5 SPI 操作指南	37
5.1 操作准备	37
5.2 操作过程	37
5.3 用户态 SPI 操作示例	37
5.3.1 用户态 SPI 读写程序示例	38
5.3.2 内核态 SPI 读写程序示例	39
6 UART 操作指南	44

6.1 操作准备	44
6.2 操作过程	44
6.3 UART 用户态程序操作示例	44
6.3.1 用户态 UART 设置属性示例	46
7 GPIO 操作指南	51
7.1 操作准备	51
7.2 操作过程	51
7.3 操作示例	51
7.3.1 用户态 GPIO 操作命令示例	51
7.3.2 内核态 GPIO 操作示例	52
8 CAN 操作指南	60
8.1 操作准备	60
8.2 可选功能配置	60
8.2.1 配置过载帧	60
8.2.2 配置重传次数	61
8.3 操作过程	61
8.4 操作示例	62
8.4.1 配置波特率	62
8.4.2 配置控制器内部回环功能	62
8.4.3 配置驱动错误上报功能	62
8.4.4 配置控制器自动重启功能	62
8.4.5 打开和关闭 CAN 节点	62
8.4.6 发送 CAN 数据	63
8.4.7 接收 CAN 数据	63
8.4.8 完整的使用流程	63
9 Watchdog 操作指南	65

9.1 操作准备	65
9.2 操作过程	65
9.3 注意事项	65
9.4 操作示例	66
9.4.1 打开 watchdog、设置超时时间、喂狗示例	66
9.4.2 使用 ioctl 命令关闭 watchdog 示例	67
9.4.3 使用 magic close 方式关闭 watchdog 示例	67
10 PWM 操作指南	68
10.1 操作准备	68
10.2 操作过程	68
10.3 操作示例	68
10.3.1 PWM 操作命令示例	68
10.3.2 内核态 PWM 操作示例	69
11 附录	71
11.1 用 fdisk 工具分区	71
11.1.1 查看当前状态	71
11.1.2 创建新的分区	71
11.1.3 保存分区信息	73
11.2 用 mkdosfs 工具格式化	73
11.3 挂载目录	73
11.4 读写文件	74

插图目录

图 1-1 IPv6 Protocol 配置示意图	2
图 2-1 成功建立网桥	17
图 3-1 在控制台下实现读写 SD 卡的操作示例	25
图 3-2 dts 配置如下图所示:	28
图 3-3 无线配置示意图	28
图 4-1 接口速率配置示意图	29
图 7-1 关闭内核标准 GPIO 示例	54

1 ETH 操作指南

1.1 操作示例

内核下使用网口的操作涉及到以下几个方面：

- 配置 ip 地址和子网掩码

```
ifconfig eth0 xxx.xxx.xxx.xxx netmask xxx.xxx.xxx.xxx up
```

- 设置缺省网关

```
route add default gw xxx.xxx.xxx.xxx
```

- mount nfs

```
mount -t nfs -o nolock xxx.xxx.xxx.xxx:/your/path /mount-dir
```

- shell 下使用 tftp 上传下载文件

前提是在 server 端有 tftp 服务软件在运行。

- 下载文件：tftp -r XX.file serverip -g

其中：XX.file 为需要下载的文件，serverip 需要下载的文件所在的 server 的 ip 地址。

- 上传文件：tftp -l xx.file remoteip -p

其中，xx.file 为需要上传的文件，remoteip 文件需要上传到的 server 的 ip 地址。

须知

调试网口前，确认单板网口 MAC 与 PHY 的接口类型（RGMII、RMII 或是 MII）、PHY 的复位输入 EPHY_RSTN 信号是否有反相，并在 Boot 表格中配置到 SYSBOOT1 寄存器，详见《U-boot 移植应用开发指南》的 Boot 表格中的板级配置章节。

1.2 IPv6 说明

发布包中默认关闭 IPv6 功能。如果要支持 IPv6，需要修改内核选项，并重新编译内核。具体操作如下：

```
cd source/kernel/linux-5.10.y
make ARCH=arm CROSS_COMPILE=<toolchain>- O=$(pwd)/../out <platform>_defconfig
make ARCH=arm CROSS_COMPILE=<toolchain>- O=$(pwd)/../out menuconfig
```

进入如下目录，将该页面选项配置如图 1-1 所示。

```
[*] Networking support --->
    Networking options --->
        <*> The IPv6 protocol --->
```

图1-1 IPv6 Protocol 配置示意图

```
--- The IPv6 protocol
[ ] IPv6: Router Preference (RFC 4191) support (NEW)
[ ] IPv6: Enable RFC 4429 Optimistic DAD (NEW)
<> IPv6: AH transformation (NEW)
<> IPv6: ESP transformation (NEW)
<> IPv6: IPComp transformation
<> IPv6: Mobility (NEW)
<> Virtual (secure) IPv6: tunneling (NEW)
<*> IPv6: IPv6-in-IPv4 tunnel (SIT driver) (NEW)
[ ] IPv6: IPv6 Rapid Deployment (6RD) (NEW)
<> IPv6: IP-in-IPv6 tunnel (RFC2473) (NEW)
[ ] IPv6: Multiple Routing Tables (NEW)
[ ] IPv6: multicast routing (NEW)
[ ] IPv6: Segment Routing Header encapsulation support (NEW)
[ ] IPv6: Segment Routing HMAC support (NEW)
[ ] IPv6: RPL Source Routing Header support (NEW)
```

IPv6 环境配置如下：

- 配置 ip 地址及缺省网关

```
ip -6 addr add <ipv6address>/<ipv6_prefixlen> dev <port>
```

示例：ip -6 addr add 2001:da8:207::9402/64 dev eth0

- Ping 某个- IPv6 地址

```
ping -6 <ipv6address>
```

示例：ping -6 2001:da8:207::9403

1.3 PHY 地址配置

U-boot 与 Kernel 下通过扫描的方式寻找 PHY，目前无需配置 PHY 的地址。

2 USB 操作指南

说明

本文档包含对 xmfalcon 和 xmorca 平台的 USB 驱动部分说明，若无特殊说明，本文所述配置步骤同时适配 Linux 4.9 与 5.10 内核版本。

- xmfalcon 可以支持两路 USB 控制器，分别为 USB 2.0/3.0 控制器，此时 USB3.0 控制器只支持工作在 host 模式，且速度为 super-speed，无法向下兼容 high-speed。当配置为一路 USB3.0 控制器时，此时 USB3.0 控制器无使用限制，同时 USB2.0 控制器不可用。
- xmorca 仅支持一路 USB2.0 控制器，默认工作速度为 high-speed，支持向下兼容。
- 每路 USB 控制器的工作模式仅为 Host 模式或 Device 模式二选一，不能同时使用。

2.1 操作准备

2.1.1 USB 控制器的配置和确认

本节内容仅适用于 xmfalcon 平台。

由于芯片支持两路 USB 控制器，因此支持对芯片进行配置，以选择合适的 USB 控制器数量。注意：如果需要切换到两路 USB 控制器的配置，请确保单板设计支持两路 USB。确认方法请参考《U-boot 移植应用开发指南》附录 “Boot 表格中的板级配置” 章节内容。

2.2 操作过程

2.2.1 Uboot 下 USB Host 操作过程

说明

当前 uboot 下只支持 U 盘，不支持硬盘。

步骤 1 选择并编译 uboot 下 USB 相关的驱动。

- 进入 u-boot 源码的根目录，执行以下命令，通过 menuconfig 界面配置 USB 选项。

```
make ARCH=arm CROSS_COMPILE=<toolchain>- <platform>_defconfig menuconfig
```

- 进入 menuconfig 的如下路径，确认以下驱动配置全部选上，默认配置是全选。

```
Lotus SoCs Configuration --->
  Device drivers --->
    [*] LOTUS USB support --->
  Command line interface --->
    Device access commands --->
      *- usb    # 已默认开启，且无法修改

  Device Drivers --->
    *- USB support --->
      [*] xHCI HCD (USB 3.0) support
      [*] USB Mass Storage support
```

步骤 2 编译烧录，启动单板按 Enter 进入 uboot。

----结束

2.2.2 内核下 USB Host 操作过程

步骤 1 选择 USB 2.0 Host 相关的内核驱动配置。

- 默认 USB Host/文件系统/存储相关模块已经全部编入内核，不需要再执行加载命令，就可以对 U 盘进行相关操作，具体操作请参见“2.3 操作示例”。**如果编译为 ko，需要手动加载下面对应的 ko 模块：**
 - 文件系统和存储设备相关 ko 模块
 - vfat
 - ext4
 - scsi_mod
 - sd_mod
 - nls_iso8859-1
 - 如未找到上述模块，请确认 CONFIG_VFAT_FS、CONFIG_EXT4_FS、CONFIG_SCSI、CONFIG_BLK_DEV_SD、CONFIG_NLS_ASCII、CONFIG_NLS_ISO8859_1 是否已经开启
 - 键盘相关 ko 模块
 - evdev
 - hid_generic
 - usbhid

- 如未找到上述模块，请确认 CONFIG_INPUT_EVDEV、CONFIG_HID_GENERIC、CONFIG_USB_HID 是否开启
- 鼠标相关 ko 模块
 - evdev
 - mousedev
 - hid_generic
 - usbhid
 - 如未找到上述模块，请确认 CONFIG_INPUT_MOUSEDEV、CONFIG_HID_GENERIC、CONFIG_USB_HID、CONFIG_INPUT_EVDEV 是否开启
- USB2.0 Host 相关 ko 模块
 - xhci-hcd
 - xhci-plat-hcd
 - usb-storage
 - 如未找到上述模块，请确认 CONFIG_USB_XHCI_HCD、CONFIG_USB_XHCI_PLATFORM、CONFIG_USB_STORAGE 是否开启
- 如确需自行开启，进入 menuconfig 的如下路径。USB2.0/3.0 Host 作为 U 盘、鼠标或者键盘所需的必要配置选择如下，其中标注为[*]的配置项可根据实际情况调整为[M]。

```

Lotus Support --->
  Lotus Device Drivers --->
    <*> Platforms Device DWC3
Device Drivers --->
  [*] USB support --->
    <*> DesignWare USB3 DRD Core Support
      DWC3 Mode Selection (Dual Role mode) --->
Input device support --->
  <*> Generic input layer (needed for keyboard, mouse, ...)
  <*> Mouse interface
    (1024) Horizontal screen resolution
    (768) Vertical screen resolution
  <*> Event interface
HID support --->
  <*> HID bus support
  <*> Generic HID driver (NEW)
USB HID support --->
  <*> USB HID transport layer
  
```

步骤 2 如需设置 SS host（仅 xmfalcon 平台的芯片可以配置）或 HS host 设备，分别找到 USB3.0 或 USB2.0 控制器设备节点，设置工作模式为 host。USB3.0 控制器节点名为 dwc3@0x10040000，USB2.0 节点名为 dwc3@0x10030000。下面以 XM7606V13 USB3.0 控制器工作模式设置为例进行说明，其他芯片的控制器工作模式设置方法一致。

进入 sdk/source/kernel/linux-5.10.y/lotus/machine/<platform>/dts 目录，编辑 <platform>.dtsi 文件，设置控制器【dr_mode】为外围设备模式【host】。

```
....
dwc3@0x10040000 {
    compatible = "snps,dwc3";
    ....
    dr_mode = "host";      # 主机工作模式
    ....
};
```

步骤 3 编译并烧录内核镜像。

编译内核命令为：

```
make ARCH=arm CROSS_COMPILE= <toolchain>- ulmage
```

步骤 4 启动单板。USB Host 具体操作请参见“2.3 操作示例”。

----结束

2.2.3 内核下 USB Device 操作过程

步骤 1 选择 USB Device 相关的内核驱动配置。

- 对于 linux-5.10 内核，进入 menuconfig 的如下路径，USB2.0/3.0 device 作为虚拟网口、虚拟串口、虚拟 u 盘、UVC 摄像头、虚拟键盘和虚拟 HID 所需的**必要配置**选择如下，其中标注为[*]的配置项可根据实际情况调整为[M]。

```
Lotus Support --->
  Lotus Device Drivers --->
    <*> Platforms Device DWC3
  Device Drivers --->
    <*> Multimedia support --->
      [*] Filter media drivers
      [] Autoselect ancillary drivers (tuners, sensors, i2c, spi, frontends)
    Media device types --->
      [*] Cameras and video grabbers
    Video4Linux options --->
```



```

[*] V4L2 sub-device userspace API
Media drivers --->
  [*] Media USB Adapters --->
    <*> USB Video Class (UVC)
    [*] UVC input events device support
<*> Sound card support --->
  <*> Advanced Linux Sound Architecture --->
  [*] USB sound devices (NEW) --->
    <*> USB Audio/MIDI driver
[*] USB support --->
  <*> DesignWare USB3 DRD Core Support
  DWC3 Mode Selection (Dual Role mode) ---->
  <*> USB Gadget Support --->
    <*> USB Gadget functions configurable through configs
    [] Generic serial bulk in/out (NEW)
    [*] Abstract Control Model (CDC ACM)
    [] Object Exchange Model (CDC OBEX) (NEW)
    [] Network Control Model (CDC NCM) (NEW)
    [*] Ethernet Control Model (CDC ECM)
    [] Ethernet Control Model (CDC ECM) subset (NEW)
    [*] RNDIS
    [] Ethernet Emulation Model (EEM) (NEW)
    [*] Mass storage
    [] Loopback and sourcesink function (for testing)
    [*] Function filesystem (FunctionFS)
    [*] Audio Class 1.0
    [] Audio Class 1.0 (legacy implementation)
    [*] Audio Class 2.0
    [] MIDI function
    [*] HID function
    [*] USB Webcam function

```

- 对于 linux-4.9 内核，Multimedia support 下的配置路径稍有不同，其他的配置选项与 linux-5.10 内核一致。

```

Device Drivers --->
  <*> Multimedia support --->
    [*] Cameras/video grabbers support
    [*] Media Controller API
    [*] V4L2 sub-device userspace API
    [*] Media USB Adapters --->
      <*> USB Video Class (UVC)
      [*] UVC input events device support

```

如在上述配置项设置为[M]，则表示以内核模块方式编译上述内核驱动。**如果编译为 ko，需要手动加载下面对应的 ko 模块：**

- 文件系统和存储设备相关模块与 Host 相同，请参考 2.2.2 步骤 1
- 虚拟 U 盘相关 ko 模块
 - libcomposite.ko
 - usb_f_mass_storage.ko
 - 如未找到上述模块，请确认 CONFIG_USB_LIBCOMPOSITE、CONFIG_USB_F_MASS_STORAGE 是否开启
- 虚拟网口设备相关 ko 模块
 - libcomposite.ko
 - u_ether.ko
 - usb_f_rndis.ko
 - usb_f_ecm.ko
 - 如未找到上述模块，请确认 CONFIG_USB_LIBCOMPOSITE、CONFIG_USB_U_ETHER、CONFIG_USB_F_RNDIS、CONFIG_USB_F_ECM 是否开启
- 虚拟串口设备相关 ko 模块
 - libcomposite.ko
 - u_serial.ko
 - usb_f_acm.ko
 - 如未找到上述模块，请确认 CONFIG_USB_LIBCOMPOSITE、CONFIG_USB_U_SERIAL、CONFIG_USB_F_ACM 是否开启
- UVC 摄像头设备相关 ko 模块
 - libcomposite.ko
 - usb_f_uvc.ko
 - u_audio.ko
 - usb_f_uac1.ko
 - usb_f_uac2.ko
 - 如未找到上述模块，请确认 CONFIG_USB_LIBCOMPOSITE、CONFIG_USB_F_UVC、CONFIG_USB_U_AUDIO、CONFIG_USB_F_UAC1、CONFIG_USB_F_UAC1 是否开启
- 虚拟键盘和虚拟 HID 相关 ko 模块
 - libcomposite.ko
 - usb_f_hid.ko
 - 如未找到上述模块，请确认 CONFIG_USB_LIBCOMPOSITE、CONFIG_USB_F_HID 是否开启

- USB Device 控制器 KO 模块：dwc3.ko。如未找到上述模块，请确认 CONFIG_USB_DWC3 是否设置为 M

步骤 2 如需设置 SS device 或 HS device 设备，分别找到 USB3.0 或 USB2.0 控制器设备节点，设置工作模式为 peripheral。USB3.0 控制器节点名为 dwc3@0x10040000，USB2.0 节点名为 dwc3@0x10030000。下面以 XM7606V13 USB3.0 控制器工作模式设置为例进行说明：

进入 sdk/source/kernel/linux-5.10.y/lotus/machine/<platform>/dts 目录，编辑 <platform>.dtsi 文件，设置 USB3.0 控制器【dr_mode】为外围设备模式【peripheral】。

```
....
dwc3@0x10040000 {
    compatible = "snps,dwc3";
    ....
    maximum-speed = "super-speed";
    dr_mode = "peripheral"; # 外围设备模式
    ....
};
```



说明

- 进入 Lotus Support 目录后选项界面如 步骤 1 所示，【Platforms Device DWC3】为 usb 运行必选项；
- 步骤 1 中，由于存在依赖关系，需要先配置 Multimedia support、Sound card support 有关配置项，USB Gadget Support 中才会出现 UAC、UVC 有关驱动选项，如 Audio Class 1.0 等配置项。
- 步骤 1 中 DWC3 Mode Selection 选项需要选择【Dual Role mode】。
- USB dtsi 文件节点配置如步骤 2 所示，在 DWC3 Mode Selection 选项为 Dual Role mode 情况下，dr_mode 参数为 “host” 则编译为 host 模式，dr_mode 参数为 “peripheral” 则编译为 device 模式。

步骤 3 编译并烧录内核镜像。如果步骤 1 中的配置项选择为 [M] 时，需要编译相应内核模块，生成.ko 文件。如果配置项选择为 [*]，则直接编译内核即可。

```
make ARCH=arm CROSS_COMPILE= <toolchain>- modules
```

注意：在编译模块时，要先编译内核，编译内核命令为：

```
make ARCH=arm CROSS_COMPILE= <toolchain>- ulmage
```

步骤 4 启动单板。USB Device 具体操作请参见 “2.3 操作示例”。

----结束

2.3 操作示例

2.3.1 Uboot 下 U 盘操作示例

2.3.1.1 设备识别



说明

Uboot 下不支持热插拔，在初始化 usb 子系统前需要将设备插入 U 口。

单板上电，进入 uboot 命令行，输入命令：usb start 初始化 usb 子系统，观察 U 盘是否识别成功。

U 口插入高速/超速 u 盘正常情况下串口打印为：

```
boot # usb start
USB0:  Register 1000140 NbrPorts 1
Starting the controller
USB XHCI 1.10
scanning bus 0 for devices... 2 USB Device(s) found
      scanning usb for storage devices... 1 Storage Device(s) found
```



说明

若 usb start 后出现识别枚举报错，如：Device not responding to set address 等，或者 usb start 后出现完全无法检测到设备，请在 uboot 命令行执行：setenv usb_pgood_delay XXX，该设置针对某些预热较慢的设备，增加 timeout 等待设备上电稳定，XXX 可根据当前设备的预热快慢设置合理的值，单位为毫秒（ms），建议取值范围为 1000-3000。

识别完成后，再输入命令：usb tree，查看识别速率。注意，仅 xmfalcon 平台的芯片支持超速 U 盘识别。

U 口插入高速/超速 u 盘正常情况下串口打印为：

```
boot # usb tree
USB device tree:
1  Hub (5 Gb/s, 0mA)
|  U-Boot XHCI Host Controller
|
+-2  Mass Storage (480 Mb/s, 250mA) # 连接的U盘为HS
      Generic Mass Storage Device 121220130416

boot # usb tree
USB device tree:
1  Hub (5 Gb/s, 0mA)
```

```
| U-Boot XHCI Host Controller
|
+-2 Mass Storage (5 Gb/s, 200mA) # 连接的U盘为SS
    Generic Mass Storage 23081710460191
```

2.3.1.2 初始化及应用

识别完成以后，进行以下操作。

步骤 1 查看设备信息。

Uboot 命令行执行：usb info [dev]，可以查看当前控制器上接的所有设备的设备信息，可在命令后加参数查看单个具体设备的信息，如：usb info 2，正常查看示例如下：

```
boot # usb info 2
config for device 2
2: Mass Storage, USB Revision 2.0
- Generic Mass Storage B92AAF26
- Class: (from Interface) Mass Storage
- PacketSize: 64 Configurations: 1
- Vendor: 0x058f Product 0x6387 Version 1.0
  Configuration: 1
    - Interfaces: 1 Bus Powered 200mA
      Interface: 0
        - Alternate Setting 0, Endpoints: 2
          - Class Mass Storage, Transp. SCSI, Bulk only
          - Endpoint 1 Out Bulk MaxPacket 512
          - Endpoint 2 In Bulk MaxPacket 512
```

步骤 2 对 U 盘进行读操作。

Uboot 命令行执行：usb read addr blk# cnt，使用命令将起始地址为 blk 大小为 cnt 的数据读到 DDR 地址 addr 的位置。正常读操作完成示例如下：

```
boot # usb read 0x82000000 0 2000
USB read: device 0 block # 0, count 8192 ... 8192 blocks read: OK
```

步骤 3 对 U 盘进行写操作。

Uboot 命令行执行：usb write addr blk# cnt，使用命令将 DDR 地址 addr 的 cnt 大小的数据写到 U 盘的 blk 位置。正常读操作完成示例如下：

```
boot # usb write 0x82000000 2000 2000
USB write: device 0 block # 8192, count 8192 ... 8192 blocks write: OK
```

----结束

2.3.2 内核下 U 盘操作示例

插入检测

直接插入 U 盘，观察是否枚举成功。

- USB 2.0 Host 插入高速 u 盘正常情况下串口打印为：

```
~ # usb 1-1: new high-speed USB device number 2 using usb-xhci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access      Kingston DT 101 G2      1.00 PQ: 0 ANSI: 4
sd 2:0:0:0: [sda] 15131636 512-byte logical blocks: (7.74 GB/7.21 GiB)
sd 2:0:0:0: [sda] Write Protect is off
sd 2:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
sda: sda1
sd 2:0:0:0: [sda] Attached SCSI removable disk
```

- 对于 xmfalcon 平台，USB 3.0 Host 插入超速 u 盘正常情况下串口打印为：

```
~ # usb 1-1: new SuperSpeed Gen 1 USB device number 3 using usb-xhci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access      Kingston DT 101 G2      1.00 PQ: 0 ANSI: 4
sd 2:0:0:0: [sda] 15131636 512-byte logical blocks: (7.74 GB/7.21 GiB)
sd 2:0:0:0: [sda] Write Protect is off
sd 2:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
sda: sda1
sd 2:0:0:0: [sda] Attached SCSI removable disk
```

其中：sda1 表示 U 盘或移动硬盘上的第一个分区，当存在多个分区时，会出现 sda1、sda2、sda3 等字样。

初始化及应用

模块插入完成后，进行如下操作：



说明

sdXY 中 X 代表磁盘号，Y 代表分区号，请根据具体系统环境进行修改。

- 分区命令操作的具体设备节点为 sdX，示例：\$ fdisk /dev/sda
- 用 mkdosfs 工具格式化的具体分区为 sdXY：~ \$ mkdosfs -F 32 /dev/sda1
- 挂载的具体分区为 sdXY：~ \$ mount -t vfat /dev/sda1 /mnt

步骤 1 查看分区信息。

- 运行命令 “ls /dev” 查看系统设备文件，若没有分区信息 sdXY，表示还没有分区，请参见 “11.1 用 fdisk 工具分区” 进行分区后，进入步骤 2。
- 若有分区信息 sdXY，则已经检测到 U 盘，并已经进行分区，进入步骤 2。

步骤 2 查看格式化信息。

- 若没有格式化，请参见 “11.2 用 mkdosfs 工具格式化” 进行格式化后，进入步骤 3。
- 若已格式化，进入步骤 3。

步骤 3 挂载目录，请参见 “11.3 挂载目录”。

步骤 4 对硬盘进行读写操作，请参见 “11.4 读写文件”。

----结束

2.3.3 内核下键盘操作示例

键盘操作过程如下：

步骤 1 插入模块。

插入键盘相关模块后，键盘会在/dev/input 目录下生成 eventX 节点（X 为设备对应的 event 节点号，根据实际情况选择，通常为 event0）。

步骤 2 接收键盘输入。其中 evtest 为开源工具，**仅用于示例说明，用户可根据自身需要下载使用（gitee 上搜索 evtest）**。

执行命令：evtest /dev/input/eventX

然后在 USB 键盘上敲击，可以看到屏幕有输出。

----结束

2.3.4 内核下鼠标操作示例

鼠标操作过程如下：

步骤 1 插入模块。

插入鼠标相关模块后，鼠标会在/dev/input 目录下生成 eventX 节点（X 为设备对应的 event 节点号，根据实际情况选择，通常为 event0）。

步骤 2 接收鼠标输入。其中 evtest 为开源工具，**仅用于示例说明，用户可根据自身需要下载使用（gitee 上搜索 evtest）**。

执行命令：evtest /dev/input/eventX

步骤 3 进行鼠标操作（点击、滑动等），可以看到串口打印出相应码值。

----结束

2.3.5 内核下虚拟 U 盘操作示例

单板作为虚拟 U 盘时支持 Flash、SD 卡、DDR 等存储介质，现以 Flash 为例，操作如下：

步骤 1 在单板端，把 libcomposite.ko、usb_f_mass_storage.ko、config_storage.sh 下载到任意目录。如编译进内核，则无需下载 ko。

步骤 2 在单板端进行如下操作，如驱动模块已编入内核，则跳过 insmod 操作。

```
insmod libcomposite.ko
insmod usb_f_mass_storage.ko
执行如下命令：

export VID="0x1d6b"           # 厂商ID
export PID="0xa4a5"           # 产品ID
export MANUFACTURER="The Linux Foundation"    # 厂商名
export PRODUCT="MassStorage"  # 产品名
export SERIALNUMBER="1234567890" # 序列号
export MEMORY=/dev/mtdblockX  # 使用Flash的第X分区作为虚拟U盘存储介质
./config_storage.sh
```

其中，mtdblockX 为 Flash 的第 X 个分区，请用户根据具体情况选择。若用户想使用其它介质，更改 dev/下节点即可。

说明

- 相关模块在 kernel 下的路径分别为：drivers/usb/gadget/libcomposite.ko、drivers/usb/gadget/function/usb_f_mass_storage.ko。
- 上述 export 所修饰的变量仅为示例，**用户需要自主配置上述字段值。**

- config_storage.sh 为启动虚拟 U 盘所进行的必要操作；disable_storage.sh 为卸载虚拟 U 盘功能的脚本，卸载时先执行卸载脚本，再卸载 ko。脚本所在目录：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。上述脚本仅供参考，用户可根据自身需要进行调整。

步骤 3 通过 USB 将单板与 Host 端相连，即可在 Host 端将单板识别成 USB 存储设备。

步骤 4 在 Host 端可将单板当成一个普通的 USB 存储设备，对其进行分区、格式化、读写等相关操作，详见“2.3.2 内核下 U 盘操作示例”。

----结束

2.3.6 内核下虚拟网口操作示例

单板作为虚拟网口设备，操作过程如下：

- 步骤 1 在单板端，把 libcomposite.ko、u_ether.ko、usb_f_rndis.ko、usb_f_ecm.ko、config_storage.sh 下载到任意目录。如编译进内核，则无需下载 ko。
- 步骤 2 在单板端进行如下操作，如驱动模块已编入内核，则跳过 insmod 操作。

```
insmod libcomposite.ko
insmod u_ether.ko
insmod usb_f_rndis.ko
insmod usb_f_ecm.ko

执行如下命令：

export MANUFACTURER="The Linux Foundation" # 厂商名

export PRODUCT="Ethernet" # 产品名

export SERIALNUMBER="1234567891" # 序列号

./config_ether.sh
```

说明

- 相关模块在 kernel 下的路径分别为：drivers/usb/gadget/libcomposite.ko、drivers/usb/gadget/function/u_ether.ko、drivers/usb/gadget/function/usb_f_ecm.ko、drivers/usb/gadget/function/usb_f_rndis.ko。
- 上述 export 所修饰的变量仅为示例，**用户需要自主配置上述字段值。**
- config_ether.sh 为启动虚拟网口所进行的必要操作；disable_ether.sh 为卸载虚拟网口功能的脚本，卸载时先执行卸载脚本，再卸载 ko。脚本所在目录：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。上述脚本仅供参考，用户可根据自身需要进行调整。

步骤 3 通过 USB 数据线将单板与 Host pc 端相连，若 pc 系统为 win10 系统，pc 端会自动加载驱动，在设备管理器的网络适配器部分可以看到 Remote NDIS Compatible Device #x 设备，但部分 pc 可能会识别为串口设备，解决方法见步骤 7 有关说明；若 pc 系统为 win7 系统，第一次可能会失败，需要自行安装驱动，方法为：

- 右击计算机，进入管理界面；
- 打开设备管理器；
- 点击其他设备会看到 Ethernet，双击；
- 打开驱动程序界面，点击更新驱动程序，进入浏览计算机以查找驱动程序软件 (R)；
- 把路径指向 linux.inf 所在的目录，将下面文件下载到本地目录，点击下一步，计算机自动进行安装驱动程序，安装成功后，点击网络适配器会看到 Ethernet Linux USB Ethernet /RNDIS Gadget #x。linux.inf 驱动文件取自发布包路径为：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。上述驱动文件仅供参考，以官方发布驱动为准。

步骤 4 在单板端 (device) 配置 IP。命令为：ifconfig usbX xx.xx.xx.xx netmask 255.255.xxx.0;route add default gw xx.xx.xx.xx。其中，usbX 为单板端实际生成的 usb 虚拟网口设备节点，可通过 ifconfig -a 查看。

步骤 5 当单板和 pc 通过 USB 数据线相连时，会在 PC 端生成 USB 网络节点，具体位置：打开网络和共享中心→更改适配器设置→Linux USB Ethernet /RNDIS Gadget #x。

步骤 6 建立网桥：

1. 右键 Linux USB Ethernet /RNDIS Gadget #x 或 Remote NDIS Compatible Device#x 节点，点击添加到桥选项；
2. PC 端连接大网的本地连接节点右键，选择添加到桥选项；
3. 等待网桥建立完成，如图 2-1 所示。

图2-1 成功建立网桥



步骤 7 此时在单板端，就可以正常访问大网，USB 可作为真正的网口操作使用，至此网口设备可以正常使用。

说明

- 作为 USB device 网口功能时，在每次重启单板后，请删除以前网桥并重新建立新的网桥
 - config_ether.sh 已经针对 win10 系统进行了适配，但发现部分 win10pc 会将网口设备识别为串口设备，解决方法为将 RNDIS.cat、RNDIS.inf 驱动文件放到本地目录，在设备管理器对新增的串口设备进行驱动更新，方法同步骤 2 中 win7pc 更新驱动方法，更新后网口设备即可正常使用。
 - win10pc 需要关闭数字签名后才可以更新驱动，关闭方法可在网上搜索，此处不再赘述。
 - RNDIS.cat、RNDIS.inf 驱动文件取自发布包路径为：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。**上述驱动文件仅供参考，以官方发布驱动为准。**
-

----结束

2.3.7 内核下虚拟串口操作示例

单板作为虚拟串口设备，操作过程如下：

- 步骤 1 在单板端，把 libcomposite.ko、u_serial.ko、usb_f_acm.ko、config_serial.sh 下载到 /root/ 路径。如编译进内核，则无需下载 ko。
- 步骤 2 如需将登录终端切换到虚拟串口，进入步骤 3；否则直接 insmod 步骤 1 中所有 ko 并执行 ./config_serial.sh，然后进入步骤 5。
- 步骤 3 在单板端执行 vi /etc/init.d/rcS，把以下命令添加进文件中，然后保存退出。如驱动编译进内核，则跳过 insmod 操作。

```
cd /root/
insmod libcomposite.ko
insmod u_serial.ko
insmod usb_f_acm.ko
./config_serial.sh
```

说明

- 相关模块在 kernel 下的路径分别为：drivers/usb/gadget/libcomposite.ko、drivers/usb/gadget/function/u_serial.ko、drivers/usb/gadget/function/usb_f_acm.ko。
 - config_serial.sh 为启动虚拟串口所进行的必要操作；disable_serial.sh 为卸载虚拟串口功能的脚本，卸载时先执行卸载脚本，再卸载 ko。脚本所在目录：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。**上述脚本仅供参考，用户可根据自身需要进行调整。**
-

- 步骤 4 在单板端，对 /etc/inittab 文件进行如下操作：

```
vi /etc/inittab # 编辑文件
```

```
#::respawn:/sbin/getty -L ttyS000 115200 vt100 -n root -l "Auto login as root ..." # 注释本行  
::respawn:/sbin/getty -L ttyGS0 115200 vt100 -n root -l "Auto login as root ..." # 添加本行
```

然后重启单板。

步骤 5 通过 USB 数据线将单板与 Host pc 端相连，

若 pc 系统为 win10 系统，pc 端会自动加载驱动。

若 pc 系统为 win7 系统，第一次可能会失败，需要自行安装驱动，方法为：

- 右击计算机，进入管理界面；
- 打开设备管理器；
- 点击其他设备会看到 Gadget Serial v2.4，双击；
- 打开驱动程序界面，点击更新驱动程序，进入浏览计算机以查找驱动程序软件 (R)。
- 把路径指向 linux-cdc-acm.inf 所在的目录，将下面文件下载到本地目录，点击下一步，计算机会自动进行安装驱动程序，安装成功后，点击端口 (COM 和 LPT)，会看到 Gadget Serial (COMx) 设备，若 pc 为 win10 系统，不需要自行安装驱动即可在端口 (COM 和 LPT) 看到 USB 串行设备 (COMx) 设备。

说明

- linux-cdc-acm.inf 驱动文件取自发布包路径为：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。上述驱动文件仅供参考，以官方发布驱动为准。

----结束

2.3.8 内核下虚拟键盘操作示例

单板作为虚拟键盘设备，操作过程如下：

步骤 1 在单板端，把 libcomposite.ko、usb_f_hid.ko、config_keyboard.sh 下载到任意目录。
如编译进内核，则无需下载 ko。

在单板端进行如下操作，如驱动模块已编入内核，则跳过 insmod 操作。

```
insmod libcomposite.ko  
insmod usb_f_hid.ko  
执行如下命令：  
./config_keyboard.sh
```

执行脚本后，虚拟键盘设备驱动会生成/dev/hidg0 节点，用户态收发数据可通过此设备节点进行。

说明

- 相关模块在 kernel 下的路径分别为：drivers/usb/gadget/libcomposite.ko、drivers/usb/gadget/function/usb_f_hid.ko。
- **用户可根据需要在 config_keyboard.sh 中自主配置 manufacturer、product、serialnumber 字段值。**
- config_keyboard.sh 为启动虚拟键盘所进行的必要操作；disable_keyboard.sh 为卸载虚拟键盘功能的脚本，卸载时先执行卸载脚本，再卸载 ko。脚本所在目录：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。**上述脚本仅供参考，用户可根据自身需要进行调整。**

步骤 2 通过 USB 将单板与 Host 端相连，即可在 Host 端将单板识别成 USB 键盘设备。

步骤 3 此时，可在单板通过 echo 命令发送 HID 协议数据，使得 Host 端看到相应键盘符号出现。以单板对接 PC 为例，在单板端执行以下命令，可在 PC 侧看到字母 a 输入：

```
echo -ne "\x00\x00\x04\x00\x00\x00\x00" > /dev/hidg0      # 板端发送字母 'a' 到PC
echo -ne "\x00\x00\x00\x00\x00\x00\x00" > /dev/hidg0      # 释放所有按键
```

----结束

2.3.9 内核下虚拟 HID 操作示例

单板作为虚拟 HID 设备，操作过程如下：

步骤 1 在单板端，把 libcomposite.ko、usb_f_hid.ko、config_hid.sh 下载到任意目录。如编译进内核，则无需下载 ko。

在单板端进行如下操作，如驱动模块已编入内核，则跳过 insmod 操作。

```
insmod libcomposite.ko
insmod usb_f_hid.ko
执行如下命令：
./config_hid.sh
```

执行脚本后，虚拟 HID 设备会生成在单板/dev/hidg0 目录下，用户态收发数据可通过此设备节点进行。

 说明

- 相关模块在 kernel 下的路径分别为：drivers/usb/gadget/libcomposite.ko、drivers/usb/gadget/function/usb_f_hid.ko。
 - 用户可根据需要在 config_hid.sh 中自主配置 manufacturer、product、serialnumber 字段值。
 - config_hid.sh 为启动虚拟键盘所进行的必要操作；disable_hid.sh 为卸载虚拟键盘功能的脚本，卸载时先执行卸载脚本，再卸载 ko。脚本所在目录：sdk/source/kernel/linux-5.10.y/lotus/tools/usb_tools。上述脚本仅供参考，用户可根据自身需要进行调整。
-

步骤 2 通过 USB 将单板与 Host 端相连，即可在 Host 端将单板识别成 USB HID 设备。

步骤 3 此时，可在单板通过 echo 命令发送 HID 协议数据，使得 Host 端看到相应数据出现。以 device 单板对接 Linux Host 为例，在 device 单板端执行以下命令，可在 Linux Host 侧看到解析的鼠标输入：

```
# Host Linux单板：接收模拟鼠标操作命令

evtest /dev/input/event0      # 通过evtest解析接收到的HID协议数据

# Device Linux单板：输入模拟鼠标操作命令

echo -n -e "\x01\x0A\x05" > /dev/hidg0      # 模拟鼠标左键按下，向右移动10个单位，向
下移动5个单位
```

----结束

2.3.10 USB 动态模式切换操作示例

本节内容仅适用于 xmorca 平台。

 说明

- 1、USB 动态模式切换依赖将 USB 控制器驱动编译为 KO 模块；
 - 2、根据 2.2.2 和 2.2.3 节对 USB 控制器驱动进行配置后，最终编译内核仅会产生 dwc3.ko。
-

假设 dwc3.ko 保存在 rootfs 的 /komod 目录下，操作过程如下：

步骤 1 内核上电进入命令行。

步骤 2 假设先指定控制器工作模式为 host，后指定控制器工作模式为 device。在内核命令行的操作如下：

```
/komod # insmod dwc3.ko dr_mode=host # 加载USB控制器驱动KO，指定模式为host
```

```

module param: host
module dr type: host
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
xhci-hcd xhci-hcd.0.auto: hcc params 0x0220fe6c hci version 0x110 quirks
0x0000000020010010
xhci-hcd xhci-hcd.0.auto: irq 66, io mem 0x10030000
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
xhci-hcd xhci-hcd.0.auto: Host supports USB 3.0 SuperSpeed
usb usb2: We don't know the algorithms for LPM for this host, disabling LPM.
hub 2-0:1.0: USB hub found
hub 2-0:1.0: config failed, hub doesn't have any ports! (err -19)

/komod # rmmod dwc3.ko # 卸载USB控制器驱动KO

xhci-hcd xhci-hcd.0.auto: remove, state 4
usb usb2: USB disconnect, device number 1
xhci-hcd xhci-hcd.0.auto: USB bus 2 deregistered
xhci-hcd xhci-hcd.0.auto: remove, state 4
usb usb1: USB disconnect, device number 1
xhci-hcd xhci-hcd.0.auto: USB bus 1 deregistered

/komod # insmod dwc3.ko dr_mode=peripheral # 加载USB控制器驱动KO, 指定模式为device

module param: peripheral
module dr type: peripheral

```

2.3.11 内核下摄像头操作示例

具体操作请参考《UVC UAC 使用指南》。

----结束

2.4 操作中需要注意的问题

操作中需要注意的问题如下：

- 在操作时请尽量按照完整的操作顺序进行操作（mount→操作文件→umount），以免造成文件系统的异常。

- 目前键盘和鼠标的驱动要和上层结合使用，比如鼠标事件要和上层的 GUI 结合。对键盘的操作只需要对/dev 下的 event 节点读取即可，而鼠标则需要标准的库支持。
- 在 Linux 系统中提供了一套标准的鼠标应用接口 libgpm，如果需要使用鼠标客户可自行编译此库。在使用时建议使用内核标准接口 gpm。
已测试通过的标准接口版本：gpm-1.20.5。
另外在 gpm 中还提供了一整套的测试工具源码（如：mev 等），用户可根据这些测试程序进行编码等操作，降低开发难度。
- USB Device 单板要插入模块后，再与 Host 端相连，否则 Host 端将不识别 Device 设备，并循环打印错误信息。

3

SD/EMMC/SDIO 操作指南

3.1 SD/EMMC 操作准备

SD/EMMC 卡的操作准备如下：

- U-boot 和 Linux 内核使用 SDK 发布的 U-boot 和 kernel。
- 文件系统。

可以使用 SDK 发布的本地文件系统 yaffs2、jffs2、ext4 或 squashFS，也可以通过本地文件系统再挂载到 NFS。

3.2 SD/EMMC 操作过程

操作过程如下：

步骤 1 启动单板，加载本地文件系统 yaffs2、jffs2、ext4 或 squashFS，也可以通过本地文件系统进一步挂载到 NFS。

步骤 2 加载内核。默认 SD/EMMC 相关模块已全部编入内核，不需要再执行加载命令。下面列出 SD/EMMC 所有相关驱动：

- 文件系统和存储设备相关模块
 - nls_base
 - nls_cp437
 - fat
 - vfat
 - msdos
 - nls_iso8859-1
 - nls_ascii
- SD/MMC/EMMC 相关模块
 - mmc_core
 - sdhci
 - sdhci-pltfm
 - sdhci-lotus

- mmc_block

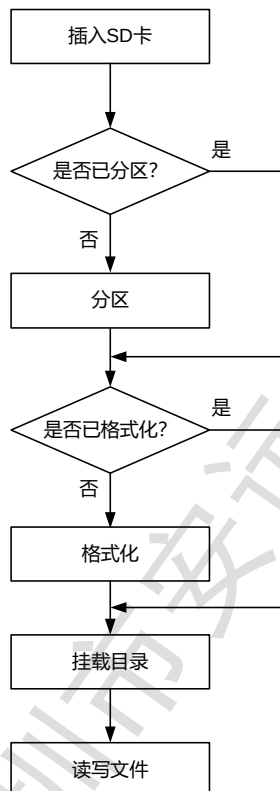
步骤 3 插入 SD/EMMC 卡，就可以对 SD/EMMC 卡进行相关的操作。具体操作请参见“3.3 SD/EMMC 操作示例”。

----结束

3.3 SD/EMMC 操作示例

此操作示例通过 SDIO 接口实现对 SD 卡的读写操作，而 eMMC 卡不支持热插拔，其读写操作也类似，这里不再举例。在控制台实现读写 SD 卡的操作示例如图 3-1 所示。

图3-1 在控制台实现读写 SD 卡的操作示例



初始化及应用，待 SD/EMMC 卡插入后，进行如下操作：



说明

其中 X 为分区号，由 fdisk 工具分区时决定。

- 命令 fdisk 操作的具体目录需改为：~\$ fdisk /dev/mmcblk0

- 用 mkdosfs 工具格式化的具体目录需改为: ~\$ mkdosfs -F 32 /dev/mmcblk0pX
- 挂载的具体目录需改为: ~\$ mount -t vfat /dev/mmcblk0pX /mnt

步骤 1 查看分区信息。

- 若没有显示出 p1, 表示还没有分区, 请参见 “11.1 用 fdisk 工具分区” 进行分区后, 进入步骤 2。
- 若有分区信息 p1, 则 SD/MMC 卡已经检测到, 并已经进行分区, 进入步骤 2。

步骤 2 查看格式化信息。

- 若没有格式化, 请参见 “11.2 用 mkdosfs 工具格式化” 进行格式化后, 进入步骤 3。
- 若已格式化, 进入步骤 3。

步骤 3 挂载目录, 请参见 “11.3 挂载目录”。

步骤 4 对 SD/EMMC 卡进行读写操作, 请参见 “11.4 读写文件”。

----结束

3.4 SD/EMMC 操作中需要注意的问题

在正常操作过程中需要遵守的事项:

- EMMC 8bit 模式: 在<platform>.dts 中将 MMC1 的 status 由 okay 改为 disable。
- 每次需要读写 SD 卡时, 必须确保 SD 卡已经创建分区, 并将该分区格式化为 vfat 文件系统 (通过 fdisk 和 mkdosfs 命令, 具体过程参见 “3.3 SD/EMMC 操作示例”)。
- 每次插入 SD 卡后, 需要做一次 mount 操作挂载文件系统, 才能读写 SD 卡; 如果 SD 卡已经挂载到文件系统, 拔卡后, 必须做一次 umount 操作, 否则, 再次插入卡时就会找不到 SD 卡的分区。
- 正常拔卡后需要 umount 挂载点 (建议正常的操作顺序是先 umount, 再拔卡), 异常拔卡后, 也需要 umount 挂载点, 否则再次插卡时就会找不到 SD 卡的分区。

在正常操作过程中不能进行的操作:

- 读写 SD 卡时不要拔卡或单板断上电操作, 否则会打印一些异常信息, 并且可能会导致卡中文件或文件系统被破坏。

- 当前目录是挂载目录如/mnt 时，不能 umount 操作，必须转到其它目录下才能 umount 操作。
- 系统中读写挂载目录的进程没有完全退出时，不能 umount 操作，必须完全结束操作挂载目录的任务才能正常 umount 操作。

在操作过程中出现异常时的操作：

- 如果在循环测试过程中异常拔卡，需要按 ctrl+c 回退出到 shell 下，否则会一直不停地打印异常操作信息。
- 拔卡后，再极其快速地插入卡时可能会出现检测不到卡的现象，因为卡的检测注册/注销过程需要一定的时间。
- 异常拔卡后，必须执行 umount 操作，否则不能读写挂载点目录如/mnt，并会打印异常信息。
- SD 有多分区时，可以通过 mount 操作切换挂载不同的分区，但最后 umount 操作次数与 mount 操作次数相等时，才会完全 umount 所有的挂载分区。
- 如果由于读写数据或其它异常原因，导致文件系统破坏，重新插卡并挂载，读写卡时可能会出现文件系统 panic，这时，需要 umount 操作，拔卡，再次插卡并 mount，才能正常读写 SD 卡。

3.5 SDIO 操作准备

SDIO 设备以 WIFI 为例其操作准备如下：

- reg 表格中添加 wifi 使用管脚的 pinmux。
- 配置 dts 支持 wifi 相关特性。
- 打开内核配置 wifi 相关配置。

3.6 DTS 配置说明

修改 linux 设备树，添加 SDIO 中断

linux-5.10.y/lotus/machine/<platform>/dts/<platform>.dtsi

文件中配置对应的 MMC 属性中增加

cap-sdio-irq

图3-2 dts 配置如下图所示:

```
mmc1: sdhci@0x10020000 {  
.....  
    full-pwr-cycle;  
    cap-sdio-irq;  
    devid = <2>;  
    status = "enable";  
};
```

3.7 内核配置说明

图3-3 无线配置示意图

```
[*] Networking support --->  
  [*] Wireless --->  
    <M>   cfg80211 - wireless configuration API  
    []   nl80211 testmode command  
    []   enable developer warnings  
    []   cfg80211 certification onus  
    [*]  enable powersave by default  
    []   cfg80211 DebugFS entries  
    [*]  support CRDA  
    [*]  cfg80211 wireless extensions compatibility  
    <M>   Generic IEEE 802.11 Networking Stack (mac80211)  
    [*]  Minstrel  
          Default rate control algorithm (Minstrel) --->  
    [*]  Enable mac80211 mesh networking support  
    []   Export mac80211 internals in DebugFS  
    []   Trace all mac80211 debug messages  
    []   Select mac80211 debugging features ----
```

4 I2C 操作指南

4.1 操作准备

I2C 的操作准备如下：

Linux 内核使能 i2c 驱动。

4.2 操作过程

操作过程如下：

步骤 1 加载内核。默认 I2C 相关模块已全部编入内核，不需要再执行加载命令。

步骤 2 在控制台下运行自己在内核态或者用户态编写 I2C 读写程序，就可以对挂载在 I2C 控制器上的外围设备进行读写操作。具体操作请参见“4.4 操作示例”。

----结束

4.3 接口速率设置说明

发布包中默认接口速率是 100kbit/s。如果要更改接口速率，需要修改 lotus/machine/<platform>/dts/<platform>.dts，并重新编译内核。具体操作如下：

i2c_bus0 节点中的 clock-frequency 属性的值，如图 4-1 所示。

图4-1 接口速率配置示意图

```
&i2c_bus0 {  
    status = "ok";  
    clock-frequency = <100000>;  
};
```

4.4 操作示例

4.4.1 用户态 I2C 读写程序示例：

此操作示例在用户态下通过 I2C 读写程序实现对 I2C 外围设备的读写操作。

步骤 1 打开 I2C 总线对应的设备文件，获取文件描述符：

```
fd = open("/dev/i2c-0", O_RDWR);
```

步骤 2 进行数据读写：

```
ioctl(fd, I2C_RDWR, &rdwr);  
write(fd, buf, (reg_width + data_width));
```

代码示例如下：



说明

此代码为示例程序，仅为客户开发用户态的 I2C 外围设备驱动程序提供参考，不提供实际应用功能。

```
int i2c_read(unsigned int i2c_num, unsigned int dev_addr, unsigned int reg_addr,  
             unsigned int reg_addr_end, unsigned int reg_width,  
             unsigned int data_width, unsigned int reg_step)  
{  
    int retval = 0;  
    int fd = -1;  
    char file_name[0x10];  
    unsigned char buf[4];  
    int cur_addr;  
    static struct i2c_rdwr_ioctl_data rdwr;  
    static struct i2c_msg msg[2];  
    unsigned int data;  
  
    memset(buf, 0x0, 4);  
    sprintf(file_name, "/dev/i2c-%u", i2c_num);  
    fd = open(file_name, O_RDWR);  
    if (fd < 0) {  
        return -1;  
    }  
  
    retval = ioctl(fd, I2C_SLAVE_FORCE, dev_addr);  
    if (retval < 0) {  
        retval = -1;  
        goto end;  
    }  
  
    return 0;  
end;  
}
```

```

    }
    msg[0].addr = dev_addr;
    msg[0].flags = 0;
    msg[0].len = reg_width;
    msg[0].buf = buf;

    msg[1].addr = dev_addr; // 器件设备地址不能包含读写位
    msg[1].flags = 0;
    msg[1].flags |= I2C_M_RD;
    msg[1].len = data_width;
    msg[1].buf = buf;

    rdwr.msgs = &msg[0];
    rdwr.nmsgs = (__u32)2;
    for (cur_addr = reg_addr; cur_addr <= reg_addr_end; cur_addr += reg_step) {
        if (reg_width == 2) {
            buf[0] = (cur_addr >> 8) & 0xff;
            buf[1] = cur_addr & 0xff;
        } else
            buf[0] = cur_addr & 0xff;

        retval = ioctl(fd, I2C_RDWR, &rdwr);
        if (retval != 2) {
            retval = -1;
            goto end;
        }

        if (data_width == 2) {
            data = buf[1] | (buf[0] << 8);
        } else
            data = buf[0];

        printf("0x%x: 0x%x\n", cur_addr, data);
    }

    retval = 0;
end:
    close(fd);
    return retval;
}

int i2c_write(unsigned int i2c_num, unsigned int dev_addr,
              unsigned int reg_addr, unsigned int data,

```



```

        unsigned int reg_width, unsigned int data_width)
{
    int retval = 0;
    int fd = -1;
    int index = 0;
    char file_name[0x10];
    unsigned char buf[4];

    sprintf(file_name, "/dev/i2c-%u", i2c_num);
    fd = open(file_name, O_RDWR);
    if (fd < 0) {
        printf("Open %s error!\n", file_name);
        return -1;
    }

    retval = ioctl(fd, I2C_SLAVE_FORCE, dev_addr);
    if (retval < 0) {
        retval = -1;
        goto end;
    }

    if (reg_width == 2) {
        buf[index] = (reg_addr >> 8) & 0xff;
        index++;
        buf[index] = reg_addr & 0xff;
        index++;
    } else {
        buf[index] = reg_addr & 0xff;
        index++;
    }

    if (data_width == 2) {
        buf[index] = (data >> 8) & 0xff;
        index++;
        buf[index] = data & 0xff;
        index++;
    } else {
        buf[index] = data & 0xff;
        index++;
    }

    retval = write(fd, buf, (reg_width + data_width));
    if (retval < 0) {

```

```

        printf("i2c write error!\n");
        retval = -1;
        goto end;
    }

    retval = 0;

end:
    close(fd);
    return retval;
}

```

----结束

4.4.2 内核态 I2C 读写程序示例：

此操作示例在内核态下通过 I2C 读写程序实现对 I2C 外围设备的读写操作。

步骤 1 调用 I2C 核心层的函数，获得描述一个 I2C 控制器的结构体 i2c_adap：

```
i2c_adap = i2c_get_adapter(0);
```



说明

假设我们已经知道新增的器件挂载在 I2C 控制器 0 上，直接设置 i2c_get_adapter 的参数为 0。

步骤 2 把 I2C 控制器和新增的 I2C 外围设备关联起来，得到描述 I2C 外围设备的客户端结构体 client：

```
client = i2c_new_device(i2c_adap, &i2c_info);
```



说明

i2c_info 结构体提供了 I2C 外围设备的设备地址

步骤 3 在非中断上下文中，调用 I2C 核心层提供的标准读写函数对外围器件进行读写：

```
ret = i2c_master_send(client, buf, count);
ret = i2c_transfer(client->adapter, msg, 2);
```



说明

- 参数 client 为步骤 2 得到的描述 I2C 外围设备的客户端结构体 client。
- 参数 buf 为需要读写的寄存器和数据。
- 参数 count 为 buf 的长度。
- 参数 msg 为读操作时的两个 i2c_msg 的首地址。

代码示例如下：



说明

此代码为示例程序，仅为客户开发内核态的 I2C 外围设备驱动程序提供参考，不提供实际应用功能。

```
static struct i2c_board_info i2c_info = {
    I2C_BOARD_INFO("i2c_test", 0x39),
};

static struct i2c_client *client = NULL;

int drv_i2c_write(unsigned int reg_addr, unsigned int reg_addr_num,
                  unsigned int data, unsigned int data_byte_num)
{
    unsigned char buf[8];
    int ret = 0;
    int idx = 0;
    struct i2c_client *client;

    client = client;

    /* reg_addr config */
    if (reg_addr_num == 2)
        buf[idx++] = (reg_addr >> 8);
    buf[idx++] = reg_addr;

    /* data config */
    if (data_byte_num == 2)
        buf[idx++] = data >> 8;
    buf[idx++] = data;

    ret = i2c_master_send(client, buf, idx); //在非中断上下文中使用
    return ret;
}

static struct i2c_msg g_msg[2];

int drv_i2c_read(unsigned int reg_addr, unsigned int reg_addr_num,
                 unsigned int data_byte_num)
{
    unsigned char buf[4];
    int ret = 0;
    int ret_data = 0xFF;
    int idx = 0;
    struct i2c_client *client;
    struct i2c_msg *msg;
```

```

client = client;

msg = &g_msg[0];

memset(msg, 0x0, sizeof(struct i2c_msg) * 2);

msg[0].addr = client->addr;
msg[0].flags = client->flags & I2C_M_TEN;
msg[0].len = reg_addr_num;
msg[0].buf = buf;

/* reg_addr config */
if (reg_addr_num == 2)
    buf[idx++] = reg_addr >> 8;
buf[idx++] = reg_addr;

msg[1].addr = client->addr;
msg[1].flags = client->flags & I2C_M_TEN;
msg[1].flags |= I2C_M_RD;
msg[1].len = data_byte_num;
msg[1].buf = buf;

ret = i2c_transfer(client->adapter, msg, 2); //在非中断上下文中使用

if (ret == 2) {
    if (data_byte_num == 2)
        ret_data = buf[1] | (buf[0] << 8);
    else
        ret_data = buf[0];
} else
    ret_data = -EIO;

return ret_data;
}

static int i2c_dev_init(void)
{
    struct i2c_adapter *i2c_adap; //分配一个I2C控制器指针

    //调用core层的函数，获得描述一个I2C控制器的结构体i2c_adap。假设我们已经知道新增的外围设备
    //挂载在编号为I2C控制器2上

    i2c_adap = i2c_get_adapter(2);

    //把I2C控制器和新增的I2C外围设备关联起来，I2C外围设备挂载在I2C控制器2，地址是0x72，就组

```

成了一个客户端client。

```
        client = i2c_new_device(i2c_adap, &i2c_info);
        i2c_put_adapter(i2c_adap);
        return 0;
    }
    static void i2c_dev_exit(void)
    {
        i2c_unregister_device(client);
    }
```

----结束

5 SPI 操作指南

5.1 操作准备

SPI 的操作准备如下：

Linux 内核使能 spi 驱动。

5.2 操作过程

操作过程如下：

步骤 1 加载内核。默认 SPI 相关模块已全部编入内核，不需要再执行加载命令。

步骤 2 编译测试程序

```
cd linux-5.10.y/tools/spi /*进入内核原生测试工具目录*/
```

```
mkdir -p include/linux/spi/ /*创建头文件目录*/
```

```
cp ../../include/uapi/linux/spi/spidev.h include/linux/spi/ /*拷贝用户态头文件*/
```

```
<toolchain>-gcc -I ./include --static spidev_test.c -o spidev_test /*手动编译*/
```

步骤 3 在控制台运行 spidev_test 测试程序或者自行在内核态或者用户态编写 SPI 读写程序，就可以对挂载在某个 SPI 控制器某个片选上的外围设备进行读写操作。具体操作请参见 5.3 “用户态 SPI 操作示例”。

----结束

5.3 用户态 SPI 操作示例

- 在控制台使用 spidev_test 应用测试程序对 spi 配置成 5MHz 速率、mode3 模式、8bit 数据位宽、使能控制器内部回环：

```
~$ ./spidev_test -v -s 5000000 -b 8 -O -H -I -D /dev/spidev0.0
```

- 在控制台使用 `spidev_test` 应用测试程序对 spi 配置成 10MHz 速率、mode2 模式、16bit 数据位宽、使能控制器内部回环：

```
~ $ ./spidev_test -v -s 10000000 -b 16 -O -I -D /dev/spidev0.0
```

5.3.1 用户态 SPI 读写程序示例

此操作示例在用户态下实现对挂载在 SPI 控制器 0 片选 0 上的 SPI 外围设备的读写操作。

步骤 1 打开 SPI 总线对应的设备文件，获取文件描述符：

```
fd = open("/dev/spidev0.0", O_RDWR);
```

步骤 2 通过 `ioctl` 设置 SPI 传输模式：

```
ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);  
ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
```



说明

`SPI_CPHA`、`SPI_CPOL` 表示 SPI 的时钟和相位。

`SPI_LSB_FIRST` 表示 SPI 传输时每个数据的格式为大端结束。

须知

`SPI_CPHA`、`SPI_CPOL` 和 `SPI_LSB_FIRST` 宏含义内核态可参考 `include/linux/spi/spi.h`，用户态可参考 `include/uapi/linux/spi/spidev.h`

SPI 的时钟、相位、大小端结束模式可参考芯片手册。

步骤 3 使用 `ioctl` 配置数据位宽

```
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
```

步骤 4 使用 `ioctl` 配置当前通信最大传输速率

```
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
```

步骤 5 使用 `ioctl` 进行数据读写

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```



说明

`tr` 表示传输一帧消息的 `spi_ioc` 结构体数组首地址，并且一次读写的数据总长度不超过 4KB。

`SPI_IOC_MESSAGE(n)` 表示全双工读写 `n` 帧消息的命令。

代码示例说明:



说明

此代码仅为客户开发生态的 SPI 外围设备操作程序提供参考，不提供实际应用功能。

具体路径为内核源码目录下: tools/spi/spidev_test.c

5.3.2 内核态 SPI 读写程序示例

此操作示例在内核态下通过 SPI 读写程序实现对 SPI 外围设备的读写操作。

步骤 1 调用 SPI 核心层的函数，获得描述一个 SPI 控制器的结构体：

```
drv_master = spi_busnum_to_master(bus_num);
```



说明

参数 bus_num 为要读写的 SPI 外围设备所连接的 SPI 控制器号。

drv_master 为描述 SPI 控制器的 spi_master 结构体类型指针变量。

步骤 2 通过每个 spi 片选在核心层的名称调用 SPI 核心层函数得到挂载在某个 spi 控制器某个片选上描述 SPI 外围设备的结构体：

```
sprintf(spi_name, "%s.%u", dev_name(&drv_master->dev), cs_n);  
d = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);  
drv_spi = to_spi_device(d);
```



说明

spi_bus_type 为核心层定义的描述 spi 总线的 bus_type 结构体类型变量。

drv_spi 为描述 SPI 外围设备的 spi_device 结构体类型指针变量。

步骤 3 设置 spi_transfer 结构体中的成员，调用 SPI 核心层函数将 spi_transfer 添加到 spi_message 的队列当中。

```
spi_message_init(&m);  
spi_message_add_tail(&t, &m);
```



说明

参数 t 为描述传输一帧消息的 spi_transfer 结构体类型变量。

参数 m 为描述传输一个消息队列的 spi_message 结构体类型变量。

步骤 4 然后调用 SPI 核心层提供的标准读写函数对外围器件进行读写：

```
status = spi_async(spi, &m);  
status = spi_sync(spi, &m);
```



说明

参数 spi 为描述 SPI 外围设备的 spi_device 结构体类型指针变量。

spi_async 函数进行 spi 异步读写操作，由于是异步操作，因此该调用返回时，spi 读写操作不一定完成，因此往该调用传的参数所指的地址空间必须是局部静态变量或全局变量，以防止函数返回时将传给 spi_async 的地址空间释放掉。spi_async 函数的原型为 int spi_async(struct spi_device *spi, struct spi_message *message)，则在这里变量 m 和 t 都必须为静态变量，并且 t 中所指的 buf 也必须是静态的。

spi_sync 函数进行 spi 同步读写操作。

代码示例如下：



说明

此代码为异步读写 SPI 外围设备 imx185 的示例程序，仅为客户开发内核态的 SPI 外围设备驱动程序提供参考，不提供实际应用功能。

```
static unsigned int bus_num = 0; //模块参数spi总线号

static unsigned int csn = 0; //模块参数spi片选号

module_param(bus_num, uint, S_IRUGO);
MODULE_PARM_DESC(bus_num, "spi bus number");
module_param(csn, uint, S_IRUGO);
MODULE_PARM_DESC(csn, "chip select number");

struct spi_master *drv_master; // spi控制器结构体

struct spi_device *drv_spi; //spi外围设备结构体

int ssp_write_alt(unsigned char devaddr, unsigned char addr, unsigned char data)
{
    struct spi_master *master = drv_master;
    struct spi_device *spi = drv_spi;
    static struct spi_transfer t;
    static struct spi_message m;
    static unsigned char buf[4];
    int status = 0;
    unsigned long flags;

    /* check spi_message is or no finish */
    spin_lock_irqsave(&master->queue_lock, flags);
    if (m.state != NULL) {
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return -EFAULT;
    }
    spin_unlock_irqrestore(&master->queue_lock, flags);
    spi->mode = SPI_MODE_3 | SPI_LSB_FIRST; //设置spi传输模式
```

```

memset(buf, 0, sizeof(buf));
buf[0] = devaddr;
buf[0] &= (~0x80);
buf[1] = addr;
buf[2] = data;

t.tx_buf = buf;
t.rx_buf = buf;
t.len = 3;
t.cs_change = 1;
t.bits_per_word = 8;
t.speed_hz = 2000000;

spi_message_init(&m); //初始化并设置spi传输队列

spi_message_add_tail(&t, &m);
m.state = &m;
status = spi_async(spi, &m);
return status;
}

int ssp_read_alt(unsigned char devaddr, unsigned char addr, unsigned char *data)
{
    struct spi_master *master = drv_master;
    struct spi_device *spi = drv_spi;
    static struct spi_transfer t;
    static struct spi_message m;
    static unsigned char buf[4];
    int status = 0;
    unsigned long flags;

    /* check spi_message is or no finish */
    spin_lock_irqsave(&master->queue_lock, flags);
    if (m.state != NULL) {
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return -EFAULT;
    }
    spin_unlock_irqrestore(&master->queue_lock, flags);

    spi->mode = SPI_MODE_3 | SPI_LSB_FIRST; //设置spi传输模式

    memset(buf, 0, sizeof buf);
    buf[0] = devaddr;
    buf[0] |= 0x80;
    buf[1] = addr;

```

```

    buf[2] = 0;

    t.tx_buf = buf;
    t.rx_buf = buf;
    t.len = 3;
    t.cs_change = 1;
    t.bits_per_word = 8;
    t.speed_hz = 2000000;

    spi_message_init(&m); //初始化并设置spi传输队列

    spi_message_add_tail(&t, &m);
    m.state = &m;
    status = spi_async(spi, &m);
    *data = buf[2];
    return status;
}

extern struct bus_type spi_bus_type;

static int __init sspdev_init(void)
{
    int            status = 0;
    struct device   *d;
    char            *spi_name;
    struct spi_master *bsp_master = spi_busnum_to_master(bus_num);
    if (bsp_master) {
        spi_name = kzalloc(strlen(dev_name(&bsp_master->dev)) + 10, GFP_KERNEL);
        if (!spi_name) {
            status = -ENOMEM;
            goto end0;
        }
        sprintf(spi_name, "%s.%u", dev_name(&bsp_master->dev), csn);
        d = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);
        if (d == NULL) {
            status = -ENXIO;
            goto end1;
        }
        drv_spi = to_spi_device(d);
        if (drv_spi == NULL) {
            status = -ENXIO;
            goto end2;
        }
        drv_master = bsp_master;
    } else {
        status = -ENXIO;
    }
}

```

```
        goto end0;
    }

    status = 0;
end2:
    put_device(d);
end1:
    kfree(spi_name);
end0:
    return status;
}
```

----结束

6 UART 操作指南

6.1 操作准备

UART 的操作准备如下：

Linux 内核使能 uart 驱动。

6.2 操作过程

操作过程如下：

步骤 1 加载内核。默认 UART 相关模块已全部编入内核，不需要再执行加载命令。

步骤 2 在控制台下运行自己在内核态或者用户态下编写 UART 操作程序，就可以对 UART 进行数据收发操作。具体操作请参见 6.3 “UART 用户态程序操作示例”。



说明

操作 UART 前，确认对应的管脚配置成 UART 功能。

----结束

6.3 UART 用户态程序操作示例

此操作示例在用户态下实现对 UART 的数据收发操作。

步骤 1 打开 UART 控制器对应的设备文件，获取文件描述符：

```
fd = open("/dev/ttyAMA1", O_RDWR);
```

步骤 2 通过 UART 文件描述符获取属性：

```
tcgetattr(fd, &options);
```

步骤 3 UART 控制器设置输入、输出传输速度：

```
ret = cfsetispeed(&options, B115200);  
ret = cfsetospeed(&options, B115200);
```

步骤 4 UART 控制器设置输入、输出传输速度：

```
ret = cfsetispeed(&options, B115200);  
ret = cfsetospeed(&options, B115200);
```

步骤 5 UART 控制器设置流控属性：

```
options.c_iflag &= ~CRTSCTS; //设置无流控制  
options.c_iflag |= CRTSCTS; //设置硬件流控制  
options.c_iflag |= (IXON | IXOFF | IXANY); //设置软件流控制
```

步骤 6 UART 控制器设置数据位宽：

```
options.c_cflag &= ~CSIZE; //使用前先把数据位清零  
options.c_cflag |= CS5; //设置数据位为5  
options.c_cflag |= CS6; //设置数据位为6  
options.c_cflag |= CS7; //设置数据位为7  
options.c_cflag |= CS8; //设置数据位为8
```

步骤 7 UART 控制器设置校验方式：

- 无校验(NONE)
options.c_cflag &= ~PARENB; //校验位 禁能
options.c_iflag &= ~INPCK;
- 奇校验(ODD)
options.c_cflag |= PARENB; //校验位 使能
options.c_cflag |= PARODD; //使用奇校验
options.c_iflag |= INPCK;
- 偶校验(EVEN)
options.c_cflag |= PARENB; //校验位 使能
options.c_cflag &= ~PARODD; //使用偶校验
options.c_iflag |= INPCK;
- 1校验(MASK)
options.c_cflag |= PARENB; //校验位 使能
options.c_cflag |= PARODD;
options.c_cflag |= CMSPAR;

```
options.c_iflag |= INPCK;
```

- 0校验(SPACE)

```
options.c_cflag |= PARENB; //校验位 使能
```

```
options.c_cflag &= ~PARODD;
```

```
options.c_cflag |= CMSPAR;
```

```
options.c_iflag |= INPCK;
```

步骤 8 UART 控制器设置停止位:

```
options.c_cflag &= ~CSTOPB; //停止位 1bit
```

```
options.c_cflag |= CSTOPB; //停止位 2bit
```

步骤 9 UART 设置串口属性:

```
ret = tcsetattr(fd, TCSANOW, &options);
```

步骤 10 UART 数据发送:

```
ret = write(fd, (unsigned char *)data, data_len);
```

步骤 11 UART 数据接收:

```
ret = read(fd, rev_buf, 1);
```

步骤 12 UART 关闭设备文件:

```
close(TTYfd);
```

6.3.1 用户态 UART 设置属性示例

```
int set_serial_rowmode(int fd, int speed, int flow_ctrl, int databit, int parity, int stopbit)
{
    int ret = 0;
    struct termios options;
    int i;
    int speed_arr[] = {
        B4000000, B3500000, B3000000, B2000000, B1500000, B1152000, B1000000,
        B921600, B576000,
        B500000, B460800, B230400, B115200, B57600, B38400, B19200, B9600, B4800,
        B2400, B1800,
        B1200, B600, B300, B200, B150, B134, B110, B75, B50, B0
    };

    int name_arr[] = {
        4000000, 3500000, 3000000, 2000000, 1500000, 1152000, 1000000, 921600, 576000,
```

```
500000, 460800, 230400, 115200, 57600, 38400, 19200, 9600, 4800, 2400, 1800,
1200, 600, 300, 200, 150, 134, 110, 75, 50, 0
```

```
};
```

```
if (tcgetattr(fd, &options) != 0) { //获取属性
```

```
    printf("Get Uart Attr Fialed!");
```

```
    return -1;
```

```
}
```

```
cfmakeraw(&options);    //使能raw模式
```

```
for (i = 0; i < sizeof(speed_arr) / sizeof(int); i++) {
```

```
    if (speed == name_arr[i]) {
```

```
        ret = cfsetispeed(&options, speed_arr[i]); //设置输入速度
```

```
        if (ret) {
```

```
            printf("Set Uart Speed Fialed!");
```

```
        }
```

```
        ret = cfsetospeed(&options, speed_arr[i]); //设置输出速度
```

```
        if (ret) {
```

```
            printf("Set Uart Speed Fialed!");
```

```
        }
```

```
        break;
```

```
    }
```

```
}
```

```
if (i >= sizeof(speed_arr) / sizeof(int)) {
```

```
    printf("Unsupported Uart Speed Fialed!");
```

```
    return -1;
```

```
}
```

```
tcflush(fd, TCIOFLUSH);
```

//刷新串口缓冲区

同时刷新收到的数据但是不读，并刷新

写入的数据但不传送

```
options.c_cflag &= ~CSIZE;    //先把数据位清零
```

```
switch (databit) {
```

```
    case 5:
```

```
        options.c_cflag |= CS5;    //设置数据位为5
```

```
        break;
```

```
    case 6:
```

```
        options.c_cflag |= CS6;    //设置数据位为6
```



```

        break;
    case 7:
        options.c_cflag |= CS7;    //设置数据位为7
        break;
    case 8:
        options.c_cflag |= CS8;    //设置数据位为8
        break;
    default:
        printf("Unsupported Uart Data bits Fialed!");
        return -1;
}

switch (flow_ctrl) {
    case 0:
        options.c_cflag &= ~CRTSCTS;    //设置无流控制
        break;
    case 1:
        options.c_cflag |= CRTSCTS;    //设置硬件流控制
        break;
    case 2:
        options.c_cflag |= (IXON | IXOFF | IXANY);    //设置软件流控制
        break;
    default:
        break;
}

options.c_cflag |= CLOCAL;
options.c_cflag |= CREAD;

//options.c_cflag 控制模式标志
switch (parity) {
    case 0: //无校验(NONE)
        options.c_cflag &= ~PARENB; //校验位 禁能
        options.c_iflag &= ~INPCK;
        break;
    case 1: //奇校验(ODD)
        options.c_cflag |= PARENB;    //校验位 使能
        options.c_cflag |= PARODD;    //使用奇校验

```

```

        options.c_iflag |= INPCK;
        break;
    case 2: //偶校验(EVEN)

        options.c_cflag |= PARENB;    //校验位 使能

        options.c_cflag &= ~PARODD; //使用偶校验

        options.c_iflag |= INPCK;
        break;
    case 3: //1校验(MASK)

        options.c_cflag |= PARENB;    //校验位 使能

        options.c_cflag |= PARODD;
        options.c_cflag |= CMSPAR;
        options.c_iflag |= INPCK;
        break;
    case 4: //0校验(SPACE)

        options.c_cflag |= PARENB;    //校验位 使能

        options.c_cflag &= ~PARODD;
        options.c_cflag |= CMSPAR;
        options.c_iflag |= INPCK;
        break;

    default:
        printf("Unsupported Uart Parity bits Fialed!");
        return -1;
}

switch (stopbit) {
    case 1:
        options.c_cflag &= ~CSTOPB; //停止位 1bit
        break;
    case 2:
        options.c_cflag |= CSTOPB;    //停止位 2bit
        break;
    default:
        printf("Unsupported Uart Parity bits Fialed!");
        return -1;
}

options.c_oflag &= ~OPOST; //禁用 输出处理 (使用原始输出)

```

```
//等待时间 单位0.1S; 最小字符to接口:1

options.c_cc[VTIME] = 128;    //指定读取每个字符的等待时间

options.c_cc[VMIN] = 1;      //指定最少读取的字符数

ret = tcsetattr(fd,TCSANOW,&options);    //设置串口属性

if (ret < 0) {
    printf("Set uart attr Fail!");
    ret = -1;
}
return ret;
}
```

7

GPIO 操作指南

7.1 操作准备

GPIO 的操作准备如下：

Linux 内核使能 GPIO 驱动

7.2 操作过程

操作过程如下：

步骤 1 加载内核。默认 GPIO 相关模块已全部编入内核，不需要再执行加载命令。

步骤 2 在控制台下运行相关命令或者自行在内核态或者用户态下编写 GPIO 操作程序，就可以对 GPIO 进行输入输出操作。具体操作请参见 7.3 “操作示例”。



说明

操作 GPIO 前，确认对应的管脚配置成 GPIO 功能。

----结束

7.3 操作示例

7.3.1 用户态 GPIO 操作命令示例

此操作示例通过命令实现对 GPIO 的读写操作。

步骤 1 在控制台使用 echo 命令将要操作的 GPIO 编号 export：

```
echo N > /sys/class/gpio/export
```



说明

每组 GPIO 有 8 个 GPIO 管脚。

N 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 $4 * 8 + 2 = 34$ 。

export 之后就会生成/sys/class/gpio/gpioN 目录

例如：exportGPIO4_2:

```
echo 34 > /sys/class/gpio/export
```

步骤 2 在控制台使用 echo 命令设置 GPIO 方向：

- 对于输入：echo in > /sys/class/gpio/gpioN/direction
- 对于输出：echo out > /sys/class/gpio/gpioN/direction

例如：设置 GPIO4_2 方向

- 对于输入：echo in > /sys/class/gpio/gpio34/direction
- 对于输出：echo out > /sys/class/gpio/gpio34/direction

 说明

- GPIO 方向只有 out 和 in 两种。
- 可使用 cat 命令查看 GPIO 方向：cat /sys/class/gpio/gpioN/direction，例如查看 GPIO4_2 方向：cat /sys/class/gpio/gpio34/direction

步骤 3 在控制台使用 cat 或 echo 命令查看 GPIO 输入值或设置 GPIO 输出值：

查看输入值：cat /sys/class/gpio/gpioN/value

输出低：echo 0 > /sys/class/gpio/gpioN/value

输出高：echo 1 > /sys/class/gpio/gpioN/value

 说明

GPIO 的电平值只有 0 和 1。0 为低电平；1 为高电平。

步骤 4 在控制台使用 echo 命令将操作的 GPIO 编号 unexport：

```
echo N > /sys/class/gpio/unexport
```

----结束

7.3.2 内核态 GPIO 操作示例

此操作示例在内核态下实现对 GPIO 的读写操作以及 GPIO 中断操作。

内核态 GPIO 读写操作示例

操作示例如下：

步骤 1 注册 GPIO：

```
gpio_request(gpio_num, NULL);
```

 说明

- 每组 GPIO 有 8 个 GPIO 管脚。
- 参数 `gpio_num` 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 $4 * 8 + 2 = 34$

步骤 2 设置 GPIO 方向：

- 对于输入：`gpio_direction_input(gpio_num)`
- 对于输出：`gpio_direction_output(gpio_num, gpio_out_val)`

 说明

如果是输出，需要设置一个输出的初始值，参数 `gpio_out_val` 为输出时的初始值。

步骤 3 查看 GPIO 输入值或设置 GPIO 输出值：

查看输入值：`gpio_get_value(gpio_num);`

输出低：`gpio_set_value(gpio_num, 0);`

输出高：`gpio_set_value(gpio_num, 1);`

 说明

输入值为 `gpio_get_value(gpio_num)` 的返回值。

步骤 4 释放注册的 GPIO 编号：

```
gpio_free(gpio_num);
```

----**结束**

内核态 GPIO 中断操作示例

须知

默认使用内核标准 GPIO。若客户自己实现了 GPIO 的驱动及中断注册，而非使用内核标准 GPIO，则需关闭内核标准 GPIO。否则两套驱动之间会产生冲突，影响使用。

关闭内核标准 GPIO 方法如下：

- 打开 dts 文件：`./lotus/machine/<platform>/dts/<platform>.dts`，将相应的 gpio 状态从 “okay” 改成 “disabled”，如图 7-1 所示。
vi `./lotus/machine/<platform>/dts/<platform>.dts`
- 修改后保存退出，重新编译内核。

图7-1 关闭内核标准 GPIO 示例

```
&gpio_chip0 {  
    //status = "ok";  
    status = "disabled";  
};
```

步骤 1 注册 GPIO：

```
gpio_request(gpio_num, NULL);
```

说明

- 每组 GPIO 有 8 个 GPIO 管脚。
- 参数 `gpio_num` 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 $4 * 8 + 2 = 34$ 。

步骤 2 设置 GPIO 方向：

```
gpio_direction_input(gpio_num);
```

说明

对于要作为中断源的 GPIO 引脚，方向必须配置为输入。

步骤 3 映射操作的 GPIO 编号对应的中断号：

```
irq_num = gpio_to_irq(gpio_num);
```

说明

中断号为 `gpio_to_irq(gpio_num)` 的返回值。

步骤 4 注册中断：

```
request_irq(irq_num, gpio_dev_test_isr, irqflags, "gpio_dev_test", &gpio_irq_type));
```

说明

`irqflags` 为需要注册的中断类型，常用类型有：

- IRQF_SHARED : 共享中断;
- IRQF_TRIGGER_RISING : 上升沿触发;
- IRQF_TRIGGER_FALLING : 下降沿触发;
- IRQF_TRIGGER_HIGH : 高电平触发;
- IRQF_TRIGGER_LOW : 低电平触发。

步骤 5 结束时释放注册的中断和 GPIO 编号:

```
free_irq(gpio_to_irq(gpio_num), &gpio_irq_type);
gpio_free(gpio_num);
```

代码示例如下:



说明

此代码为 GPIO 操作的示例程序, 仅为客户开发内核态的 GPIO 操作程序提供参考, 不提供实际应用功能。

```
#include <linux/delay.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/module.h>

//模块参数, GPIO组号、组内偏移、方向、输出时的输出初始值

static unsigned int gpio_chip_num = 4;
module_param(gpio_chip_num, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_chip_num, "gpio chip num");

static unsigned int gpio_offset_num = 2;
module_param(gpio_offset_num, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_offset_num, "gpio offset num");

static unsigned int gpio_dir = 1;
module_param(gpio_dir, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_dir, "gpio dir");

static unsigned int gpio_out_val = 1;
module_param(gpio_out_val, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_out_val, "gpio out val");

//模块参数, 中断触发类型
/*
* 0 - disable irq
* 1 - rising edge triggered
* 2 - falling edge triggered
* 3 - rising and falling edge triggered
```



```

* 4 - high level triggered
* 8 - low level triggered
*/
static unsigned int gpio_irq_type = 0;
module_param(gpio_irq_type, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_irq_type, "gpio irq type");

spinlock_t lock;

static int gpio_dev_test_in(unsigned int gpio_num)
{
    //设置方向为输入
    if (gpio_direction_input(gpio_num)) {
        return -EIO;
    }
    //读出GPIO输入值
    pr_info("[%s %d]gpio%d_%d in %d\n", __func__, __LINE__,
            gpio_num / 8, gpio_num % 8,
            gpio_get_value(gpio_num));

    return 0;
}

//中断处理函数
static irqreturn_t gpio_dev_test_isr(int irq, void *dev_id)
{
    pr_info("[%s %d]\n", __func__, __LINE__);

    return IRQ_HANDLED;
}

static int gpio_dev_test_irq(unsigned int gpio_num)
{
    unsigned int irq_num;
    unsigned int irqflags = 0;
    //设置方向为输入
    if (gpio_direction_input(gpio_num)) {
        return -EIO;
    }

    switch (gpio_irq_type) {
        case 1:

```

```

        irqflags = IRQF_TRIGGER_RISING;
        break;
    case 2:
        irqflags = IRQF_TRIGGER_FALLING;
        break;
    case 3:
        irqflags = IRQF_TRIGGER_RISING |
                    IRQF_TRIGGER_FALLING;
        break;
    case 4:
        irqflags = IRQF_TRIGGER_HIGH;
        break;
    case 8:
        irqflags = IRQF_TRIGGER_LOW;
        break;
    default:
        pr_info("[%s %d]gpio_irq_type error!\n",
                __func__, __LINE__);
        return -1;
}

pr_info("[%s %d]gpio_irq_type = %d\n", __func__, __LINE__, gpio_irq_type);
irqflags |= IRQF_SHARED;
//根据GPIO编号映射中断号
irq_num = gpio_to_irq(gpio_num);
//注册中断
if (request_irq(irq_num, gpio_dev_test_isr, irqflags,
                "gpio_dev_test", &gpio_irq_type)) {
    return -1;
}

return 0;
}

static void gpio_dev_test_irq_exit(unsigned int gpio_num)
{
    unsigned long flags;

    pr_info("[%s %d]\n", __func__, __LINE__);
    //释放注册的中断
    spin_lock_irqsave(&lock, flags);
    free_irq(gpio_to_irq(gpio_num), &gpio_irq_type);
}

```

```

        spin_unlock_irqrestore(&lock, flags);
    }
    static int gpio_dev_test_out(unsigned int gpio_num, unsigned int gpio_out_val)
    {
        //设置方向为输出，并输出一个初始值
        if (gpio_direction_output(gpio_num, !!gpio_out_val)) {
            return -EIO;
        }

        return 0;
    }

    static int __init gpio_dev_test_init(void)
    {
        unsigned int gpio_num;
        int status = 0;

        pr_info("[%s %d]\n", __func__, __LINE__);

        spin_lock_init(&lock);

        gpio_num = gpio_chip_num * 8 + gpio_offset_num;
        //注册要操作的GPIO编号
        if (gpio_request(gpio_num, NULL)) {
            return -EIO;
        }

        if (gpio_dir) {
            status = gpio_dev_test_out(gpio_num, gpio_out_val);
        } else {
            if (gpio_irq_type)
                status = gpio_dev_test_irq(gpio_num);
            else
                status = gpio_dev_test_in(gpio_num);
        }

        if (status)
            gpio_free(gpio_num);

        return status;
    }
    module_init(gpio_dev_test_init);

```

```
static void __exit gpio_dev_test_exit(void)
{
    unsigned int gpio_num;

    gpio_num = gpio_chip_num * 8 + gpio_offset_num;

    if (gpio_irq_type)
        gpio_dev_test_irq_exit(gpio_num);
    //释放注册的GPIO编号
    gpio_free(gpio_num);
}

module_exit(gpio_dev_test_exit);

MODULE_DESCRIPTION("GPIO device test Driver sample");
MODULE_LICENSE("GPL");
```

----结束

8 CAN 操作指南



说明

本章节仅适用于 xmfalcon 平台包含 CAN 控制器的芯片。

8.1 操作准备

CAN 的操作准备如下：

- 软件版本包含配置和测试的 CAN 工具。
在 SDK 目录下，输入 make menuconfig 进行图形化配置，依次进入 Linux System，Tools 菜单下，选中 CAN Tools Support，保存退出，重新编译软件。

```
Linux System --->
                  Tools --->
                        [*] CAN Tools Support
```

- 烧录 sdk 镜像后，单板能正常启动进入 kernel 或者 uboot。

8.2 可选功能配置



说明

必须先使能 FTCAN 设备驱动，才能找到 FTCAN 配置选项。

8.2.1 配置过载帧

CAN 控制器支持当 RX fifo 过载时收到 can 帧的时候，发出一个过载帧来推迟总线上其他 CAN 节点的发送请求。

```
Lotus Support --->
  Lotus Device Drivers --->
    <*> FTCAN support
      FTCAN config --->
        [*] FTCAN support overload frame
```



说明

驱动默认使能过载帧的功能。

8.2.2 配置重传次数

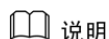
当 CAN 控制器发送帧失败的时候，会做一定次数的重传。

目前支持四种重传配置：

1. 当发送失败的时候，会一直重传。
2. 当发送失败的时候，不会发生重传。
3. 当发送失败的时候，最多发生 3 次重传。
4. 当发送失败的时候，最多发生 8 次重传。

```
Lotus Support --->
  Lotus Device Drivers --->
    <*> FTCAN support
      FTCAN config --->
        Choose retransmission times --->
```

```
( ) Message will always be retransmitted
( ) Message will not be retransmitted
(X) Message will be retransmitted at most 3 times
( ) Message will be retransmitted at most 8 times
```



说明

驱动默认使用重传三次的配置。

8.3 操作过程

操作过程如下：

- 步骤 1 启动单板，加载本地文件系统 yaffs2、jffs2、Ext4 或 SquashFS，也可以通过本地文件系统进一步挂载到 NFS。
- 步骤 2 加载内核。默认 CAN 相关模块已全部编入内核，不需要再执行加载命令。
- 步骤 3 配置管脚复用。
- 步骤 4 在控制台运行相关命令或者自行在用户态下编写 CAN 操作程序，就可以对 CAN 进行功能配置和数据收发操作。具体操作请参见 8.4 “操作示例”。



说明

操作 CAN 前，对应的管脚需要复用到 CAN 功能。

----结束

8.4 操作示例



说明

配置 CAN 的属性必须要在接口处于 down 的状态下，否则配置不会生效。

8.4.1 配置波特率

配置支持经典 CAN：canconfig can0 dev nbtrrate 1000000

配置支持 CAN FD：canconfig can0 dev dbtrrate 1000000 5000000；第一个波特率是仲裁段的，第二个波特率是数据段的。

8.4.2 配置控制器内部回环功能

canconfig can0 ctrlmode loopback on

canconfig can0 ctrlmode loopback off

8.4.3 配置驱动错误上报功能

CAN 驱动支持上报位错误、应答错误、填充错误、校验错误和格式错误。当开启错误上报功能后，驱动以标准的错误 CAN 帧格式上报错误，上层应用接收错误帧，按标准解析错误即可。

canconfig can0 ctrlmode berr-reporting on

canconfig can0 ctrlmode berr-reporting off

8.4.4 配置控制器自动重启功能

CAN 可能会因为收发错误到达了上限，而导致总线关闭，可以用以下命令设置重启间隔时间或者关闭自动重启功能。

canconfig can0 restart-ms 1000

canconfig can0 restart-ms 0

8.4.5 打开和关闭 CAN 节点

ifconfig canx up；ifconfig canx down。canx 对应 ifconfig 中对应的节点

8.4.6 发送 CAN 数据

cangen 是一个可以产生各种 CAN 帧的工具，主要可选功能如下：

-g：发送间隔，单位毫秒，默认为 200ms；

-l：帧 ID；

-L：帧数据长度；

-D：数据内容；

-n：产生 CAN 帧的次数；

-e：产生扩展帧；

-f：产生 CAN-FD 帧；

-b：产生变速率的 CAN-FD 帧；

例如：cangen -g 5 -l 333 -L 1 -D 11 -n 1 can0。发送一次一个 CAN id 为 0x333，1 字节，数据内容为 0x11 的帧。

8.4.7 接收 CAN 数据

candump 是一个打印 CAN 帧内容的程序，主要可选功能如下：

-f：保存 CAN 帧到指定的文件；

-e：以可阅读的方式打印 CAN 错误帧；

例如：candump -e any,0:0,#FFFFFFFF。打印所有的错误帧和数据帧。

8.4.8 完整的使用流程

1. 将两块板子 CAN0 连接。

2. 分别设置 CAN 配置。

```
canconfig can0 dev bitrate 1000000 5000000 (必须)
```

```
canconfig can0 ctrlmode berr-reporting on (可选)
```

```
canconfig can0 restart-ms 1000 (可选)
```

3. 分别使能 CAN0 接口。

```
ifconfig can0 up
```


cangen can0 -g 5 -l 1 -L 8 -D i -x -n 1 (2.0A 标准帧)

- [illegible]

9

Watchdog 操作指南

9.1 操作准备

watchdog 的操作准备如下：

能正常启动的硬件及烧录了正常的版本。

9.2 操作过程

操作过程如下：

- 1、默认 watchdog 模块已全部编入内核，不需要再执行加载命令。
- 2、用户空间打开 watchdog 设备节点/dev/watchdogX，使用 X 对应不同序号的 watchdog 硬件。设置超时时间，按时喂狗。
- 3、不同多个 watchdog 喂狗需要在用户空间应用程序绑定不同 cpu 喂狗，监控对应 cpu 的运行情况。

9.3 注意事项

- 1、watchdog 设备一旦被应用空间进程打开，就需要应用空间进程喂狗，否则如果 watchdog 超时会触发重启。
- 2、设置的 watchdog 超时时间是以秒为单位，watchdog 在超时时间没有喂狗就会触发重启，比如设置超时时间为 5 秒，如果 5 秒内没有正确喂狗就会触发重启。
- 3、Watchdog 的最大超时时间为 356 秒，如果设置的时间超过最大值，将会设置最大超时时间；如果设置的超时时间为 0 秒，实际也会设置最大超时实际 356 秒。
- 4、应用空间喂狗进程如果退出、奔溃、或者卡死导致 watchdog 没有在超时时间之内喂狗都会触发重启。

- 5、如果确实需要关闭 watchdog，使用 ioctl 命令关闭或者使用 magic close 的方式关闭 watchdog。
- 6、往 watchdog 节点写任意数据，或者使用 ioctl 命令 WDIOC_KEEPALIVE 都可以喂狗。

9.4 操作示例

9.4.1 打开 watchdog、设置超时时间、喂狗示例

```
int timeout = 0;
int ret = 0;
int fd = 0;
int cnt = 0;
int newtime = DEFAULT_TIMEOUT_VAL;

fd = open("/dev/watchdog", DEV_OPEN_FLAG);
if (fd == -1) {
    printf("open /dev/watchdog error\n");
    return -1;
}

ret = ioctl(fd, WDIOC_GETTIMEOUT, &timeout);
printf("get watchdog old timeout %d ret %d\n", timeout, ret);

ret = ioctl(fd, WDIOC_SETTIMEOUT, &newtime);
printf("set watchdog new timeout %d ret %d\n", newtime, ret);

ret = ioctl(fd, WDIOC_GETTIMEOUT, &timeout);
printf("get watchdog new timeout %d ret %d\n", timeout, ret);

while (1) {
    sleep(newtime - 1);
    if ((cnt % 2) == 0) {
        write(fd, "\0", 1);
    } else {
        ret = ioctl(fd, WDIOC_KEEPALIVE, &timeout);
    }
}
```

9.4.2 使用 ioctl 命令关闭 watchdog 示例

```
int ret = 0;
int fd = 0;
int status;

fd = open("/dev/watchdog", DEV_OPEN_FLAG);
if (fd == -1) {
    printf("open /dev/watchdog error\n");
    return -1;
}
status = WATCHDOG_DISABLE;
ret = ioctl(fd, WDIOC_SETOPTIONS, &status);
printf("disable watchdog ret %d\n", ret);
```

9.4.3 使用 magic close 方式关闭 watchdog 示例

```
int fd = 0;
fd = open("/dev/watchdog", DEV_OPEN_FLAG);
if (fd == -1) {
    printf("open /dev/watchdog error\n");
    return -1;
}
write(fd, "V", 1);
close(fd);
```

10 PWM 操作指南

10.1 操作准备

pwm 的操作准备如下：

能正常启动的硬件及烧录了正常的版本。

10.2 操作过程

操作过程如下：

步骤 1 加载内核，默认 pwm 模块已全部编入内核，不需要再执行加载命令。

步骤 2 确认管脚复用。

步骤 3 在控制台下运行相关命令或者自行在内核态或者用户态下编写 pwm 操作程序，就可以对 pwm 进行输入输出操作。具体操作请参见 10.3 “操作示例”。

 说明

此文档说明的是通用 pwm 模块，区别于 svb_pwm，通用 pwm 有用于 cpu 调压，svb_pwm 用于 vcore 调压。

操作 pwm 前，对应的管脚需要复用到 pwm 功能。

----结束

10.3 操作示例

10.3.1 PWM 操作命令示例

此操作示例通过命令实现对 PWM 的输出配置操作。

步骤 1 在控制台使用 echo 命令将要操作的 PWM 通道 export：

```
echo N > /sys/class/pwm/pwmchip0/export
```



说明

N 为 pwm 通道。

步骤 2 在控制台使用 echo 命令设置 PWM 周期：

```
echo xxx > /sys/class/pwm/pwmchip0/pwmN/period
```



说明

N 为 pwm 通道。

xxx 为配置的时间周期，单位为 ns。

步骤 3 在控制台使用 echo 命令设置 PWM 占空比：

```
echo yyy > /sys/class/pwm/pwmchip0/pwmN/duty_cycle
```



说明

N 为 pwm 通道。

yyy 为配置的占空比。比如，要配置 75% 占空比输出， $yyy = xxx * 75\%$ 。

步骤 4 如果需要反转波形：

```
echo inversed > /sys/class/pwm/pwmchip0/pwmN/polarity
```

步骤 5 在控制台使用 echo 命令将操作的 PWM 使能：

```
echo 1 > /sys/class/pwm/pwmchip0/pwmN/enable
```



说明

N 为 pwm 通道。

----结束

10.3.2 内核态 PWM 操作示例

此操作示例在内核态下实现对 PWM 的输出配置。

操作示例如下：

步骤 1 申请 PWM：

```
struct pwm_device *pwm = NULL;  
pwm = pwm_request(5, "lotus_pwm");
```



说明

5 为 pwm5 通道，lotus_pwm 为控制器固定匹配 name。

步骤 2 配置 PWM：

```
struct pwm_state state = { 0 };  
state.period = 250000;
```

```
state.duty_cycle = 187500;
state.enabled = true;
state.polarity = PWM_POLARITY_INVERSED; // 如果需要反转波形，才需要加此步骤
pwm_apply_state(pwm, &state);
```



说明

输出频率为 4kHz，高电平占 75%（即占空比）。

步骤 3 释放 PWM：

```
pwm_free(pwm);
```

----结束

11 附录

11.1 用 fdisk 工具分区

通过“11.1.1 查看当前状态”，对应以下情况选择操作：

- 若已有分区，本操作可以跳过，直接到“11.2 用 mkdosfs 工具格式化”。
- 若没有分区，则在控制台的提示符下，输入命令 fdisk，具体格式如下：

```
~ $ fdisk 设备节点
```

回车后，输入命令 m，根据帮助信息继续进行以下的操作。

其中设备节点与实际接入的设备类型有关，具体名称在以上各章节的“操作示例”中均有说明。

11.1.1 查看当前状态

在控制台的提示符下，输入命令 p，查看当前分区状态：

```
Command (m for help): p
```

控制台显示出分区状态信息：

```
Disk /dev/mmcblk1/disc: 127 MB, 127139840 bytes
8 heads, 32 sectors/track, 970 cylinders
Units = cylinders of 256 * 512 = 131072 bytes
Device Boot Start End Blocks Id System
```

上面信息表明设备没有分区，需要按照“11.1.2 创建新的分区”和“11.1.3 保存分区信息”的描述对设备进行分区。

11.1.2 创建新的分区

创建新的分区步骤如下：

步骤 1 创建新的分区。

在提示符下输入命令 n，创建新的分区：


```
Command (m for help): n
```

控制台显示出如下信息：

```
Command action
e extended
p primary partition (1-4)
```

步骤 2 建立主分区。

输入命令 p，选择主分区：

```
p
```

步骤 3 选择分区数。

本例中选择为 1，输入数字 1：

```
Partition number (1-4): 1
```

控制台显示出如下信息：

```
First cylinder (1-970, default 1):
```

步骤 4 选择起始柱面。

本例选择默认值 1，直接回车：

```
Using default value 1
```

步骤 5 选择结束柱面。

本例选择默认值 970，直接回车：

```
Last cylinder or +size or +sizeM or +sizeK (1-970, default 970):
Using default value 970
```

步骤 6 选择系统格式。

由于系统默认为 Linux 格式，本例中选择 Win95 FAT 格式，输入命令 t 进行修改：

```
Command (m for help): t
Selected partition 1
```

输入命令 b，选择 Win95 FAT 格式：

```
Hex code (type L to list codes): b
```

输入命令 l，可以查看 fdisk 所有分区的详细信息：

```
Changed system type of partition 1 to b (Win95 FAT32)
```

步骤 7 查看分区状态。

输入命令 p, 查看当前分区状态:

```
Command (m for help): p
```

控制台显示出当前分区状态信息, 表示成功分区。

----结束

11.1.3 保存分区信息

输入命令 w, 写入并保存分区信息到设备:

```
Command (m for help): w
```

控制台显示出当前设备信息, 表示成功写入分区信息到设备:

```
The partition table has been altered!
Calling ioctl() to re-read partition table.

.....
~ $
```

11.2 用 mkdosfs 工具格式化

存在以下情况选择操作:

- 若已格式化, 本操作可以跳过, 直接到“11.3 挂载目录”。
- 若没有格式化, 则输入命令 mkdosfs 进行格式化:

```
~ $ mkdosfs -F 32 设备分区名
```

其中设备分区名与实际接入的设备类型有关, 具体名称在以上各章节的“操作示例”中均有说明。

控制台没有显示错误提示信息, 表示成功格式化。

11.3 挂载目录

使用命令 mount 挂载到 mnt 目录下, 就可以进行读写文件操作:

```
~ $ mount -t vfat 设备分区名 /mnt
```

其中设备分区名与实际接入的设备类型有关，具体名称在以上各章节的“操作示例”中均有说明。

11.4 读写文件

读写操作的具体情况很多，在本例中使用命令 `cp` 实现读写操作。

使用命令 `cp` 拷贝当前目录下的 `test.txt` 文件到 `mnt` 目录下，即拷贝至设备，实现写操作，如：

```
~ $ cp ./test.txt /mnt
```