Tina Linux 配置
开发指南

版本号: 1.4
发布日期: 2021.04.20

版本历史

版本号	日期	制/修订人	内容描述
1.0	2019.03.01	AWA1046	整合多份文档，形成初始版本
1.1	2020.09.07	AWA1046	补充部分说明
1.2	2021.01.06	AWA1610	补充 dts 配置文件合并，覆盖规则
1.3	2021.04.09	AWA1046	补充 RISCV 路径，调整文档结构
1.4	2021.04.20	AWA1046	补充部分说明，修复格式

目　录

插　图

1 概述

1.1 编写目的

介绍 TinaLinux 的配置文件，配置方法。

1.2 适用范围

Tina V3.0 及其后续版本。

1.3 相关人员

适用于 TinaLinux 平台的客户及相关技术人员。

2 menuconfig

Tina 采用 Kconfig 机制，对 SDK 和内核进行配置。

具体用法，可以参考 Kconfig 机制的相关介绍。

2.1 tina menuconfig

Tina Linux SDK 的根目录下，执行 make menuconfig 命令可进入 Tina Linux 的配置界面。

对于具体软件包：

```
<*> (按y) ： 表示该软件包将包含在固件中。
<M>(按 m) ： 表示该软件将会被编译，但不会包含在固件中。
< >(按n) ： 表示该软件不会被编译。
```

配置文件保存在：

```
target/allwinner/${borad}/defconfig
```

make menuconfig 修改后的文件，会保存回上述配置文件。

2.2 kernel menuconfig

Tina Linux SDK 的根目录下，执行 make kernel_menuconfig 命令可进入对应内核的配置界面。

每个方案有对应的内核版本，如 3.4，3.10，4.4，4.9 等，记为 x.y。

对于 Tina3.5.0 及之前版本，配置后文件会保存在：

```
target/allwinner/${borad}/config-x.y
```

对于 Tina3.5.1 及之后版本，配置后文件会保存在：

```
device/config/chips/${chip}/configs/${borad}/linux
```

3 sysconfig

3.1 说明

3.1.1 文档说明

- 描述 GPIO 配置的形式：Port: 端口 + 组内序号。
- 文中的 <X>=0,1,2,3,4,5.....，如 twi0，twi1....；uart0，uart1....。
- 部分模块的配置项目可能是多余的，同时配置举例仅供参考，不一定为真实可用的，实际使用时需向技术支持人员询问。
- 跟模块说明文档冲突的，以模块文档为准。

3.1.2 配置文件路径

在方案的 configs 目录下，可用 cconfig 命令跳转过去。

Tina 3.5.0 及之前版本，路径为：

```
/target/allwinner/${borad}/configs/sys_config.fex
```

Tina 3.5.1 及之后版本，路径为：

```
device/config/chips/${chip}/configs/${borad}/sys_config.fex
```

3.2 linux-5.4 方案

像以往其他方案 (如 linux-4.9,linux-4.4 的)，会在 pack 阶段解析并将 sys_config 合并到 dtb 中，而 linux-5.4 使用的是原生未改动的 dtc 工具，没法解析 sys_config，所有内核用到的配置肯定都得在 board.dts 中设定好。目前方案目录中仍保存着 sys_config.fex 文件 (device/config/chips/r528/configs/evb1/sys_config.fex)，它的作用主要是：打包阶段根据 sys_config 配置更新 boot0, uboot, optee 等 bin 文件的头部等信息，例如更新 dram 参数、uart 参数等

3.3 系统

3.3.1 [product]

配置项	配置项含义
version	配置的版本号
machine	方案名字

示例：

```
[product]
version     = "100"
machine     = "m2ultra"
```

3.3.2 [platform]

配置项	配置项含义
eraseflag	量产时是否擦除。0：不擦，1：擦除（仅仅对量产工具，升级工具无效），0x11：强制擦除 (包括 private 分区) 0x12：强制擦除 (擦除 private 分区及 secure storage)
next_work	PhoenixUSBPro 量产完成后：1-不做任何动作，2-重启，3-关机，4-量产，5-正常启动，6-量产结束进入关机关机充电
debug_mode	Uboot 阶段打印等级：0-不打印，1-打印

示例：

```
[platform]
eraseflag  = 1
debug_mode = 1
```

3.3.3 [target]

配置项	配置项含义
boot_clock	启动频率; xx 表示多少 MHz
storage_type	启动介质选择 0:nand 1:sd 2:emmc 3:spinor 4:emmc3 5:spinand 6:sd1 -1:(defualt) 自动扫描启动介质

配置项	配置项含义
burn_key	支持 DragonSN_V2.0 烧录 sn 号

注，目前 nor 和其他介质不兼容，若为 nor，请配置为 3. 否则请配置为对应的介质或-1。

示例：

```
[target]
boot_clock   = 1008
storage_type = -1
burn_key     = 1
```

3.3.4 [power_sply]

配置项配	置项含义
dcdc<X>_vol d	cdc<X> 模块输出电压
aldo<X>_vol a	ldo<X> 模块输出电压
dc1sw_vol d	c1sw 模块输出电压
dc5ldo_vol d	c5ldo 模块输出电压
dldo<X>_vol d	ldo<X> 模块输出电压
gpio<X>_vol g	pio<X> 的输出电压

示例：

```
[power_sply]
dcdc1_vol    = 1003300
dcdc2_vol    = 1001160
dcdc3_vol    = 1001100
dcdc4_vol    = 1100
aldo1_vol    = 2800
aldo2_vol    = 1001500
aldo3_vol    = 1003000
dc1sw_vol    = 3000
dc5ldo_vol   = 1100
dldo1_vol    = 3300
dldo2_vol    = 3300
dldo3_vol    = 3300
dldo4_vol    = 2500
eldo1_vol    = 2800
eldo2_vol    = 1500
eldo3_vol    = 1200
gpio0_vol    = 3300
gpio1_vol    = 1800
```

补充说明：

电压名称 = 100XXXX： 表示把该路电压设置为 XXXX 指定的电压值，同时打开输出开关。

电压名称 = 000XXXX： 表示把该路电压设置为 XXXX 指定的电压值，同时关闭输出开关，当有需要时由内核驱动打开。

电压名称 = 0： 表示关闭该路电压输出开关，不修改原有的值。

这里的电压值单位为 mV。

3.3.5 [card_boot]

配置项	配置项含义
logical_start	启动卡逻辑起始扇区
sprite_gpio0	卡量产 gpio led 灯配置
next_work	卡量产完成后：1-不做任何动作，2-重启，3-关机，4-量产，5-正常启动

示例：

```
[card_boot]
logical_start    = 40960
sprite_gpio0     = port:PH21<1><default><default><default>
```

3.3.6 [card0_boot_para]

配置项	配置项含义
card_ctrl=0	卡量产相关的控制器选择 0
card_high_speed	速度模式 0 为低速，1 为高速
card_line	1，4，8 线卡可以选择，需看具体芯片是否支持
sdc_clk	sdc 卡时钟信号的 GPIO 配置
sdc_cmd	sdc 命令信号的 GPIO 配置
sdc_d<X>	sdc 卡数据 <X> 线信号的 GPIO 配置

示例：

```
[card0_boot_para]
card_ctrl       = 0
card_high_speed = 1
card_line       = 4
sdc_d1          = port:PF0<2><1><2><default>
sdc_d0          = port:PF1<2><1><2><default>
```

```
sdc_clk         = port:PF2<2><1><2><default>
sdc_cmd         = port:PF3<2><1><2><default>
sdc_d3          = port:PF4<2><1><2><default>
sdc_d2          = port:PF5<2><1><2><default>
```

3.3.7 [card2_boot_para]

配置项	配置项含义
card_ctrl=2	卡启动控制器选择 2
card_high_speed	速度模式 0 为低速，1 为高速
card_line	1，4，8 线卡可以选择，需看具体芯片是否支持
sdc_clk	sdc 卡时钟信号的 GPIO 配置
sdc_d<X>	sdc 卡数据 <X> 线信号的 GPIO 配置
sdc_emmc_rst	sdc 卡 rst 引脚
sdc_ex_dly_used	
sdc_io_1v8	

示例：

```
[card2_boot_para]
card_ctrl          = 2
card_high_speed    = 1
card_line          = 8
sdc_clk            = port:PC7<3><1><3><default>
sdc_cmd            = port:PC6<3><1><3><default>
sdc_d0             = port:PC8<3><1><3><default>
sdc_d1             = port:PC9<3><1><3><default>
sdc_d2             = port:PC10<3><1><3><default>
sdc_d3             = port:PC11<3><1><3><default>
sdc_d4             = port:PC12<3><1><3><default>
sdc_d5             = port:PC13<3><1><3><default>
sdc_d6             = port:PC14<3><1><3><default>
sdc_d7             = port:PC15<3><1><3><default>
sdc_emmc_rst       = port:PC24<3><1><3><default>
sdc_ds             = port:PC5<3><1><3><default>
sdc_ex_dly_used    = 2
;sdc_io_1v8        =
```

3.3.8 [twi_para]

配置项	配置项含义
twi_port	Boot 的 twi 控制器编号
twi_scl	Boot 的 twi 的时钟的 GPIO 配置
twi_sda	Boot 的 twi 的数据的 GPIO 配置

配置项	配置项含义
twi_regulator	上拉配置

示例：

```
[twi_para]
twi_port   = 0
twi_scl    = port:PB0<2><default><default><default>
twi_sda    = port:PB1<2><default><default><default>
```

3.3.9 [uart_para]

配置项	配置项含义
uart_debug_port	Boot 串口控制器编号
uart_debug_tx	Boot 串口发送的 GPIO 配置
uart_debug_rx	Boot 串口接收的 GPIO 配置
uart_regulator	上拉配置

示例：

```
[uart_para]
uart_debug_port = 0
uart_debug_tx   = port:PF02<3><1><default><default>
uart_debug_rx   = port:PF04<3><1><default><default>
```

3.3.10 [jtag_para]

配置项	配置项含义
jtag_enable	JTAG 使能
jtag_ms	测试模式选择输入 (TMS) 的 GPIO 配置
jtag_ck	测试时钟输入 (TMS) 的 GPIO 配置
jtag_do	测试数据输出 (TDO) 的 GPIO 配置
jtag_di	测试数据输入（TDI）的 GPIO 配置

示例：

```
[jtag_para]
jtag_enable = 1
```

```
jtag_ms     = port:PB14<3><default><default><default>
jtag_ck     = port:PB15<3><default><default><default>
jtag_do     = port:PB16<3><default><default><default>
jtag_di     = port:PB17<3><default><default><default>
```

3.3.11 [clock]

配置项	配置项含义
pll4	pll4 时钟频率 (MHz)
pll8	pll8 时钟频率 (MHz)
pll9	pll9 时钟频率 (MHz)
pll12	pll12 时钟频率 (MHz)

示例：

```
[clock]
pll4  = 297
pll8  = 297
pll9  = 384
pll12 = 297
```

3.3.12 [pm_para]

配置项	配置项含义
standby_mode	1：支持 super standby 0：支持 normal standby

示例：

```
[pm_para]
standby_mode = 1
```

3.4 DRAM

3.4.1 [dram_para]

配置项	配置项含义
dram_clk	DRAM 的时钟频率，单位为 MHz; 它为 24 的整数倍，最低不得低于 120
dram_type	DRAM 类型：2 为 DDR2，3 为 DDR3
dram_zq	DRAM 控制器内部参数，由原厂来进行调节，请勿修改
dram_odt_en	ODT 是否需要使能 0：不使能 1：使能，一般情况下，为了省电，此项为 0
dram_para1	DRAM 控制器内部参数，由原厂来进行调节，请勿修改
dram_para2	DRAM 控制器内部参数，由原厂来进行调节，请勿修改
dram_mr0	DRAM CAS 值，可为 6，7，8，9；具体需根据 DRAM 的规格书和速度来确定
dram_mr<X>	DRAM 控制器内部参数，由原厂来进行调节，请勿修改

示例：

```
[dram_para]
dram_clk    = 648
dram_type   = 7
dram_zq     = 0x3b3bfb
dram_odt_en = 0x31
dram_para1  = 0x10e410e4
dram_para2  = 0x1000
dram_mr0    = 0x1840
dram_mr1    = 0x40
dram_mr2    = 0x18
dram_mr3    = 0x2
dram_tpr0   = 0x0048A192
dram_tpr1   = 0x01b1a94b
dram_tpr2   = 0x00061043
dram_tpr3   = 0xB47D7D96
dram_tpr4   = 0x0000
dram_tpr5   = 0x198
dram_tpr6   = 0x21000000
dram_tpr7   = 0x2406C1E0
dram_tpr8   = 0x0
dram_tpr9   = 0
dram_tpr10  = 0x0008
dram_tpr11  = 0x44450000
dram_tpr12  = 0x9777
dram_tpr13  = 0x4090950
```

3.5 Ethernet MAC Controller

3.5.1 [gmac_para]

配置项	配置项含义
gmac_used	是否使用 Ethernet
gmac_txd<X>	发送数据 GPIO 配置
gmac_txclk	发送时钟信号
gmac_txen	发送使能信号
gmac_gtxclk	gtx 时钟信号
gmac_rxd<X>	接收数据 GPIO 配置
gmac_rxdv	接收有效指示
gmac_rxclk	接收时钟信号
gmac_txerr	接收出错指示
gmac_col	冲突检测
gmac_crs	crs GPIO 配置
gmac_clkin	clkin GPIO 配置
gmac_mdc	配置接口时钟
gmac_mdio	配置接口 I/O

示例：

```
gmac_used   = 0
gmac_txd0   = port:PA00<2><default><default><default>
gmac_txd1   = port:PA01<2><default><default><default>
gmac_txd2   = port:PA02<2><default><default><default>
gmac_txd3   = port:PA03<2><default><default><default>
gmac_txd4   = port:PA04<2><default><default><default>
gmac_txd5   = port:PA05<2><default><default><default>
gmac_txd6   = port:PA06<2><default><default><default>
gmac_txd7   = port:PA07<2><default><default><default>
gmac_txclk  = port:PA08<2><default><default><default>
gmac_txen   = port:PA09<2><default><default><default>
gmac_gtxclk = port:PA10<2><default><default><default>
gmac_rxd0   = port:PA11<2><default><default><default>
gmac_rxd1   = port:PA12<2><default><default><default>
gmac_rxd2   = port:PA13<2><default><default><default>
gmac_rxd3   = port:PA14<2><default><default><default>
gmac_rxd4   = port:PA15<2><default><default><default>
gmac_rxd5   = port:PA16<2><default><default><default>
gmac_rxd6   = port:PA17<2><default><default><default>
gmac_rxd7   = port:PA18<2><default><default><default>
gmac_rxdv   = port:PA19<2><default><default><default>
gmac_rxclk  = port:PA20<2><default><default><default>
gmac_txerr  = port:PA21<2><default><default><default>
gmac_rxerr  = port:PA22<2><default><default><default>
gmac_col    = port:PA23<2><default><default><default>
gmac_crs    = port:PA24<2><default><default><default>
gmac_clkin  = port:PA25<2><default><default><default>
gmac_mdc    = port:PA26<2><default><default><default>
gmac_mdio   = port:PA27<2><default><default><default>
```

3.6 I2C 总线

3.6.1 [twi<X>]

配置项	配置项含义
twiX_used	TWI 使用控制：1 使用，0 不用
twiX_scl	TWI SCK 的 GPIO 配置
twiX_sda	TWI SDA 的 GPIO 配置

示例：

```
[twi0]
twiX_used = 1
twiX_scl  = port:PB00<2><default><default><default>
twiX_sda  = port:PB01<2><default><default><default>
```

3.7 串口 (UART)

3.7.1 [uart<X>]

配置项	配置项含义
uart_used	UART 使用控制：1 使用，0 不用
uart_port	UART 端口号
uart_type	UART 类型，有效值为：2/4/8; 表示 2/4/8 线模式
uartX_tx	UART TX 的 GPIO 配置
uartX_rx	UART RX 的 GPIO 配置
uartX_rts	UART RTS 的 GPIO 配置
uartX_cts	UART CTS 的 GPIO 配置
uartX_dtr	UART DTR 的 GPIO 配置
uartX_dsr	UART DSR 的 GPIO 配置
uartX_dcd	UART DCD 的 GPIO 配置
uartX_ring	UART RING 的 GPIO 配置

示例：

```
[uart1]
uart1_used = 0
uart1_port = 1
```

```
uart1_type = 8
uart1_tx   = port:PA10<4><1><default><default>
uart1_rx   = port:PA11<4><1><default><default>
uart1_rts  = port:PA12<4><1><default><default>
uart1_cts  = port:PA13<4><1><default><default>
uart1_dtr  = port:PA14<4><1><default><default>
uart1_dsr  = port:PA15<4><1><default><default>
uart1_dcd  = port:PA16<4><1><default><default>
uart1_ring = port:PA17<4><1><default><default>
```

3.8 SPI 总线

3.8.1 [spi<X>]

配置项	配置项含义
spiX_used	SPI 使用控制：1 使用，0 不用
spiX_cs_number	spiX 片选个数，最多 2 个
spiX_cs_bitmap	由于 SPI 控制器支持多个 CS，这一个参数表示 CS 的掩码
spiX_cs0	SPI CS0 的 GPIO 配置
spiX_cs1	SPI CS1 的 GPIO 配置
spiX_sclk	SPI CLK 的 GPIO 配置
spiX_mosi	SPI MOSI 的 GPIO 配置
spiX_miso	SPI MISO 的 GPIO 配置

示例：

```
[spi0]
spi0_used      = 0
spi0_cs_number = 2
spi0_cs_bitmap = 3
spi0_cs0       = port:PC23<3><1><default><default>
spi0_cs1       = port:PI14<2><1><default><default>
spi0_sclk      = port:PC2<3><default><default><default>
spi0_mosi      = port:PC0<3><default><default><default>
spi0_miso      = port:PC1<3><default><default><default>
```

3.8.2 [spiX/spi_boardX]

配置项	配置项含义
compatible	设备名称
spi-max-frequency	工作最大频率
reg	片选

配置项	配置项含义
spi-cpha	时钟相位
spi-cpol	时钟极性
spi-cs-high	默认 0，为 1 表示 flash 的片选为 high active

示例：

```
[spi0/spi_board0]
compatible        = "m25p80"
spi-max-frequency = 1000000
reg               = 0
;spi-cpha
;spi-cpol
;spi-cs-high
```

3.9 gpadc

3.9.1 [gpadc]

配置项	配置项含义
gpadc_used	whether use gpadc or not
channel_num	maxinum number of channels supported on the platform.
channel_select	channel enable setection. channel0:0x01 channel1:0x02 channel2:0x04 channel3:0x08
channel_data_select	channel data enable. channel0:0x01 channel1:0x02 channel2:0x04 channel3:0x08.
channel_compare_select	compare function enable channel0:0x01 channel1:0x02 channel2:0x04 channel3:0x08.
channel_cld_select	compare function low data enable setection: channel0:0x01 channel1:0x02 channel2:0x04 channel3:0x08.
channel_chd_select	compare function hig data enable setection: channel0:0x01 channel1:0x02 channel2:0x04 channel3:0x08.

示例：

```
[gpadc]
gpadc_used               = 1
channel_num              = 1
channel_select           = 0x01
channel_data_select      = 0
channel_compare_select   = 0x01
channel_cld_select       = 0x01
channel_chd_select       = 0
channel0_compare_lowdata = 1700000
channel0_compare_higdata = 1200000
key_cnt                  = 5
key0_vol                 = 115
key0_val                 = 115
key1_vol                 = 240
key1_val                 = 114
key2_vol                 = 360
key2_val                 = 139
key3_vol                 = 480
key3_val                 = 28
key4_vol                 = 600
key4_val                 = 102
```

3.10 触摸屏配置

3.10.1 [rtp_para]

配置项	配置项含义
rtp_used	该模块在方案中是否启用
rtp_screen_size	屏幕尺寸设置，以斜对角方向长度为准，以寸为单位
rtp_regidity_level	表屏幕的硬度，以指覆按压，抬起时开始计时，多少个 10ms 时间单位之后，硬件采集不到数据为准; 通常，我们建议的屏，5 寸屏设为 5，7 寸屏设为 7，对于某些供应商提供的屏，硬度可能不合要求，需要适度调整
rtp_press_threshold_enable	是否开启压力的门限制，建议选 0 不开启
rtp_press_threshold	这配置项当 rtp_press_threshold_enable 为 1 时才有效，其数值可以是 0 到 0xFFFFFF 的任意数值，数值越小越敏感，推荐值为 0xF
rtp_sensitive_level	敏感等级，数值可以是 0 到 0xF 之间的任意数值，数值越大越敏感，0xF 为推荐值
rtp_exchange_x_y_flag	当屏的 x，y 轴需要转换的时候，这个项目该置 1，一般情况下则该置 0

示例：

```
[rtp_para]
rtp_used                    = 0
rtp_screen_size             = 5
rtp_regidity_level          = 5
rtp_press_threshold_enable = 0
rtp_press_threshold         = 0x1f40
rtp_sensitive_level         = 0xf
rtp_exchange_x_y_flag       = 0
```

3.10.2 [ctp]

配置项	配置项含义
ctp_used	该选项为是否开启电容触摸，支持的话置 1，反之置 0
ctp_name	tp 的 name，必须配，与驱动保持一致
ctp_twi_id	用于选择 i2c adapter，可选 1，2
ctp_twi_addr	指明 i2c 设备地址，与具体硬件相关
ctp_screen_max_x	触摸板的 x 轴最大坐标
ctp_screen_max_y	触摸板的 y 轴最大坐标
ctp_revert_x_flag	是否需要翻转 x 坐标，需要则置 1，反之置 0
ctp_revert_y_flag	是否需要翻转 y 坐标，需要则置 1，反之置 0
ctp_exchange_x_y_flag	是否需要 x 轴 y 轴坐标对换
ctp_int_port	电容屏中断信号的 GPIO 配置
ctp_wakeup	电容屏唤醒信号的 GPIO 配置
ctp_power_ldo	电容屏供电 ldo
ctp_power_ldo_vol	电容屏供电 ldo 电压
ctp_power_io	当电容屏供电 gpio

示例：

```
[ctp]
ctp_used              = 1
ctp_twi_id            = 1
ctp_twi_addr          = 0x5d
ctp_screen_max_x      = 1280
ctp_screen_max_y      = 800
ctp_revert_x_flag     = 1
ctp_revert_y_flag     = 1
ctp_exchange_x_y_flag = 1
ctp_int_port          = port:PI10<6><default><default><default>
ctp_wakeup            = port:PH10<1><default><default><1>
ctp_power_ldo         = "vcc-ctp"
ctp_power_ldo_vol     = 3300
ctp_power_io          =
```

3.10.3 [acc_gpio]

配置项	配置项含义
compatible	设备名字
acc_gpio_used	该选项是否开启，1：开启，0：关闭
acc_int	acc gpio 配置引脚，用作判断是否需要进入睡眠

示例：

```
[acc_gpio]
compatible    = "allwinner,sunxi-acc-det"
acc_gpio_used = 1
acc_int       = port:power0<6><default><default><default>
```

3.10.4 [ctp_list]

配置项	配置项含义
ctp_det_used	支持触摸屏 list
ft5x_ts	是否支持 ft5x_ts 模组
gt82x	是否支持 gt82x 模组
gslX680	是否支持 gslX680 模组
gt9xx_ts	是否支持 gt9xx_ts 模组
gt9xxnew_ts	是否支持 gt9xxnew_ts 模组
gt811	是否支持 gt811 模组
zet622x	是否支持 zet622x 模组
aw5306_ts	是否支持 d5306_ts 模组
ctp_det_used	支持触摸屏 list
tu_ts	
gt818ts	
icn83xx_ts	

示例：

```
[ctp_list]
compatible   = "allwinner,sun50i-ctp-list"
ctp_det_used = 1
ft5x_ts      = 1
gt82x        = 1
gslX680      = 0
gslX680new   = 1
gt9xx_ts     = 1
gt9xxf_ts    = 0
```

```
tu_ts        = 0
gt818_ts     = 0
zet622x      = 0
aw5306_ts    = 0
icn83xx_ts   = 0
```

3.11 触摸按键

3.11.1 [tkey_para]

配置项	配置项含义
tkey_used	支持触摸按键的置 1，反之置 0
tkey_twi_id	用于选择 i2c adapter，可选 1，2
tkey_twi_addr	指明 i2c 设备地址，与具体硬件相关
tkey_int	触摸按键中断信号的 GPIO 配置

示例：

```
[tkey_para]
tkey_used     = 0
tkey_twi_id   =
tkey_twi_addr =
tkey_int      =
```

3.12 马达

3.12.1 [motor_para]

配置项	配置项含义
motor_used	是否启用马达，启用置 1，反之置 0
motor_shake	马达使用的 GPIO 配置

示例：

```
[motor_para]
motor_used   = 0
motor_shake  = port:power3<1><default><default><1>
```

注意事项：

motor_shake = port:power3<1>

<1>

默认 io 口的输出应该为 1，这样就不会初始化之后就开始震动了。

假设 motor_shake = 0 ，说明没有指定 gpio 引脚，那么就会设置 axp 的引脚为马达供电，优先考虑 gpio 配置。

3.13 闪存

3.13.1 [nand<X>_para]

配置项	配置项含义
nand_support_2ch	nand0 是否使能双通道
nand0_used	nand0 模块使能标志
nand0_we	nand0 写时钟信号的 GPIO 配置
nand0_ale	nand0 地址使能信号的 GPIO 配置
nand0_cle	nand0 命令使能信号的 GPIO 配置
nand0_ce1	nand0 片选 1 信号的 GPIO 配置
nand0_ce0	nand0 片选 0 信号的 GPIO 配置
nand0_nre	nand0 读时钟信号的 GPIO 配置
nand0_rb0	nand0 Read/Busy 1 信号的 GPIO 配置
nand0_rb1	nand0 Read/Busy 0 信号的 GPIO 配置
nand0_d[X]	nand0 数据总线信号的 GPIO 配置，[X]=0，1，2...
nand0_nwp	
nand0_ce[X]	nand0 片选 [X] 信号的 GPIO 配置，[X]=0，1，2...
nand0_ndqs	
nand0_regulator1	
nand0_regulator2	
nand0_cache_level	
nand0_flush_cache_num	
nand0_capacity_level	
nand0_id_number_ctl	
nand0_print_level	
nand0_p0	
nand0_p1	
nand0_p2	
nand0_p3	

示例：

```
[nand0_para]
nand0_support_2ch     = 0
nand0_used            = 1
nand0_we              = port:PC00<2><0><1><default>
nand0_ale             = port:PC01<2><0><1><default>
nand0_cle             = port:PC02<2><0><1><default>
nand0_ce1             = port:PC03<2><1><1><default>
nand0_ce0             = port:PC04<2><1><1><default>
nand0_nre             = port:PC05<2><0><1><default>
nand0_rb0             = port:PC06<2><1><1><default>
nand0_rb1             = port:PC07<2><1><1><default>
nand0_d0              = port:PC08<2><0><1><default>
nand0_d1              = port:PC09<2><0><1><default>
nand0_d2              = port:PC10<2><0><1><default>
nand0_d3              = port:PC11<2><0><1><default>
nand0_d4              = port:PC12<2><0><1><default>
nand0_d5              = port:PC13<2><0><1><default>
nand0_d6              = port:PC14<2><0><1><default>
nand0_d7              = port:PC15<2><0><1><default>
nand0_nwp             = port:PC16<2><1><1><default>
nand0_ce2             = port:PC17<2><1><1><default>
nand0_ce3             = port:PC18<2><1><1><default>
nand0_ce4             = port:PC19<2><1><1><default>
nand0_ce5             = port:PC20<2><1><1><default>
nand0_ce6             = port:PC21<2><1><1><default>
nand0_ce7             = port:PC22<2><1><1><default>
nand0_ndqs            = port:PC24<2><0><1><default>
nand0_regulator1      = "vcc-nand"
nand0_regulator2      = "none"
nand0_cache_level     = 0x55aaaa55
nand0_flush_cache_num = 0x55aaaa55
nand0_capacity_level  = 0x55aaaa55
nand0_id_number_ctl   = 0x55aaaa55
nand0_print_level     = 0x55aaaa55
nand0_p0              = 0x55aaaa55
nand0_p1              = 0x55aaaa55
nand0_p2              = 0x55aaaa55
nand0_p3              = 0x55aaaa55
```

3.14 显示

3.14.1 [boot_disp]

配置项	配置项含义
output_disp	支持显示用户自定义 bootlogo
output_type	1:LCD 2:TV 3:HDMI 4:VGA
output_mode	(用于 tv/hdmi 输出，0:480i，1:576i，2:480p，3:576p 4:720p50，5:720p60，6:1080i50，7:1080i60，8:1080p24，9:1080p5，10:1080p60，11:pal 14:ntsc)

3.14.2 [disp]

配置项	配置项含义
disp_init_enable	是否进行显示的初始化设置
disp_mode	显示模式：0:screen0<screen0,fb0> 1:screen1<screen1,fb0>
screen<X>_output_type	屏 0 输出类型 (0:none; 1:lcd; 2:tv; 3:hdmi; 4:vga)
screen<X>_output_mode	屏 0 输出模式 (用于 tv/hdmi 输出，0:480i 1:576i 2:480p 3:576p 4:720p50 5:720p60 6:1080i50 7:1080i60 8:1080p24 9:1080p50 10:1080p60 11:pal 14:ntsc)
screen<X>_output_format	0:RGB 1:yuv444 2:yuv422 3:yuv420
screen<X>_output_bits	0:8bit 1:10bit 2:12bit 2:16bit
screen<X>_output_eotf	0:reserve 4:SDR 16:HDR10 18:HLG
screen<X>_output_cs	0:undefined 257:BT709 260:BT601 263:BT2020
fb<X>_format	fb<X> 的格式 (0:ARGB 1:ABGR 2:RGBA 3:BGRA)
fb<X>_width	fb<X> 的宽度，为 0 时将按照输出设备的分辨率
fb<X>_height	fb<X> 的高度，为 0 时将按照输出设备的分辨率
lcd<X>_backlight	lcd<X> 的背光初始值，0~55
lcd<X>_bright	lcd<X> 的亮度值，0~100
lcd<X>_contrast	lcd<X> 的对比度，0~100
lcd<X>_saturation	lcd<X> 的饱和度，0~100
lcd<X>_hue	lcd<X> 的色度，0~100

示例：

```
[disp]
disp_init_enable     = 1
disp_mode            = 0

screen0_output_type = 1
screen0_output_mode = 5

screen1_output_type = 3
screen1_output_mode = 4

fb0_format           = 0
fb0_width            = 0
fb0_height           = 0

fb1_format           = 0
fb1_width            = 0
fb1_height           = 0

lcd0_backlight       = 50
lcd1_backlight       = 50
```

```
lcd0_bright         = 50
lcd0_contrast       = 50
lcd0_saturation     = 57
lcd0_hue            = 50

lcd1_bright         = 50
lcd1_contrast       = 50
lcd1_saturation     = 57
lcd1_hue            = 50
```

3.14.3 [edp<X>]

配置项	配置项含义
used	whether use edp0 or not
edp_io_power	power of edp controller
edp_x	width in panel’s resolution
edp_y	height in panel’s resolution
edp_hbp	horizon back porch(pixel)
edp_ht	horizon totoal(pixel)
edp_hspw	horizon sync pulse width(pixel)
edp_vbp	vertical back porch(line)
edp_vt	vertical totoal (line)
edp_vspw	vertical sync pulse width(line)
edp_rate	(0:1.62 Gbps, 1:2.7 Gbps, 2:5.4 Gbps)
edp_lane	number of lanes of panel
edp_fps	frame per second of panel
edp_colordepth	color depth of panel.(0:8 bits, 1:6 bits)

示例：

```
[edp0]
used=1
edp_io_power = "vcc-edp"
edp_x=2048
edp_y=1536
edp_hbp=10
edp_ht=2208
edp_hspw=5
edp_vbp=10
edp_vt=1570
edp_vspw=1
edp_rate=0
edp_lane=4
edp_fps=60
edp_colordepth=0
```

3.14.4 [lcd<X>_suspend]

配置项	配置项含义
lcdd<X>	lcd 数据 <X> 线信号休眠状态下的 GPIO 配置

示例：

```
[lcd0_suspend]
;lcdd0 = port:PD00<7><0><default><default>
;lcdd1 = port:PD01<7><0><default><default>
;lcdd2 = port:PD02<7><0><default><default>
;lcdd3 = port:PD03<7><0><default><default>
;lcdd4 = port:PD04<7><0><default><default>
;lcdd5 = port:PD05<7><0><default><default>
;lcdd6 = port:PD06<7><0><default><default>
;lcdd7 = port:PD07<7><0><default><default>
;lcdd8 = port:PD08<7><0><default><default>
;lcdd9 = port:PD09<7><0><default><default>
```

3.14.5 [car_reverse]

配置项	配置项含义
compatible	匹配设备的 token
used	模块使用配置项
tvd_id	倒车模块使用的 tvd 通道
screen_width	倒车预览图像宽度
screen_height	倒车预览图像高度
rotation	是否使能旋转
reverse_pin	倒车信号输入管脚

示例：

```
[car_reverse]
compatible    = "allwinner, sunxi-car-reverse"
used          = 1
tvd_id        = 0
screen_width  = 720
screen_height = 480
rotation      = 1
reverse_pin   = port:PH20<6><0><default><default>
```

3.14.6 [lcd<X>]

配置项	配置项含义
lcd_used	是否使用 lcd0
lcd_driver_name	定义驱动名称
lcd_bl_0_percent	
lcd_bl_40_percent	
lcd_bl_100_percent	
cd_backlight	LCD 背光值
lcd_if	lcd 接口 (0:hv(sync+de); 1:8080; 2:ttl; 3:lvds，4:dsi; 5:edp)
lcd_x	lcd 分辨率 x
lcd_y	lcd 分辨率 y
lcd_width	lcd 屏宽度
lcd_height	lcd 屏高度
lcd_dclk_freq	lcd 频率
lcd_pwm_used	pwm 是否使用
lcd_pwm_ch	pwm 通道
lcd_pwm_freq	pwm 频率
lcd_pwm_pol	pwm 属性，0:positive; 1:negative
lcd_pwm_max_limit	pwm 最大值
lcd_hbp	lcd 行后沿时间
lcd_ht	lcd 行时间
lcd_hspw	lcd 行同步脉宽
lcd_vbp	lcd 场后沿时间
lcd_vt	lcd 场时间
lcd_vspw	lcd 场同步脉宽
lcd_dsi_if	
lcd_dsi_lane	
lcd_dsi_format	
lcd_dsi_te	
lcd_dsi_eotp	
lcd_lvds_if	lcd lvds 接口，0:single link; 1:dual link
lcd_lvds_colordepth	lcd lvds 颜色深度 0:8bit; 1:6bit
lcd_lvds_mode	lcd lvds 模式，0:NS mode; 1:JEIDA mode
lcd_frm	lcd 格式，0:disable; 1:enable rgb666 dither; 2:enable rgb656 dither
lcd_io_phase	
lcd_hv_clk_phase	lcd hv 时钟相位 0:0 degree; 1:90 degree; 2: 180 degree; 3: 270 degree
lcd_hv_sync_polarity	lcd io 属性，0:not invert; 1:invert
lcd_gamma_en	lcdgamma 校正使能

配置项	配置项含义
lcd_bright_curve_en	lcd 亮度曲线校正使能
lcd_cmap_en	lcd 调色板函数使能
deu_mode	deu 模式 0:smoll lcd screen; 1:large lcd screen(larger than 10inch)
lcdgamma4iep	使能背光参数，lcd gamma vale*10;decrease it while lcd is not bright enough; increase while lcd is too bright
lcd_dsi_port_num	
lcd_tcon_mode	
lcd_slave_stop_pos	
lcd_sync_pixel_num	
lcd_sync_line_num	
smart_color	丽色系统，90:normal lcd screen 65:retina lcd screen(9.7inch)
lcd_bl_en	背光使能的 GPIO 配置
lcd_power	lcd 电源
lcd_gpio_<X>	lcd 数据 <X> 线信号的 GPIO 配置

示例：

```
[lcd0]
lcd_used            = 1

lcd_driver_name     = "S070WV20_MIPI_RGB"
lcd_backlight       = 50
lcd_if              = 4
lcd_x               = 800
lcd_y               = 480
lcd_width           = 86
lcd_height          = 154
lcd_dclk_freq       = 20
lcd_pwm_used        = 1
lcd_pwm_ch          = 0
lcd_pwm_freq        = 50000
lcd_pwm_pol         = 1
lcd_pwm_max_limit = 255
lcd_hbp             = 88
lcd_ht              = 928
lcd_hspw            = 48
lcd_vbp             = 32
lcd_vt              = 525
lcd_vspw            = 3

lcd_frm             = 0
lcd_cmap_en         = 0
lcd_dsi_if          = 0
lcd_dsi_lane        = 4
lcd_dsi_format      = 0
lcd_dsi_te          = 0

deu_mode            = 0
```

```
lcdgamma4iep      = 22
smart_color       = 90

lcd_bl_en         = port:PH16<1><0><2><1>
lcd_power         = "vcc-3v"
;lcd_power        = "vcc-mipi"
lcd_gpio_0        = port:PH17<1><0><2><1>
lcd_gpio_1        = port:PH18<1><0><2><1>
```

3.15 PWM

3.15.1 [pwm<X>]

配置项	配置项含义
pwm_used	是否使用 PWM0
pwm_positive	PWM 输出 GPIO 配置

示例：

```
[pwm0]
pwm_used     = 1
pwm_positive = port:PB2<3><0><default><default>
```

3.15.2 [pwm<X>_suspend]

配置项	配置项含义
pwm<X>_suspend	pwm suspend

示例：

```
pwm_positive = port:PB2<3><0><default><default>
```

3.15.3 [spwm<X>]

配置项	配置项含义
s_pwm<X>_used	是否使用 s_pwm<X>
pwm_positive PWM	输出 GPIO 配置

示例：

```
pwm_positive = port:PL16<2><0><default><default>
```

3.15.4 [spwm<X>_suspend]

配置项	配置项含义
s_pwm<X>_suspend	s_pwm<X> suspend

3.16 HDMI

3.16.1 [hdmi]

配置项	配置项含义
hdmi_used	是否使用 hdmi。1：使用;0：不使用
hdmi_hdcp_enable	是否使能 hdcp
hdmi_cts_compatibility	cts 兼容性使能设置
hdmi_power	内核阶段 hdmi 电源配置

示例：

```
[hdmi]
hdmi_used               = 1
hdmi_hdcp_enable        = 0
hdmi_cts_compatibility = 0
```

3.17 tvd 摄像头

3.17.1 [tvd]

配置项	配置项含义
tvd_used	是否使用 TVD。1：使用;0：不使用
tvd_if	tvd interface 0:CVBS ，1:YPBPRI ，2: YPBPRP
fliter_used	使能 3D 滤波功能，设置为 1
cagc_enable	使能 cagc 功能，设置为 1

配置项	配置项含义
agc_auto_enable	使能 agc 功能，设置为 1
tvd_power0	AXP power，具体参考原理图配置
tvd_hot_plug	支持 TVD 动态插拔功能，1 to enable hot plug function ，0 to disable ，default disable
tvd_gpio0	gpio control power output or not

示例：

```
[tvd0]
tvd_used        = 1
tvd_if          = 0
fliter_used     = 1
cagc_enable     = 1
agc_auto_enable = 1
```

3.18 vind 摄像头

3.18.1 [vind<X>]

配置项	配置项含义
vind0_used	Vin 框架使能配置

示例：

```
[tvd0]
tvd_used        = 1
tvd_if          = 0
fliter_used     = 1
cagc_enable     = 1
agc_auto_enable = 1
```

3.18.2 [vind<X>/csi<X>]

配置项	配置项含义
csi0_used	vin 框架对应的 csi 使能配置
csi0_pck	csi pclock 时钟 GPIO 配置
csi0_hsync	hsync 信号 GPIO 配置
csi0_vsync	vsync 信号 GPIO 配置

配置项	配置项含义
csi0_d<X>	csi 数据引脚 GPIO 配置

示例：

```
[vind0/csi0]
csi0_used    = 1
csi0_pck     = port:PE00<2><default><default><default>
csi0_hsync   = port:PE02<2><default><default><default>
csi0_vsync   = port:PE03<2><default><default><default>
csi0_d0      = port:PE04<2><default><default><default>
csi0_d1      = port:PE05<2><default><default><default>
csi0_d2      = port:PE06<2><default><default><default>
csi0_d3      = port:PE07<2><default><default><default>
csi0_d4      = port:PE08<2><default><default><default>
csi0_d5      = port:PE09<2><default><default><default>
csi0_d6      = port:PE10<2><default><default><default>
csi0_d7      = port:PE11<2><default><default><default>
```

3.18.3 [vind<X>/csi_cci<X>]

配置项	配置项含义
csi_cci0_used	csi 的 cci 使能配置
csi_cci0_sck	cci 的 i2c 通信 sck GPIO 配置
csi_cci0_sda	cci 的 i2c 通信 sda GPIO 配置

示例：

```
[vind0/csi_cci0]
csi_cci0_used = 1
csi_cci0_sck  = port:PE12<2><default><default><default>
csi_cci0_sda  = port:PE13<2><default><default><default>`
```

3.18.4 [vind<X>/flash<X>]

配置项	配置项含义
flash0_used	vin 框架对应的闪光灯使能配置
flash0_type	闪光灯的类型
flash0_en	闪光灯使能
flash0_mode	闪光灯工作模式
flash0_flvdd	闪光灯电压配置

配置项	配置项含义
flash0_flvdd_vol	闪光灯电压值

示例：

```
[vind0/flash0]
flash0_used      = 1
flash0_type      = 2
flash0_en        =
flash0_mode      =
flash0_flvdd     = ""
flash0_flvdd_vol =
```

3.18.5 [vind<X>/actuator<X>]

配置项	配置项含义
actuator0_used	对焦马达使能配置
actuator0_name	对焦马达名称
actuator0_slave	对焦马达的 i2c 地址
actuator0_af_pwdn	对焦马达的 pwm 控制
actuator0_afvdd	对焦马达电压配置
actuator0_afvdd_vol	对焦马达电压配置值

示例：

```
[vind0/actuator0]
actuator0_used     = 0
actuator0_name     = "ad5820_act"
actuator0_slave    = 0x18
actuator0_af_pwdn  =
actuator0_afvdd    = "afvcc-csi"
actuator0_afvdd_vol = 2800000
```

3.18.6 [vind<X>/sensor<X>]

配置项	配置项含义
sensor1_used	sensor 使能控制
sensor1_mname	sensor 名称，需要和驱动文件的对应
sensor1_twi_cci_id	sensor 通信使用的 twi 索引
sensor1_twi_addr	sensor 的 i2c 地址

配置项	配置项含义
sensor1_pos	sensor 索引
sensor1_isp_used	sensor 的 ISP 使能
sensor1_fmt	sensor 的数据格式
sensor1_stby_mode	sensor 的 stby 模式选择
sensor1_vflip	sensor 垂直镜像使能配置
sensor1_hflip	sensor 水平镜像使能配置
sensor1_iovdd	sensor io 电压配置
sensor1_iovdd_vol	sensor io 电压值
sensor1_avdd	sensor avdd 电压配置
sensor1_avdd_vol	sensor avdd 电压值
sensor1_dvdd	sensor dvdd 电压配置
sensor1_dvdd_vol	sensor dvdd 电压值
sensor1_power_en	sensor 电源使能
sensor1_reset	sensor reset GPIO 配置
sensor1_pwdn	sensor pwdn GPIO 配置

示例：

```
sensor1_used       = 1
sensor1_mname      = "ov5647"
sensor1_twi_cci_id = 0
sensor1_twi_addr   = 0x6c
sensor1_pos        = "front"
sensor1_isp_used   = 0
sensor1_fmt        = 0
sensor1_stby_mode  = 1
sensor1_vflip      = 0
sensor1_hflip      = 0
sensor1_iovdd      = "iovdd-csi"
sensor1_iovdd_vol  = 2800000
sensor1_avdd       = "avdd-csi"
sensor1_avdd_vol   = 2800000
sensor1_dvdd       = "dvdd-csi"
sensor1_dvdd_vol   = 1800000
sensor1_power_en   =
sensor1_reset      = port:PE16<0><0><1><0>
sensor1_pwdn       = port:PE17<0><0><1><0>
```

3.18.7 [vind<X>/vinc<X>]

配置项	配置项含义
vinc<X>_used	vin core 使能配置
vinc<X>_csi_sel	vin core 对应的 csi 索引
vinc<X>_mipi_sel	vin core 对应的 mipi 索引

配置项	配置项含义
vinc<X>_isp_sel	vin core 对应的 isp 索引
vinc<X>_rear_sensor_sel	vin core 对应的 rear sensor 索引
vinc<X>_front_sensor_sel	vin core 对应的 front sensor 索引
vinc<X>_sensor_list	vin core 对应的 sensor 列表

示例：

```
[vind0/vinc1]
vinc1_used             = 1
vinc1_csi_sel          = 0
vinc1_mipi_sel         = 0xff
vinc1_isp_sel          = 0
vinc1_rear_sensor_sel  = 0
vinc1_front_sensor_sel = 1
vinc1_sensor_list      = 0
```

3.19 摄像头 (CSI)

3.19.1 [csi<X>]

配置项	配置项含义
csi<X>_used	摄像头使能配置
csi<X>_sensor_list	
csi<X>_pck	pclk 信号的 GPIO 配置
csi<X>_mck	mclk 信号的 GPIO 配置
csi<X>_hsync	hsync 信号的 GPIO 配置
csi<X>_vsync	vsync 信号的 GPIO 配置
csi<X>_d<X>	csi d<X> 信号的 GPIO 配置

示例：

```
[csi0]
csi0_used        = 1
csi0_sensor_list = 0
csi0_pck         = port:PE00<3><default><default><default>
csi0_mck         = port:PE01<1><0><1><0>
csi0_hsync       = port:PE02<3><default><default><default>
csi0_vsync       = port:PE03<3><default><default><default>
csi0_d0          = port:PE04<3><default><default><default>
csi0_d1          = port:PE05<3><default><default><default>
csi0_d2          = port:PE06<3><default><default><default>
csi0_d3          = port:PE07<3><default><default><default>
```

```
csi0_d4          = port:PE08<3><default><default><default>
csi0_d5          = port:PE09<3><default><default><default>
csi0_d6          = port:PE10<3><default><default><default>
csi0_d7          = port:PE11<3><default><default><default>
```

3.19.2 [csi<X>/csi0_dev0]

配置项	配置项含义
csi<X>_dev0_used	是否使用 csi0_dev0
csi<X>_dev0_mname	设置 sensor 0 名称
csi<X>_dev0_twi_addr	请参考实际原理图填写
csi<X>_dev0_twi_id	请参考实际模组的 8bit ID 填写
csi<X>_dev0_pos	摄像头位置前置填 “front”，后置填 “rear”
csi<X>_dev0_isp_used	YUV 填 0
csi<X>_dev0_fmt	YUV 填 0
csi<X>_dev0_stby_mode	填 0
csi<X>_dev0_vflip	Sensor 图像垂直翻转
csi<X>_dev0_hflip	Sensor 图像水平翻转
csi<X>_dev0_iovdd	IOVDD 配置，请参考实际原理图填写
csi<X>_dev0_iovdd_vol	IOVDD 电压值一般为 2.8V(2800000)
csi<X>_dev0_avdd	AVDD 配置，如”csi-avdd”
csi<X>_dev0_avdd_vol	AVDD 电压值，一般为 2.8V(2800000)
csi<X>_dev0_dvdd	DVDD 配置，如 “csi-dvdd”
csi<X>_dev0_dvdd_vol	DVDD 电压值参考 datasheet，1.2/1.5/1.8V
csi<X>_dev0_afvdd	Isp-dvdd 配置，如 isp-dvdd12
csi<X>_dev0_afvdd_vol	电压值为 1.2V
csi<X>_dev0_power_en	Sensor power enable 引脚 GPIO 配置
csi<X>_dev0_reset	Sensor reset 引脚 GPIO 配置
csi<X>_dev0_pwdn	Sensor power down 引脚 GPIO 配置
csi<X>_dev0_flash_used	填 0
csi<X>_dev0_flash_type	填 0
csi<X>_dev0_flash_en	不需填写
csi<X>_dev0_flash_mode	不需填写
csi<X>_dev0_flvdd	不需填写
csi<X>_dev0_flvdd_vol	不需填写
csi<X>_dev0_af_pwdn	不需填写
csi<X>_dev0_act_used	不需填写
csi<X>_dev0_act_name	不需填写
csi<X>_dev0_act_slave	不需填写

示例：

```
[csi0/csi0_dev0]
csi0_dev0_used       = 1
csi0_dev0_mname      = "ov5640"
csi0_dev0_twi_addr   = 0x78
csi0_dev0_twi_id     = 4
csi0_dev0_pos        = "rear"
csi0_dev0_isp_used   = 0
csi0_dev0_fmt        = 0
csi0_dev0_stby_mode  = 0
csi0_dev0_vflip      = 0
csi0_dev0_hflip      = 0
csi0_dev0_iovdd      = "csi-iovcc"
csi0_dev0_iovdd_vol  = 2800000
csi0_dev0_avdd       = "csi-avdd"
csi0_dev0_avdd_vol   = 2800000
csi0_dev0_dvdd       = "csi-dvdd"
csi0_dev0_dvdd_vol   = 1500000
csi0_dev0_afvdd      = "csi-afvcc"
csi0_dev0_afvdd_vol  = 2800000
csi0_dev0_power_en   =
csi0_dev0_reset      = port:PI07<1><0><1><0>
csi0_dev0_pwdn       = port:PI06<1><0><1><0>
csi0_dev0_flash_used = 0
csi0_dev0_flash_type = 2
csi0_dev0_flash_en   =
csi0_dev0_flash_mode =
csi0_dev0_flvdd      = ""
csi0_dev0_flvdd_vol  =
csi0_dev0_af_pwdn    =
csi0_dev0_act_used   = 0
csi0_dev0_act_name   = "ad5820_act"
csi0_dev0_act_slave  = 0x18
```

3.20 tvout/tvin

3.20.1 [tvout_para]

配置项	配置项含义
tvout_used	是否使用 tvout。1：使用 0：不使用
tvout_channel_num	使用的 tvout 通道号
tv_en	tvout 通道使能

示例：

```
[tvout_para]
tvout_used        =
tvout_channel_num =
tv_en             =
```

3.20.2 [tvin_para]

配置项	配置项含义
tvin_used	是否使用 tvint。1：使用 0：不使用
tvin_channel_num	使用的 tvin 通道号

示例：

```
[tvout_para]
tvout_used        =
tvout_channel_num =
tv_en             =
```

3.20.3 [di]

配置项	配置项含义
di_used	是否使用反交错。1：使用 0：不使用

示例：

```
[di]
di_used = 1
```

3.21 SD/MMC

3.21.1 [sdc<X>]

配置项	配置项含义
sdc<X>_used	SDC 使用控制：1 使用，0 不用
bus-width	位宽：1-1bit，4-4bit，8-8bit
sdc<X>_d<X>	SDC DATA<X> 的 GPIO 配置
sdc<X>_clk	SDC CLK 的 GPIO 配置
sdc<X>_cmd	SDC CMD 的 GPIO 配置
sdc<X>_d<X>	SDC DATA<X> 的 GPIO 配置
sd-uhs-sdr50	
sd-uhs-ddr50	

配置项	配置项含义
sd-uhs-sdr104	
broken-cd	
cd-inverted	
non-removable	
sdc<X>_emmc_rst	
cd-gpios	SDC 卡检测信号的 GPIO 配置
card-pwr-gpios	
data3-detect	
sunxi-power-save-mode	SDC CLK 信号无数据传输时暂停
sunxi-dis-signal-vol-sw	
mmc-ddr-1_8v	
mmc-hs200-1_8v	
mmc-hs400-1_8v	
max-frequency	
sdc_tm4_sm0_freq0	
sdc_tm4_sm0_freq1	
sdc_tm4_sm1_freq0	
sdc_tm4_sm1_freq1	
sdc_tm4_sm2_freq0	
sdc_tm4_sm2_freq1	
sdc_tm4_sm3_freq0	
sdc_tm4_sm3_freq1	
sdc_tm4_sm4_freq0	
sdc_tm4_sm4_freq1	
vmmc	SDC 供电电源配置
vqmmc	SDC IO 供电电源配置
vdmmc	是否是 sdio card，0：不是，1：是

示例：

```
[sdc0]
sdc0_used             = 1
bus-width             = 4
sdc0_d1               = port:PF00<2><1><2><default>
sdc0_d0               = port:PF01<2><1><2><default>
sdc0_clk              = port:PF02<2><1><2><default>
sdc0_cmd              = port:PF03<2><1><2><default>
sdc0_d3               = port:PF04<2><1><2><default>
sdc0_d2               = port:PF05<2><1><2><default>
cd-gpios              = port:PH13<0><1><2><default>
sunxi-power-save-mode =
vmmc                  = "vcc-sdcv"
vqmmc                 = "vcc-sdcvq33"
vdmmc                 = "vcc-sdcvd"
```

注，以上仅说明常用配置项，未说明的配置项，可参考

```
linux-X.X/Documentation/devicetree/bindings/mmc/mmc.txt"
```

3.21.2 [smc]

配置项	配置项含义
smc_used	是否使用 sim 卡控制器。1：使用 0：不使用
smc_rst	rst gpio
smc_vppen	vppen gpio
smc_vppp	vppp gpio
smc_det	det gpio
smc_vccen	vccen gpio
smc_sck	sck gpio
smc_sda	sda gpio

示例：

```
[smc_para]
smc_used            = 1
smc_rst             = port:PA09<2><default><default><default>
smc_vppen           = port:PA20<3><default><default><default>
smc_vppp            = port:PA21<3><default><default><default>
smc_det             = port:PA10<2><default><default><default>
smc_vccen           = port:PA06<2><default><default><default>
smc_sck             = port:PA07<2><default><default><default>
smc_sda             = port:PA08<2><default><default><default>
```

3.22 [gpio_para]

配置项	配置项含义
compatible	该配置的名字
gpio_used	内核 GPIO 初始化使能功能，1：开启 0：禁用
gpio_num	GPIO 引脚数目
gpio_pin_1	GPIO 引脚配置
gpio_pin_2	GPIO 引脚配置
normal_led	正常状态灯使用的 GPIO
standby_led	休眠状态灯使用的 GPIO

示例：

```
[gpio_para]
compatible  = "allwinner,sunxi-init-gpio"
gpio_used   = 1
gpio_num    = 2
gpio_pin_1  = port:PL08<1><default><default><1>
gpio_pin_2  = port:power0<1><default><default><0>
normal_led  = "gpio_pin_1"
standby_led = "gpio_pin_2"
```

3.23 USB 控制标志

3.23.1 [usbc<X>]

配置项	配置项含义
usb_used	USB 使能标志 (xx=1 or 0)。1 表示系统中 USB 模块可用，0 则表示系统 USB 禁用。此标志只对具体的 USB 控制器模块有效。
usb_port_type	USB 端口的使用情况。(xx=0/1/2) 0:device only 1:host only 2:OTG
usb_detect_type	USB 端口的检查方式。0：无检查方式 1：vbus/id 检查
usb_detect_mode	usb otg 的检测方法，0-thread scan，1-id gpio interrupt
usb_id_gpio	USB ID pin 脚配置
usb_det_vbus_gpio	USB DET_VBUS pin 脚配置
usb_drv_vbus_gpio	USB DRY_VBUS pin 脚配置
usb_host_init_state	host only 模式下，Host 端口初始化状态。0：初始化后 USB 不工作 1：初始化后 USB 工作
usb_regulator_io	usb 供电的 regulator GPIO
usb_wakeup_suspend	支持 usb 唤醒功能 0：关闭 usb 唤醒功能 1：当进入 normal standby 时候，支持 usb 唤醒 (例如鼠标等外设)
usb_luns	使用 mass storage 功能时的盘符数量
usb_serial_unique	usb device 的序列号是否唯一。1：唯一，使用 chip id; 0：相同，由 usb_serial_number 指定
usb_serial_number	usb device 的序列号

示例：

```
[usbc0]
usbc0_used            = 1
usb_port_type         = 2
usb_detect_type       = 1
usb_detect_mode       = 0
usb_id_gpio           = port:PI4<0><1><default><default>
```

```
usb_det_vbus_gpio     = port:PI8<0><1><default><default>
usb_drv_vbus_gpio     = "axp_ctrl"
usb_host_init_state   = 0
usb_regulator_io      = "nocare"
usb_regulator_vol     = 0
usb_wakeup_suspend    = 0
;---       USB Device
usb_luns              = 3
usb_serial_unique     = 0
usb_serial_number     = "20080411"
```

3.24 [serial_feature]

配置项	配置项含义
sn_filename	该配置的名字

示例：

```
[serial_feature]
sn_filename = "ULI/factory/snum.txt"
```

3.25 重力感应 (G Sensor)

3.25.1 [gsensor_para]

配置项	配置项含义
gsensor_used	是否支持 gsensor
gsensor_twi_id	I2C 的 BUS 控制选择 0：TWI0; 1:TWI1; 2:TWI2
gsensor_twi_addr	芯片的 I2C 地址
gsensor_int1	中断 1 的 GPIO 配置
gsensor_int2	中断 2 的 GPIO 配置

示例：

```
[gsensor_para]
gsensor_used     = 1
gsensor_twi_id   = 2
gsensor_twi_addr = 0x18
gsensor_int1     = port:PA09<6><1><default><default>
gsensor_int2     =
```

3.25.2 [gsensor_list]

配置项	配置项含义
compatible	配置名字
gsensor_list_used	是否支持 gsensor list
da380	是否支持 da380 模组

示例：

```
[gsensor_list_para]
compatible          = "allwinner,sun50i-gsensor-list-para"
gsensor_list__used  = 1
da380               = 1
```

3.26 WiFi

3.26.1 [wlan]

配置项	配置项含义
wlan_used	是否要使用 wifi
compatible	wlan 名称
clocks	低功耗时钟，此值固定为 &clk_outa
wlan_power	wifi 模组使用哪一路 AXP 供电
wlan_io_regulator	wifi 模组 io 使用哪一路 AXP 供电
wlan_busnum	所使用的 SDIO 号，如使用的是 SDIO1，则此值为 1
wlan_regon	Wifi 使能脚
wlan_hostwake	wifi 唤醒主控脚
wlan_clk_gpio	wifi 模块 32K 时钟输出硬件

示例：

```
[wlan]
wlan_used         = 1
compatible        = "allwinner,sunxi-wlan"
clocks            = "outa"
wlan_power        = "vcc-wifi"
wlan_io_regulator = "vcc-io-wifi"
wlan_busnum       = 1
wlan_regon        = port:PG10<1><1><1><0>
wlan_hostwake     = port:ower0<0><default><default><default>
```

3.27 蓝牙 (blueteeth)

3.27.1 [bt]

配置项	配置项含义
bt_used	蓝牙使用控制：1 使用，0 不用
compatible	“allwinner,sunxi-bt”
clocks	低功耗时钟，此值固定为 &clk_outa
clock_io	32K 时钟的 clock io
bt_power	bt 模组使用哪一路 AXP 供电 (通常情况下和 wifi 相同)
bt_io_regulator	bt 模组 io 使用哪一路 AXP 供电 (通常情况下和 wifi 相同)
bt_rst_n	uart 电平转换芯片使能脚

示例：

```
[bt]
bt_used         = 1
compatible      = "allwinner,sunxi-bt"
clocks          = "outa"
pinctrl-names   = "default"
clock_io        = port:PI12<4><0><0><0>
bt_power        = "vcc-wifi"
bt_io_regulator = "vcc-io-wifi"
bt_rst_n        = port:PH12<1><1><1><0>
```

3.27.2 [btlpm]

配置项	配置项含义
btlpm_used	蓝牙使用控制：1 使用，0 不用
uart_index	使用的串口序号，如使用 ttyS1，则此值为 1
bt_wake	主控唤醒 bt 引脚
bt_host_wake	bt 唤醒主控引脚

示例：

```
[btlpm]
btlpm_used   = 0
compatible   = "allwinner,sunxi-btlpm"
uart_index   = 3
bt_wake      = port:PG11<1><1><1><0>
bt_host_wake = port:power1<0><default><default><default>
```

3.28 光感 (light sensor)

3.28.1 [ls_para]

配置项	配置项含义
ls_used	是否支持 ls
ls_twi_id	I2C 的 BUS 控制选择，0：TWI0;1:TWI1;2:TWI2
ls_twi_addr	芯片的 I2C 地址
ls_int	中断的 GPIO 配置

示例：

```
[ls_para]
ls_used     = 0
ls_twi_id   = 1
ls_twi_addr = 0x23
ls_int      = port:PB07<4><1><default><default>
```

3.29 陀螺仪传感器 (gyroscope sensor)

3.29.1 [gy_para]

配置项	配置项含义
gy_used	是否支持 gyroscope
gy_twi_id	I2C 的 BUS 控制选择，0:twi0;1:twi1;2:twi2
gy_twi_addr	芯片的 I2C 地址
gy_int1	中断 1 的 GPIO 配置
gy_int2	中断 2 的 GPIO 配置

示例：

```
[gy_para]
gy_used     = 1
gy_twi_id   = 2
gy_twi_addr = 0x6a
gy_int1     = port:PA10<6><1><default><default>
gy_int2     =
```

3.30 罗盘 Compass

3.30.1 [compass_para]

配置项	配置项含义
compass_used	是否支持 compass
compass_twi_id	I2C 的 BUS 控制选择，0：TWI0;1:TWI1;2:TWI2
compass_twi_addr	芯片的 I2C 地址
compass_int	中断的 GPIO 配置

示例：

```
[compass_para]
compass_used     = 1
compass_twi_id   = 2
compass_twi_addr = 0x0d
compass_int      = port:PA11<6><1><default><default>
```

3.31 数字音频总线（SPDIF）

请参考音频相关文档

3.32 内置音频 codec

请参考音频相关文档

3.33 [s_cir0]

配置项	配置项含义
s_cir0_used	是否使能
ir_power_key_code	power key 码
ir_addr_code0	地址码
ir_addr_cnt	地址数

示例：

```
[s_cir0]
s_cir0_used       = 1
ir_power_key_code = 0x0
ir_addr_code0     = 0x04
ir_addr_cnt       = 0x1
```

3.34 PMU 电源

3.34.1 [pmu<X>]

配置项	配置项含义
compatible	AXP 名字
used	是否使用 AXPxx：0：不使用，1：使用
pmu_id	Pmu 的 id 号
reg	Twi id 号
pmu_vbusen_func	Vubs 引脚 0：输出 1：输入
pmu_reset	长按 16s，0：不操作 1：重启
pmu_irq_wakeup	是否允许中断唤醒，0：not wake up 1：wakeup
pmu_hot_shutdowm	是否允许 pmu 高温关机
pmu_inshort	启动是否检测电池电量

示例：

```
[pmu0]
compatible       = "axp221s"
used             = 1
pmu_id           = 2
reg              = 0x34
pmu_vbusen_func  = 0
pmu_reset        = 0
pmu_irq_wakeup   = 1
pmu_hot_shutdowm = 1
pmu_inshort      = 0
pmu_start        = 0
```

3.34.2 [charger<X>]

配置项	配置项含义
compatible	AXP 名字
used	是否使用 AXPxx：0：不使用，1：使用
pmu_bat_unused	是否使用电池，1：不使用，0：使用

配置项	配置项含义
pmu_chg_ic_temp	是否开启充电智能温度检测，0 关闭，1 开启
pmu_battery_rdc	电池通路内阻，单位 mΩ
pmu_battery_cap	电池容量，单位 mAh，如果配置改值，计量方式为库仑计方式，否则为电压方式。
pmu_runtime_chgcur	设置开机时充电电流大小，单位 mA，仅支持: 300/450/600/750/900/1050/1200/1350/ 1500/1650/1800/1950/2100
pmu_suspend_chgcur	设置待机时充电电流大小，单位 mA，仅支持: 300/4500/600/750/900/1050/1200/1350/ 1500/1650/1800/1950/2100
pmu_shutdown_chgcur	设置关机时充电电流大小，单位 mA，仅支持: 300/4500/600/750/900/1050/1200/1350/ 1500/1650/1800/1950/2100
pmu_init_chgvol	设置充电完成时电池目标电压，单位 mV，仅支持: 4100/4200/4220/4240
pmu_ac_vol	usb-ac 限制电压
pmu_ac_cur	usb-ac 限制电流
pmu_usbpc_vol	usb-pc 限制电压
pmu_usbpc_cur	usb-pc 限制电流
pmu_battery_warning_level1	低电量警告 level1
pmu_battery_warning_level2	低电量警告 level2
pmu_chgled_func	CHGKED 引脚控制。0：PMU 1：充电器
pmu_chgled_type	CHGLED 类型。0：Type A 1：Type B
pmu_ocv_en	
pmu_cou_en	
pmu_update_min_time	
pmu_bat_para1	电池空载电压为 3.13V 对应的电量值
pmu_bat_para2	电池空载电压为 3.27V 对应的电量值
pmu_bat_para3	电池空载电压为 3.34V 对应的电量值
pmu_bat_para4	电池空载电压为 3.41V 对应的电量值
pmu_bat_para5	电池空载电压为 3.58V 对应的电量值
pmu_bat_para6	电池空载电压为 3.52V 对应的电量值
pmu_bat_para7	电池空载电压为 3.55V 对应的电量值
pmu_bat_para8	电池空载电压为 3.57V 对应的电量值
pmu_bat_para9	电池空载电压为 3.59V 对应的电量值
pmu_bat_para10	电池空载电压为 3.61V 对应的电量值
pmu_bat_para11	电池空载电压为 3.63V 对应的电量值
pmu_bat_para12	电池空载电压为 3.64V 对应的电量值
pmu_bat_para13	电池空载电压为 3.66V 对应的电量值
pmu_bat_para14	电池空载电压为 3.7V 对应的电量值
pmu_bat_para15	电池空载电压为 3.73V 对应的电量值

配置项	配置项含义
pmu_bat_para16	电池空载电压为 3.77V 对应的电量值
pmu_bat_para17	电池空载电压为 3.78V 对应的电量值
pmu_bat_para18	电池空载电压为 3.8V 对应的电量值
pmu_bat_para19	电池空载电压为 3.82V 对应的电量值
pmu_bat_para20	电池空载电压为 3.84V 对应的电量值
pmu_bat_para21	电池空载电压为 3.85V 对应的电量值
pmu_bat_para22	电池空载电压为 3.87V 对应的电量值
pmu_bat_para23	电池空载电压为 3.91V 对应的电量值
pmu_bat_para24	电池空载电压为 3.94V 对应的电量值
pmu_bat_para25	电池空载电压为 3.98V 对应的电量值
pmu_bat_para26	电池空载电压为 4.01V 对应的电量值
pmu_bat_para27	电池空载电压为 4.05V 对应的电量值
pmu_bat_para28	电池空载电压为 4.08V 对应的电量值
pmu_bat_para29	电池空载电压为 4.1V 对应的电量值
pmu_bat_para30	电池空载电压为 4.12V 对应的电量值
pmu_bat_para31	电池空载电压为 4.14V 对应的电量值
pmu_bat_para32	电池空载电压为 4.15V 对应的电量值
pmu_bat_temp_enable	电池温度检测使能
pmu_bat_charge_ltf	电池充电低温门限电压
pmu_bat_charge_htf	电池充电高温门限电压
pmu_bat_shutdown_ltf	关机电池低温门限电压
pmu_bat_shutdown_htf	关机电池高温门限电压
pmu_bat_temp_para1	电池温度-25 度对应的电压
pmu_bat_temp_para2	电池温度-15 度对应的电压
pmu_bat_temp_para3	电池温度-10 度对应的电压
pmu_bat_temp_para4	电池温度-5 度对应的电压
pmu_bat_temp_para5	电池温度 0 度对应的电压
pmu_bat_temp_para6	电池温度 5 度对应的电压
pmu_bat_temp_para7	电池温度 10 度对应的电压
pmu_bat_temp_para8	电池温度 20 度对应的电压
pmu_bat_temp_para9	电池温度 30 度对应的电压
pmu_bat_temp_para10	电池温度 40 度对应的电压
pmu_bat_temp_para11	电池温度 45 度对应的电压
pmu_bat_temp_para12	电池温度 50 度对应的电压
pmu_bat_temp_para13	电池温度 55 度对应的电压
pmu_bat_temp_para14	电池温度 60 度对应的电压
pmu_bat_temp_para15	电池温度 70 度对应的电压
pmu_bat_temp_para16	电池温度 80 度对应的电压

配置项	配置项含义
power_start	当充电状态下的关机动作。1：关机; 非 1：重启。当有接入电池的情况下，插入外部电源时：0：关机状态下，插入外部电源时，电池电量充足时，不允许开机，会进入充电模式；电池电量不足，则关机。1：关机状态下，插入外部电源，电池电量充足时，直接开机进入系统；电池电量不足，则关机。2：关机状态下，插入外部电源时，不允许开机，会进入充电模式；无视电池电量。3：关机状态下，插入外部电源，直接开机进入系统；无视电池电量。

示例：

```
[charger0]
compatible                  = "axp221s-charger"
pmu_chg_ic_temp             = 0
pmu_battery_rdc             = 100
pmu_battery_cap             = 0
pmu_runtime_chgcur          = 450
pmu_suspend_chgcur          = 1500
pmu_shutdown_chgcur         = 1500
pmu_init_chgvol             = 4200
pmu_ac_vol                  = 4000
pmu_ac_cur                  = 0
pmu_usbpc_vol               = 4400
pmu_usbpc_cur               = 500
pmu_battery_warning_level1  = 15
pmu_battery_warning_level2  = 0
pmu_chgled_func             = 0
pmu_chgled_type             = 0
power_start                 = 0

pmu_bat_para1               = 0
pmu_bat_para2               = 0
pmu_bat_para3               = 0
pmu_bat_para4               = 0
pmu_bat_para5               = 0
pmu_bat_para6               = 0
pmu_bat_para7               = 0
pmu_bat_para8               = 0
pmu_bat_para9               = 5
pmu_bat_para10              = 8
pmu_bat_para11              = 9
pmu_bat_para12              = 10
pmu_bat_para13              = 13
pmu_bat_para14              = 16
pmu_bat_para15              = 20
pmu_bat_para16              = 33
pmu_bat_para17              = 41
pmu_bat_para18              = 46
pmu_bat_para19              = 50
pmu_bat_para20              = 53
pmu_bat_para21              = 57
pmu_bat_para22              = 61
pmu_bat_para23              = 67
```

```
pmu_bat_para24              = 73
pmu_bat_para25              = 78
pmu_bat_para26              = 84
pmu_bat_para27              = 88
pmu_bat_para28              = 92
pmu_bat_para29              = 93
pmu_bat_para30              = 94
pmu_bat_para31              = 95
pmu_bat_para32              = 100

pmu_bat_temp_enable         = 0
pmu_bat_charge_ltf          = 2261
pmu_bat_charge_htf          = 388
pmu_bat_shutdown_ltf        = 3200
pmu_bat_shutdown_htf        = 237
pmu_bat_temp_para1          = 7466
pmu_bat_temp_para2          = 4480
pmu_bat_temp_para3          = 3518
pmu_bat_temp_para4          = 2786
pmu_bat_temp_para5          = 2223
pmu_bat_temp_para6          = 1788
pmu_bat_temp_para7          = 1448
pmu_bat_temp_para8          = 969
pmu_bat_temp_para9          = 664
pmu_bat_temp_para10         = 466
pmu_bat_temp_para11         = 393
pmu_bat_temp_para12         = 333
pmu_bat_temp_para13         = 283
pmu_bat_temp_para14         = 242
pmu_bat_temp_para15         = 179
pmu_bat_temp_para16         = 134
```

3.34.3 [powerkey<X>]

配置项	配置项含义
compatible	设备名字
pmu_powkey_off_timet	系统起来后，长按关机时间
pmu_powkey_off_func	系统起来后，长按功能 0：shutdown，1：restart
pmu_powkey_off_en	系统起来后，是否使用长按功能
pmu_powkey_long_time	短按响应时间
pmu_powkey_on_time	关机后，长按开机时间
pmu_hot_shutdowm	是否允许 pmu 高温关机
pmu_inshort	启动是否检测电池电量

示例：

```
[powerkey0]
compatible          = "axp221s-powerkey"
pmu_powkey_off_time  = 6000
pmu_powkey_off_func  = 0
```

```
pmu_powkey_off_en    = 1
pmu_powkey_long_time = 1500
pmu_powkey_on_time   = 1000
```

3.34.4 [regulator<X>]

配置项	配置项含义
compatible	设备名
regulator_count	regulator 数量
regulator<X>	regulator<X> 对应的别名，请勿修改

示例：

```
[regulator0]
compatible      = "axp221s-regulator"
regulator_count = 20
regulator1      = "axp221s_dcdc1 none vcc-hdmi vcc-io vcc-dsi vcc-usb vdd-efuse vcc-hp vcc-
    audio vcc-emmc vcc-card vcc-pc vcc-pd vcc-3v vcc-tvout vcc-tvin vcc-emmcv vcc-sdcv vcc-
    sdcvq33 vcc-sdcvd vcc-nand vcc-sdcv-p3 vcc-sdcvq33-p3 vcc-sdcvd-p3"
regulator2      = "axp221s_dcdc2 none vdd-cpua"
regulator3      = "axp221s_dcdc3 none vdd-sys vdd-gpu"
regulator4      = "axp221s_dcdc4 none"
regulator5      = "axp221s_dcdc5 none vcc-dram"
regulator6      = "axp221s_rtc none vcc-rtc"
regulator7      = "axp221s_aldo1 none vcc-25 csi-avdd"
regulator8      = "axp221s_aldo2 none vcc-pa ehpy-vdd25"
regulator9      = "axp221s_aldo3 none avcc vcc-pll"
regulator10     = "axp221s_dldo1 none vcc-io-wifi vcc-pg "
regulator11     = "axp221s_dldo2 none vcc-wifi"
regulator12     = "axp221s_dldo3 none"
regulator13     = "axp221s_dldo4 none vdd-sata-25 vcc-pf"
regulator14     = "axp221s_eldo1 none vcc-pe csi-iovcc csi-afvcc"
regulator15     = "axp221s_eldo2 none csi-dvdd"
regulator16     = "axp221s_eldo3 none vdd-sata-12"
regulator17     = "axp221s_ldoio0 none vcc-ctp"
regulator18     = "axp221s_ldoio1 none vcc-i2s-18"
regulator19     = "axp221s_dc1sw none ephy-dvdd33"
regulator20     = "axp221s_dc5ldo none"
```

3.34.5 [axp_gpio<X>]

配置项	配置项含义
compatible	设备名

示例：

```
[axp_gpio0]
compatible = "axp221s-gpio"
```

3.34.6 [psensor_table]

配置项	配置项含义
psensor_count	psensor 数量
prange_min_2	范围最小值 2
prange_max_2	范围最大值 2
prange_min_1	范围最小值 1
prange_max_1	范围最大值 1
prange_min_0	范围最小值 0
prange_max_0	范围最大值 0

示例：

```
psensor_count = 3

prange_min_2 = 4800
prange_max_2 = 6500

prange_min_1 = 4500
prange_max_1 = 4800

prange_min_0 = 0
prange_max_0 = 4500
```

3.35 DVFS

3.35.1 [dvfs_table]&&[dvfs_table_[X]]

配置项	配置项含义
extremity_freq	极限频率
max_freq	最大运行频率
min_freq	最小运行频率
lv_count	VF 表项数
lvn_freq	对应的最大频率 (n 表示级数)
lvn_volt	第 n 级的电压

示例：

```
[dvfs_table]
max_freq = 1200000000
min_freq = 240000000

lv_count = 8

lv1_freq = 1200000000
lv1_volt = 1300

lv2_freq = 1104000000
lv2_volt = 1240

lv3_freq = 1008000000
lv3_volt = 1160

lv4_freq = 912000000
lv4_volt = 1100

lv5_freq = 720000000
lv5_volt = 1000

lv6_freq = 0
lv6_volt = 1000

lv7_freq = 0
lv7_volt = 1000

lv8_freq = 0
lv8_volt = 1000
```

⚠ 警告

vf 表（电压频率对应表）关乎系统稳定性，请勿私自修改！

3.36 s_uart<X>

配置项	配置项含义
s_uart_used	是否使用 cpus 的 uart 模块；0-否，1-是
s_uart_tx	cpus TX GPIO 配置
s_uart_rx	cpus RX GPIO 配置

示例：

```
[s_uart0]
s_uart_used = 1
s_uart_tx   = port:PL02<2><default><default><default>
s_uart_rx   = port:PL03<2><default><default><default>
```

3.37 s_twi<X>

配置项	配置项含义
s_twi0_used	0-否，1-是
s_twi0_sck	cpus i2c sck GPIO 配置
s_twi0_sda	cpus i2c sda GPIO 配置

示例：

```
[s_twi0]
s_twi0_used = 1
s_twi0_sck  = port:PL00<2><1><2><default>
s_twi0_sda  = port:PL01<2><1><2><default>
```

3.38 s_jtag<X>

配置项	配置项含义
s_jtag_used	0-否，1-是
s_jtag_tms	cpus jtag 模式选择输入 GPIO 配置
s_jtag_tck	cpus jtag 时钟选择输入 GPIO 配置
s_jtag_tdo	cpus jtag 数据输出 GPIO 配置
s_jtag_tdi	cpus jtag 数据输入 GPIO 配置

示例：

```
s_jtag_used = 0
s_jtag_tms  = port:PL04<2><1><2><default>
s_jtag_tck  = port:PL05<2><1><2><default>
s_jtag_tdo  = port:PL06<2><1><2><default>
s_jtag_tdi  = port:PL07<2><1><2><default>
```

3.39 Virtual device

3.39.1 [Vdevice]

配置项	配置项含义
Vdevice_used	作为 pinctrl test 的虚拟设备，为 1 使能
Vdevice_0	虚拟设备的 gpio0 脚设置
Vdevice_1	虚拟设备的 gpio1 脚设置

示例：

```
[Vdevice]
Vdevice_used = 1
Vdevice_0    = port:PB00<4><1><2><default>
Vdevice_1    = port:PB01<4><1><2><default>
```

4 设备树介绍

4.1 Device tree 介绍

ARM Linux 中，arch/arm/mach-xxx 中充斥着大量描述板级细节的代码，而这些板级细节对于内核来讲，就是垃圾，如板上的 platform 设备、resource、i2c_board_info、spi_board_info 以及各种硬件的 platform_data。

内核社区为了改变这个局面，引用了 PowerPC 等其他体系结构下已经使用的 Flattened Device Tree(FDT)。采用 Device Tree 后，许多硬件的细节可以直接透过它传递给 Linux，而不再需要在 kernel 中进行大量的冗余编码。

Device Tree 是一种描述硬件的数据结构，它表现为一颗由电路板上 cpu、总线、设备组成的树，Device Tree 由一系列被命名的结点 (node) 和属性 (property) 组成，而结点本身可包含子结点。所谓属性，其实就是成对出现的 name 和 value。在 Device Tree 中，可描述的信息包括：

- CPU 的数量和类别
- 内存基地址和大小
- 总线
- 外设
- 中断控制器
- GPIO 控制器
- Clock 控制器

Bootloader 会将这棵树传递给内核，内核可以识别这棵树，并根据它展开出 Linux 内核中的 platform_device、i2c_client、spi_device 等设备，而这些设备用到的内存、IRQ 等资源，也会通过 dtb 传递给了内核，内核会将这些资源绑定给展开的相应的设备。

Device tree 牵扯的东西还是比较多的，对 device tree 的理解，可以分为 5 个步骤：

1. 用于描述硬件设备信息的文本格式，如 dts/dtsi。
2. 认识 DTC 工具。
3. Bootloader 怎么把二进制文件写入到指定的内存位置。
4. 内核时如何展开文件，获取硬件设备信息。
5. 设备驱动如何使用。

4.2 Device tree source file

.dts 文件是一种 ASCII 文本格式的 Device Tree 描述，在 ARM Linux 中，一个.dts 文件对应一个 ARM 的 machine。* ARMv7 架构下，dts 文件放置在内核的 arch/arm/boot/dts/目录。* ARMv8 架构下，dts 文件放置在内核的 arch/arm64/boot/dts/目录。* RISCV 架构下，dts 文件放置在内核的 arch/riscv/boot/dts/目录。

由于一个 SoC 可能对应多个 machine（一个 SoC 可以对应多个产品和电路板），势必这些.dts 文件需包含许多共同的部分。Linux 内核为了简化，把 SoC 公用的部分或者多个 machine 共同的部分一般提炼为.dtsi，类似于 C 语言的头文件，其他的 machine 对应的.dts 就 include 这个.dtsi。

设备树是一个包含节点和属性的简单树状结构。属性就是键－值对，而节点可以同时包含属性和子节点。例如，以下就是一个.dts 格式的简单树：

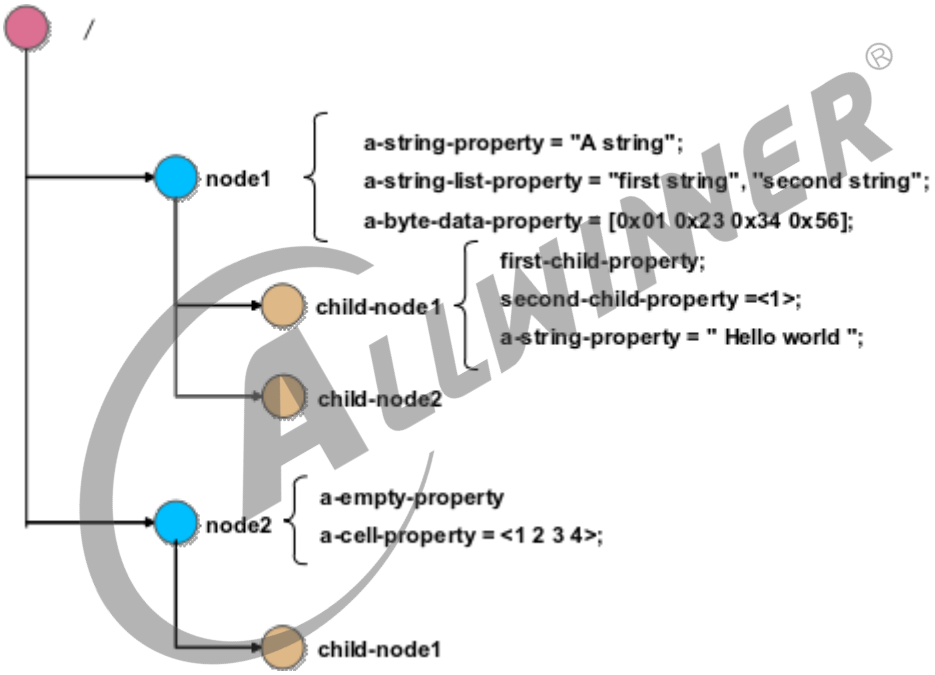

图 4-1：dts 简单树示例

这棵树显然是没什么用的，因为它并没有描述任何东西，但它确实体现了节点的一些属性：

1. 一个单独的根节点：“/”。
2. 两个子节点：“node1” 和 “node2”。
3. 两个 node1 的子节点：“child-node1” 和 “child-node2”。
4. 一堆分散在树里的属性。

属性是简单的键－值对，它的值可以为空或者包含一个任意字节流。虽然数据类型并没有编码进数据结构，但在设备树源文件中仍有几个基本的数据表示形式。

1. 文本字符串（无结束符）可以用双引号表示：a-string-property=”hello world”。
2. 二进制数据用方括号限定。
3. 不同表示形式的数据可以使用逗号连在一起。
4. 逗号也可用于创建字符串列表：a-string-list-property=”first string”,”second string”。

4.2.1 Device tree 结构约定

4.2.1.1 节点名称 (node names)

规范：device tree 中每个节点的命名必须遵从一下规范：node-name@unit-address

详注：

1. node-name：节点的名称，小于 31 字符长度的字符串，可以包括图中所示字符。节点名称的首字符必须是英文字母，可大写或者小写。通常，节点的命名应该根据它所体现的是什么样的设备。

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

图 4-2: 节点名称支持字符

2. @unit-address：如果该节点描述的设备有一个地址，则应该加上设备地址（unit-address）。通常，设备地址就是用来访问该设备的主地址，并且该地址也在节点的 reg 属性中列出。
3. 同级节点命名必须是唯一的，但只要地址不同，多个节点也可以使用一样的通用名称（例如 serial@101f1000 和 serial@101f2000）。
4. 根节点没有 node-name 或者 unit-address，它通过 “/” 来识别。

实例

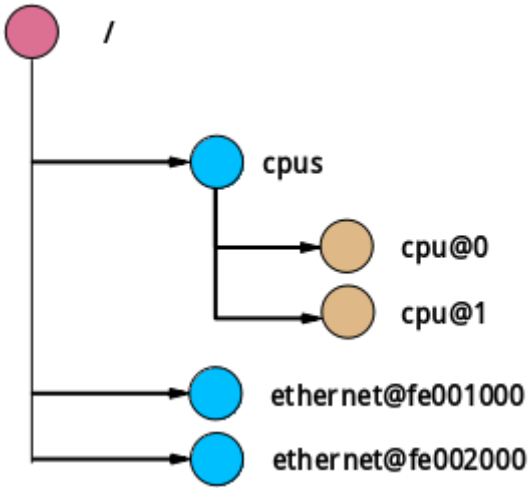

图 4-3: 节点名称规范示例

在实例中，一个根节点/下有 3 个子节点；节点名称为 cpu 的节点，通过地址 0 和 1 来区别；节点名称为 ethernet 的节点，通过地址 fe001000 和 fe002000 来区别。

4.2.1.2 路径名称 (path names)

在 device tree 中唯一识别节点的另一个方法，通过给节点指定从根节点到该节点的完整路径。

device tree 中约定了完整路径表达方式：

```
/node-name-1/node-name-2/.../node-name-N
```

实例：

如图 2-3 节点名称规范示例，

指定根节点路径：/

指定 cpu#1 的完整路径：/cpus/cpu@1

指定 ethernet#fe002000：/cpus/ethernet@fe002000

说明

注：如果完整的路径可以明确表示我们所需的节点，那么 *unit-address* 可以省略

4.2.1.3 属性 (properties)

Device tree 中，节点可以用属性来描述该节点的特征，属性由两个部分组成：名称和值。

属性名称 (property names)

由长度小于 31 的字符串组成。属性名称支持的字符如下图：

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

图 4-4: 属性名称支持字符

非标准的属性名称，需要指定一个唯一的前缀，用来识别是哪个公司或者机构定义了该属性。

例如：

```
fsl,channel-fifo-len 29
ibm,ppc-interrupt-server#s 30
linux,network-index
```

属性值 (property values)

属性值是一个包含属性相关信息的数组，数组可能有 0 个或者多个字节。

当属性是为了传递真伪信息时，属性值可能为空值，这个时候，属性值的存在或者不存在，就已经足够描述属性的相关信息了。

value	description
<empty>	属性表达真伪信息，判断值有没有存在就可以识别
<u32>	大端格式的 32 位整数. 例如：值 0x11223344 ，则 address 11; address+1 22; address+2 33; address+3 44;
<u64>	大端格式的 64 位整数，有两个 <u32> 组成。第一 <u32> 表示高位，第二 <u32> 表示地位。例如，0x1122334455667788 由两个单元组成 <0x11223344,0x55667788> address 11 address+1 22 address+2 33 address+3 44 address+4 55 address+5 66 address+6 77 address+7 88
<string>	字符串可打印，并且有终结符。例如：“hello” address 68 address+1 65 address+2 6c address+3 6c address+4 6f address+5 00
<prop-encoded-array>	跟特定的属性有关
<phandle>	一个 <u32> 值，phandle 值提供了一种引用设备树中其他节点的方法。通过定义 phandle 属性值，任何节点都可以被其他节点引用。

value	description
<stringlist>	由一系列 <string> 值串连在一起，例如”hello”,”world”. address 68 address+1 65 address+2 6C address+3 6C address+4 6F address+5 00 address+6 77 address+7 6F address+8 72 address+9 6C address+10 64 address+11 00

4.2.1.4 标准属性类型

Compatible

```
1. 属性：compatible
2. 值类型：<stringlist>
3. 说明：
   树中每个表示一个设备的节点都需要一个 compatible 属性。
   compatible 属性是操作系统用来决定使用哪个设备驱动来绑定到一个设备上的关键因素。
   compatible 是一个字符串列表，之中第一个字符串指定了这个节点所表示的确切的设备，该字符串的格式为：
        “<制造商>,<型号>”
   剩下的字符串的则表示其它与之相兼容的设备。
   例如：compatible = “fsl,mpc8641-uart”, “ns16550";
   系统首先会查找跟fsl,mpc8641-uart相匹配的驱动，如果找不到，就找更通用的，跟ns16550相匹配的驱动。
```

Model

```
1. 属性：model
2. 值类型：<string>
3. 说明：
  model 属性值是<string>,该值指定了设备的型号。推荐的使用形式如下：
   “manufacturer, model”
  其中，字符manufacturer表示厂商的名称，字符model表示设备的型号。
  例如：model = “fsl,MPC8349EMITX”;
```

Phandle

```
1. 属性：phandle
2. 值类型：<u32>
3. 说明：
device tree 中，定义了phandle属性，它是一个u32的值。
每个节点都可以拥有一个相关的phandle，通过它的值来唯一标识。(实际实现中常采用指针或者偏移)。
phandle 常用于查询或者遍历设备树，也有用于指向设备树中的其它节点。
例如，在设备树中，pic节点如下所示：
        pic@10000000 {
            phandle = <1>;
            interrupt-controller;
        };
定义pic节点的phandle 为1，那么其他设备节点引用pic节点时，只需要在本节点中添加：
        interrupt-parent = <1>;
```

Status

1. 属性：status
2. 值类型：<string>
3. 说明：
   该属性指明设备的运行状态，见表格

value	description
“okay”	表明设备可运行
“disabled”	表明设备当前不可运行，但条件满足，它还是可以运行的。
“fail”	表明设备不可运行，设备产生严重错误，如果不修复，将一直不可运行
“fail-sss”	表明设备不可运行，设备产生严重错误，如果不修复，将一直不可运行.sss 部分特定设备相关，指明错误检测条件。

#address-cells 和 #size-cells

1.属性：#address-cells，#size-cells
2.值类型：<u32>
3.说明：
  #address-cells 和#size-cells 属性常备用在拥有孩子节点的父节点上，用来描述孩子节点时如何编址的。
  父结点的#address-cells 和#size-cells 分别决定了子结点的reg 属性的address 和length 字段的长度。
  例如：

```
/ {
        compatible = "acme,coyotes-revenge";
        #address-cells = <1>;
        #size-cells = <1>;
        interrupt-parent = <&intc>;
        cpus {
                #address-cells = <1>;
                #size-cells = <0>;
                cpu@0 {
                        compatible = "arm,cortex-a9";
                        reg = <0>;
                };
                cpu@1 {
                        compatible = "arm,cortex-a9";
                        reg = <1>;
                };
        };
        serial@101f0000 {
                compatible = "arm,pl011";
                reg = <0x101f0000 0x1000 >;
                interrupts = < 1 0 >;
        };
        serial@101f2000 {
                compatible = "arm,pl011";
                reg = <0x101f2000 0x1000 >;
                interrupts = < 2 0 >;
        };
        gpio@101f3000 {
                compatible = "arm,pl061";
                reg = <0x101f3000 0x1000
                0x101f4000 0x0010>;
                interrupts = < 3 0 >;
        };
        intc: interrupt-controller@10140000 {
                compatible = "arm,pl190";
                reg = <0x10140000 0x1000 >;
                interrupt-controller;
                #interrupt-cells = <2>;
        };
        spi@10115000 {
                compatible = "arm,pl022";
                reg = <0x10115000 0x1000 >;
                interrupts = < 4 0 >;
        };
        external-bus {
                #address-cells = <2>
                #size-cells = <1>;
                ranges = <0 0 0x10100000 0x10000     // Chipselect 1, Ethernet
                          1 0 0x10160000 0x10000     // Chipselect 2, i2c controller
                          2 0 0x30000000 0x1000000>; // Chipselect 3, NOR Flash
                ethernet@0,0 {
                        compatible = "smc,smc91c111";
                        reg = <0 0 0x1000>;
                        interrupts = < 5 2 >;
                };
                i2c@1,0 {
                        compatible = "acme,a1234-i2c-bus";
                        #address-cells = <1>;
                        #size-cells = <0>;
                        reg = <1 0 0x1000>;
                        interrupts = < 6 2 >;
                        rtc@58 {
                                compatible = "maxim,ds1338";
                                reg = <58>;
                                interrupts = < 7 3 >;
                        };
                };
                flash@2,0 {
                        compatible = "samsung,k8f1315ebm", "cfi-flash";
                        reg = <2 0 0x4000000>;
                };
        };
 [...]
};
```

图 4-5: address-cells 和 size-cells 示例

```
root 结点的#address-cells = <1>和#size-cells =<1>;
决定了serial、gpio、spi 等结点的address 和length 字段的长度分别为1。

cpus 结点的#address-cells = <1>和#size-cells = <0>;
决定了2 个cpu 子结点的address 为1，而length 为空，于是形成了2 个cpu 的reg = <0>和reg = <1>。

external-bus 结点的#address-cells = <2>和#size-cells = <1>;
决定了其下的ethernet、i2c、flash 的reg 字段形如reg = <0 0 0x1000>;reg = <1 0 0x1000>和reg = <2 0
     0x4000000>。
其中，address字段长度为0，开始的第1个cell（0、1、2）是对应的片选，第2 个cell（0，0，0）是相对该片选的基地
    址，
第3 个cell（0x1000、0x1000、0x4000000）为length。

特别要留意的是i2c 结点中定义的 #addresscells= <1>和#size-cells = <0>;
又作用到了I2C 总线上连接的RTC，它的address 字段为0x58，是设备的I2C 地址。
```

Reg

```
1. 属性：reg
2. 值类型：<address1 length1 [address2 length2] [address3 length3] ... >
3. 说明
  reg 属性描述了设备拥有资源的地址信息，其中的每一组address  length 表明了设备使用的一个地址范围。
  address 为1 个或多个32 位的整型（即cell），而length 则为cell 的列表或者为空（若#size-cells = 0）。
  address 和length 字段是可变长的，父结点的#address-cells 和#size-cells
  分别决定了子结点的reg 属性的address 和length 字段的长度。
```

Virtual-reg

```
1. 属性：virtual-reg
2. 值类型：<u32>
3. 说明：virutal-reg 属性指定一个有效的地址映射到物理地址。
```

Ranges

```
1. 属性：ranges
2. 值类型：<empty>或者<prop-encoded-array>
3. 说明：
  前边reg属性说明中，我们已经知道如何给设备分配地址，但目前来说这些地址还只是设备节点的本地地址，
  我们还没有描述如何将这些地址映射成 CPU 可使用的地址。
  根节点始终描述的是 CPU 视角的地址空间。根节点的子节点已经使用的是 CPU 的地址域，
  所以它们不需要任何直接映射。例如，serial@101f0000 设备就是直接分配的 0x101f0000 地址。
  那些非根节点直接子节点的节点就没有使用 CPU 地址域。为了得到一个内存映射地址，
  设备树必须指定从一个域到另一个域地址转换的方法，而 ranges 属性就为此而生。
  还以图2-5的设备数来分析：

        ranges = < 0 0 0x10100000 0x10000    // Chipselect 1,  Ethernet
                   1 0 0x10160000 0x10000    // Chipselect 2,  i2c controller
                   2 0 0x30000000 0x1000000 >; // Chipselect 3,  NOR Flash

  ranges 是一个地址转换列表。ranges 表中的每一项都是一个包含子地址、父地址和在子地址空间中区域大小的元组。
  每个字段的值都取决于子节点的 #address-cells 、父节点的 #address-cells 和子节点的 #size-cells。
  以本例中的外部总线来说，子地址是 #address-cells是2、父地址#address-cells是 1 、区域大小#size-cells
      是1。
  那么三个 ranges 被翻译为：
    从片选 0 开始的偏移量 0 被映射为地址范围：0x10100000..0x1010ffff
    从片选 0 开始的偏移量 1 被映射为地址范围：0x10160000..0x1016ffff
    从片选 0 开始的偏移量 2 被映射为地址范围：0x30000000..0x10000000
```

另外，如果父地址空间和子地址空间是相同的，那么该节点可以添加一个空的 range 属性。
一个空的 range 属性意味着子地址将被 1:1 映射到父地址空间。

4.2.2 常用节点类型

所有 device tree 都必须拥有一个根节点，还必须在根节点下边有以下的节点：

1. Cpu 节点
2. Memory 节点

4.2.2.1 根节点 (root node)

设备树都必须有一个根节点，树中其它的节点都是根节点的后代，根节点的完整路径是/。

根节点具有如下属性：

属性名称	需要使用	属性值类型	定义
#address-cells	需要	<u32>	表示子节点寄存器属性中的地址
#size-cells	需要	<u32>	表示子节点寄存器属性中的大小
model	需要	<string>	指定一个字符串用来识别不同板子
compatible	需要	<stringlist>	指定平台的兼容列表
Epapr-version	需要	<string>	这个属性必须包含下边字符串“ePAPR-<ePAPR version>”其中，<ePAPR version> 是平台遵从的 PAPR 规范版本号，例如：Epapr-version =”ePAPR-1.1”

4.2.2.2 别名节点 (aliases node)

Device tree 中采用别名节点来定义设备节点全路径的别名，别名节点必须是根节点的孩子，而且还必须采用 aliases 的节点名称。

/aliases 节点中每个属性定义了一个别名，属性的名字指定了别名，属性值指定了 device tree 中设备节点的完整路径。例如：

```
serial0 = “/simple-bus@fe000000/serial@llc500”
```

指定该路径下 serial@llc500 设备节点全路径的别名为 serial0。当用户想知道的只是“那个设备是 serial”时，这样的全路径就会变得很冗长，采用 aliases 节点指定一个设备节点全路径的别名，好处就在这个时候体现出来了

4.2.2.3 内存节点 (memory node)

ePAPR 规范中指定了内存节点是 device tree 中必须的节点。内存节点描绘了系统物理内存的信息，如果系统中有多个内存范围，那么 device tree 中可能会创建多个内存节点，或者在一个单独的内存节点中通过 reg 属性指定内存的范围。节点的名称必须是 memory。

内存节点属性如下：

属性名称	是否使用	值类型	定义
Device_type	需要	<string>	属性值必须为”memory”
reg	需要	<prop-encoded-array>	包含任意数量的用来指示地址和地址空间大小的对
Initial-mapped-area	可选择	<prop-encoded-array>	指定初始映射区的内存地址和地址空间的大小

假设一个 64 位系统具有以下的物理内存块：

1. RAM：起始地址 0x0，长度 0x80000000(2GB)
2. RAM：起始地址 0x100000000，长度 0x100000000(4GB)

内存节点的定义可以采用以下方式，假设 #address-cells =2，#size-cells =2。

方式 1：

```
memory@0 {
    device_type = "memory";
    reg = < 0x000000000 0x00000000 0x00000000 0x80000000
            0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

方式 2：

```
memory@0 {
    device_type = "memory";
    reg = < 0x000000000 0x00000000 0x00000000 0x80000000>;
};
memory@100000000 {
    device_type = "memory";
    reg = < 0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

4.2.2.4 chosen 节点

chosen 节点并不代表一个真正的设备，只是作为一个为固件和操作系统之间传递数据的地方，比如引导参数。chosen 节点里的数据也不代表硬件。通常，chosen 节点在.dts 源文件中为空，并

在启动时填充。它必须是根节点的孩子。节点属性如下：

属性名称	是否使用	值类型	定义
bootargs	可选择	<string>	为用户指定 boot 参数
Stdout-path	可选择	<string>	指定 boot 控制台输出路径
Stdin-path	可选择	<string>	指定 boot 控制台输入路径

例子：

```
chosen {
    bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

4.2.2.5 cpus 节点

ePAPR 规范指定 cpus 节点是 device tree 中必须的节点，它并不代表系统中真实设备，可以理解 cpus 节点仅作为存放子节点 cpu 的一个容器。节点属性如下：

属性名称	是否使用	值类型
#address-cells	必须	<u32>
#size-cells	必须	<u32>

4.2.2.6 cpu 节点

Device tree 中每一个 cpu 节点描述一个具体的硬件执行单元。每个 cpu 节点的 compatible 属性是一个 “,” 形式的字符串，并指定了确切的 cpu，就像顶层的 compatible 属性一样。如果系统的 cpu 拓扑结构很复杂，还必须在 binding 文档中详细说明。

cpu 节点所拥有的属性：

属性名称	是否使用	值类型	定义
Device_type	必须	<string>	属性值必须是 “cpu” 的字符串
reg	必须	<prop-encoded-array>	定义 cpu/thread id
Clock-frequency	必须	<prop-encodec-array>	指定 cpu 的时钟频率

属性名称	是否使用	值类型	定义
Timebase-frequency	必须	<prop-encoded-array>	指定当前 timebase 的是时钟频率信息
status		<u32>	描述 cpu 的状态 okay/disabled
Enable-method		<stringlist>	指定了 cpu 从 disabled 状态到 enabled 的方式
Mmu-type	可选	<string>	指定 cpu mmu 的类型

cpu 节点实例：

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
        device_type = "cpu";
        compatible = "arm,cortex-a8";
        reg = <0x0>;
    };
};
```

4.2.2.7 soc 节点

这个节点用来表示一个系统级芯片（soc），如果处理器就是一个系统级芯片，那么这个节点就必须包含，soc 节点的顶层包含 soc 上所有设备可见的信息。

节点名字必须包含 soc 的地址并且以”soc” 字符开头。

实例：

```
soc@01c20000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x01c20000 0x300000>;
        ranges;

        intc: interrupt-controller@01c20400 {
            compatible = "allwinner,sun4i-ic";
            reg = <0x01c20400 0x400>;
            interrupt-controller;
            #interrupt-cells = <1>;
        };

        pio: pinctrl@01c20800 {
            compatible = "allwinner,sun5i-a13-pinctrl";
            reg = <0x01c20800 0x400>;
            interrupts = <28>;
            clocks = <&apb0_gates 5>;
            gpio-controller;
```

```
            interrupt-controller;
            #address-cells = <1>;
            #size-cells = <0>;
            #gpio-cells = <3>;

            uart1_pins_a: uart1@0 {
                allwinner,pins = "PE10", "PE11";
                allwinner,function = "uart1";
                allwinner,drive = <0>;
                allwinner,pull = <0>;
            };
          .........
}
```

4.2.3 Binding

对于 Device Tree 中的结点和属性具体是如何来描述设备的硬件细节的，一般需要文档来进行讲解，这些文档位于内核的 Documentation/devicetree/bindings/arm 路径下。

4.3 Device tree block file

4.3.1 DTC (device tree compiler)

将.dts 编译为.dtb 的工具。DTC 的源代码位于内核的 scripts/dtc 目录，在 Linux 内核使能了 Device Tree 的情况下，编译内核时同时会编译 dtc。通过 scripts/dtc/Makefile 中的“hostprogs-y := dtc” 这一 hostprogs 编译 target。

在 Linux 内核的 arch/arm/boot/dts/Makefile 中，描述了当某个 SoC 被选中后，哪些.dtb 文件会被编译出来，如与 sunxi 对应的.dtb 包括

```
dtb-$(CONFIG_ARCH_SUNXI) += \
    sun4i-a10-cubieboard.dtb \
    sun4i-a10-mini-xplus.dtb \
    sun4i-a10-hackberry.dtb \
    sun5i-a10s-olinuxino-micro.dtb \
    sun5i-a13-olinuxino.dtb
```

4.3.2 Device Tree Blob (.dtb)

.dtb 是.dts 被 DTC 编译后的二进制格式的 Device Tree 描述，可由 Linux 内核解析。通常在我们为电路板制作 NAND、SD 启动 image 时，会为.dtb 文件单独留下一个很小的区域以存放之，之后 bootloader 在引导 kernel 的过程中，会先读取该.dtb 到内存。

4.3.3 DTB 的内存布局

Device tree block 内存布局大致如下 (地址从上往下递增)。我们可以看到，dtb 文件结构主要由 4 个部分组成，一个小的文件头、一个 memory reserve map、一个 device tree structure 、一个 device-tree strings。这几个部分构成一个整体，一起加载到内存中。

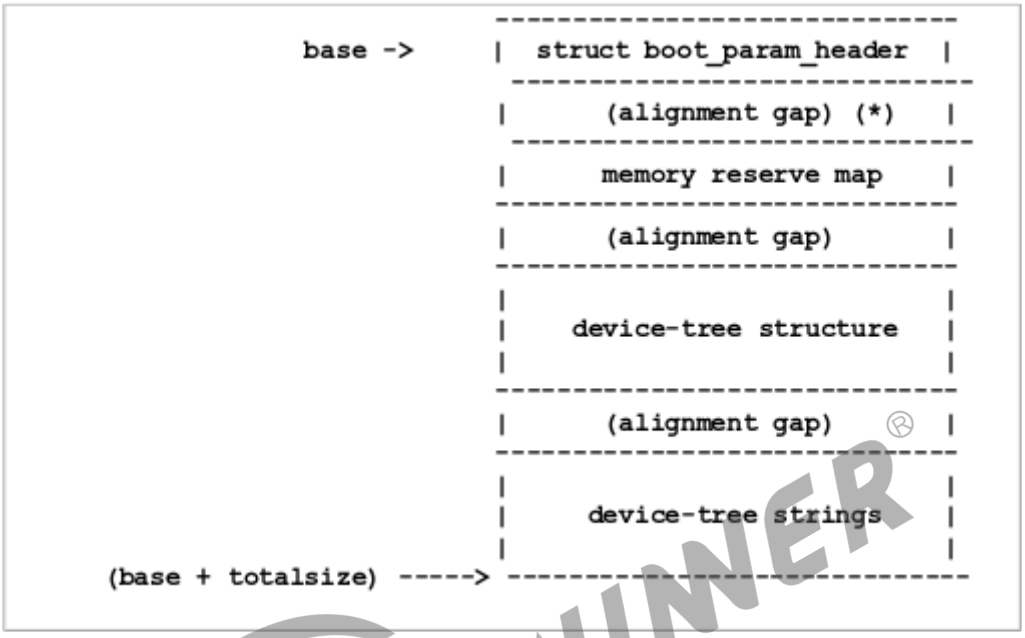

图 4-6: dtb 内存布局

4.3.3.1 文件头-boot_param_header

内核的物理指针指向的内存区域在 structure boot_param_header 这个结构体中大概描述到了：

```
include/linux/of_fdt.h

/* Definitions used by the flattened device tree */
#define OF_DT_HEADER        0xd00dfeed  /* marker */
#define OF_DT_BEGIN_NODE    0x1     /* Start of node, full name */
#define OF_DT_END_NODE      0x2     /* End node */
#define OF_DT_PROP      0x3        /* Property: name off, size,* content */
#define OF_DT_NOP       0x4          /* nop */
#define OF_DT_END       0x9
#define OF_DT_VERSION       0x10

struct boot_param_header {
    __be32  magic;                  /* magic word OF_DT_HEADER */
    __be32  totalsize;             /* total size of DT block */
    __be32  off_dt_struct;          /* offset to structure */
    __be32  off_dt_strings;         /* offset to strings */
    __be32  off_mem_rsvmap;     /* offset to memory reserve map */
    __be32  version;                    /* format version */
    __be32  last_comp_version;      /* last compatible version */
```

```
    /* version 2 fields below */
    __be32  boot_cpuid_phys;        /* Physical CPU id we're booting on */
    /* version 3 fields below */
    __be32  dt_strings_size;            /* size of the DT strings block */
    /* version 17 fields below */
    __be32  dt_struct_size;         /* size of the DT structure block */
};
```

具体这个结构体怎么用，在后边会有具体描述。

4.3.3.2 device-tree structure

这一部分主要存储了各个结点的信息。每一个结点都都可以嵌套子结点，其中的结点以 OF_DT_BEGIN_NODE 做起始标志，接下来就是结点名。如果结点带有属性，那么就紧接就是结点的属性，其以 OF_DT_PROP 为起始标志。嵌套的子结点紧跟着父子结点之后，也是以 OF_DT_BEGIN_NODE 起始。OF_DT_END_NODE 标志着一结点的终止。

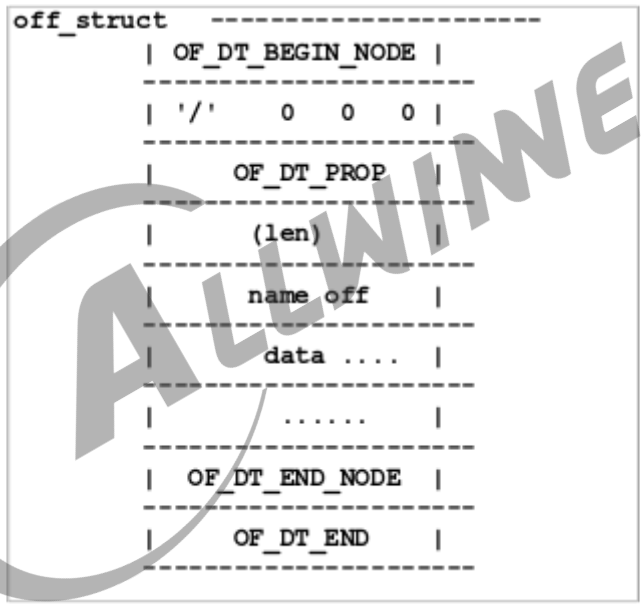

图 4-7: device-tree 的 structure 结构

上面提到一个结点的属性，每一个属性有如下的结构：

```
Scripts/dtc/libfdt/fdt.h
struct fdt_property {
        uint32_t tag;
        uint32_t len;
        uint32_t nameoff;
        char data[0];
};
```

4.3.3.3 Device tree string

最后一部分就是 String，没有固定格式。其主要是把一些公共的字符串线性排布，以节约空间。

4.3.3.4 dtb 实例

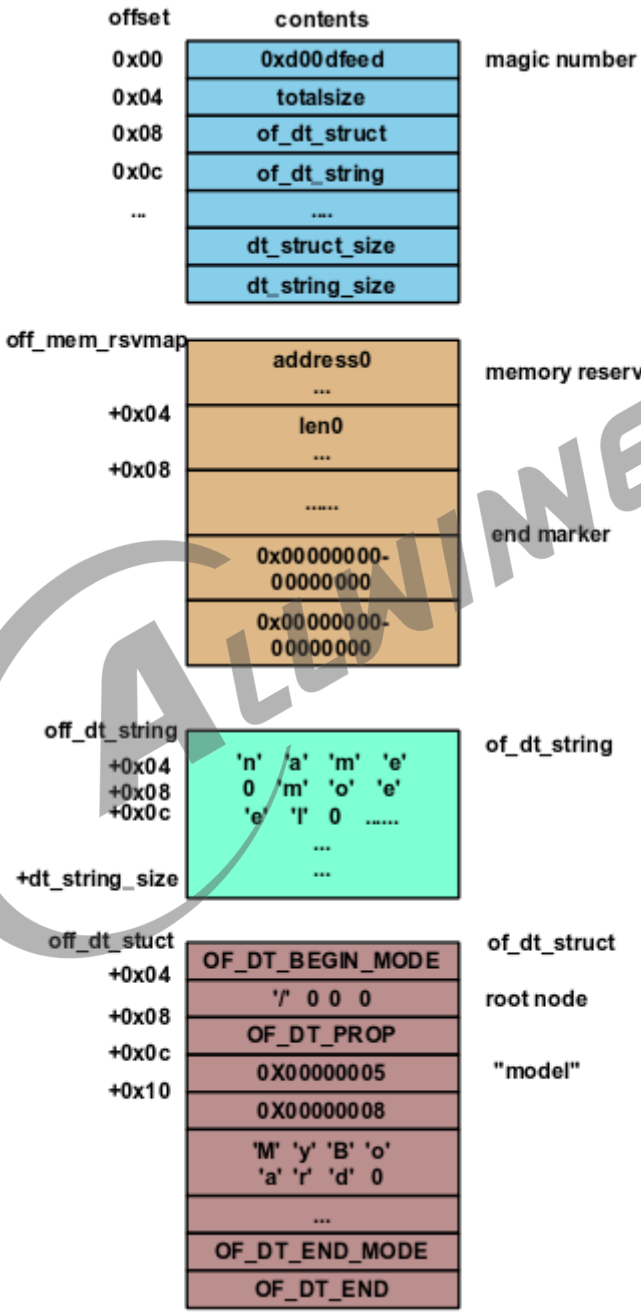

图 4-8: dtb 实例

可以看出 dtb 结构由 4 个部分组成。

memory reserve table：给出了 kernel 不能使用的内存区域列表。

Of_device-struct：结构包含了 device tree 的属性。每个节点以 OF_DT_BEGIN_NODE 标签开始，接着紧跟着节点的名称。如果节点有属性，那么紧跟着就是节点的属性，每个属性值以 OF_DT_PROP 标签开始，紧接着是嵌套在节点中的子节点，子节点也是以 OF_DT_BEGIN_NODE 起始，以 OF_DT_END_NODE 结束，最后以标签 OF_DT_END 标示根节点结束。

对每个属性，在标签 OF_DT_PROP 之后，由一个 32 位的数指明属性名称存放在偏移 of_dt_string 结构体起始地址多少 byte 的地方。之所以采用这种做法，是因为有很多节点都有很多相同的属性名称，比如 compatible、reg 等，这些节点的名称如果一个个存放起来，显然挺浪费空间的，采用一个偏移量，来指定它在 of_dt_string 的哪个地方，在 of_dt_string 中只需要保存一份属性值就可以了，有利于降低 block 占用的空间。

4.4 内核常用 API

4.4.1 of_device_is_compatible

原型

```
int of_device_is_compatible(const struct device_node *device,const char *compat);
```

函数作用

判断设备结点的 compatible 属性是否包含 compat 指定的字符串。当一个驱动支持 2 个或多个设备的时候，这些不同.dts 文件中设备的 compatible 属性都会进入驱动 OF 匹配表。因此驱动可以透过 Bootloader 传递给内核的 Device Tree 中的真正结点的 compatible 属性以确定究竟是哪一种设备，从而根据不同的设备类型进行不同的处理。

4.4.2 of_find_compatible_node

原型

```
struct device_node *of_find_compatible_node(struct device_node *from,
          const char *type, const char *compatible);
```

函数作用

根据 compatible 属性，获得设备结点。遍历 Device Tree 中所有的设备结点，看看哪个结点的类型、compatible 属性与本函数的输入参数匹配，大多数情况下，from、type 为 NULL。

4.4.3 of_property_read_u32_array

原型

```
int of_property_read_u8_array(const struct device_node *np,
          const char *propname, u8 *out_values, size_t sz);
int of_property_read_u16_array(const struct device_node *np,
          const char *propname, u16 *out_values, size_t sz);
int of_property_read_u32_array(const struct device_node *np,
          const char *propname, u32 *out_values, size_t sz);
int of_property_read_u64(const struct device_node *np,
          const char *propname, u64 *out_value);
```

函数作用

读取设备结点 np 的属性名为 propname，类型为 8、16、32、64 位整型数组的属性。对于 32 位处理器来讲，最常用的是 of_property_read_u32_array()。

4.4.4 of_property_read_string

原型

```
int of_property_read_string(struct device_node *np,
          const char *propname, const char **out_string);
int of_property_read_string_index(struct device_node *np,
          const char *propname, int index, const char **output);
```

函数作用

前者读取字符串属性，后者读取字符串数组属性中的第 index 个字符串。

4.4.5 bool of_property_read_bool

原型

```
static inline bool of_property_read_bool(const struct device_node *np,
              const char *propname);
```

函数作用

如果设备结点 np 含有 propname 属性，则返回 true，否则返回 false。一般用于检查空属性是否存在。

4.4.6 of_iomap

原型

```
void __iomem *of_iomap(struct device_node *node, int index);
```

函数作用

通过设备结点直接进行设备内存区间的 ioremap()，index 是内存段的索引。若设备结点的 reg 属性有多段，可通过 index 标示要 ioremap 的是哪一段，只有 1 段的情况，index 为 0。采用 Device Tree 后，大量的设备驱动通过 of_iomap() 进行映射，而不再通过传统的 ioremap。

4.4.7 irq_of_parse_and_map

原型

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index);
```

函数作用

透过 Device Tree 或者设备的中断号，实际上是从.dts 中的 interrupts 属性解析出中断号。若设备使用了多个中断，index 指定中断的索引号。

4.5 Device tree 配置 demo

以 pinctrl 为例：

```
soc@01c20000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        pio: pinctrl@01c20800 {
            compatible = "allwinner,sun50i-pinctrl";
            reg = <0x01c20800 0x400>;
            interrupts = <0 11 1>, <0 15 1>, <0 16 1>, <0 17 1>;
            clocks = <&apb1_gates 5>;
            gpio-controller;
            interrupt-controller;
            #address-cells = <1>;
            #size-cells = <0>;
            #gpio-cells = <6>;
            uart0_pins_a: uart0@0 {
                allwinner,pins = "PH20", "PH21";    //设备需要用到的pin
                allwinner,function = "uart0";        //复用名字
                allwinner,drive = <0>;              //设置驱动力
                allwinner,pull = <0>;               //设置上下拉
                allwinner,data=<0>;                //设置数据属性
```

```
            };
        };
};
```

5 设备树使用

5.1 引言

5.1.1 编写目的

介绍 Device Tree 配置、设备驱动如何获取 Device Tree 配置信息等内容，让用户明确掌握 Device Tree 配置与使用方法。

5.1.2 术语与缩略语

术语/缩略语	解释说明
DTS	Device Tree Source File，设备树源码文件
DTB	Device Tree Blob File，设备树二进制文件
sys_config.fex	Allwinner 配置文件

5.2 模块介绍

Device Tree 是一种描述硬件的数据结构，可以把嵌入式系统资源抽象成一颗树形结构，可以直观查看系统资源分布；内核可以识别这棵树，并根据它展开出 Linux 内核中的 platform_device 等。

5.2.1 模块功能介绍

Device Tree 改变了原来用 hardcode 方式将 HW 配置信息嵌入到内核代码的方法，消除了 arch/arm64 下大量的冗余编码。使得各个厂商可以更专注于 driver 开发，开发流程遵从 mainline kernel 的规范。

5.2.2 相关术语介绍

术语/缩略语	解释说明
FDT	嵌入式 PowerPC 中，为了适应内核发展 && 嵌入式 PowerPC 平台的千变万化，推出了 Standard for Embedded Power Architecture Platform Requirements（ePAPR）标准，吸收了 Open Firmware 的优点，在 U-boot 引入了扁平设备树 FDT 接口，使用一个单独的 FDT blob 对象将系统硬件信息传递给内核。
DTS	device tree 源文件，包含用户配置信息。对于 32bit Arm 架构，dts 文件存放在 arch/arm/boot/dts 路径下。对于 64bit Arm 架构，dts 文件存放在 arch/arm64/boot/dts 路径下。对于 Tina3.5.1 及之后版本，会使用 device 目录下的 board.dts，即 device/config/chips/*chip/configs/*{borad}/board.dts。
DTB	DTS 被 DTC 编译后二进制格式的 Device Tree 描述，可由 Linux 内核解析，并为设备驱动提供硬件配置信息。

5.3 如何配置

5.3.1 配置文件位置

设备树文件，存放在具体内核的目录下。

- ARMv7 架构下，dts 文件放置在内核的 arch/arm/boot/dts/目录。
- ARMv8 架构下，dts 文件放置在内核的 arch/arm64/boot/dts/目录。
- RISCV 架构下，dts 文件放置在内核的 arch/riscv/boot/dts/目录。

对于 Tina3.5.1 及之后版本，会使用 device 目录下的 board.dts，即：

```
device/config/chips/${chip}/configs/${borad}/board.dts
```

lunch 选择具体方案后，可以使用快捷命令跳到该目录：

```
cdts
```

要确认具体的设备树，首先确定 chip，在 lunch 选择方案之后，会有打印：

```
TARGET_CHIP=xxx
```

如，TARGET_CHIP=sun8iw18p1，则使用 sun8iw18p1 开头的配置文件。

对应 CHIP 开头的设备树有多份。编译内核的时候都会进行编译，生成中间文件。在打包的时候，才结合 sys_config，生成最终的设备树。

在打包脚本，即 scripts/pack_img.sh 中，do_ini_to_dts 函数进行相关的处理。

对于 Tina3.5.0 及之前版本，如果有方案同名的设备树，则优先使用方案同名的设备树。没有的话，则使用通用的设备树。

如 cowbell-perf1 方案，会先找 sun8iw18p1-cowbell-perf1.dts，如果找不到，就使用 sun8iw18p1-soc.dts。

对于 Tina3.5.1 及之后版本，会使用 device 目录下的 board.dts，即：

```
device/config/chips/${chip}/configs/${borad}/board.dts
```

5.3.2 配置文件关系

5.3.2.1 不存在 sys_config.fex 配置情况

当不存在 sys_config.fex 时，一份完整的配置可以包括两个部分 (以 sun50iw1p1 平台为例)：

- soc 级配置文件：定义了 SOC 级配置，如设备时钟、中断等资源，如图 sun50iw1p1.dtsi[1]。
- board 级配置文件：定义了板级配置，包含一些板级差异信息，如图 sun50iw1p1-t1.dtsi。

示例如图：

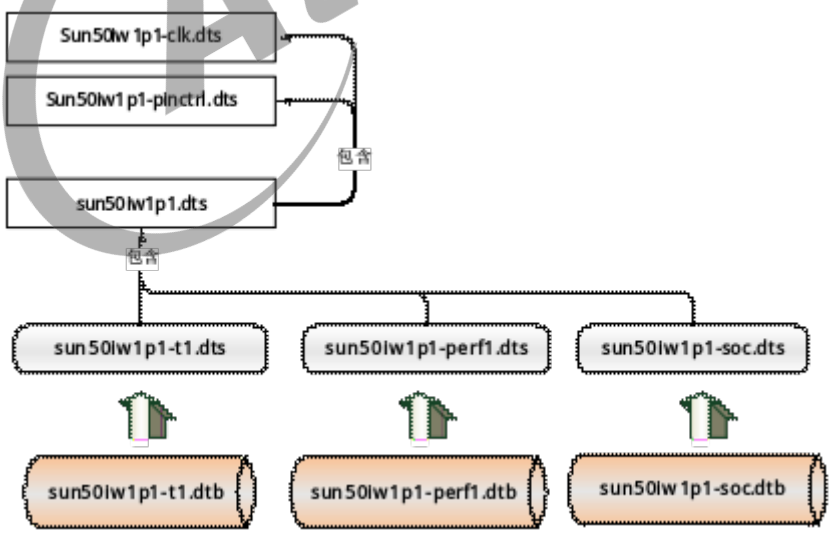

图 5-1：不存在 sysconfig 的配置结构

说明

注，对应图中为 *sun50iw1p1.dts*，*.dts* 与*.dtsi* 的区别在于 *dtsi* 是 *dts* 公共部分的提炼，应用时 *dts* 包含 *dtsi* 即可。在本文中并不作严格区分。

上图以 R18 为例子，展示了三个方案的设备树配置信息，其中：

- 每个方案 dtb 文件，依赖于 sun50iw1p1-*board.dtssun50iw1p1*−{board}.dtsi 又包含 sun50iw1p1.dtsi，当 board 级配置文件跟 soc 级配置文件出现相同节点的同名属性时，Board 级配置文件的属性值会去覆盖 soc 级的相同属性值。
- 图示中 sun50iw1p1-soc.dts 文件跟 sun50iw1p1-t1.dts 与 sun50iw1p1-perf1.dts 一样，都属于 board 配置文件。该配置文件定义为一种通用的 board 配置文件，主要为了防止客户移植新的方案时，没有在内核 linux-3.10/arch/arm64/boot/dts/目录下定义客户方案的 board 级配置文件。如果出现这样的情况，内核编译的时候，就会采用 sun50iw1p1-soc.dts，作为该客户方案的 board 级配置文件。注：Tina3.5.1 及之后的版本，方案 dts 配置，迁移到了 device/config/chips/*chip*/*configs*/{board}/board.dts

5.3.2.2 存在 sys_config.fex 配置情况

当存在 sys_config.fex 时，一份完整的配置可以包括三个部分 (以 sun50iw1p1 平台为例)：

- soc 级配置文件：定义了 SOC 级配置，如设备时钟、中断等资源，如图 sun50iw1p1.dtsi。
- board 级配置文件：定义了板级配置，包含一些板级差异信息，如图 sun50iw1p1-t1.dtsi。
- sys_config.fex 配置文件，为方便客户使用而定义，优先级比 board 级配置、soc 级配置都高。

示例如图：

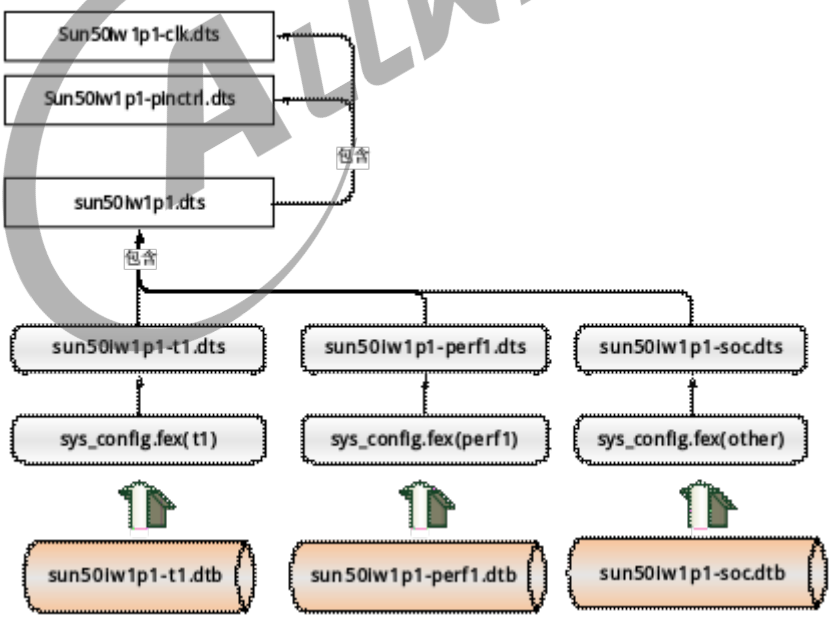

图 5-2: 存在 sysconfig 的配置结构

以 R18 为例子，上图展示了三个方案的设备树配置信息，其中：

- 每个方案 dtb 文件，既包含 sys_config.fex 配置信息，同时又依赖于 sun50iw1p1-*board.dtssun50iw1p1*−{board}.dtsi 又包含 sun50iw1p1.dtsi，sys_config.fex 配置文件的

优先级别最高，sys_config.fex 跟 devicetree 文件都存在配置项时，sys_config.fex 的配置项内容会更新到 board 级配置文件或者 soc 级配置文件对应的配置项上去。

- 图示中 sun50iw1p1-soc.dts 文件跟 sun50iw1p1-t1.dts 与 sun50iw1p1-perf1.dts 一样，都属于 board 配置文件。该配置文件定义为一种通用的 board 配置文件，主要为了防止客户移植新的方案时，没有在内核 linux-4.4/arch/arm64/boot/dts/sunxi/目录下定义客户方案的 board 级配置文件。如果出现这样的情况，内核编译的时候，就会采用 sun50iw1p1-soc.dts，作为该客户方案的 board 级配置文件。注：Tina3.5.1 及之后的版本，方案 dts 配置，迁移到了 device/config/chips/*chip*/*configs*/{board}/board.dts

5.3.2.3 soc 级配置文件与 board 级配置文件

soc 级配置文件与 board 级配置文件都是 dts 配置文件，对于相同设备节点的描述可能存在重合关系。因此，需要对重合的部分采取合并或覆盖的特殊处理，我们一般考虑两种情况：

- 1、一般地，soc 级配置文件保存公共配置，board 级配置文件保存差异化配置，如果公共配置不完善或需要变更，则一般需要通过 board 级配置文件修改补充，那么只需在 board 级配置文件中创建相同的路径的节点，补充差异配置即可。此时采取的合并规则是：两个配置文件中不同的属性都保留到最终的配置文件，即合并不同属性配置项；相同的属性，则优先选取 board 级配置文件中属性值保留，即 board 覆盖 soc 级相同属性配置项。如下：

```
soc级定义：
/ {
    soc {
        thermal-zones {
            xxx {
                aaa = "1";
                bbb = "2";
            }
        }
    }
}

board级定义：
/ {
    soc {
        thermal-zones {
            xxx {
                aaa = "3";
                ccc = "4";
            }
        }
    }
}

最终生成：
/ {
    soc {
        thermal-zones {
            xxx {
                aaa = "3";
```

```
                bbb = "2";
                ccc = "4";
            }
        }
    }
}
```

- 2、如果 soc 级保存的公共配置无法满足部分方案的特殊要求，且使用这项公共配置的其他方案众多，直接修改难度较大。那么我们考虑在 board 级配置文件中，使用/delete-node/语句删除 soc 级的配置，并重新定义。如下：

```
soc级定义：
/ {
    soc {
        thermal-zones {
            xxx {
                aaa = "1";
                bbb = "2";
            }
        }
    }
}

board级定义：
/ {
    soc {
        /delete-node/ thermal-zones;
        thermal-zones {
            xxx {
                aaa = "3";
                ccc = "4";
            }
        }
    }
}

最终生成：
/ {
    soc {
        thermal-zones {
            xxx {
                aaa = "3";
                ccc = "4";
            }
        }
    }
}
```

删除节点的语法如下：

```
/delete-node/ 节点名;
```

需要注意的是注意：（1）/delete-node/与节点名之间有空格。

（2）如果节点中有地址信息，节点名后也需要加上。

删除属性的语法如下：

```
/delete-property/ 属性名;
```

5.3.3 配置 sys_config.fex

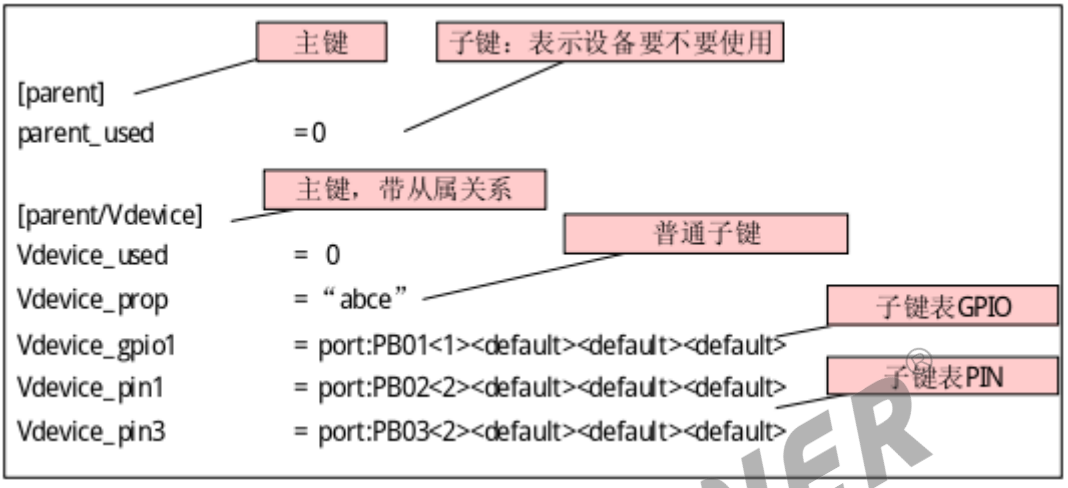

图 5-3: sysconfig 配置

5.3.4 配置 device tree

```
Vdevice: vdevice@0{                    (详见(1)): (详见(2))
    compatible=” allwinner,sun50i-vdevice” ;
    device_type=” Vdevice” ;                (详见(3))
    pinctrl_name=” default” ;               (详见(4))
    pinctrl_0=<&vdevice_pins_a>
    test_prop=” adb”                    (详见(5))
    status=” okay”                      (详见(6))
};
```

详注：

(1) label，此处名字必须与 sys_config.fex 主键一致。

(2) 节点名字。

(3) 特定属性表示设备类型，必须与 label 一致。

(4) 特定属性，用来 PIN 配置。

(5) 普通属性。

(6) 特定属性，用来表示设备是否使用。

5.4 接口描述

Linux 系统为 device tree 提供了标准的 API 接口。

5.4.1 常用外部接口

使用内核提供的 device tree 接口，必须引用 Linux 系统提供的 device tree 接口头文件，包含且不限于以下头文件：

```
#include<linux/of.h>
#include<linux/of_address.h>
#include<linux/of_irq.h>
#include<linux/of_gpio.h>
```

5.4.1.1 irq_of_parse_and_map

类别	介绍
函数原型	unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
参数	dev：要解析中断号的设备；index：dts 源文件中节点 interrupt 属性值索引；
返回	如果解析成功，返回中断号，否则返回 0。

DEMO：

```
以timer节点为例子：
Dts配置：
/{
     timer0: timer@1c20c00 {
          ...

          interrupts = <GIC_SPI 18 IRQ_TYPE_EDGE_RISING>;
          ...
     };
};

驱动代码片段：
static void __init sunxi_timer_init(struct device_node *node){
     int irq;
          ....
     irq = irq_of_parse_and_map(node, 0);
        if (irq <= 0)
            panic("Can't parse IRQ");
}
```

5.4.1.2 of_iomap

类别	介绍
函数原型	void __iomem *of_iomap(struct device_node *np, int index);
参数	np：要映射内存的设备节点，index：dts 源文件中节点 reg 属性值索引；
返回	如果映射成功，返回 IO memory 的虚拟地址，否则返回 NULL。

DEMO：

```
以timer节点为例子，dts配置：
/{
     timer0: timer@1c20c00 {
         ...
         reg = <0x0 0x01c20c00 0x0 0x90>;
         ...
     };
};

以timer为例子，驱动代码片段：
static void __init sunxi_timer_init(struct device_node *node){
     ...
    timer_base = of_iomap(node, 0);
}
```

5.4.1.3 of_property_read_u32

类别	介绍
函数原型	static inline int of_property_read_u32(const struct device_node *np, const char *propname, u32 *out_value)
参数	np：想要获取属性值的节点; propname：属性名称; out_value：属性值
返回	如果取值成功，返回 0。

DEMO：

```
//以timer节点为例子，dts配置例子：
/{
        soc_timer0: timer@1c20c00 {
            clock-frequency = <24000000>;
            timer-prescale = <16>;
```

```
        };
};
//以timer节点为例子，驱动中获取clock-frequency属性值的例子:
int rate=0;
if (of_property_read_u32(node, "clock-frequency", &rate)) {
    pr_err("<%s> must have a clock-frequency property\n",node->name);
    return;
}
```

5.4.1.4 of_property_read_string

类别	介绍
函数原型	static inline int of_property_read_string(struct device_node *np, const char *propname, const char **output)
参数	np：想要获取属性值的节点；propname：属性名称；output：用来存放返回字符串
返回	如果取值成功，返回 0
功能描述	该函数用于获取节点中属性值。（针对属性值为字符串）

DEMO:

```
//例如获取string-prop的属性值， Dts配置:
/{
    soc@01c20800{
            vdevice: vdevice@0{
                ...
                   string_prop = "abcd";
            };
    };
};
例示代码:
test{
    const char *name;
    ....
    err = of_property_read_string(np, "string_prop", &name);
    if (WARN_ON(err))
            return;
}
```

5.4.1.5 of_property_read_string_index

类别	介绍
函数原型	static inline int of_property_read_string_index(struct device_node *np, const char *propname,int index, const char **output)
参数	np：想要获取属性值的节点 Propname：属性名称; Index：用来索引配置在 dts 中属性为 propname 的值。Output：用来存放返回字符串
返回	如果取值成功，返回 0。
功能描述	该函数用于获取节点中属性值。（针对属性值为字符串）。

DEMO：

```
//例如获取string-prop的属性值， Dts配置：
/{
    soc@01c20800{
            vdevice: vdevice@0{
                ...
                   string_prop = "abcd";
            };
    };
};
例示代码：
test{
    const char *name;
    ....
    err = of_property_read_string_index(np, "string_prop", 0, &name);
    if (WARN_ON(err))
            return;
}
```

5.4.1.6 of_find_node_by_name

类别	介绍
函数原型	extern struct device_node *of_find_node_by_name(struct device_node *from, const char *name);
参数	clk：待操作的时钟句柄；From：从哪个节点开始找起 Name：想要查找节点的名字
返回	如果成功，返回节点结构体，失败返回 null。
功能描述	该函数用于获取指定名称的节点。

DEMO：

```
//获取名字为vdeivce的节点， dts配置
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
            string_prop = "abcd";
        };
    };
};

例示代码片段:
test{
     struct device_node *node;
     ....
        node = of_find_node_by_name(NULL, "vdevice");
        if (!node){
          pr_warn("can not get node.\n");

    };
       of_node_put(node);
}
```

5.4.1.7 of_find_node_by_type

类别	介绍
函数原型	extern struct device_node *of_find_node_by_name(struct device_node *from, const char *type);
参数	clk：待操作的时钟句柄；From：从哪个节点开始找起 type：想要查找节点中 device_type 包含的字符串
返回	如果成功，返回节点结构体，失败返回 null。
功能描述	该函数用于获取指定 device_type 的节点。

DEMO：

```
//获取名字为vdeivce的节点， dts配置。
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
         device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};

例示代码片段:
test{
     struct device_node *node;
     ....
        node = of_find_node_by_type(NULL, "vdevice");
```

```
        if (!node){
          pr_warn("can not get node.\n");
    };
       of_node_put(node);
}
```

5.4.1.8 of_find_node_by_path

类别	介绍
函数原型	extern struct device_node *of_find_node_by_path(const char *path);
参数	path：通过指定路径查找节点；
返回	如果成功，返回节点结构体，失败返回 null。
功能描述	该函数用于获取指定路径的节点。

DEMO：

```
//获取名字为vdeivce的节点， dts配置。
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
         device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};

例示代码片段：
test{
     struct device_node *node;
     ....
        node = of_find_node_by_path("/soc@01c2000/vdevice@0");
        if (!node){
          pr_warn("can not get node.\n");
    };
       of_node_put(node);
}
```

5.4.1.9 of_get_named_gpio_flags

类别	介绍
函数原型	int of_get_named_gpio_flags(struct device_node *np, const char *propname, int index, enum of_gpio_flags *flags)

类别	介绍
参数	np：包含所需要查找 GPIO 的节点 propname：包含 GPIO 信息的属性 Index：属性 propname 中属性值的索引 Flags：用来存放 gpio 的 flags
返回	如果成功，返回 gpio 编号，flags 存放 gpio 配置信息，失败返回 null。
功能描述	该函数用于获取指定名称的 gpio 信息。

DEMO：

```
//获取名字为vdeivce的节点， dts配置。
/{
    soc@01c20800{
        vdevice: vdevice@0{
            ...
         device_type = "vdevice";
            string_prop = "abcd";
        };
    };
};

例示代码片段：
test{
     struct device_node *node;
     ....
        node = of_find_node_by_path("/soc@01c2000/vdevice@0");
        if (!node){
          pr_warn("can not get node.\n");
    };
       of_node_put(node);
}

/{
        soc@01c20800{
                vdevice: vdevice@0{
                    ...
                        test-gpios=<&pio PA 1 1 1 1 0>;
                };
        };
 };

static int gpio_test(struct platform_device *pdev)
{
          struct gpio_config config;
       ....
          node=of_find_node_by_type(NULL, "vdevice");
          if(!node){
                 printk(" can not find node\n");
      }
        ret = of_get_named_gpio_flags(node, "test-gpios", 0, (enum of_gpio_flags *)&config)
    ;
        if (!gpio_is_valid(ret)) {
              return -EINVAL;
         }
};
```

5.4.2 sys_config 接口 &&dts 接口映射

5.4.2.1 获取子键内容

script API：

原型	script_item_value_type_e script_get_item(char *main_key, char *sub_key, script_item_u *item);
作用	通过主键名和子键名字，获取子键内容（该接口可以自己识别子键的类型）。

dts API：

说明	dts 标准接口支持通过节点和属性名，获取属性值（用户需要知道属性值得类型）
原型	int of_property_read_u32(const struct device_node *np, const char *propname, u32 *out_value)
作用	获取属性值，使用于属性值为整型数据。
原型	int of_property_read_string(struct device_node *np, const char *propname, const char **out_string)
作用	获取属性值，使用于属性值为字符串。
原型	int of_get_named_gpio_flags(struct device_node *np, const char *list_name, int index, enum of_gpio_flags *flags)
作用	获取 GPIO 信息。

5.4.2.2 获取主键下 GPIO 列表

script API：

原型	int script_get_pio_list(char *main_key, script_item_u **list);
作用	获取主键下 GPIO 列表。

dts API：

说明无对应接口。

5.4.2.3 获取主键数量

script API：

原型	unsigned int script_get_main_key_count(void);
作用	获取主键数量。

dts API：

说明无对应接口。

5.4.2.4 获取主键名称

script API：

原型	char *script_get_main_key_name(unsigned int main_key_index);
作用	通过主键索引号，获取主键名字。

dts API：

说明无对应接口。

5.4.2.5 判断主键是否存在

script API：

原型	bool script_is_main_key_exist(char *main_key);
作用	判断主键是否存在。

dts API：

说明

dts 标准接口支持四种方式判断节点是否存在。

原型	struct device_node *of_find_node_by_name(struct device_node *from, const char *name)
作用	通过节点名字。
原型	struct device_node *of_find_node_by_path(const char *path)
作用	通过节点路径。
原型	struct device_node *of_find_node_by_phandle(phandle handle)
作用	通过节点 phandle 属性。
原型	struct device_node *of_find_node_by_type(struct device_node *from, const char *type)
作用	通过节点 device_type 属性。

5.5 接口使用例子

5.5.1 配置比较

下表展示了设备 vdevice 在 sys_config.fex 与 dts 中的配置，两种配置形式不一样，但实现的功能是等价的。

dts：

```
/*
device config in dts:
/{
    soc@01c20000{
        vdevice@0{
            compatible = "allwinner,sun50i-vdevice";
            device_type= "vdevice";
            vdevice_0=<&pio 1 1 1 1 1 0>;
            vdevice_1=<&pio 1 2 1 1 1 0>;
            vdevice-prop-1=<0x1234>;
            vdevice-prop-3="device-string";
            status = "okay";
        };
    };
};
```

sys_config.fex：

```
device config in sys_config.fex
[vdevice]
compatible          = "allwinner,sun50i-vdevice";
vdevice_used        = 1
vdevice_0           = port:PB01<1><1><2><default>
vdevice_1           = port:PB02<1><1><2><default>
vdevice-prop-1       = 0x1234
```

```
vdevice-prop-3       = "device-string"
*/
```

说明：GPIO_IN/GPIO_OUT/EINT 采用下边的配置方式，PIN 采用另外配置，参考 pinctrl 使用说明文档。

```
      vdevice_0=<&pio 1 1 1 1 1 0>;
         |         |  | | | | | |-------------------电平
         |         |  | | | | |-----------------------上下拉
         |         |  | | | |-------------------------驱动力
         |         |  | | |---------------------------复用类型，0-GPIOIN 1-GPIOOUT..
         |         |  | |-----------------------------pin bank内偏移.
         |         |  |-------------------------------哪个bank，PA=0， PB=1...以此类推
         |         |-------------------------------------指向哪个pio，属于cpus要用&r_pio
         |-----------------------------------------------------属性名字，相当sys_config子键名
```

5.5.2 获取整形属性值

通过 script 接口：

```
#include <linux/sys_config.h>
int get_subkey_value_int(void)
{
    script_item_u script_val;
    script_item_value_type_e type;

    type = script_get_item("vdevice", "vdevice-prop-1", &script_val);
    if (SCIRPT_ITEM_VALUE_TYPE_INT != type) {
        return -EINVAL;
    }
    return 0;

}
```

通过 dts 接口：

```
#include <linux/of.h>
int get_subkey_value_int(void)
{
    int ret;
    u32 value;
    struct device_node *node;

    node = of_find_node_by_type(NULL,"vdevice");
    if(!node){
        return -EINVAL;
    }
    ret = of_property_read_u32(node, "vdevice-prop-1", &value);
    if(ret){
        return -EINVAL;
    }
    printk("prop-value=%x\n", value);

    return 0;
```

```
}
```

5.5.3 获取字符型属性值

通过 script 接口：

```
#include <linux/sys_config.h>
int get_subkey_value_string(void)
{
    script_item_u script_val;
    script_item_value_type_e type;

    type = script_get_item("vdevice", "vdevice-prop-3", &script_val);
    if (SCIRPT_ITEM_VALUE_TYPE_STR!= type) {
        return -EINVAL;
    }
    return 0;
}
```

通过 dts 接口：

```
#include <linux/of.h>
int get_subkey_value_string(void)
{
    int ret;
    const char *string;
    struct device_node *node;

    node = of_find_node_by_type(NULL,"vdevice");
    if(!node){
        return -EINVAL;
    }
    ret = of_property_read_string(node, "vdevice-prop-3", &string);
    if(ret){
        return -EINVAL;
    }
    printk("prop-vlalue=%s\n", string);

    return 0;

}
```

5.5.4 获取 gpio 属性值

通过 script 接口：

```
#include <linux/sys_config.h>
int get_gpio_info(void)
{
    script_item_u script_val;
```

```
    script_item_value_type_e type;

    type = script_get_item("vdevice", "vdevice_0", &script_val);
    if (SCIRPT_ITEM_VALUE_TYPE_PIO!= type) {
        return -EINVAL;
    }
    return 0;

}
```

通过 dts 接口：

```
#include <linux/sys_config.h>
#include <linux/of.h>
#include <linux/of_gpio.h>
int get_gpio_info(void)
{
    unsigned int gpio;
    struct gpio_config config;
    struct device_node *node;

    node = of_find_node_by_name(NULL,"vdevice");
    if(!node){
        return -EINVAL;
    }
    gpio = of_get_named_gpio_flags(node, "vdevice_0", 0, (enum of_gpio_flags *)&config);
    if (!gpio_is_valid(gpio)) {
        return -EINVAL;
    }
    printk("pin=%d  mul-sel=%d  drive=%d  pull=%d  data=%d gpio=%d\n",
            config.gpio,
            config.mul_sel,
            config.drv_level,
            config.pull,
            config.data,
            gpio);
    return 0;
}
```

5.5.5 获取节点

通过 scritp 接口：

```
#include <linux/sys_config.h>
int check_mainkey_exist(void)
{
    int ret;
    ret = script_is_main_key_exist("vdevice");
    if(!ret){
        return -EINVAL;
    }
}
```

通过 dts 接口：

```
int check_mainkey_exist(void)
{
    struct device_node *node_1, *node_2;
    /* mode 1*/
    node_1 = of_find_node_by_name(NULL,"vdevice");
    if(!node_1){
        printk("can not find node in dts\n");
        return -EINVAL;
    }
    /*mode 2 */
    node_2 =  of_find_node_by_type(NULL,"vdevice");
If(!node_2){
        return -EINVAL;
    }
    return 0;

}
```

5.6 其他

5.6.1 sysfs 设备节点

device tree 会解析 dtb 文件中，并在/sys/devices 目录下会生成对应设备节点，其节点命名规则如下：

5.6.1.1 “单元地址. 节点名”

节点名的结构是“单元地址. 节点名”，例如 1c28000.uart、1f01400.prcm。

形成这种节点名的设备，在 device tree 里的节点配置具有 reg 属性。

```
    uart0: uart@01c28000 {
        compatible = "allwinner,sun50i-uart";
        reg = <0x0 0x01c28000 0x0 0x400>;
        ......
    };

    prcm {
        compatible = "allwinner,prcm";
        reg = <0x0 0x01f01400 0x0 0x400>;
    };
```

5.6.1.2 “节点名. 编号”

节点名的结构是“节点名. 编号”，例如 soc.0、usbc0.5。

形成这种节点名的设备，在 device tree 里的节点配置没有 reg 属性。

```
soc: soc@01c00000 {
    compatible = "simple-bus";
    ......
};

usbc0:usbc0@0 {
    compatible  = "allwinner,sunxi-otg-manager";
    ......
};
```

编号是按照在 device tree 中的出现顺序从 0 开始编号，每扫描到这样一个节点，编号就增加 1，如 soc 节点是第 1 个出现的，所以编号是 0，而 usbc0 是第 6 个出现的，所以编号是 5。

device tree 之所以这么做，是因为 device tree 中允许配置同名节点，所以需要通过单元地址或者编号来区分这些同名节点。可以参见内核的具体实现代码：

```
arm64_device_init()
    ->of_platform_populate()
        ->of_platform_bus_create()
            ->of_platform_device_create_pdata()
                ->of_device_alloc()
                    ->of_device_make_bus_id()
of_device_make_bus_id()
{
    ......
    reg = of_get_property(node, "reg", NULL);
    if (reg) {
    ......
    dev_set_name(dev, "%llx.%s", (unsigned long long)addr, node->name);
    return;
    }

    magic = atomic_add_return(1, &bus_no_reg_magic);
    dev_set_name(dev, "%s.%d", node->name, magic - 1);
}
```

6 设备树调试

介绍在不同阶段 Device Tree 配置信息查看方式。

6.1 测试环境

Kernel Menuconfig 配置：

```
Device Drivers-->
Device Tree and Open Firmware support-->
Suppot for device tree in /proc
```

6.2 Build 阶段

6.2.1 输出文件描述

用户在 tina 根目录下执行 make 命令，执行编译固件的动作，该动作会生成 device tree 的二进制文件。

这些文件会存放到：

```
out/<方案名字>/compile_dir/target/linux-<方案名字>/<内核版本>/arch/arm/boot/dts
```

以 sitar 方案为例：

```
(R6平台)
    1.sun3iw1p1-sitar-soc.dtb
    2.sun3iw1p1-sitar-pd4.dtb
    3.sun3iw1p1-sitar-mic.dtb
    3.sun3iiw1p1-sitar-cuckoo.dtb
```

这些文件存放在：out/ sitar-{board} /compile_dir/target/linux-sitar-{board} /linux-3.10.65+/arch/arm/boot/dts 路径下边

这个阶段的 dtb 文件只包含 linux-3.10.65+/arch/arm/boot/dts 的配置信息，不包括 sys_config.fex 配置信息。

6.2.2 配置信息查看

查看 dtb 配置信息的方法（在 tina 根目录下输入）：

以 R6 平台为例：

```
./lichee/linux-3.10/scripts/dtc/dtc -I dtb -O dts -o output_sunxi.dts \
./out/sitar -{board} /compile_dir/target/linux-sitar –{board} /linux-3.10.65+/arch/arm/boot
    /dts /sun3iw1p1-soc.dtb
```

解析出来的 output_sunxi.dts 文件包含的信息，跟 lichee/linux-3.10/arch/arm/boot/dts 的配置信息完全一致。

6.3 Pack 阶段

6.3.1 输出文件描述

用户在 tina 根目录下执行 pack 命令，执行打包生成固件的动作，该动作会生成 dtb 文件：

以 R6 平台为例：

sunxi.dtb 存放在 out/sitar-{board}/image。

这个阶段 sunxi.dtb 文件包含 lichee/linux-3.10/arch/arm/boot/dts/的配置信息，还包含 sys_config.fex 配置信息。

6.3.2 配置信息查看

查看 dtb 配置信息的方法：

可以通过反编译，从最终的 dts 生成 dtb。

以 R6 平台为例：

```
./lichee/linux-3.10/scripts/dtc/dtc -I dtb -O dts -o output.dts ./out/sitar-${board} /image
    /sunxi.dtb
```

解析出来的 output.dts 文件包含的信息，跟 lichee/linux-3.10/arch/arm/boot/dts/的配置信息不完全一致，因为有些配置会被 sys_config.fex 更新。

默认 pack 的时候，会反编译一份，输出到：

```
out/方案/image/.sunxi.dtb
```

6.4 系统启动 boot 阶段

当 firmware 下载到 target device 之后，target device 启动到 uboot 的时候，也可以查看 dtb 配置信息。

在 uboot 的控制台输入：

```
fdt --help
```

可以看到 uboot 提供的可以查看、修改 dtb 的方法：

```
fdt - flattened device tree utility commands
Usage:
fdt addr [-c]  <addr> [<length>]   - Set the [control] fdt location to <addr>
fdt move   <fdt> <newaddr> <length> - Copy the fdt to <addr> and make it active
fdt resize                          - Resize fdt to size + padding to 4k addr
fdt print  <path> [<prop>]          - Recursive print starting at <path>
fdt list   <path> [<prop>]          - Print one level starting at <path>
fdt get value <var> <path> <prop>   - Get <property> and store in <var>
fdt get name <var> <path> <index>   - Get name of node <index> and store in <var>
fdt get addr <var> <path> <prop>    - Get start address of <property> and store in <var>
fdt get size <var> <path> [<prop>]  - Get size of [<property>] or num nodes and store in <
    var>
fdt set    <path> <prop> [<val>]    - Set <property> [to <val>]
fdt mknode <path> <node>            - Create a new node after <path>
fdt rm     <path> [<prop>]          - Delete the node or <property>
fdt header                          - Display header info
fdt bootcpu <id>                    - Set boot cpuid
fdt memory <addr> <size>            - Add/Update memory node
fdt rsvmem print                    - Show current mem reserves
fdt rsvmem add <addr> <size>        - Add a mem reserve
fdt rsvmem delete <index>           - Delete a mem reserves
fdt chosen [<start> <end>]          - Add/update the /chosen branch in the tree
                                        <start>/<end> - initrd start/end addr
fdt save                           - write fdt to flash
```

常用的比如：

```
fdt print                                       --打印整棵设备树。
fdt printf   /soc/vdevice                       --打印“/soc/vdevice”路径下的配置信息。
fdt set  /soc/vdevice  status  "disabled"       --设置“/soc/vdevice”下status属性的属性值。
fdt save                    --fdt set之后需要执行fdt save才能真正写入flash保存。
```

6.5 系统启动 kernel 阶段

内核配置了 CONFIG_PROC_DEVICETREE = y 之后，在/proc/device-tree 文件夹下的文件节点可以读取到 dtb 的配置信息。

说明

注：该文件节点下配置信息只能读不能写。

7 分区表

请参考，TinaLinux 存储管理开发指南。

8 env

8.1 配置文件路径

env.cfg 根据使用内核版本（linux3.4 或 linux3.10 或 linux4.4）分为 env-3.4.cfg，env-3.10.cfg，env-4.4.cfg，用于环境变量。

Tina 下的配置文件可能有几个路径。

Tina 3.5.0 及之前版本：

Tina 默认配置文件路径：

```
target/allwinner/generic/configs
```

芯片默认配置文件路径：

```
target/allwinner/xxx-common/configs
```

具体方案配置文件路径：

```
target/allwinner/xxx-xxx/configs
```

Tina 3.5.1 及之后版本：

芯片默认配置文件路径：

```
device/config/chips/${chip}/configs/default
```

具体方案配置文件路径：

```
device/config/chips/${chip}/configs/${borad}/linux
```

优先级依次递增，即优先使用具体方案下的配置文件，没有方案配置，则使用芯片默认配置文件，没有方案配置和芯片配置，才使用 tina 默认配置文件。

8.2 常用配置项说明

配置项	含义
bootdelay	串口选择是否进入 uboot 命令行的等待时间，单位秒。例如：为 0 时自动加载内核，为 3 则等待 3 秒，期间按任何按键都可进入 uboot 命令行。
bootcmd	默认为 run setargs_nand boot_normal，但 uboot 会根据实际介质正确修改 setargs_nand，称为 " update_bootcmd "。另外，在烧录固件时 bootcmd 会被修改成 run sunxi_sprite_test，即此时不会去加载内核，而是去执行烧录固件命令。
setargs_xxx	会去设置 bootargs、console、root、init、loglevel、partitions，这些都是内核需要用到的环境变量。其中 partitions 会根据分区进行自适应。
boot_normal	正常启动加载内核。
boot_recovery	正常启动加载恢复系统。
boot_fastboot	正常启动加载 fastboot。
console	设置内核的串口。
loglevel	设置内核的 log 级别。
verify	设置是否校验内核，默认会进行校验，设置为 =no 则不校验。
cmd	设置 cmd 内存大小。

8.3 uboot 中的修改方式

进入 uboot 命令行执行 env 相关命令可以查看，修改，保存 env 环境变量。常用的命令如：

```
env print                   --打印所有环境变量。
env set bootdelay 1         --设置 bootdelay 为1。
env save                    --保存环境变量， env set之后需要执行env save才能真正写入flash保存。
```

8.4 用户空间的修改方式

Tina 中提供了 uboot-envtools 软件包，选中即可：

```
make menuconfig --->    Utilities  --->  <*>uboot-envtools
```

可在用户空间，调用 fw_setenv 和 fw_printenv，对 env 进行读写。

fw_printenv 使用方法：

```
Usage: fw_printenv [OPTIONS]... [VARIABLE]...
Print variables from U-Boot environment

 -h, --help           print this help.
```

```
 -v, --version        display version
 -c, --config         configuration file, default:/etc/fw_env.config
 -n, --noheader       do not repeat variable name in output
 -l, --lock           lock node, default:/var/lock
```

fw_setenv 使用方法：

```
fw_setenv: option requires an argument -- 'h'
Usage: fw_setenv [OPTIONS]... [VARIABLE]...
Modify variables in U-Boot environment

 -h, --help           print this help.
 -v, --version        display version
 -c, --config         configuration file, default:/etc/fw_env.config
 -l, --lock           lock node, default:/var/lock
 -s, --script         batch mode to minimize writes

Examples:
  fw_setenv foo bar   set variable foo equal bar
  fw_setenv foo       clear variable foo
  fw_setenv --script file run batch script

Script Syntax:
  key [space] value
  lines starting with '#' are treated as comment

  A variable without value will be deleted. Any number of spaces are
  allowed between key and value. Space inside of the value is treated
  as part of the value itself.

Script Example:
  netdev          eth0
  kernel_addr     400000
  foo             empty empty empty    empty empty empty
  bar
```

9 nor/nand 介质配置

由于 spinor 一般容量远小于其他介质，为此分离出了独立的分区表 sys_partition_nor.fex 。打包时需要特殊处理，跟其他介质不兼容。即，可以用一个固件，兼容 nand 和 emmc，但不能兼容 spinor。

一般需要跟 spinor 互换的，是 spinand。

配置的目的是：

- 设置 sys_config.fex，方便打包时进行判断
- 选上适配的驱动，即 nor 要选 nor 的驱动，nand 要选 nand 的驱动
- 选上 UDISK 使用的文件系统支持，nor 默认使用 jffs2，nand 默认使用 ext4
- 选上对应的文件系统工具，nor 需要 mtd-utils，nand 需要 e2fsprogs

具体配置方法如下。

9.1 spinand 切换为 spinor

9.1.1 sys_config

设置介质为 nor：

```
[target]
storage_type = 3
```

配置所用 nor 的大小，如 16M：

```
[norflash]
size       = 16
```

9.1.2 内核配置

```
make kernel_menuconfig --->
    Device Drivers --->
        < >Block devices  (取消选中)
    Device Drivers --->
        <*>Memory Technology Device (MTD) support
            <*>OpenFirmware partitioning information support
            <*>SUNXI partitioning support
            <*>   Caching block device access to MTD devices
            <*>   SPI-NOR device support  (对于linux4.9，先选这个，下面的选项才出现)
            Self-contained MTD device drivers  --->
                 <*> Support most SPI Flash chips (AT26DF, M25P, W25X, ...)
    File systems  --->
        < > The Extended 4 (ext4) filesystem (取消选中)
    File systems  --->
        [*] Miscellaneous filesystems  --->
            <*>   Journalling Flash File System v2 (JFFS2) support (选中)
    [*] Enable the block layer  --->
        [ ]   Support for large (2TB+) block devices and files (取消选中)
```

9.1.3 menuconfig 配置

```
make menuconfig --->
    Utilities  --->
        <*> mtd-utils (选择) --->
            <*> mtd-utils-mkfs.jffs2

make menuconfig --->
    Utilities  --->
        Filesystem  --->
            < > e2fsprogs(取消选择)
```

9.2 spinor 切换为 spinand

9.2.1 sys_config

设置介质为 spinand：

```
[target]
storage_type = 5
```

9.2.2 内核配置

```
 make kernel_menuconfig --->
    Device Drivers --->
        [*]Block devices --->
            <*>   sunxi nand flash driver

make kernel_menuconfig --->
    Device Drivers --->
        < >Memory Technology Device (MTD) support（取消选择）

make kernel_menuconfig --->
    [*] Enable the block layer  --->
        [*]   Support for large (2TB+) block devices and files

make kernel_menuconfig --->
    File systems  --->
        <*> The Extended 4 (ext4) filesystem
```

9.2.3 menuconfig 配置

```
make menuconfig --->
    Utilities  --->
        < > mtd-utils (取消选择)

make menuconfig --->
    Utilities  --->
        Filesystem  --->
            <*> e2fsprogs
```

著作权声明

商标声明

免责声明