



CV186AH 外设驱动使用手册

Version: 1.0.1

Release date: 2023/12

©2022 北京晶视智能科技有限公司
本文件所含信息归北京晶视智能科技有限公司所有。
未经授权，严禁全部或部分复制或披露该等信息。

目录

1	声明	2
2	Ethernet 操作指南	3
2.1	操作示例	3
2.2	IPv6 说明	4
2.3	IEEE 802.3x 流控功能	4
2.3.1	流控功能描述	4
2.3.2	流控功能配置	4
2.3.3	ethtool 配置接口流控功能	5
3	USB 操作指南	6
3.1	操作准备	6
3.2	linux Host	6
3.2.1	USB 3.0 Host 操作过程 (以 usb1 为例)	6
3.2.2	U 盘操作范例	6
3.3	linux Device	7
3.3.1	USB 3.0 Device 操作过程 (以 usb0 为例)	7
3.3.2	USB Device 终端设备操作范例	8
3.3.3	USB Device RNDIS 设备操作范例	8
3.4	操作中需注意问题	10
4	SD/MMC 卡操作指南	11
4.1	操作准备	11
4.2	操作过程	11
4.3	操作示例	11
4.4	操作中需要注意的问题	12
5	I2C 操作指南	13
5.1	操作准备	13
5.2	操作过程	13
5.3	接口速率设置说明	13
5.3.1	I2C 读写命令示例:	14
5.3.2	内核态 I2C 读写程序示例:	14
5.3.3	用户态 I2C 读写程序示例:	15
6	SPI 操作指南	17
6.1	操作准备	17
6.2	操作过程	17
6.3	操作示例	17
6.3.1	内核态 SPI 读写程序示例:	17
6.3.2	用户态 SPI 读写程序示例:	20

7	GPIO 操作指南	22
7.1	GPIO 的操作准备如下:	22
7.2	操作过程	22
7.3	操作示例	22
7.3.1	GPIO 操作命令示例:	22
7.3.2	内核态 GPIO 操作程序示例:	23
7.3.3	用户态 GPIO 操作程序示例:	25
8	UART 操作指南	26
8.1	UART 的操作准备如下	26
8.2	操作过程	26
8.2.1	命令行操作示例:	26
8.2.2	用户态 UART 操作程序示例:	27
9	Watchdog 操作指南	28
9.1	Watchdog 的操作准备如下:	28
9.2	模块编译	28
9.3	操作示例	28
10	PWM 操作指南	30
10.1	PWM 的操作准备如下:	30
10.2	操作过程	30
10.3	操作示例	30
10.3.1	PWM 操作命令示例:	30
10.3.2	通过文件 IO 操作程序示例:	31
11	ADC 操作指南	33
11.1	ADC 的操作准备如下:	33
11.2	操作过程	33
11.3	操作示例	34
11.3.1	ADC 操作命令示例:	34
11.3.2	用户态 ADC 读取操作程序示例:	34
12	I2S 操作指南	35
12.1	概述	35
12.2	配置说明	35
12.3	常见问题	39

修订记录

Revision	Date	Description
1.0.0	2022/10/31	Initial version
1.0.1	2023/02/06	Modified to be compatible with CV186AH
1.0.2	2024/12/20	Add I2S operation guide

1 声明



法律声明

本数据手册包含北京晶视智能科技有限公司（下称“晶视智能”）的保密信息。未经授权，禁止使用或披露本数据手册中包含的信息。如您未经授权披露全部或部分保密信息，导致晶视智能遭受任何损失或损害，您应对因之产生的损失/损害承担责任。

本文件内信息如有更改，恕不另行通知。晶视智能不对使用或依赖本文件所含信息承担任何责任。本数据手册和本文件所含的所有信息均按“原样”提供，无任何明示、暗示、法定或其他形式的保证。晶视智能特别声明未做任何适销性、非侵权性和特定用途适用性的默示保证，亦对本数据手册所使用、包含或提供的任何第三方的软件不提供任何保证；用户同意仅向该第三方寻求与此相关的任何保证索赔。此外，晶视智能亦不对任何其根据用户规格或符合特定标准或公开讨论而制作的可交付成果承担责任。

联系我们

地址 北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

深圳市宝安区福海街道展城社区会展湾云岸广场 T10 栋

电话 +86-10-57590723 +86-10-57590724

邮编 100094（北京）518100（深圳）

官方网站 <https://www.sophgo.com/>

技术论坛 <https://developer.sophgo.com/forum/index.html>

2 Ethernet 操作指南

2.1 操作示例

Ethernet 模块默认为编入内核，不需另外执行加载操作。

内核下使用网口的操作步骤如下：

- 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
```

- 挂载 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 -g -r [remote file name] [server ip]

备注: remote file name 为欲下载的文件名称，server ip 为下载文件所在 server 的 ip 地址 (ex: tftp -g -r test.txt 192.168.0.11)。

- 上传文件：tftp -p -l [local file name] [server ip]

备注: local file name 为本地欲上传的文件名称，server ip 为上传文件的目标 server 的 ip 地址 (ex: tftp -l -p test.txt 192.168.0.11)。

备注说明：CV186X Ethernet 模块不支持 TSO 功能。

备注说明：nfs 工具默认不会编入文件系统，需要用户在需要时候加入。

2.2 IPv6 说明

SDK 包中默认关闭 IPv6 功能。如果要开启 IPv6，需要修改内核选项。具体操作如下：

修改

```
build/boards/{chip_name}/{board_name}/linux/{board_name}_defconfig。
```

新增或修改成 `CONFIG_IPV6=y`。然后重新编译内核软件。

IPv6 环境配置方法如下：

- 配置 ip 地址以及网关

```
#ip -6 addr add <ipv6 address>/ipv6 prefixlen dev <port name>
```

```
Ex: ip -6 addr add 2020:abc:102::8888/24 dev eth0
```

- Ping 指定的 IPv6 地址

```
#ping -6 <ipv6 address>
```

```
Ex: ping -6 2020:abc:102::6666
```

2.3 IEEE 802.3x 流控功能

2.3.1 流控功能描述

CV186X Ethernet 支持 IEEE 802.3x 所定义的流控功能，透过发送流控帧以及接收对端所发送过来的流控帧的方式来达到流控的目的。

- 发送流控帧：

在接收由对端发送过来的封包的过程中，若发现目前接收端的接收对列可能无法满足接收后续送达的封包时，则本地端会发送流控帧至对端，要求对端暂停一段时间不发送封包，藉此来进行流量控制。

- 接收流控帧：

当本地端接收到由对端发送过来的流控帧时，本地端会根据帧内的流控时间描述延迟发送封包至对端，等到过了流控延迟时间后，再启动发送。若在等待过程中收到了由对端发送的流控帧其描述的流控时间为 0 时，则会直接启动发送。

2.3.2 流控功能配置

接收流控帧功能默认是关闭的，亦没有提供软件接口配置。

发送流控帧功能的相关文件配置在 `linux/drivers/net/ethernet/stmicro/stmmac/stmmac_main.c`

```
static int flow_ctrl = FLOW_OFF;
module_param(flow_ctrl, int, 0644);
MODULE_PARM_DESC(flow_ctrl, "Flow control ability [on/off]");
```

(下页继续)

(续上页)

```
static int pause = PAUSE_TIME;
module_param(pause, int, 0644);
MODULE_PARM_DESC(pause, "Flow Control Pause Time");
```

若欲默认开启流控功能，可修改 `flow_ctrl = FLOW_AUTO`。

若欲修改默认 pause time，可配置 pause 至目标值。

2.3.3 ethtool 配置接口流控功能

用户可以通过标准 ethtool 工具接口进行流控功能的使能。

`ethtool -a eth0` 命令查看 eth0 口流控功能状态；打印如下

```
# ethtool -a eth0
Pause parameters for eth0:
Autonegotiate: on
RX: off
TX: off
```

其中，RX 流控是关闭的，TX 流控是关闭的；用户可以通过以下命令打开或关闭 TX 流控：

```
# ethtool -A eth0 tx off (关闭TX流控)
# ethtool -A eth0 tx on (打开TX流控)
```

备注：ethtool 工具默认不会编入文件系统，需要用户在需要时候加入。

3 USB 操作指南

3.1 操作准备

USB 3.0 Host/Device 的操作准备如下:

- Linux 内核使用 SDK 发布 Kernel。
- 文件系统可以使用本地文件系统 ext4 或 squashfs, 也可以使用 NFS。
- Shell script “run_usb.sh”. run_usb.sh 使用内核的 USB ConfigFS 功能来客制化 USB device 装置. 使用者可参考并修改 run_usb.sh 来变更 PID/VID 与 function 的相关参数. 详细操作可参考内核文件” linux/Documentation/usb/gadget_configfs.txt”。

3.2 linux Host

3.2.1 USB 3.0 Host 操作过程 (以 usb1 为例)

步骤 1. 启动平台, 加载 ext4 或 squashfs, 也可以使用 NFS.

步骤 2. 设置 usb 角色

```
echo host > /sys/kernel/debug/usb/39110000.usb/mode
```

3.2.2 U 盘操作范例

插入检测:

直接插入 U 盘, 观察是否枚举成功. 正常情况下串口打印为:

```
[ 72.061964] usb 1-1: new high-speed USB device number 2 using dwc2
[ 72.315816] usb-storage 1-1:1.0: USB Mass Storage device detected
[ 72.335934] scsihost0: usb-storage 1-1:1.0
[ 73.363027] scsi 0:0:0:0: Direct-Access Generic STORAGE DEVICE1532 PQ: 0 ANSI: 6
[ 73.374407] sd 0:0:0:0: Attached scsigeneric sg0 type 0
[ 73.558597] sd 0:0:0:0: [sda] 30253056 512-byte logical blocks:(15.5 GB/14.4 GiB)
[ 73.566961] sd 0:0:0:0: [sda] Write Protect is off
[ 73.571922] sd 0:0:0:0: [sda] Mode Sense: 21 00 00 00
```

(下页继续)

(续上页)

```
[ 73.577899] sd 0:0:0:0: [sda] Write cache: disabled, read cache:enabled, doesn't support DPO or FUA
[ 73.593961] sda: sda1[ 73.602607] sd 0:0:0:0: [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, sd2, sda3 等字样。

初始化及应用:

插入检测后, 进行如下操作:

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。

- 分区命令操作的设备节点为 sdX, 范例: ~\$ 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 /dev/sda1 /mnt。

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. 请用 fdisk 进行分区后进入步骤 2。
- 若有分区信息 sdXY, 则已经检测到 U 盘分区, 进入步骤 2。

2. 查看格式化信息

- 若没有格式化, 请使用 mkdosfs 进行格式化后进入步骤 3。
- 若已格式化, 进入步骤 3。

3. 挂载目录

- 运行” mount /dev/sdaXY /mnt” 挂载目录。

4. 对硬盘进行读写操作

3.3 linux Device

3.3.1 USB 3.0 Device 操作过程 (以 usb0 为例)

步骤 1. 编译 USB3.0 Device 相关的内核驱动模块

- 进入 menuconfig 的如下路径, 并配置如下。

```
Device Driver --->
[*] USB support --->
  <*> USB Gadget Support --->
    <M> USB functions configurable through configs
      [*] Abstract Control Model (CDC ACM)
      [*] Mass storage
```

- 编译内核模块, 生成.ko 文件。

步骤 2. 加载驱动

```
insmod /mnt/system/ko/usb_f_mass_storage.ko
```

步骤 3. 启动平台，加载 ext4 或 squashfs 文件系统，也可以使用 NFS。

步骤 4. 将 otg controller 切换至 device mode

步骤 5. `echo device > /sys/kernel/debug/usb/39010000.usb/mode`

步骤 6. 运行 shell script “run_usb.sh”

```
/etc/run_usb.sh probe msc /dev/mmcblkXY
```

```
/etc/run_usb.sh start
```

其中 mmcblkXY 为第 X 个磁盘的 eMMC 或 SD 中第 Y 个分区。请用户根据具体情况选择。

步骤 7. 在 Host 端可将平台当成一个普通的 USB 存储设备，对其进行分区，格式化，读写等。

3.3.2 USB Device 终端设备操作范例

平台作 Device 时可当作终端设备，操作如下：

步骤 1. 插入模块。

```
insmod /mnt/system/ko/u_serial.ko
```

```
insmod /mnt/system/ko/usb_f_acm.ko
```

```
insmod /mnt/system/ko/usb_f_serial.ko
```

步骤 2. 将 otg controller 切换至 device mode

```
echo device > /sys/kernel/debug/usb/39010000.usb/mode
```

步骤 3. 运行 shell script “run_usb.sh”

```
/etc/run_usb.sh probe acm
```

```
/etc/run_usb.sh start
```

通过 USB 将平台与 Host 端相连，即可在 Host 端将平台识别成 USB 终端设备，并在 /dev 目录下生成相应的设备节点 ttyACMX，X 为同类型终端设备号码。在 device 端 /dev 目录下会生成 ttyGSY，Y 为同类型终端设备号码。

Host 和 Device 可透过终端设备进行数据传输。

3.3.3 USB Device RNDIS 设备操作范例

平台作 Device 时可当作 RNDIS 设备，操作如下：

步骤 1. 插入模块。

```
insmod /mnt/system/ko/u_ether.ko
```

```
insmod /mnt/system/ko/usb_f_ecm.ko
```

```
insmod /mnt/system/ko/usb_f_eem.ko
```

```
insmod /mnt/system/ko/usb_f_rndis.ko
```

步骤 2. 将 otg controller 切换至 device mode

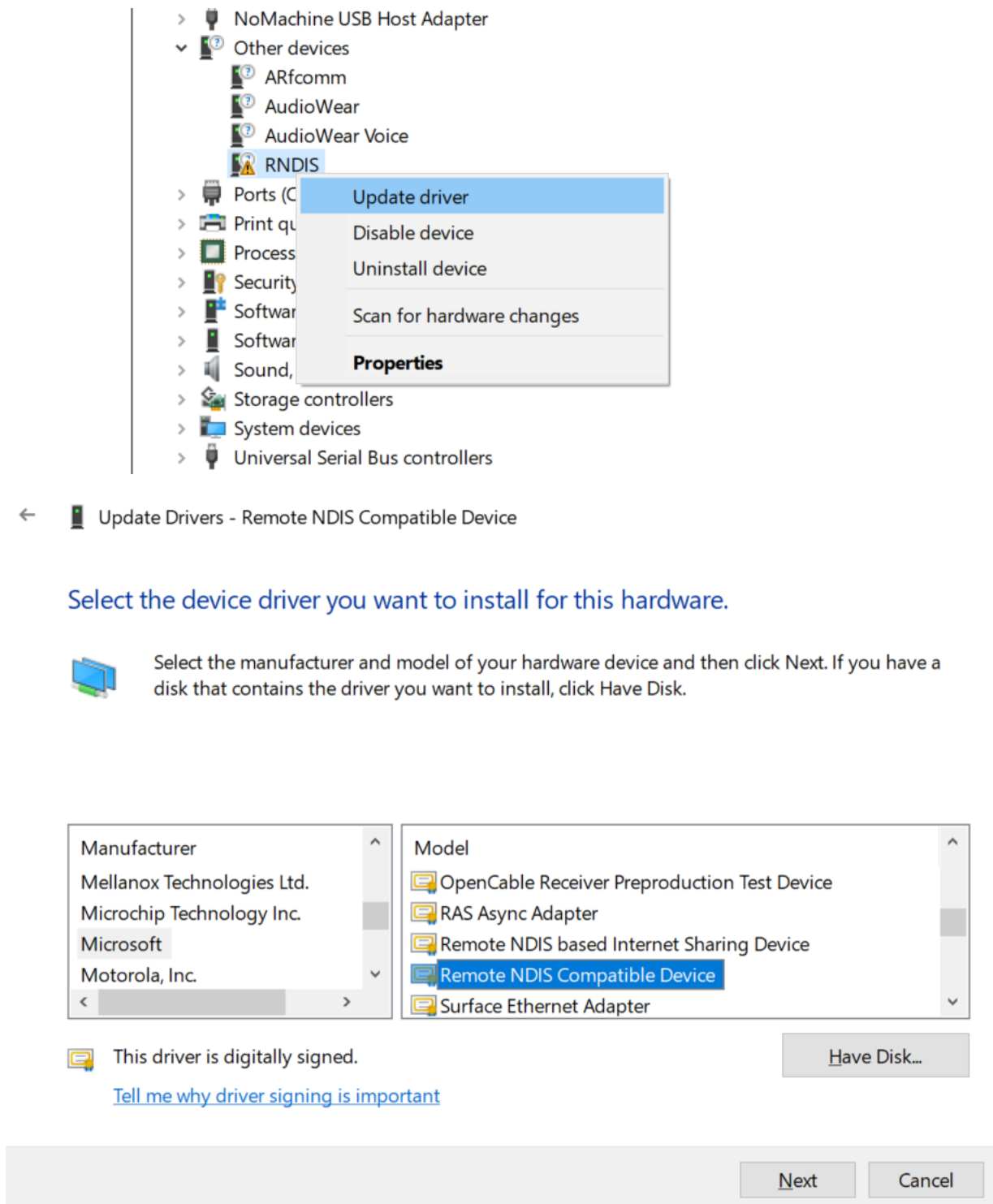
```
echo device > /sys/kernel/debug/usb/39010000.usb/mode
```

步骤 3. 运行 shell script “run_usb.sh”

```
/etc/run_usb.sh probe rndis
```

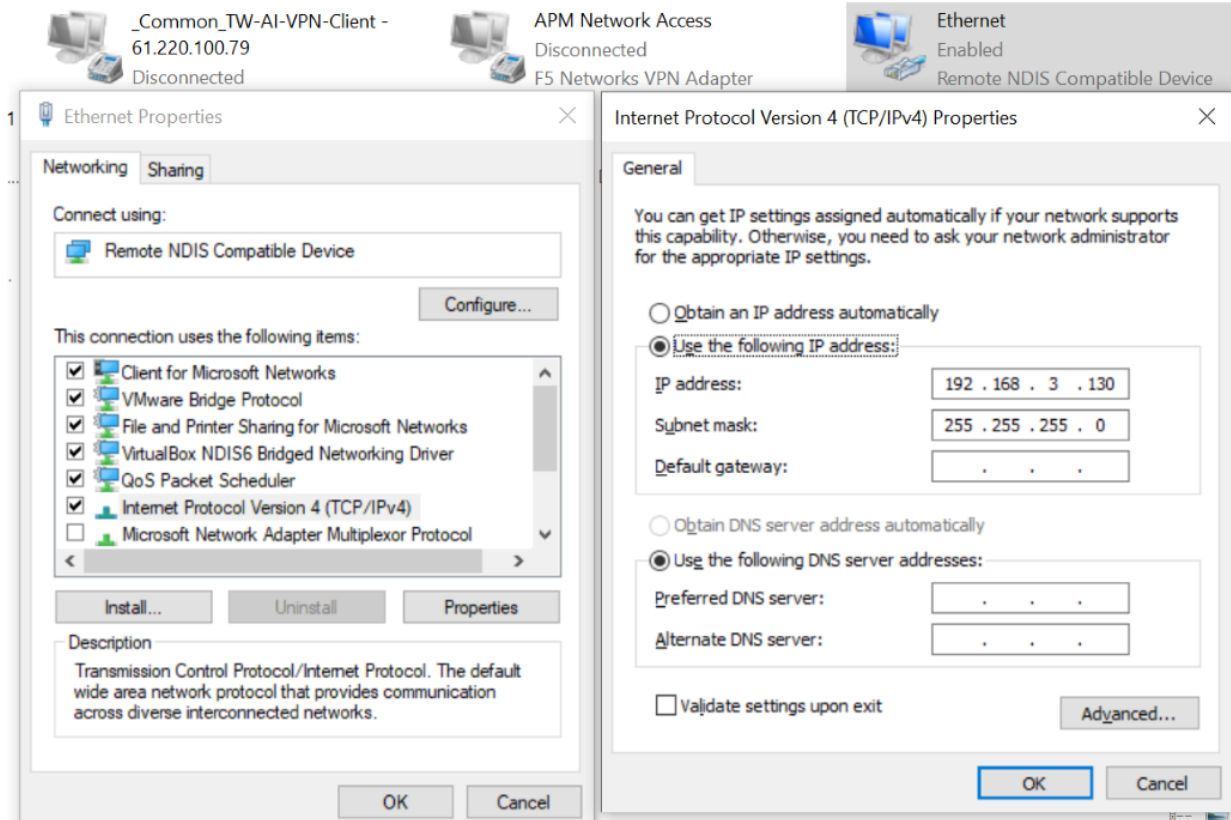
```
/etc/run_usb.sh start
```

步骤 4. 通过 USB 将平台与 Host 端相连，即可在 Host 端将平台识别成 USB Remote NDIS 设备，在 Windows 安装” Remote NDIS Compatible Device” 驱动。



步骤 5. 在单板设定 IP 地址，例如” ifconfig usb0 192.168.3.101 up”。

步骤 6. 在 Window 设定 IP 地址。



Host 和 Device 可透过 RNDIS 设备进行数据传输。

3.4 操作中需注意问题

操作中需要注意的问题如下：

- 系统开机后默认是 Host mode. 若要使用 Device mode 须加载模块并执行 USB ConfigFS 脚本. 当切换 Device 前，用户须确认以下事项：
 - USB Cable 未连接 Host。
 - 平台上的硬件须切换到对应的 USB mode。例如：在切换到 Device mode 前，须关闭平台上的 USB 5V 供电。若平台上有带 Hub, 需关闭 Hub 电源并切换路径 Switch 到 Device mode connector。
- 切换为 Device mode 后，若要再使用 Host mode，用户须重新启动平台。
- 当平台做终端设备时，因 TTY 终端特性，若在短时间内传送大量数据，可能造成数据遗失。用户使用此功能须注意此限制。
- 在 Uboot 下使用 USB Host 读取 U 盘时，注意若平台上有 Hub，须打开 Hub 电源并切换路径 Switch 到正确的 Connector。

4 SD/MMC 卡操作指南

4.1 操作准备

1. 使用 SDK 发布的 U-boot 和 kernel。
2. 文件系统：
对于 SD/MMC 卡来说，SDK 仅支持 FAT 文件系统，支持可读可写。
启动至 kernel 后需挂载至 /mnt/sd 目录或根据项目需求的目录即可。
3. 可透过 fdisk 工具实现分区的工作。
4. CV186AH SD 支持 2.0 与 3.0:

目前 SD 卡支持 1.8/3.3V VDDIO，EMMC 仅支持 1.8V，使用者需注意。

4.2 操作过程

5. 默认 SD/MMC 相关驱动模块已全部编入内核，不需再额外执行加载命令。
6. 插入卡片上电启动，可以在 U-boot 下，通过 fat 相关指令查看卡片内容。启动平台至 kernel 后，会自扫卡识别产生响应节点：/dev/mmcblk0 和 /dev/mmcblk0p1。
7. Uboot 下卡片不支持热插拔操作，kernel 下支持热插拔。在 kernel 下插入 SD 卡，就可以对 SD 卡进行相关的操作。具体操作请参见“3.3 操作示例”

4.3 操作示例

SD 卡的读写操作示例如下。

初始化及应用：

待 SD 卡插入后，进行如下操作（下文 X 为分区号，其值由 fdisk 工具进行分区时决定）：
指定 fdisk 操作的具体目录为：~ \$ fdisk /dev/mmcblk0

步骤 1. 检查分区信息

- a. 若没有显示出 p1, 表示 SD 卡还没有分区, 请在 Linux 下用 fdisk 工具进行分区或是在 windows 系统上将 SD 卡进行格式化之后, 进入步骤 2。
- b. 若有显示分区信息 p1, 则表示 SD 卡已经被检测到, 并已进行过分区, 可进入步骤 2 进行挂载。

步骤 2. 挂载

1. `~ $ mount /dev/mmcblk1pX /mnt/sd`, 此命令会将 SD 卡上第 X 个分区挂载至 /mnt/sd 目录

4.4 操作中需要注意的问题

1. 需确保 SD 卡与卡槽硬件脚位接触良好, 如若接触不良, 有可能会出现检测错误或读写数据错误相关错误信息, 并导致读写失败。
2. 每次插入 SD 卡后, 都需要做一次挂载操作, 才能读写 SD 卡; 如果 SD 卡已经挂载到文件系统, 拔卡前则必须做一次卸载 (umount) 操作, 否则有可能在下次插入 SD 卡后看不到 SD 卡分区。另, 异常拔卡亦需要进行卸载动作。
3. 必须确保 SD 卡已经创建分区, 并将该分区格式化为 FAT 或 FAT32 文件系统 (LINUX 下通过 fdisk 命令, Windows 下使用磁盘管理工具)。
4. 在正常操作过程中不能进行的操作:
 - 读写 SD 卡时不要拔卡, 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。
 - 若当前目录是处于挂载目录之下如 /mnt/sd 时, 则无法进行卸载操作, 必须离开当前目录如 /mnt/sd, 才能进行卸载操作。
 - 系统中读写挂载目录的进程没有完全结束前, 不能进行卸载操作, 必须完全结束操作挂载目录的任务才能正常卸载。
 - 在操作过程中出现异常时的操作:
 1. 如果因为读写数据或其它不明原因导致文件系统被破坏, 读写 SD 卡时可能会出现文件系统错误信息, 这时需要进行卸载操作, 拔卡, 再次插卡并挂载, 才能再次正常读写 SD 卡。
 2. 因为 SD 卡的注册, 检测/注销过程需要一定的时间, 因此拔卡后若再快速地插入卡, 有可能会出现检测不到 SD 卡的现象。
 3. 如果在测试过程中异常拔卡, 使用者需要按 ctrl+c 以回退出到 kernel shell 下, 否则会一直不停地打印异常信息。
 4. SD 卡上有一个以上的分区时, 可以通过挂载操作切换挂载不同的分区, 但最后需确认挂载操作的次数与卸载操作次数相等, 才能确保完全卸载所有的挂载分区。

5 I2C 操作指南

5.1 操作准备

I2C 的操作准备如下：

- 使用 SDK 发布的 kernel。

5.2 操作过程

- 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。
- 在控制台下运行 I2C 读写命令或者自行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。

5.3 接口速率设置说明

如果要更改接口速率，需要修改 build/boards/default/dts/cv186x/cv186x_base.dtsi 中 i2c node 里的 clock_frequency，如下所示，并重新编译内核。

```
i2c0: i2c@29000000 {
    compatible = "snps,designware-i2c";
    clocks = <&clk CV186X_CLK_I2C>;
    reg = <0x0 0x29000000 0x0 0x1000>;
    clock-frequency = <400000>;

    #size-cells = <0x0>;
    #address-cells = <0x1>;
    resets = <&rst RST_I2C0>;
    reset-names = "i2c0";
};
```


5.3.1 I2C 读写命令示例:

可在 linux 终端上发 iic 相关命令 detect 总线设备和对总线上 i2c 设备进行读写。

1. i2cdetect -l

检测系统中的 iic 总线（在 bm1688 中可为 i2c-0~8）

2. i2cdetect -y -r N 检测接到 i2c-N 总线上的所有设备地址，如下检测 i2c-2 上有哪些设备：

```
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[root@cvitek]~# i2cdetect -y -r 2
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- 29 -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[root@cvitek]~#
```

3. i2cdump -f -y N M 查看 i2c-N 上地址为 M 的设备中所有寄存器的值
4. i2cget -f -y 0 0x3c 0x00//读取 i2c-0 上地址为 0x3c 的设备上 0x00 寄存器的值
5. i2cset -f -y 0 0x3c 0x40 0x12//写入 i2c-0 上地址为 0x3c 的设备上 0x40 寄存器

5.3.2 内核态 I2C 读写程序示例:

此示例说明在内核态下如何通过 I2C 读写程序对 I2C 外围设备进行读写操作。

步骤 1. 假设已知外围设备挂载在 I2C 控制器 0 上，调用 i2c_get_adapter() 函数以获得 I2C 控制器结构体 adapter:

```
adapter = i2c_get_adapter(0);
```

步骤 2. 透过 i2c_new_device() 函数关连 I2C 控制器与 I2c 外围设备，以得到 I2C 外围设备的客户端结构体 client:

```
client = i2c_new_device(adapter, &info)
```

备注: info 结构体提供 i2c 外围设备的设备地址

步骤 3. 调用 I2C 核心层提供的标准读写函数对外围设备进行读写操作:

```
ret = i2c_master_send(client, buf, count);
```

```
ret = i2c_master_recv(client, buf, count);
```

备注: client 为步骤 2 所得之客户端结构体，buf 为需要读写的寄存器地址以及数据，count 为 buf 的长度。

代码示例如下:

```
//宣告一个外围设备名字叫做" dummy", 设备地址为0x3c
static struct i2c_board_info info = {
    I2C_BOARD_INFO("dummy", 0x3C),
};
static struct i2c_client *client;

static int cvi_i2c_dev_init(void) {
    //分配 i2c控制器指针
    struct i2c_adapter *adapter;

    adapter = i2c_get_adapter(0);
    client = i2c_new_device(adapter, &info);
    i2c_put_adapter(adapter);
    return 0;
}

static int i2c_dev_write(char *buf, unsigned int count){
    int ret;

    ret = i2c_master_write(client, buf, count);
    return ret;
}

static int i2c_dev_read(char *buf, unsigned int count) {
    int ret;

    ret = i2c_master_recv(client, buf, count);
    return ret;
}
```

5.3.3 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:

```
i2c_file = open("/dev/i2c-0", O_RDWR);
if (i2c_file < 0) {
    printf("open I2C device failed %d\n", errno);
    return -ENODEV;
}
```

步骤 2.
进行数据读写:

```
ret = ioctl(file, I2C_RDWR, &packets);
if (ret < 0) {
    perror("Unable to send data");
    return ret;
}
```

备注: 需于 flags 指定读写操作

```
struct i2c_msg messages[2];
int ret;

/*
 * In order to read a register, we first do a "dummy write" by writing
 * 0 bytes to the register we want to read from. This is similar to
 * the packet in set_i2c_register, except it's 1 byte rather than 2.
 */
outbuf = reg;
messages[0].addr = addr;
messages[0].flags = 0;
messages[0].len = sizeof(outbuf);
messages[0].buf = &outbuf;

/* The data will get returned in this structure */
messages[1].addr = addr;
/* | I2C_M_NOSTART */
messages[1].flags = I2C_M_RD;
messages[1].len = sizeof(inbuf);
messages[1].buf = &inbuf;
```

6 SPI 操作指南

6.1 操作准备

SPI 的操作准备如下：

- 使用 SDK 发布的内核以及文件系统。文件系统可使用 SDK 所发布的 squashFS 或 ext4。也可透过本地文件系统再透过网络挂载至 NFS。

6.2 操作过程

- 加载内核。把 SPI 相关模块全部编入内核，不需要再执行加载命令。
- 在控制台下运行 SPI 读写命令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 SPI 控制器上的外围设备进行读写操作。

6.3 操作示例

6.3.1 内核态 SPI 读写程序示例：

此操作示例说明在内核态下如何通过 SPI 读写程序实现对 SPI 外围设备的读写操作。

步骤 1. 调用 SPI 核心层函数 `spi_busnum_to_master()`，以获取一个描述 SPI 控制器结构体：

```
master = spi_busnum_to_master(bus_num);  
//bus_num 为要读写的 SPI 外围设备所在的控制器号  
//master 为描述 SPI 控制器的 spi_master 结构体类型指针
```

步骤 2. 通过 spi 外围设备在核心层的名称调用 SPI 核心层函数取得挂载在 SPI 控制器上描述 SPI 外围设备的结构体：

```
snprintf(str, sizeof(str), "%s.%u", dev_name(&master->dev), cs);  
dev = bus_find_device_by_name(&spi_bus_type, NULL, str);  
spi = to_spi_device(dev);
```

//spi_buf_type 为描述 SPI 总线的 bus_type 结构体类型变量

//spi 为描述 SPI 外围设备 spi_device 结构体类型指针

步骤 3. 调用 SPI 核心层函数将 spi_transfer 添加到 spi_message 队列中。

```
spi_message_init(&m)
```

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

//t 为 spi_transfer 结构体类型变量

//m 为 spi_message 结构体类型变量

步骤 4. 调用 SPI 核心层停工的读写函数对外围设备进行读写操作

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

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

//spi 为描述 SPI 外围设备的 spi_device 结构体类型指针

//spi_sync 函数为进行 spi 同步读写操作

//spi_async 函数为进行 spi 异步读写操作

代码示例如下：

此段代码示例仅供参考，而非实际应用功能。

```
//传入SPI控制器总线号和片选号
static unsigned int busnum;module_param(busnum, uint, 0);
MODULE_PARM_DESC(busnum, "SPI busnumber (default=0)");

static unsigned int cs;
module_param(cs, uint, 0);
MODULE_PARM_DESC(cs, "SPI chip select (default=0)");

extern struct bus_type spi_bus_type;

//宣告SPI控制器的结构体
static struct spi_master *master;

//宣告SPI外围设备的结构体
static struct spi_device *spi_device;

static int __init spidev_init(void) {
    char *spi_name;
    struct device *spi;
    master = spi_busnum_to_master(busnum);
    spi_name = kzalloc(strlen(dev_name(&master->dev)), GFP_KERNEL);

    if (!spi_name)
        return -ENOMEM;

    snprintf(spi_name, sizeof(spi_name), "%s.%u", dev_name(&master->dev), cs);
    spi = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);
    if (spi == NULL)
        return -EPERM;
```

(下页继续)

(续上页)

```

spi_device = to_spi_device(spi);
if (spi_device == NULL)
    return -EPERM;

put_device(spi);
kfree(spi_name);

return 0;
}

int spi_dev_write(, void *buf, unsigned long len, int buswidth)
{
    struct spi_device *spi = spi_device;
    struct spi_transfer t = {
        .speed_hz = 2000000,
        .tx_buf = buf,
        //buf里需依外围设备规范填入device addr, register addr, write data等资讯
        .len = len,
    };
    struct spi_message m;
    spi->mode = SPI_MODE_0;

    if (buswidth == 16)
        t.bits_per_word = 16;
    else
        t.bits_per_word = 8;
    if (!spi) {
        return -ENODEV;
    }
    spi_message_init(&m);
    spi_message_add_tail(&t, &m);
    return spi_sync(spi, &m);
}

int spi_dev_read(unsigned char devaddr, unsigned char reg_addr, void *buf, size_t len)
{
    struct spi_device *spi = spi_device;
    int ret;
    u8 txbuf[4] = { 0, };
    struct spi_transfer t = {
        .speed_hz = 2000000,
        .rx_buf = buf,
        .len = len,
    };
    struct spi_message m;
    spi->mode = SPI_MODE_0;

    if (!spi) {
        return -ENODEV;
    }
    txbuf[0] = devaddr;
    txbuf[1] = 0;
    txbuf[2] = reg_addr; //txbuf[1]&txbuf[2]需根据外围设备位宽来决定填写1 byte or 2
    ↪ bytes, 此范例为2 | bytes位宽
    t.tx_buf = txbuf;

```

(下页继续)

(续上页)

```
spi_message_init(&m);
spi_message_add_tail(&t, &m);
ret = spi_sync(spi, &m);

return ret;
}
```

6.3.2 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 上的 SPI 外围设备的读写操作。(具体实现可参考 tools/spi/spidev_test.c)

步骤 1: 打开 SPI0 总线对应的设备文件, 获取文件描述符。至多有 4 个 spi 设备: spidev0.0 spidev1.0 spidev2.0 spidev3.0

```
static const char *device = "/dev/spidev0.0";
...
fd = open(device, O_RDWR);
if (fd < 0)
    pabort("can't open device");
```

备注: SPI 控制器 0 上挂载的外围设备默认节点为 “dev/spidev0.0”

SPI 控制器 1 上挂载的外围设备默认节点为 “dev/spidev1.0”

SPI 控制器 2 上挂载的外围设备默认节点为 “dev/spidev2.0”

SPI 控制器 3 上挂载的外围设备默认节点为 “dev/spidev3.0”

仅需替换节点名称即可, 其余操作与 SPI 控制器 0 上挂载的外围设备相同。

步骤 2: 通过 ioctl 设置 SPI 传输模式:

```
/*
 *spi mode
 */
ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
if (ret == -1)
    pabort("can't set spi mode");
ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
if (ret == -1)
    pabort("can't get spi mode");
```

备注: mode 值配置请参考下图或是内核代码 include/linux/spi/spi.h,

Ex. mode = SPI_MODE_3 | SPI_LSB_FIRST;

```
#define SPI_CPHA 0x01 /* clock phase */
#define SPI_CPOL 0x02 /* clock polarity */
#define SPI_MODE_0 (0|0) /* (original MicroWire) */
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
```

步骤 3: 通过 ioctl 设置 SPI 传输频宽:

```
/*
 * bits per word
 */
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
    pabort("can't set bits per word");
ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
if (ret == -1)
    pabort("can't get bits per word");
```

步骤 4: 通过 ioctl 设置 SPI 传输速度 (一般建议 speed = 25M):

```
/*
 * max speed hz
 */
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
    pabort("can't set max speed hz");

ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
if (ret == -1)
    pabort("can't get max speed hz");
```

备注: 使用 ioctl 进行 speed、mode、width 等设置后不会立即更新 SPI 的状态, 而是将所有的修改保存到

controller 的结构体中, 在下次传输的 update config 阶段统一更新到 IP 寄存器。因此若想更新的配置立即生效,

可以在调用 ioctl 更新之后传输一段 dummy 无效数据更新 SPI 工作状态。

步骤 5: 使用 ioctl 进行数据读写:

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
if (ret < 1)
    pabort("can't send spi message");
```

备注: tr 传输一帧消息的 spi_ioc 结构体数组首地址。

7 GPIO 操作指南

7.1 GPIO 的操作准备如下：

- 使用 SDK 发布的 kernel

7.2 操作过程

- 默认 GPIO 相关模块已全部编入内核，不需要再执行加载命令。
- 在控制台下运行 GPIO 读写命令或者自行在内核态或者用户态编写 GPIO 读写程序，就可以对 GPIO 进行输入输出操作。

7.3 操作示例

7.3.1 GPIO 操作命令示例：

步骤 1: 在控制台使用 echo 命令, 指定待操作的 GPIO 编号 N:

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

N 为待操作的 GPIO 编号, $\text{GPIO 编号} = \text{GPIO 组号值} + \text{偏移值}$.

以原理图中 GPIO_num 管脚为例, $\text{num} \% 32 = \text{base} \cdots \text{off}$, 对应的组号为 base, 偏移值为 off。例如 GPIO32, $32 \% 32 = 1 \cdots 0$, 则其组号为 1, 偏移值为 0, 组号 1 对应的组号值为 448

因此 GPIO 编号 N 为 $448 + 0 = 448$

组号值对应如下:

组号 0 对应 linux 组号值为 480

组号 1 对应 linux 组号值为 448

组号 2 对应 linux 组号值为 416

组号 3 对应 linux 组号值为 384

组号 4 对应 linux 组号值为 352
组号 5 对应 linux 组号值为 320
pwr_gpio 对应 linux 组号值为 288

echo N > /sys/class/gpio/export 之后, 生成/sys/class/gpio/gpioN 目录

步骤 2: 在控制台使用 echo 命令设置 GPIO 方向:

设置为输入: echo in > /sys/class/gpio/gpioN/direction

设置为输出: echo out > /sys/class/gpio/gpioN/direction

例: 设置 GPIO32 (即编号 448) 方向为输入:

echo in > /sys/class/gpio/gpio448/direction

设置 GPIO32 (即编号 448) 方向为输出:

echo out > /sys/class/gpio/gpio448/direction

步骤 3: 在控制台使用 cat 命令查看 GPIO 输入值, 或使用 echo 命令设置 GPIO 输出值:

查看输入值: cat /sys/class/gpio/gpioN/value

或

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

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

步骤 4: 使用完毕后, 在控制台使用 echo 命令释放资源:

echo N > /sys/class/gpio/unexport

注: 可以通过打开 CONFIG_DEBUG_FS 选项开启 gpio 的 sysfs debug 功能, 在操作前通过如下命令查看 gpio pin 具体对应的组号值:

cat /sys/kernel/debug/gpio

7.3.2 内核态 GPIO 操作程序示例:

内核态 GPIO 读写操作程序示例:

步骤 1: 注册 GPIO:

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

gpio_num 为要操作的 GPIO 编号, 该编号等于” GPIO 组号 + 组内偏移号”

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)
```

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);
```

步骤 4: 释放注册的 GPIO 编号:

```
gpio_free(gpio_num);
```

内核态 GPIO 中断操作程序示例:

步骤 1: 注册 GPIO:

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

gpio_num 为要操作的 GPIO 编号, 该编号等于” GPIO 组号 + 组内偏移号”

步骤 2: 设置 GPIO 方向:

```
gpio_direction_input(gpio_num);
```

对于要作为中断源的 GPIO 引脚, 方向必须配置为输入对于输出

步骤 3: 映射操作的 GPIO 编号对应的中断号:

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

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))
```

Irqflags 为需要注册的中断类型, 常用类型为:

- 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);
```

7.3.3 用户态 GPIO 操作程序示例：

用户态 GPIO 读写操作程序示例：

步骤 1: 将要操作的 GPIO 编号 export:

```
fp = fopen("/sys/class/gpio/export", "w");
fprintf(fp, "%d", gpio_num);
fclose(fp);
```

gpio_num 为要操作的 GPIO 编号，该编号等于” GPIO 组号 + 组内偏移号”

步骤 2: 设置 GPIO 方向:

```
fp = fopen("/sys/class/gpio/gpio%d/direction", "rb+");
对于输入: fprintf(fp, "in");
对于输出: fprintf(fp, "out");

fclose(fp);
```

步骤 3: 查看 GPIO 输入值或设置 GPIO 输出值:

```
fp = fopen("/sys/class/gpio/gpio%d/direction", "rb+");
查看输入: fread(buf, sizeof(char), sizeof(buf) - 1, fp);
输出低:
    strcpy(buf, "0" );
    fwrite(buf, sizeof(char), sizeof(buf) - 1, fp);
输出高:
    strcpy(buf, "1" );
    fwrite(buf, sizeof(char), sizeof(buf) - 1, fp);
```

步骤 4: 将操作的 GPIO 编号 unexport:

```
fp = fopen("/sys/class/gpio/unexport", "w");
fprintf(fp, "%d", gpio_num);
fclose(fp);
```

8 UART 操作指南

8.1 UART 的操作准备如下

- 使用 SDK 发布的 kernel;
- 将需要测试的设备树节点 status 改为 “okay”;

```
uart0: serial@29180000 {  
    compatible = "snps,dw-apb-uart";  
    reg = <0x0 0x29180000 0x0 0x1000>;  
    clock-frequency = <200000000>;  
    reg-shift = <2>;  
    reg-io-width = <4>;  
    status = "okay";  
};
```

8.2 操作过程

8.2.1 命令行操作示例:

- 设置好对应引脚的 PINMUX;
- 执行以下指令设定波特率为 115200, 数据位 8, 停止位 1, 无奇偶校验位 (ttySx, x 为 0~8):

```
stty -F /dev/ttySx 115200 cs8 -cstopb -parenb
```

- 进行收发数据:

```
echo 123 > /dev/ttySx # 向 uartx 输入 123  
cat /dev/ttySx
```

8.2.2 用户态 UART 操作程序示例：

```
/* Open your specific device (e.g., /dev/mydevice): */
fd = open ("/dev/ttyS0", O_RDWR);
if (fd < 0) {
    /* Error handling. See errno. */
    return -1;
}

struct termios t;
tcgetattr(fd, &t);
cfsetispeed(&t, B115200); // 设置输入波特率为115200
cfsetospeed(&t, B115200); // 设置输出波特率为115200
t.c_cflag &= ~PARENB; // 不使用奇偶校验
t.c_cflag &= ~CSTOPB; // 使用1位停止位
t.c_cflag &= ~CSIZE; // 清除数据位设置
t.c_cflag |= CS8; // 设置数据位为8位
tcsetattr(fd, TCSANOW, &t);
// read/write ...
```

9 Watchdog 操作指南

9.1 Watchdog 的操作准备如下：

- 使用 SDK 发布的 kernel。

9.2 模块编译

- 插入模块:cv186x 则 insmod soph_wdt.ko.
- 在控制台下运行 Watchdog 读写命令或者自行在内核态或者用户态编写 Watchdog 读写程序，即可操作 Watchdog 。

9.3 操作示例

Watchdog 采用标准的 linux 框架, 提供硬件 watchdog。用户只需打开, 关闭或设定 timeout, 即可使用 watchdog。当 watchdog timeout 发生时, 系统重新启动。

Watchdog 默认是关闭的, 客户可自行决定是否开启。可指定的 timeout 时间分别为 1 秒, 2 秒, 5 秒, 10 秒, 21 秒, 42 秒, 85 秒。

当使用者输入 timeout 时间为 8 秒, 驱动会选择大于等于该值的 timeout 即 10 秒; 若未设定 timeout, 驱动采用默认 42 秒。

- 打开 WATCHDOG

打开/dev/watchdog 设备, watchdog 即被启动。打开之后应该立刻 ping (喂狗), 否则 wdt 将会立刻重启。

```
int wdt_fd = -1;
wdt_fd = open("/dev/watchdog", O_WRONLY);
if (wdt_fd == -1)
{
    // fail to open watchdog device
}
ioctl(fd, WDIOC_KEEPAIVE, 0);
```

- 关闭 WATCHDOG

驱动支持” Magic Close”，在关闭 watchdog 前必須將 magic 字符’ V’ 写入 watchdog 设备。

如果 userspace daemon 没有发送’ V’ 而直接关闭設備，則 watchdog 驱

动持續計數, 在给定时间内未喂狗仍会导致 timeout, 系統重启.

参考代码如下:

```
int option = WDIOF_DISABLECARD;
ioctl(wdt_fd, WDIOF_SETOPTIONS, &option);
if (wdt_fd != -1)
{
    write(wdt_fd, "V", 1);
    close(wdt_fd);
    wdt_fd = -1;
}
```

- 设定 TIMEOUT 值

通过标准的 IOCTL 命令 WDIOF_SETTIMEOUT 设定 timeout, 单位为秒. 可指定的 timeout 时间分别为 1 秒, 2 秒, 5 秒, 10 秒, 21 秒, 42 秒, 85 秒.

```
#define WATCHDOG_IOCTL_BASE 'W'
#define WDIOF_SETTIMEOUT \_IOWR(WATCHDOG_IOCTL_BASE, 6, int)

int timeout = 10;
ioctl(wdt_fd, WDIOF_SETTIMEOUT, &timeout);
```

- PING watchdog (喂狗)

通过标准的 IOCTL 命令 WDIOF_KEEPAWAKE 喂狗.

```
while (1) {
    ioctl(fd, WDIOF_KEEPAWAKE, 0);
    sleep(1);
}
```


10 PWM 操作指南

10.1 PWM 的操作准备如下：

- 使用 SDK 发布的 kernel。

10.2 操作过程

- 插入模块:cv186x 则 insmod soph_pwm.ko;
- 在控制台下运行 PMW 读写命令或者自行在内核态或者用户态编写 PWM 读写程序，就可以对 PWM 进行输入输出操作;
- PWM 操作在定频时钟 100MHz，共有 20 路，每路可单独控制;
- CV186AH 共有 5 个 PWM IP (pwmchip0/ pwmchip4/ pwmchip8/ pwmchip12/ pwmchip16), 各 IP 控制 5 路讯号，总共可控制 20 路讯号；电路图上以 pwm0~pwm15 表示

在 Linux sysfs 中, pwm0~pwm3 的 device node 各自如下:

```
/sys/class/pwm/pwmchip0/pwm0~3
```

在 Linux sysfs 中, pwm4~pwm7 的 device node 各自如下:

```
/sys/class/pwm/pwmchip:mark:4/pwm0~3
```

以此类推

10.3 操作示例

10.3.1 PWM 操作命令示例：

步骤 1:

在控制面板使用 echo 命令，配置待操作的 PWM 编号，此例为 PWM1:

```
echo 1 > /sys/class/pwm/pwmchip0/export
```

步骤 2:

设置 PWM 一个周期的持续时间, 单位为 ns:

```
echo 1000000 >/sys/class/pwm/pwmchip0/pwm1/period
```

步骤 3:

设置一个周期中的” ON” 时间, 单位为 ns, 即占空比 =duty_cycle/period=50% :

```
echo 500000 >/sys/class/pwm/pwmchip0/pwm1/duty_cycle
```

步骤 4:

设置 PWM 使能

```
echo 1 >/sys/class/pwm/pwmchip0/pwm1/enable
```

10.3.2 通过文件 IO 操作程序示例:

用户态 GPIO 读写操作程序示例:

步骤 1: 配置待操作的 PWM 编号, 以 PWM1 为例:

```
fd = open("/sys/class/pwm/pwmchip0/export", O_WRONLY);
if(fd < 0)
{
    dbmsg("open export error\n");
    return -1;
}
ret = write(fd, "1", strlen("0"));
if(ret < 0)
{
    dbmsg("Export pwm1 error\n");
    return -1;
}
```

步骤 2: 设置 PWM 一个周期的持续时间, 单位为 ns:

```
fd_period = open("/sys/class/pwm/pwmchip0/pwm1/period", O_RDWR);
ret = write(fd_period, "1000000", strlen("1000000"));
if(ret < 0)
{
    dbmsg("Set period error\n");
    return -1;
}
```

步骤 3: 设置一个周期中的” ON” 时间, 单位为 ns: (此例占空比为 50%)

```
fd_duty = open("/sys/class/pwm/pwmchip0/pwm1/duty_cycle", O_RDWR);
ret = write(fd_duty, "500000", strlen("500000"));
if(ret < 0)
{
    dbmsg("Set period error\n");
    return -1;
}
```

步骤 4: 设置 PWM 使能

```
fd_enable = open("/sys/class/pwm/pwmchip0/pwm1/enable", O_RDWR);
ret = write(fd_enable, "1", strlen("1"));
if(ret < 0)
{
    dbmsg("enable pwm0 error\n");
    return -1;
}
```

11 ADC 操作指南

11.1 ADC 的操作准备如下：

- 使用 SDK 发布的 kernel。

11.2 操作过程

- 插入模块：CV186AH 则 `insmod soph_saradc.ko`。
- 在控制台下运行 ADC 读写命令或者自行在内核态或者用户态编写 ADC 读写程序，就可以对 ADC 进行输入输出操作。
- 用户层通过访问 IIO 接口来实现 5 通道，12-bit ADC 的触发、采样等操作。
- 1.5v ref 参考电压。
- adc 引脚和 sysfs 文件对应关系如下：

adc1 对应 sysfs 文件为 `in_voltage1_raw`

adc2 对应 sysfs 文件为 `in_voltage2_raw`

adc3 对应 sysfs 文件为 `in_voltage3_raw`

sar0 对应 sysfs 文件为 `in_voltage4_raw`

sar1 对应 sysfs 文件为 `in_voltage5_raw`

- 电压值计算公式： $vol = val * 1500 / 4096$ ，单位：mV

11.3 操作示例

11.3.1 ADC 操作命令示例：

步骤 1: | 指定 ADC 通道 1~5, 此例为 ADC1:

```
echo 1 > /sys/bus/iio/devices/iio:device0/in_voltage1_raw
```

步骤 2:

读出刚才指定的 ADC channel 值:

```
cat /sys/bus/iio/devices/iio:device0/in_voltage1_raw
```

11.3.2 用户态 ADC 读取操作程序示例：

用户态 ADC 读写操作程序示例：

```
fd = open("/sys/bus/iio/devices/iio:device0/in_voltage1_raw", O_RDWR|
O_NOCTTY|O_NDELAY);
if (fd < 0)
    printf("open adc err!\n");

write(fd, "1", 1);
lseek(fd, -1, SEEK_CUR);

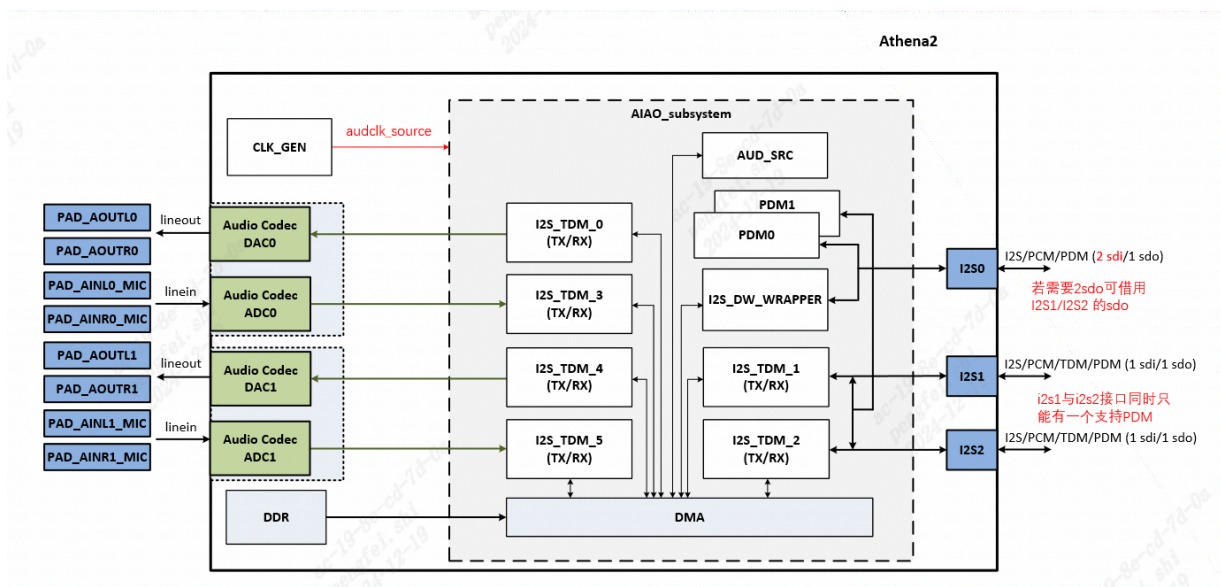
char buffer[512] = {0};
int len = 0;
unsigned int adc_value = 0;

len = read(fd, buffer, 5);
if (len != 0) {
    printf("read buf: %s\n", buffer);
    adc_value = atoi(buffer);
    printf("adc value is %d\n", adc_value);
}
write(fd, "0", 1);
close(fd);
```

12 I2S 操作指南

12.1 概述

本文章说明如何配置外接 Audio Codec 进行输入输出



若外接 Codec 仅有一路 I2S interface, 假设是与 CV186X I2S0 作对接, 那么就必須针对 I2S0 做 pinmux 配置。

12.2 配置说明

步骤 1. 编辑 u-boot-2021.10/board/cvitek/cv186x/board.c

在 `board_init()` 中将 I2S0 的 `pinmux` 打开。

```
PINMUX_CONFIG(RMII0_TXD0, EPHY_SPD_LED); //EPHY_LNK_LED
PINMUX_CONFIG(PWM3, PWM_3); //white_led
#endif

pinmux_config(PINMUX_I2S0);
return 0;
```

另外，若 codec 是通过 I2C5 与 CV186X 进行控制，那么还需要配置 u-boot-2021.10/board/cvitek/cvi_board_init.c 于 cvi_board_init() 里将 I2C5 的 pinmux 打开

```
PINMUX_CONFIG(SD1_D0, IIC5_SDA, G11);
PINMUX_CONFIG(SD1_D1, IIC5_SCL, G11);
```

步骤 2. 编辑 build/boards/sophon/cv186ah_wevb_emmc/linux/cv186ah_wevb_emmc_defconfig (不同产品线分支该配置文件目录和名称可能不一样，请客户以自己实际目录文件为准)

需开启同时可以进行 TX/RX 的 CONFIG 以及将 Codec 相关 CONFIG 打开 (此以外接 es8316 为例)。

```
#
# AUDIO
#
#dwi2s-rt5616
CONFIG_SND_SOC_ES8316=y
CONFIG_SND_SOC_CV186X_DWI2S=y
CONFIG_SND_SOC_CV186X_DWI2S_ES8316=y
#
```

确认 Linux/sound/soc/codecs/Kconfig 是否有配置：

```
config SND_SOC_CV186X_DWI2S_ES8316
tristate "Support for the CV186X-ES8316 card"
help

Say y or m if you want to add support for the analog
codec ES8316 connect to the CVITEK CV186X boards.
Then select it for built in or module.
If not, please don't select it
```

确认 Linux/sound/soc/codecs/Makefile 是否有配置：

```
obj-$(CONFIG_SND_SOC_CV186X_DWI2S_ES8316) += cv186x_es8316.o
```

如果发现配置后无法编译相关文件可以清除一次 kernel。

查看是否配置上：

```
vi linux_5.10/build/cv186ah_wevb_emmc/.config
```

查看是否有编译对应的 c 文件为相关.o 文件：

```
ls linux_5.10/build/cv186ah_wevb_emmc/sound/soc/cvitek
ls linux_5.10/build/cv186ah_wevb_emmc/sound/soc/codecs
```

步骤 3. 编辑 build/boards/default/dts/sophon/soph_base.dtsi (不同产品线分支该配置文件目录和名称可能不一样，请客户以自己实际目录文件为准)

以 Codec 通过 I2C5 进行控制为例

```
i2c5: i2c@29050000 {
    compatible = "snps,designware-i2c";
    clocks = <&clk CV186X_CLK_I2C5>;
```

(下页继续)

(续上页)

```
reg = <0x0 0x29050000 0x0 0x1000>;
clock-frequency = <400000>;

#size-cells = <0x0>;
#address-cells = <0x1>;
resets = <&rst RST_I2C5>;
reset-names = "i2c5";

es8316: es8316@10 {
    compatible = "everest,es8316";
    reg = <0x10>;
    clocks = <&i2s_mclk>;
    clock-names = "mclk";
};
```

```
i2s6: i2s@29178000 {
    compatible = "cvitek,cvitek-dwi2s";
    reg = <0x0 0x29178000 0x0 0x400>;
    clocks = <&i2s_mclk 0>;
    clock-names = "i2sclk";

    dev-id = <6>;
    #sound-dai-cells = <0>;
    dmas = <&dmac0 4 1 1>, <&dmac0 5 1 1>; /* read channel */
    dma-names = "rx", "tx";
    capability = "txrx";
    mclk_out = "true";
};
```

```
sound_ext3 {
    compatible = "cvitek,cv186x-es8316";
    cvi,model = "CV1835";
    cvi,mode = "I2S";
    cvi,fmt = "NBNF";
    cvi,card_name = "cvi_cv186x_dw1";
    cvi,slot_no=<2>;

    dai@0 {
        cvi,stream_name = "ES8316 HiFi";
        cvi,role = "master"; //soc为master, codec为slave
    };
};
```

配置完成后实际机器情况：


```
[root@cvitek]/# ls /proc/device-tree/
#address-cells      i2c@29020000      restart-controller
#size-cells         i2c@29030000      rgn
adc@28109100        i2c@29040000      rtc
adc@2810A100        i2c@29050000      rtos_cmdqu
aliases             i2c@29060000      rx-queues-config
audio_clock         i2c@29070000      saradc
base                i2c@29080000      sata@20bc0000
bmtpu               i2c@29090000      serial-number
bt_pin              i2s@29110000      serial@05022000
can@29240000        i2s@29120000      serial@29180000
can@29250000        i2s@29130000      serial@29190000
chosen              i2s@29140000      serial@291a0000
cif                 i2s@29150000      serial@291b0000
cif_v4l2            i2s@29160000      serial@291c0000
clk_100k            i2s@29178000      serial@291d0000
clk_32k             i2s_mclk           serial@291e0000
clk_xtal_free       i2s_subsys         serial@291f0000
```

```
[root@cvitek]~# ls /proc/device-tree/i2c@29050000/
#address-cells  clocks      interrupts      reset-names
#size-cells     compatible  name            resets
clock-frequency es3316@10  reg
```

```
[root@cvitek]~# ls /proc/device-tree/
#address-cells      i2c@29020000      rgn
#size-cells         i2c@29030000      rtc
adc@28109100        i2c@29040000      rtos_cmdqu
adc@2810A100        i2c@29050000      rx-queues-config
aliases             i2c@29060000      saradc
audio_clock         i2c@29070000      sata@20bc0000
base                i2c@29080000      serial-number
bmtpu               i2c@29090000      serial@05022000
bt_pin              i2s@29110000      serial@29180000
can@29240000        i2s@29120000      serial@29190000
can@29250000        i2s@29130000      serial@291a0000
chosen              i2s@29140000      serial@291b0000
cif                 i2s@29150000      serial@291c0000
cif_v4l2            i2s@29160000      serial@291d0000
clk_100k            i2s@29178000      serial@291e0000
clk_32k             i2s_mclk           serial@291f0000
clk_xtal_free       i2s_subsys         sophgo-cooling
clock-controller    interrupt-controller sound_PDM
compatible          interrupt-parent   sound_adc
cpus                irrx@0502E000     sound_dac
cv-emmc@29300000    ispv4l2           sound_ext1
cv-sd1@29320000     ive              sound_ext2
cv-sd@29310000      jpu              sound_ext3
cv-wd@0x27000000    keyscan@27030000  spacc
```

可以在 shell 执行 `cat` 查看每个设备树里每一项是否和代码配置的一样。备注：上例是在 186x 使用 linux5.10 版本 I2C 启动 I2C0-I2C9 的状况，因内核会认定 I2C0 对应 id 0, I2C1 对应 id 1, I2C2 对应 id 2, 依此类推。当发现外置 codec 无法注册成功可以确认该索引号是否配对。用以下方法可以确定：

```
[root@cvitek]/# ls /sys/bus/i2c/drivers/es8316/
5-0010 bind uevent unbind
```

12.3 常见问题

问题 1. 配置后无声音，首先确保对应的 pcm 节点有出来：ls /dev/snd/

```
[root@cvitek]~# ls /dev/snd
controlC0 controlC2 pcmC0D1c pcmC1D1p pcmC2D0p
controlC1 pcmC0D0c pcmC1D0p pcmC2D0c timer

[root@cvitek]~# cat /proc/asound/cards
0 [cv186xadc ]: cv186x_adc - cv186x_adc
cv186x_adc
1 [cv186xdac ]: cv186x_dac - cv186x_dac
cv186x_dac
2 [cvi cv186xdw1 ]: cvi_cv186x_dw1 - cvi_cv186x_dw1
cvi cv186x dw1
```

问题 2. 检查./tinyplay dukou_16K_2_32b.wav -D 2 -d 0 可以正常播放，使用示波器抓 bclk, lrclk, Dout pin 脚有正常波形。使用./tinycap play.wav -D 2 -d 0 -c 2 -r 16000 -b 32 -T 10 可以正常录制，使用示波器抓 bclk, lrclk, Din pin 脚有正常波形。

问题 3. 如果有正常波形，但是播音无声，请先参考 datasheet 测试 codec 在 bypass adc&dac 情况下是否有声，以及 codec loopback 测试是否有声音。

问题 4. 有些 codec 的默认配置是关闭的，如 es8316, 可以用 tinymix 进行修改。(不同 codec 请参考原厂 datasheet)

```
./tinymix -D 2
```

ctl	type	num	name	value
0	INT	2	Headphone Playback Volume	3 3
1	INT	2	Headphone Mixer Volume	0 0
2	ENUM	1	Playback Polarity	Normal
3	INT	2	DAC Playback Volume	192 192 //默认是 0
4	BOOL	1	DAC Soft Ramp Switch	Off
5	INT	1	DAC Soft Ramp Rate	4
6	BOOL	1	DAC Notch Filter Switch	Off
7	BOOL	1	DAC Double Fs Switch	Off
8	INT	1	DAC Stereo Enhancement	0
9	BOOL	1	DAC Mono Mix Switch	Off
10	ENUM	1	Capture Polarity	Normal
11	BOOL	1	Mic Boost Switch	On
12	INT	1	ADC Capture Volume	192 //默认是 0
13	INT	1	ADC PGA Gain Volume	5 //默认是 0
14	BOOL	1	ADC Soft Ramp Switch	On
15	BOOL	1	ADC Double Fs Switch	Off
16	BOOL	1	ALC Capture Switch	Off
17	INT	1	ALC Capture Max Volume	28
18	INT	1	ALC Capture Min Volume	0
19	INT	1	ALC Capture Target Volume	11

(下页继续)

(续上页)

20	INT	1	ALC Capture Hold Time	0
21	INT	1	ALC Capture Decay Time	3
22	INT	1	ALC Capture Attack Time	2
23	BOOL	1	ALC Capture Noise Gate Switch	Off
24	INT	1	ALC Capture Noise Gate Threshold	0
25	ENUM	1	ALC Capture Noise Gate Type	Constant PGA Gain

在播音前需设置：

```
./tinymix -D 2 3 192 192
./tinymix -D 2 32 1
./tinymix -D 2 34 1
```

在录音前需设置：

```
./tinymix -D 2 12 192
./tinymix -D 2 13 5
```