Tina Linux LCD
调试指南

版本号: 1.4
发布日期: 2023.11.22

版本历史

版本号	日期	制/修订人	内容描述
1.0	2019.07.03	AWA1422	1. 初始版本
1.1	2020.03.06	AWA1422	1. 增加更多的参数说明和常见问题
1.2	2021.04.06	AWA1422	1. 适配 linux 5.4
1.3	2021.04.06	AWA1810	1. 增加 checklist 2. 补充调试命令 3. 增加 runtime 功能说明 4. 新增支持 linux-5.10
1.4	2023.11.22	AWA1810	1. 补充 ESD 相关说明 2. 补充 dts 新增项说明

目　录

插　图

1 概述

1.1 编写目的

本文档将介绍 sunxi 平台 Display Engine 模块中 LCD 的调试方法。

1. LCD 调试方法，调试手段。
2. LCD 驱动编写。
3. lcd0 节点下各个属性的解释。
4. 典型 LCD 接口配置。

1.2 适用范围

sunxi 平台 DE1.0/DE2.0 中 LCD 屏幕参数设置。

1.3 相关人员

系统整合人员，显示开发相关人员，客户。

2 相关术语介绍

表 2-1: LCD 相关术语

术语	解释说明
SUNXI	Allwinner 一系列 SoC 硬件平台
LCD	Liquid Crystal Display, 液晶显示器
MIPI	Mobile Industry Processor Interface
DSI	Display Serial Interface，显示串行接口
I8080	Intel 8080LCD 接口
RGB	这里指一种 LCD 接口，该接口发送不经过任何编码的 RGB 分量
LVDS	Low-Voltage Differential Signaling 一种 LCD 接口，低压和差分传输是其特点

3 IC 规格

LCD 接口相关规格参考（每个平台支持的规格不同，需要以该型号的 datasheet 为准）：

1. 支持双显，异显。也就是显示内容可以不一样，显示分辨率可以不一样，屏接口也可以不一样。
2. 支持 MIPI-DSI 接口, 数量一个。最大支持 1920x1200@60 分辨率，满足宽高不要超过 2048，像素时钟不超过 180MHz 都支持。
3. 支持 RGB 接口，数量 2 个。其中主显支持并行 RGB666，副显并行支持 RGB888, 并行 RGB 接口最大支持 1920x1200@60 分辨率，满足宽高不要超过 2048，像素时钟不超过 180MHz 都支持。或者两个串行 RGB 接口，串行 RGB 的最高分辨率最大不超过 800*480@60
4. 支持两个 dual-link LVDS 接口, 最大支持 1920x1200@60 分辨率，满足宽高不要超过 2048，像素时钟不超过 180MHz 都支持。或者 4 个 single-link LVDS 接口，分辨率最高支持 1366*768@60。
5. 两个 I8080 接口。分辨率最高支持 800*480@60。
6. LVDS 接口支持信号同显。每两个 single link LVDS 接口必须连接到完全一样的 LVDS 接口的屏上，将完全一样的数据发送到这两个屏上，做到信号一样。所以理论上，T509 能做到 4 显，其中前 2 个和后 2 个分辨率可以不一样，2 个之间的分辨率必须一样，且必须连接一样的 LCD 屏。

说明

在多显的场景下，以上接口可以自由搭配，除了 MIPI-DSI 必须用在主显上。

技巧

一个 dual link LVDS 接口共 20 条线，它可以拆分成两个 single link 的 LVDS 接口，假设为 lvds0 和 lvds1，选择一个 single link 的时候做显示的时候，必须选择 lvds0。

4 模块介绍

4.1 添加屏驱动步骤

1. 对于 linux4.9 及以下版本总共需要修改三处地方（即下列前三项），对于 linux5.4 则需要修改四处地方，具体可参考屏驱动源码位置。

- linux 源码仓库。
- uboot 源码仓库。在 uboot 中也有显示和屏驱动，目的是显示 logo。
- 板级 dts 配置仓库。目的是通过 board.dts 来配置一些通用的 LCD 配置参数。对于 linux4.9，该配置同时对内核及 uboot 生效，对于 linux-5.4，请参照下条。
- 对于 linux5.4，还需额外配置 uboot 专用板级 dts 配置仓库。

2. 确保全志显示框架的内核配置有使能，查看menuconfig 配置说明。
3. 前期准备以下资料和信息：

- 屏手册。主要是描述屏基本信息和电气特性等，向屏厂索要。
- Driver IC 手册。主要是描述屏 IC 的详细信息。这里主要是对各个命令进行详解，对我们进行初始化定制有用，向屏厂索要。
- 屏时序信息。请向屏厂索要。请看屏时序参数说明 以了解更多信息。
- 屏初始化代码。，请向屏厂索要。一般情况下 DSI 和 I8080 屏等都需要初始化命令对屏进行初始化。
- 万用表。调屏避免不了测量相关电压。

4. 动手添加屏驱动之前，先了解屏驱动，请看屏驱动分解。
5. 通过第 3 步的资料，定位该屏的类型，然后选择一个已有同样类型的屏驱动作为模板进行屏驱动添加或者直接在上面修改。
6. 修改屏驱动目录下的panel.c和panel.h。在全局结构体变量panel_array中新增刚才添加strcut __lcd_panel的变量指针。panel.h中新增strcut __lcd_panel的声明。
7. 修改 Makefile。在 lcd 屏驱动目录的上一级的 Makefile 文件中的disp-objs中新增刚才添加屏驱动.o。
8. 修改 board.dts 中的 lcd0。可以看RGB 接口，MIPI-DSI 接口，I8080 接口和LVDS 接口，里面有介绍各种接口典型配置。硬件参数说明，这一章有所有 lcd0 节点下可配置属性详细解释。
9. 编译 uboot，kernel，打包烧写。注意不同 SDK，编译方式有所不同，部分 SDK 默认不编译 uboot。
10. 调试。通过调试方法我们可以初步定位问题，还有FAQ，对调屏也有帮助。

4.2 屏驱动说明

4.2.1 屏驱动源码位置

linux 3.4 版本内核：

linux3-4/drivers/video/sunxi/disp2/disp/lcd/

linux 3.10 版本内核：

linux3-10/drivers/video/sunxi/disp2/disp/lcd/

linux 4.9 版本内核：

linux-4.9/drivers/video/fbdev/sunxi/disp2/disp/lcd/

linux 5.4 版本内核：

linux-5.4/drivers/video/fbdev/sunxi/disp2/disp/lcd/

linux 5.4-ansc 版本内核：

bsp/drivers/video/sunxi/disp2/disp/lcd/

linux 5.10 版本及其以上内核：

bsp/drivers/video/sunxi/disp2/disp/lcd/

uboot-2014:

brandy/u-boot-2014.07/drivers/video/sunxi/disp2/disp/lcd

uboot-2018:

brandy/brandy-2.0/u-boot-2018/drivers/video/sunxi/disp2/disp/lcd

板级配置，其中 “芯片型号” 比如 R818，和 “板子名称” 比如 demo，请根据实际替换。

device/config/chips/芯片型号/configs/板子名称/board.dts

针对 linux5.4 时使用的 uboot 板级配置：

device/config/chips/芯片型号/configs/板子名称/uboot-board.dts

针对 linux5.4 时使用的 kernel 板级配置：

device/config/chips/芯片型号/configs/板子名称/linux-5.4/board.dts

4.2.2 menuconfig 配置说明

LCD 相关代码包含在 disp 驱动模块中，进入内核根目录，执行 make ARCH=arm menuconfig 或者 make ARCH=arm64 menuconfig(64bit 平台) 进入配置主界面。并按以下步骤操作：

DE1.0 对应平台：R6(linux-3.10)、R16(linux-3.4)。

DE2.0 对应平台：除 R6 和 R16 之外的。

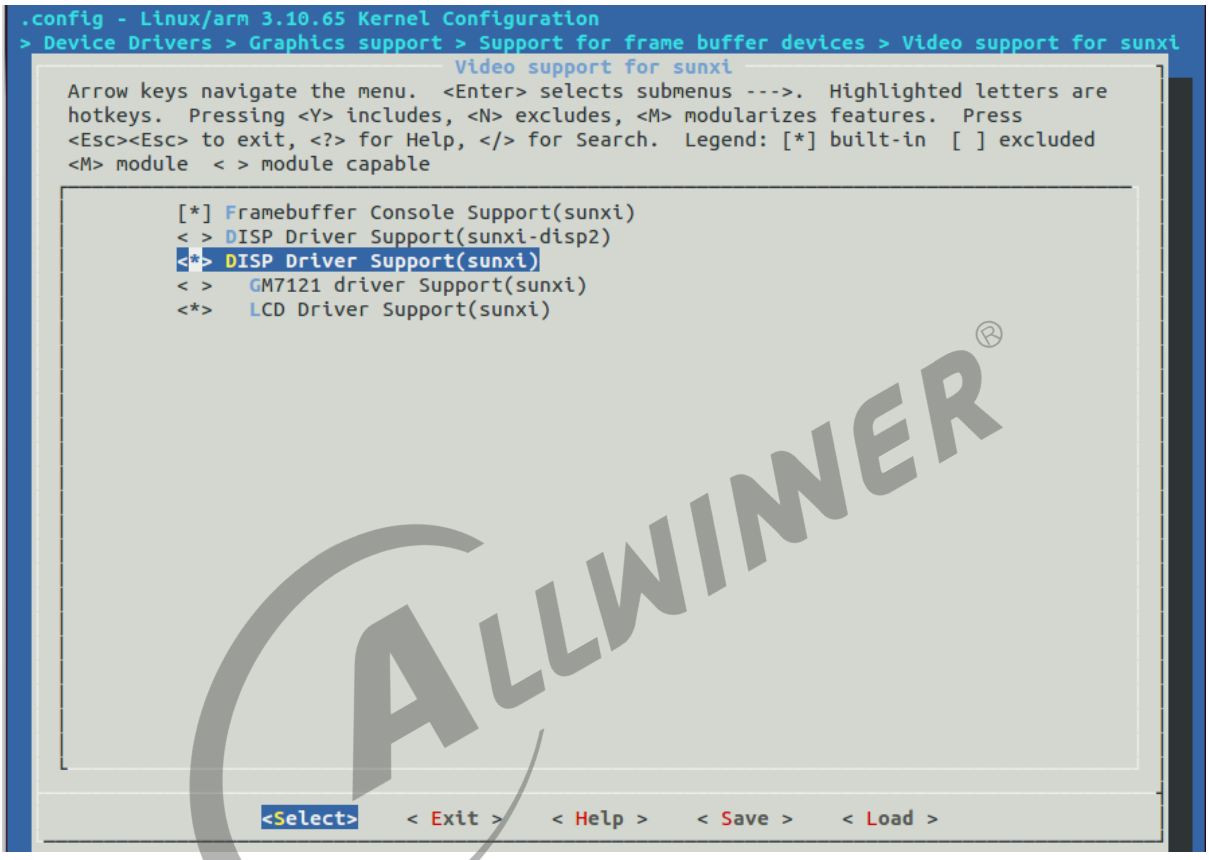

图 4-1: DE1.0 menuconfig 配置图

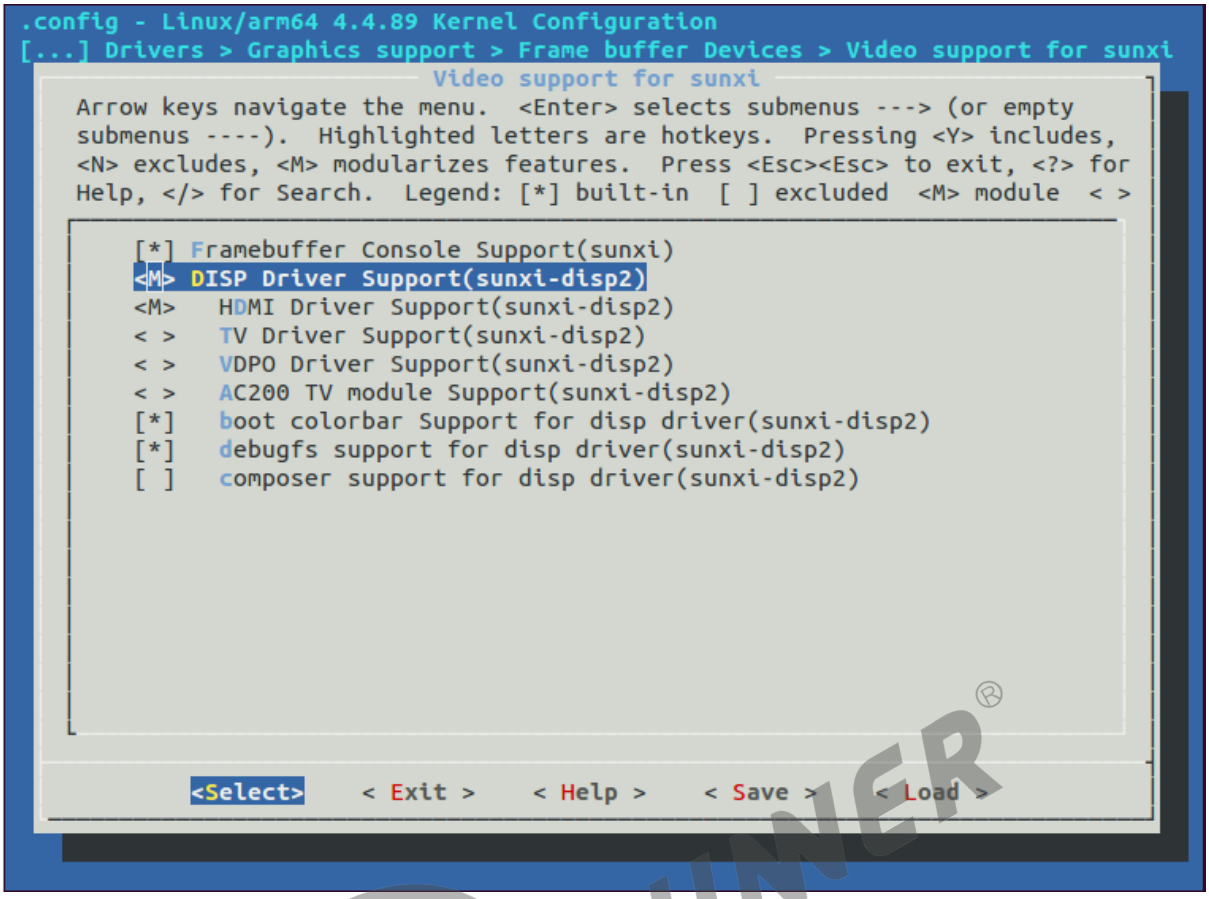

图 4-2: DE2.0 menuconfig 配置图

以 R40 为例，具体配置路径为：Device Drivers->Graphics support->Support for frame buffer devices->Video Support for sunxi -> DISP Driver Support(sunxi-disp2)。

如果是 linux-5.4-ansc 或 linux-5.10 及其以上的内核版本，配置路径为：

Allwinner BSP > Device Drivers > Video Drivers > DISP Driver Support(sunxi-disp2)

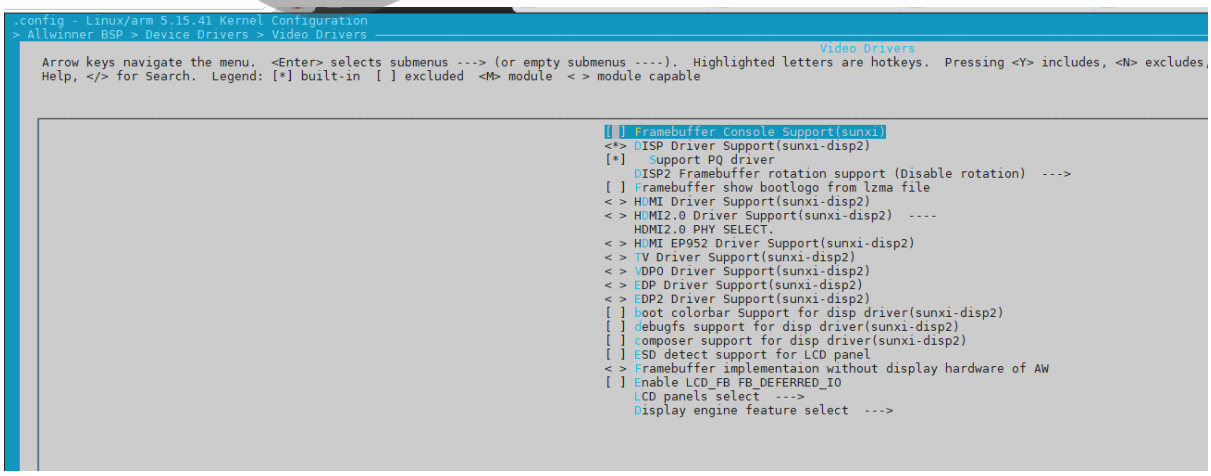

图 4-3: linux-5.10 及其以上版本 menuconfig 配置图

4.2.3 屏驱动分解

在屏驱动源码位置中，主要分为四类文件

1. panel.c和panel.h，当用户添加新屏驱动时，是需要修改这两个文件的，需要将屏结构体变量添加到全局结构体变量panel_array中。
2. lcd_source.c和lcd_source.h，这两个文件实现的是给屏驱动使用的函数接口，比如电源开关，gpio，dsi 读写接口等，用户不需要修改只需要用。
3. 屏驱动。除了上面提到的源文件外，其它的一般一个 c 文件和一个 h 文件就代表一个屏驱动。
4. 在屏驱动源码位置的上一级，有用户需要修改的 Makefile 文件。

我们可以打开屏驱动源码位置下的default_panel.c文件作为屏驱动的例子，在该文件的最后

```
struct __lcd_panel default_panel = {
   /* panel driver name, must mach the lcd_drv_name in board.dts */
   .name = "default_lcd",
   .func = {
      .cfg_panel_info = LCD_cfg_panel_info,
      .cfg_open_flow = LCD_open_flow,
      .cfg_close_flow = LCD_close_flow,
      }
   ,
};
```

该全局变量default_panel的成员name与lcd_driver_name必须一致，这个关系到驱动能否找到指定的文件。

接下来是func成员的初始化，这里最主要实现三个回调函数。LCD_cfg_panel_info，LCD_open_flow和LCD_close_flow。

开关屏流程即屏上下电流程，屏手册或者 driver IC 手册中里面的 Power on Sequence 和 Power off Sequence。

开关屏的操作流程如下图所示。

其中，LCD_open_flow 和 LCD_close_flow 称为开关屏流程函数，方框中的函数，如 LCD_power_on，称为开关屏步骤函数。

不需要进行初始化操作的 LCD 屏，比如 lvds 屏，RGB 屏等，LCD_panel_init 及 LCD_panel_exit 这函数可以为空。

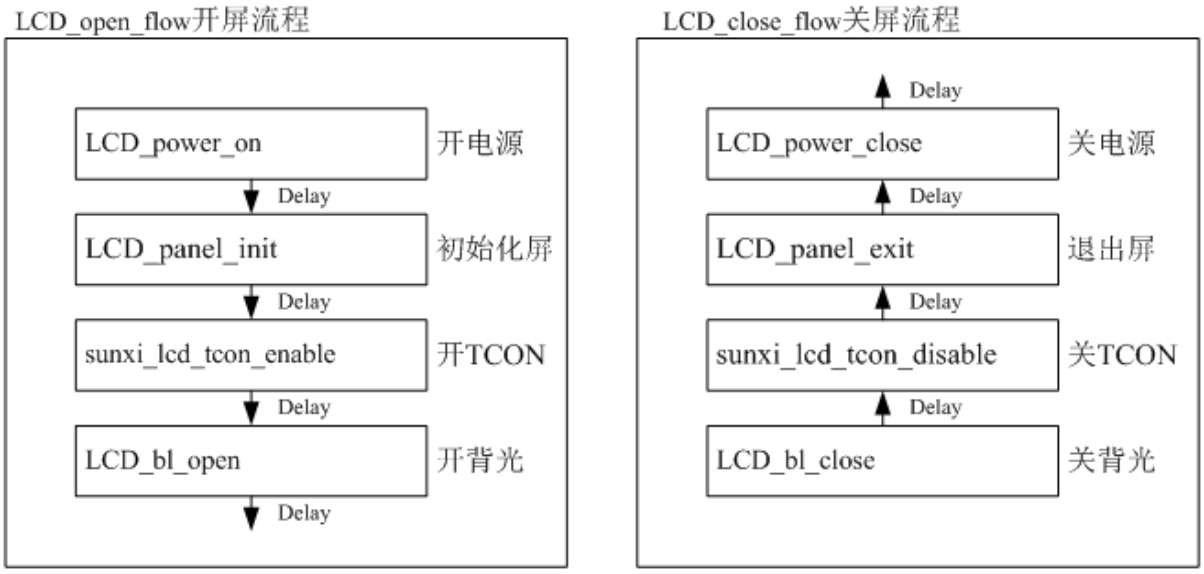

图 4-4: LCD 开关屏流程

函数：LCD_open_flow

功能：LCD_open_flow 函数只会系统初始化的时候调用一次，执行每个 LCD_OPEN_FUNC 即是把对应的开屏步骤函数进行注册，先注册先执行，但并没有立刻执行该开屏步骤函数。

原型：

```
static __s32 LCD_open_flow(__u32 sel)
```

函数常用内容为：

```
static __s32 LCD_open_flow(__u32 sel)
{
   LCD_OPEN_FUNC(sel, LCD_power_on,10);
   LCD_OPEN_FUNC(sel, LCD_panel_init, 50);
   LCD_OPEN_FUNC(sel, sunxi_lcd_tcon_enable, 100);
   LCD_OPEN_FUNC(sel, LCD_bl_open, 0);
   return 0;
}
```

如上，调用四次 LCD_OPEN_FUNC 注册了四个回调函数，对应了四个开屏流程, 先注册先执行。实际上注册多少个函数是用户自己的自由，只要合理即可。

1. LCD_power_on 即打开 LCD 电源，再延迟 10ms；这个步骤一般用于打开 LCD 相关电源和相关管脚比如复位脚。这里一般是使用电源控制函数说明和管脚控制函数说明进行操作。
2. LCD_panel_init 即初始化屏，再延迟 50ms；不需要初始化的屏，可省掉此步骤，这个函数一般用于发送初始化命令给屏进行屏初始化。如果是 DSI 屏看DSI 相关函数说明，如果是 I8080 屏用I8080 接口函数说明，如果是其它情况比如 i2c 或者 spi 可以看使用 twi/spi 串行接口初始化，也可以用 GPIO 来进行模拟。
3. sunxi_lcd_tcon_enable 打开 TCON，再延迟 100ms；这一步是固定的，表示开始发送图像信号。

4. LCD_bl_open 打开背光，再延迟 0ms。前面三步搞定之后才开背光，这样不会看到闪烁。这里一般使用的函数请看背光控制函数说明。

如下图，这是屏手册中典型的上电时序图，我们编写屏驱动的时候，也要注意，该延时就得延时。

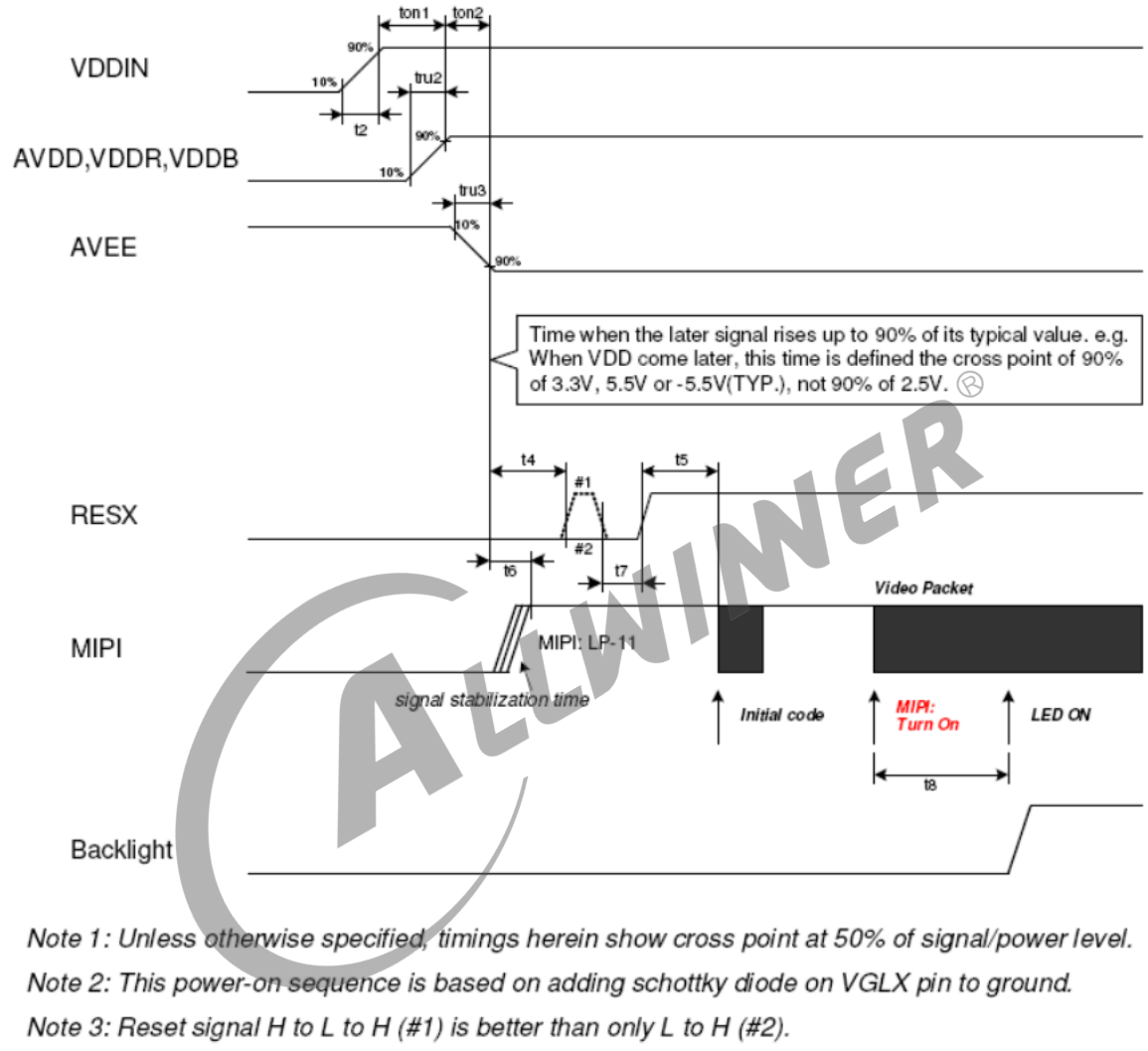

图 4-5: power on

函数：LCD_OPEN_FUNC

功能：注册开屏步骤函数到开屏流程中，记住这里是注册不是执行！

原型：

```
void LCD_OPEN_FUNC(__u32 sel, LCD_FUNC func, __u32 delay)
```

参数说明：

func 是一个函数指针，其类型是：void (*LCD_FUNC) (__u32 sel)，用户自己定义的函数必须也要用统一的形式。比如：

```
void user_defined_func(__u32 sel)
{
  //do something
}
```

delay 是执行该步骤后，再延迟的时间，时间单位是毫秒。

LCD_OPEN_FUNC 的第二个参数是前后两个步骤的延时长度，单位 ms，注意这里的数值请按照屏手册规定去填，乱填可能导致屏初始化异常或者开关屏时间过长，影响用户体验。

与 LCD_open_flow 对应的是 LCD_close_flow 是，用于注册关屏函数，使用 LCD_CLOSE_FUNC 进行函数注册，先注册先执行，这里只是注册回调函数不是立刻执行。

```
static s32 LCD_close_flow(u32 sel)
{
  /* close lcd backlight, and delay 0ms */
  LCD_CLOSE_FUNC(sel, LCD_bl_close, 0);
  /* close lcd controller, and delay 0ms */
  LCD_CLOSE_FUNC(sel, sunxi_lcd_tcon_disable, 50);
  /* open lcd power, than delay 200ms */
  LCD_CLOSE_FUNC(sel, LCD_panel_exit, 100);
  /* close lcd power, and delay 500ms */
  LCD_CLOSE_FUNC(sel, LCD_power_off, 0);

  return 0;
}
```

1. 先关闭背光，这样整个关屏过程，用户不会看到闪烁的过程。
2. 关闭 TCON，也就是停止发送数据，这是必要的。再延迟 50ms。
3. 执行关屏代码，再延迟 200ms（不需要初始化的屏，可省掉此步骤）。
4. 最后关闭电源，再延迟 0ms。

如下图是典型关屏时序图。

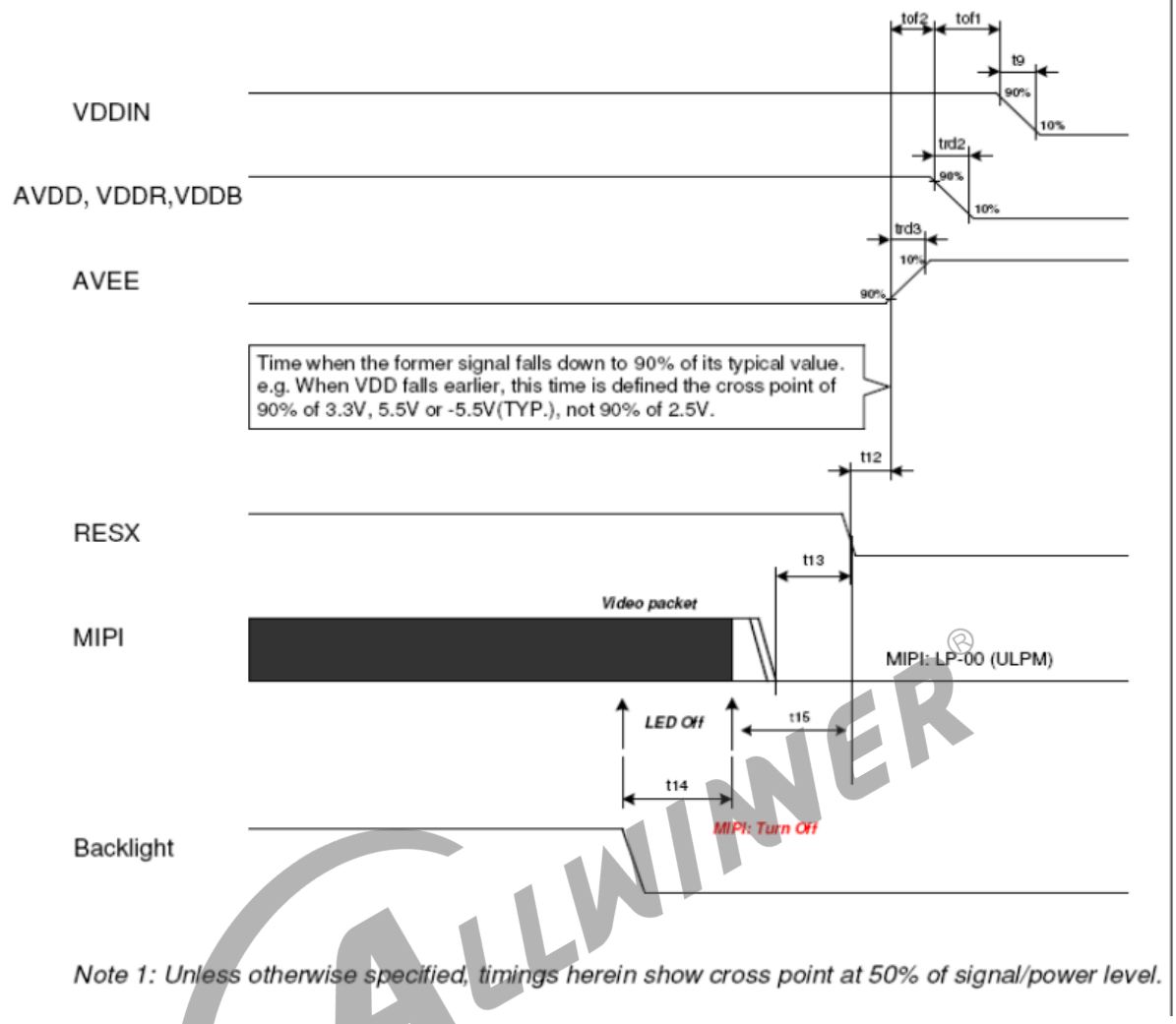

图 4-6: power off

函数：LCD_cfg_panel_info

功能：配置的 TCON 扩展参数，比如 gamma 功能和颜色映射功能。

原型：

```
static void LCD_cfg_panel_info(__panel_extend_para_t * info)
```

TCON 的扩展参数只能在屏文件中配置，参数的定义见显示效果相关参数。

需要 gamma 校正，或色彩映射，在 board.dts 中将相应模块的 enable 参数置 1，lcd_gamma_en，lcd_cmap_en，并且填充 3 个系数表，lcd_gamma_tbl，lcd_cmap_tbl，如下所示代码部分。注意的是：gamma，模板提供了 18 段拐点值，然后再插值出所有的值（255 个）。如果觉得还不细，可以往相应表格里添加子项。cmap_tbl 的大小是固定了，不能减小或增加表的大小。

最终生成的 gamma 表项是由 rgb 三个 gamma 值组成的，各占 8bit，目前提供的模板中，三个 gamma 值是相同的。

```
static void LCD_cfg_panel_info(struct panel_extend_para *info)
{
  u32 i = 0, j = 0;
  u32 items;
  u8 lcd_gamma_tbl[][2] = {
    /* {input value, corrected value} */
    {0, 0},
    {15, 15},
    {30, 30},
    {45, 45},
    {60, 60},
    {75, 75},
    {90, 90},
    {105, 105},
    {120, 120},
    {135, 135},
    {150, 150},
    {165, 165},
    {180, 180},
    {195, 195},
    {210, 210},
    {225, 225},
    {240, 240},
    {255, 255},
  };

  u32 lcd_cmap_tbl[2][3][4] = {
    {
    {LCD_CMAP_G0, LCD_CMAP_B1, LCD_CMAP_G2, LCD_CMAP_B3},
    {LCD_CMAP_B0, LCD_CMAP_R1, LCD_CMAP_B2, LCD_CMAP_R3},
    {LCD_CMAP_R0, LCD_CMAP_G1, LCD_CMAP_R2, LCD_CMAP_G3},
    },
    {
    {LCD_CMAP_B3, LCD_CMAP_G2, LCD_CMAP_B1, LCD_CMAP_G0},
    {LCD_CMAP_R3, LCD_CMAP_B2, LCD_CMAP_R1, LCD_CMAP_B0},
    {LCD_CMAP_G3, LCD_CMAP_R2, LCD_CMAP_G1, LCD_CMAP_R0},
    },
  };

  items = sizeof(lcd_gamma_tbl) / 2;
  for (i = 0; i < items - 1; i++) {
    u32 num = lcd_gamma_tbl[i + 1][0] - lcd_gamma_tbl[i][0];

    for (j = 0; j < num; j++) {
      u32 value = 0;

      value =
        lcd_gamma_tbl[i][1] +
        ((lcd_gamma_tbl[i + 1][1] -
          lcd_gamma_tbl[i][1]) * j) / num;
      info->lcd_gamma_tbl[lcd_gamma_tbl[i][0] + j] =
        (value << 16) + (value << 8) + value;
    }
  }
  info->lcd_gamma_tbl[255] =
    (lcd_gamma_tbl[items - 1][1] << 16) +
    (lcd_gamma_tbl[items - 1][1] << 8) + lcd_gamma_tbl[items - 1][1];

  memcpy(info->lcd_cmap_tbl, lcd_cmap_tbl, sizeof(lcd_cmap_tbl));
```

```
}
```

4.2.4 延时函数说明

函数：**sunxi_lcd_delay_ms / sunxi_lcd_delay_us**

功能：延时函数，分别是毫秒级别/微秒级别的延时。

原型：s32 sunxi_lcd_delay_ms(u32 ms) / s32 sunxi_lcd_delay_us(u32 us)

4.2.5 图像数据使能函数说明

函数：**sunxi_lcd_tcon_enable / sunxi_lcd_tcon_disable**

功能：打开 LCD 控制器，开始刷新 LCD 显示。关闭 LCD 控制器，停止刷新数据。

原型：void sunxi_lcd_tcon_enable(u32 screen_id)

void sunxi_lcd_tcon_disable(u32 screen_id)

4.2.6 背光控制函数说明

函数：**sunxi_lcd_backlight_enable / sunxi_lcd_backlight_disable**

功能：打开/关闭背光，操作的是 board.dts 中 lcd_bl 配置的 gpio。见lcd_bl_en。

原型：void sunxi_lcd_backlight_enable(u32 screen_id)

void sunxi_lcd_backlight_disable(u32 screen_id)

函数：**sunxi_lcd_pwm_enable / sunxi_lcd_pwm_disable**

功能：打开/关闭 pwm 控制器，打开时 pwm 将往外输出 pwm 波形。对应的是 lcd_pwm_ch 所对应的那一路 pwm。

原型：s32 sunxi_lcd_pwm_enable(u32 screen_id)

s32 sunxi_lcd_pwm_disable(u32 screen_id)

4.2.7 电源控制函数说明

函数：**sunxi_lcd_power_enable / sunxi_lcd_power_disable**

功能：打开/关闭 Lcd 电源，操作的是 board.dts 中的 lcd_power/lcd_power1/lcd_power2（pwr_id 标识电源索引）。

原型：void sunxi_lcd_power_enable(u32 screen_id, u32 pwr_id)

void sunxi_lcd_power_disable(u32 screen_id, u32 pwr_id)

1. pwr_id = 0：对应于 board.dts 中的 lcd_power。
2. pwr_id = 1：对应于 board.dts 中的 lcd_power1。
3. pwr_id = 2：对应于 board.dts 中的 lcd_power2。
4. pwr_id = 3：对应于 board.dts 中的 lcd_power3。

函数：**sunxi_lcd_pin_cfg**

功能：配置 lcd 的 io。

原型：s32 sunxi_lcd_pin_cfg(u32 screen_id, u32 bon)

说明：配置 lcd 的 data/clk 等 pin，对应 board.dts 中的 lcdd0-lcdd23/lcddclk/lcdde/lcdhsync/lcdvsync。

由于 dsi 是专用 pin, 所以 dsi 接口屏不需要在 board.dts 中配置这组 pin，但同样会在此函数接口中打开与关闭对应的 pin。

Bon: 1: 为开，0：为配置成 disable 状态。

4.2.8 DSI 相关函数说明

MIPI DSI 屏，大部分需要初始化，使用的是 DSI-D0 通道的 LP 模式进行初始化。提供的接口函数说明如下：

函数：**sunxi_lcd_dsi_clk_enable / sunxi_lcd_dsi_clk_disble**

功能：仅限 dsi 接口屏使用，使能/关闭 dsi 输出的高速时钟 clk 信号，必须在初始化的时候调用。

原型：s32 sunxi_lcd_dsi_clk_enable(u32 scree_id)

技巧

sunxi_lcd_dsi_clk_enable 函数必须在发送完初始化命令后调用，否则屏有可能会初始化失败。

s32 sunxi_lcd_dsi_clk_disable(u32 scree_id)

函数：**sunxi_lcd_dsi_dcs_wr**

功能：对屏的 dcs 写操作。

原型：__s32 sunxi_lcd_dsi_dcs_wr(__u32 sel,__u8 cmd,__u8* para_p,__u32 para_num)

参数说明：

- cmd：dcs 写命令内容。
- para_p：dcs 写命令的参数起始地址。
- para_num：dcs 写命令的参数个数，单位为 byte。

函数：**sunxi_lcd_dsi_dcs_wr_2para**

功能：对屏的 dcs 写操作，该命令带有两个参数。

原型：__s32 sunxi_lcd_dsi_dcs_wr_2para(__u32 sel,__u8 cmd,__u8 para1,__u8 para2)

参数说明：

- cmd：dcs 写命令内容。
- para1：dcs 写命令的第一个参数内容。
- para2：dcs 写命令的第二个参数内容。

sunxi_dsi_dcs_wr_0para，sunxi_dsi_dcs_wr_1para，sunxi_dsi_dcs_wr_3para，sunxi_dsi_dcs_wr_4para，sunxi_dsi_dcs_wr_5para定义与dsi_dcs_wr_2para类似，差别就是参数数量。

函数：**sunxi_lcd_dsi_dcs_read**

功能：dsi 读操作。

原型：s32 sunxi_lcd_dsi_dcs_read(u32 sel, u8 cmd, u8 result, u32 num_p)

参数说明：

- sel, 显示 id。
- cmd, 要读取的寄存器。
- result，用于存放读取接口的数组，用户必须自行保证其有足够空间保存读取的接口。
- num_p，指针用于存放读取字节数，用户必须保证其非空指针。

4.2.9 I8080 接口函数说明

显示驱动提供 5 个接口函数可供使用。如下：

函数：**sunxi_lcd_cpu_write**

功能：设定 CPU 屏的指定寄存器为指定的值。

原型：void sunxi_lcd_cpu_write(__u32 sel, __u32 index, __u32 data)

函数内容为：

```
Void sunxi_lcd_cpu_write(__u32 sel, __u32 index, __u32 data)
{
   sunxi_lcd_cpu_write_index(sel, index);
   sunxi_lcd_cpu_wirte_data(sel, data);
}
```

实现了 8080 总线上的两个写操作。

sunxi_lcd_cpu_write_index 实现第一个写操作，这时 PIN 脚 RS（A1）为低电平，总线数据上的数据内容为参数 index 的值。

Sunxi_lcd_cpu_wirte_data 实现第二个写操作，这时 PIN 脚 RS（A1）为高电平，总线数据上的数据内容为参数 data 的值。

函数：**sunxi_lcd_cpu_write_index**

功能：设定 CPU 屏为指定寄存器。

原型：

```
void sunxi_lcd_cpu_write_index(__u32 sel,__u32 index)
```

具体说明见 sunxi_lcd_cpu_write。

函数：**sunxi_lcd_cpu_write_data**

功能：设定 CPU 屏寄存器的值为指定的值。

原型：

```
void Sunxi_lcd_cpu_write_data(__u32 sel, __u32 data);
```

函数: tcon0_cpu_rd_24b_data

功能：读操作。

原型：

```
s32 tcon0_cpu_rd_24b_data(u32 sel, u32 index, u32 *data, u32 size)
```

参数说明：

- sel：显示 id。
- index: 要读取的寄存器。
- data：用于存放读取接口的数组指针，用户必须保证其有足够空间存放数据。
- size：要读取的字节数。

4.2.10 管脚控制函数说明

函数：**sunxi_lcd_gpio_set_value**

功能：LCD_GPIO PIN 脚上输出高电平或低电平。

原型：s32 sunxi_lcd_gpio_set_value(u32 screen_id, u32 io_index, u32 value)

参数说明：

- io_index = 0：对应于 board.dts 中的 lcd_gpio_0。
- io_index = 1：对应于 board.dts 中的 lcd_gpio_1。
- io_index = 2：对应于 board.dts 中的 lcd_gpio_2。
- io_index = 3：对应于 board.dts 中的 lcd_gpio_3。
- value = 0：对应 IO 输出低电平。
- Value = 1：对应 IO 输出高电平。

只用于该 GPIO 定义为输出的情形。

函数：**sunxi_lcd_gpio_set_direction**

功能：设置 LCD_GPIO PIN 脚为输入或输出模式。

原型：

```
s32 sunxi_lcd_gpio_set_direction(u32 screen_id, u32 io_index, u32 direction);
```

参数说明：

- io_index = 0：对应于 board.dts 中的 lcd_gpio_0。
- io_index = 1：对应于 board.dts 中的 lcd_gpio_1。
- io_index = 2：对应于 board.dts 中的 lcd_gpio_2。
- io_index = 3：对应于 board.dts 中的 lcd_gpio_3。
- direction = 0：对应 IO 设置为输入。
- direction = 1：对应 IO 设置为输出。

一部分屏需要进行初始化操作，在开屏步骤函数中，对应于 LCD_panel_init 函数，提供了几种方式对屏的初始化。

对于 DSI 屏，是通过 DSI-D0 通道进行初始化。对于 CPU 屏，是通过 8080 总线的方式，使用的是 LCDIO（PD,PH）进行初始化。这种初始化方式，其总线的引脚位置定义与 CPU 屏一致。

以下这些接口在屏驱动分解中提到路径的 lcd_source.c 和 lcd_source.h 中定义和实现。

4.2.11 使用 twi/spi 串行接口初始化

需要在屏驱动中注册 twi/spi 设备对串行接口的访问。

使用硬件 spi 对屏或者转接 IC 进行初始化，如下代码片段。

首先调用 spi_init 函数对 spi 硬件进行初始化，spi_init 函数可以分为几个步骤，第一获取 master；根据实际的硬件连接，选择 spi（代码中选择了 spi1），如果这一步返回错误说 spi 没有配置好，找 spi 驱动负责人。第二步设置 spi device，这里包括最大速度，spi 传输模式，以及每个字包含的比特数。最后调用 spi_setup 完成 master 和 device 的关联。

comm_out 是一个 spi 传输的例子，核心就是 spi_sync_transfer 函数。

```
static int spi_init(void)
{
    int ret = -1;
    struct spi_master *master;

    master = spi_busnum_to_master(1);
    if (!master) {
        lcd_fb_wrn("fail to get master\n");
        goto OUT
    }


spi_device = spi_alloc_device(master);
    if (!spi_device) {
        lcd_fb_wrn("fail to get spi device\n");
        goto OUT;
    }

spi_device->bits_per_word = 8;
    spi_device->max_speed_hz = 60000000; /*50MHz*/
    spi_device->mode = SPI_MODE_0;

    ret = spi_setup(spi_device);
    if (ret) {
        lcd_fb_wrn("Faile to setup spi\n");
        goto FREE;
    }

    lcd_fb_inf("Init spi1:bits_per_word:%d max_speed_hz:%d mode:%d\n",
          spi_device->bits_per_word, spi_device->max_speed_hz,
          spi_device->mode);

    ret = 0;
    goto OUT;

FREE:
    spi_master_put(master);
    kfree(spi_device);
    spi_device = NULL;
OUT:
    return ret;
}

static int comm_out(unsigned int sel, unsigned char cmd)
{
    struct spi_transfer t;
    if (!spi_device)
        return -1;
    DC(sel, 0);
    memset(&t, 0, sizeof(struct spi_transfer));
    t.tx_buf    = &cmd;
    t.len       = 1;
```

```
    t.bits_per_word = 8;
    t.speed_hz = 24000000;
    return spi_sync_transfer(spi_device, &t, 1);
}
```

使用硬件 twi 对 LCD& 转接 IC 进行初始化，初始化 twi 硬件的核心函数是 i2c_add_driver，而你要做的是初始化好其参数 struct i2c_driver。

it66121_id 包含设备名字以及 i2c 总线索引（i2c0，i2c1…）。

it66121_i2c_probe 能进到这个函数，你就可以开始使用 twi 了。代码段里面仅仅将后面需要的参数 cilent 赋值给一个全局指针变量。

it66121_match，这是 dts 的 match table，由于你是给 disp2 加驱动，所以这里的 match table 就是 disp2 的 match table，这个 table 关系到能否使用 twi，注意不要填错。

tv_i2c_detect 函数，这里是非常关键的，这个函数早于 probe 函数被调用，只有成功被调用后才能开始使用 twi，其中 strlcpy 的调用意味着成功。

normal_i2c 是从设备地址列表，填写的 LCD 或者转接 IC 的从设备地址以及 i2c 索引。

以 probe 函数是否被调用来决定你是否可以开始使用 twi。

用 i2c_smbus_write_byte_data 或者 i2c_smbus_read_byte_data 来读写可以满足大部分场景。

```
#define IT66121_SLAVE_ADDR 0x4c
#define IT66121_I2C_ID 0

 static const struct i2c_device_id it66121_id[] = {
    { "IT66121", IT66121_I2C_ID },
    { /* END OF LIST */ }
};
MODULE_DEVICE_TABLE(i2c, it66121_id);
static int it66121_i2c_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    this_client = client;
    return 0;
}

static const struct of_device_id it66121_match[] = {
    {.compatible = "allwinner,sun8iw10p1-disp",},
    {.compatible = "allwinner,sun50i-disp",},
    {.compatible = "allwinner,sunxi-disp",},
    {},
};

static int tv_i2c_detect(struct i2c_client *client, struct i2c_board_info *info)
{
    const char *type_name = "IT66121";

    if (IT66121_I2C_ID == client->adapter->nr) {
        strlcpy(info->type, type_name, 20);
    } else
        pr_warn("%s:%d wrong i2c id:%d, expect id is :%d\n", __func__, __LINE__,
            client->adapter->nr, IT66121_I2C_ID);
    return 0;
```

```
}

static  unsigned short normal_i2c[] = {IT66121_SLAVE_ADDR, I2C_CLIENT_END};

static struct i2c_driver it66121_i2c_driver = {
   .class = I2C_CLASS_HWMON,
   .id_table = it66121_id,
   .probe = it66121_i2c_probe,
   .remove = it66121_i2c_remove,
   .driver = {
      .owner = THIS_MODULE,
      .name = "IT66121",
      .of_match_table = it66121_match,
   },
   .detect      = tv_i2c_detect,
   .address_list    = normal_i2c,
};

static void LCD_panel_init(u32 sel)
{
   int ret = -1;

   ret = i2c_add_driver(&it66121_i2c_driver);
   if (ret) {
      pr_warn("Add it66121_i2c_driver fail!\n");
      return;
   }
//start init chip with i2c

}


void it6612_twi_write_byte(it6612_reg_set* reg)
{
   u8 rdata = 0;
   u8 tmp = 0;

   rdata = i2c_smbus_read_byte_data(this_client, reg->offset);
   tmp = (rdata & (~reg->mask))|(reg->mask&reg->value);
   i2c_smbus_write_byte_data(this_client, reg->offset, tmp);
}
```

4.2.12 U-boot 屏驱动注意事项

U-boot 编写屏驱动的步骤和内核是一样的，代码路径文件组织方式都是一样的，这里要讲的是需要注意的事项。

1. 为了加快 U-boot 的显示速度，开屏的几个函数之间采取异步调用的方式，原理是利用 timer 中断，定时调用开屏函数，所以这种情况下 bootGUI 框架加载完毕并不意味着开屏完成，而是当你见到LCD open finish的打印的时候。

建议：为了尽量利用异步调用的优点，请把需要的延时尽量在注册回调的时候指定，比如下面延时 10ms 就是利用 timer 异步来进行回调的，这 10ms 时间，uboot 就可以做其它事情，以达到异步调用的目的。

```
LCD_OPEN_FUNC(sel, LCD_power_on,10);
```

2. sunxi_lcd_power_enable 函数和 sunxi_lcd_pin_cfg 不能在LCD_power_on之外调用，否则 uboot 会异常。

严格讲，只能在用LCD_OPEN_FUNC注册的回调第一个函数里面调用。

4.3 RGB 接口

4.3.1 概述

下面介绍全志平台的 RGB 以及配置示例，至于 lcd0 下面每个属性的详解细节请看硬件参数说明。

RGB 接口在全志平台又称 HV 接口（Horizontal 同步和 Vertical 同步）。

对于 RGB 屏的初始化:

有些 LCD 屏支持高级的功能比如 gamma，像素格式的设置等，但是 RGB 协议本身不支持图像数据之外的传输，所以无法通过 RGB 管脚进行对 LCD 屏进行配置，所以拿到一款 RGB 接口屏，要么不需要初始化命令，要么这个屏会提供额外的管脚给 SoC 来进行配置，比如 SPI 和 I2C 等。

4.3.2 RGB 接口管脚

Signal	Description	Type
Vsync	Vertical sync, indicates one new frame	O
Hsync	Horizontal sync, indicate one new scan line	O
DCLK	Dot clock, pixel data are sync by this clock	O
DE	LCD data enable	O
D[23..0]	24Bit RGB output from input FIFO for panel	O

图 4-7: RGB 管脚

上面这些脚具体到 SoC 哪根管脚以及第几个功能（管脚复用功能）请参考 pin mux 表格，管脚复用功能的名字一般以 “LCDX_” 开头，其中 X 是数字。

其中数据脚的数量不一定是 24 根。RGB 又细分几种接口，通过设置lcd_hv_if来选择。

表 4-1: RGB 接口分类

位宽	时钟周期数	颜色数量和格式
24 bits	1 cycle	16.7M colors, RGB888
18 bits	1 cycle	262K colors, RGB666
16 bits	1 cycle	65K colors, RGB565
6 bits	3 cycles	262K colors, RGB666
6 bits	3 cycles	65K colors, RGB565

说明

时钟周期数的意思：是一个像素需要用多少个时钟周期发送完毕的意思。
当时钟周期为 1 时，我们称这种 RGB 接口为并行接口，其它的情况则是串行接口，更为普遍的原则就是只要需要多个时钟周期才能发送完一个像素的接口都是串行接口。
如何判断是否支持 24bit 的位宽，最简单的方式就是在 pinmux 表格中数一数数据脚的数量，如果有 24 根则支持 24bit，如果只有 18 根则支持 18bit。

硬件连接

对于并行 RGB 的接口，当位宽小于 24 时，硬件连接应该选择连接每个分量中的高位而放弃低位，这样做的原因是损失较少的颜色数量。

对于串行 RGB 接口，硬件连接可参考RGB 和 I8080 管脚配置示意图中 sync RGB 那几列。

RGB 接口有两种同步方式，根据经验来说尽量使用第二种方式，硬件上请保证连接好 DE 脚。

1. Hsync+Vsync
2. DE（Data Enable）

4.3.3 并行 RGB 接口配置示例

当我们配置并行 RGB 接口时，在配置里面并不需要区分是 24 位，18 位和 16 位，最大位宽是哪种是参考 pin mux 表格，如果 LCD 屏本身支持的位宽比 SoC 支持的位宽少，当然只能选择少的一方。

因为不需要初始化，RGB 接口极少出现问题，重点关注 lcd 的 timing 的合理性，也就是lcd_ht，lcd_hspw，lcd_hbp，lcd_vt，lcd_vspw和lcd_vbp这个属性的合理性。

下面是典型并行 RGB 接口 board.dts 配置示例，其中用空行把配置分成几个部分

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化，这里是 default_lcd，是针对不需要初始化设置的 RGB 屏
2. 第二部分决定下面的配置是一个并行 RGB 的配置。
3. 第三部分决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。
4. 第四部分决定了背光（pwm 和 lcd_bl_en）。请看背光相关参数。

5. 第五部分是显示效果部分的配置，如果非 24 位的 RGB，那么一般情况下需要设置lcd_frm。
6. 第六部分就是电源和管脚配置。是用 RGB666 还是 RGB888，需要根据实际 pinmux 表来决定，如果该芯片只有 18 根 rgb 数据则只能 rgb18。请看电源和管脚参数。

```
&lcd0 {
    /* part 1 */
    lcd_used            = <1>;
    lcd_driver_name     = "default_lcd";

    /* part 2 */
    lcd_if              = <0>;
    lcd_hv_if           = <0>;

    /* part 3 */
    lcd_width           = <150>;
    lcd_height          = <94>;
    lcd_x               = <800>;
    lcd_y               = <480>;
    lcd_dclk_freq       = <33>;
    lcd_hbp             = <46>;
    lcd_ht              = <1055>;
    lcd_hspw            = <0>;
    lcd_vbp             = <23>;
    lcd_vt              = <525>;
    lcd_vspw            = <0>;

    /* part 4 */
    lcd_backlight       = <50>;
    lcd_pwm_used        = <1>;
    lcd_pwm_ch          = <8>;
    lcd_pwm_freq        = <10000>;
    lcd_pwm_pol         = <1>;
    lcd_bl_en           = <&pio PD 27 1 0 3 1>;
    lcd_bright_curve_en = <0>;

    /* part 5 */
    lcd_frm             = <0>;
    lcd_io_phase        = <0x0000>;
    lcd_gamma_en        = <0>;
    lcd_cmap_en         = <0>;
    lcd_hv_clk_phase    = <0>;
    lcd_hv_sync_polarity= <0>;

    /* part 6 */
    lcd_power           = "vcc-lcd";
    lcd_pin_power       = "vcc-pd";
    pinctrl-0 = <&rgb24_pins_a>;
    pinctrl-1 = <&rgb24_pins_b>;
};
```

4.3.4 串行 RGB 接口的典型配置

串行 RGB 是相对于并行 RGB 来说，而并不是说它只用一根线来发数据，只要通过多个时钟周期才能把一个像素的数据发完，那么这样的 RGB 接口就是串行 RGB。

同样与并行 RGB 接口一样，配置中并不需要也无法体现具体是哪种串行 RGB 接口，需要做的就是把硬件连接对。

下面是典型串行 RGB 接口 board.dts 配置示例，它只有 8 根数据脚，其中用空行把配置分成几个部分

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。
2. 第二分部决定下面的配置是一个串行 RGB 的配置。
3. 第三部分决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。

技巧

这里需要注意的是，对于该接口，SoC 总共需要三个周期才能发完一个 pixel，所以我们配置时序的时候，需要满足lcd_dclk_freq*3=lcd_ht*lcd_vt*60，或者lcd_dclk_freq=lcd_ht*3*lcd_vt*60要么 3 倍lcd_ht要么 3 倍lcd_dclk_freq。

4. 第四部分决定了背光。就是 pwm 和 lcd_bl_en。请看背光相关参数
5. 第五部分是显示效果方面的设置。
6. 第六部分管脚和电源的定义。请看电源和管脚参数。

说明

下面实例的 lcd driver IC 是 stv7789v，是需要初始化，初始化的接口协议是 SPI，所以这多了几根 spi 管脚配置，驱动里面用 gpio 模拟 spi 协议，所以这里都是配置 gpio 功能。

```
&lcd0 {
    /* part 1 */
    lcd_used            = <1>;
    lcd_driver_name     = "st7789v";

    /* part 2 */
    lcd_if              = <0>;
    lcd_hv_if           = <8>;

    /* part 3 */
    lcd_x               = <240>;
    lcd_y               = <320>;
    lcd_width           = <108>;
    lcd_height          = <64>;
    lcd_dclk_freq       = <19>;
    lcd_hbp             = <120>;
    ;10 + 20 + 10 + 240*3 = 760  real set 1000
    lcd_ht              = <850>;
    lcd_hspw            = <2>;
    lcd_vbp             = <13>;
    lcd_vt              = <373>;
    lcd_vspw            = <2>;

    /* part 4 */
    lcd_backlight       = <50>;
    lcd_pwm_used        = <1>;
    lcd_pwm_ch          = <8>;
    lcd_pwm_freq        = <50000>;
    lcd_pwm_pol         = <1>;
    lcd_pwm_max_limit   = <255>;
```

```
    lcd_bl_en       = <&pio PB 1 1 0 3 1>;
    lcd_bright_curve_en = <1>;

    /* part 5 */
    lcd_frm         = <1>;
    lcd_hv_clk_phase    = <0>;
    lcd_hv_sync_polarity= <0>;
    lcd_hv_srgb_seq     = <0>;
    lcd_io_phase        = <0x0000>;
    lcd_gamma_en        = <0>;
    lcd_cmap_en         = <0>;
    lcd_rb_swap         = <0>;

    /* part 6 */
    lcd_power       = "vcc-lcd";
    lcd_pin_power       = "vcc-pd";
    /*reset */
    lcd_gpio_0      = <&pio PD 9 1 0 3 1>;
    /* cs */
    lcd_gpio_1      = <&pio PD 10 1 0 3 0>;
    /*sda */
    lcd_gpio_2      = <&pio PD 13 1 0 3 0>;
    /*sck */
    lcd_gpio_3      = <&pio PD 12 1 0 3 0>;
    pinctrl-0 = <&rgb8_pins_a>;
    pinctrl-1 = <&rgb8_pins_b>;
};
```

4.4 MIPI-DSI 接口

4.4.1 概述

MIPI-DSI，即 Mobile Industry Processor Interface Display Serial Interface，移动通信行业处理器接口显示串行接口。

对于用户来说，需要了解：

1. Command mode，类似 MPU 接口，需要 IC 内部有 GRAM 来缓冲。
2. Video mode。类似 RGB 接口，没有 GRAM，需要不停往 panel 刷数据。其中 video mode 又分为三个子 mode。

- Non-burst mode with sync pulses
- Non Burst mode with sync Events
- Burst mode。简单理解就是有效数据比率更高，传输效率更高。

3. lane 的意思是指一对差分管脚。

4.4.2 MIPI-DSI 的管脚

MIPI-DSI 的管脚是在大部分 IC 中是专用，在 board.dtsi 里面不需要配置，只要硬件上连接好就行。

但是有一部分 IC 的 DSI 管脚不是专用的，与其它功能的脚复用，这个时候就需要配置好pinctrl-0和pinctrl-1。

mipi-dsi 的管脚是差分的，分为两种管脚，一种是时钟管脚，另外一种是数据管脚，数据管脚的数量是可变的，数量的单位是 lane，每一条 lane 实际包含两条线。一般来说 LCD 屏说明书里面的说的 lane 的数量是指数据管脚的数量不包括时钟管脚。比如说某 4 lane MIPI-DSI 屏就总共有(4+1)*2根脚。

4.4.3 MIPI-DSI 的电源

一般都有一路电源供给 MIPI-DSI 这个模块，你可以理解为管脚电，也可以理解成模块电，不同 IC 这路电的电压要求可能不同，一旦确定 IC 型号之后，这路电的电压就不变，如果擅自改变此路电的电压可能导致模块异常。

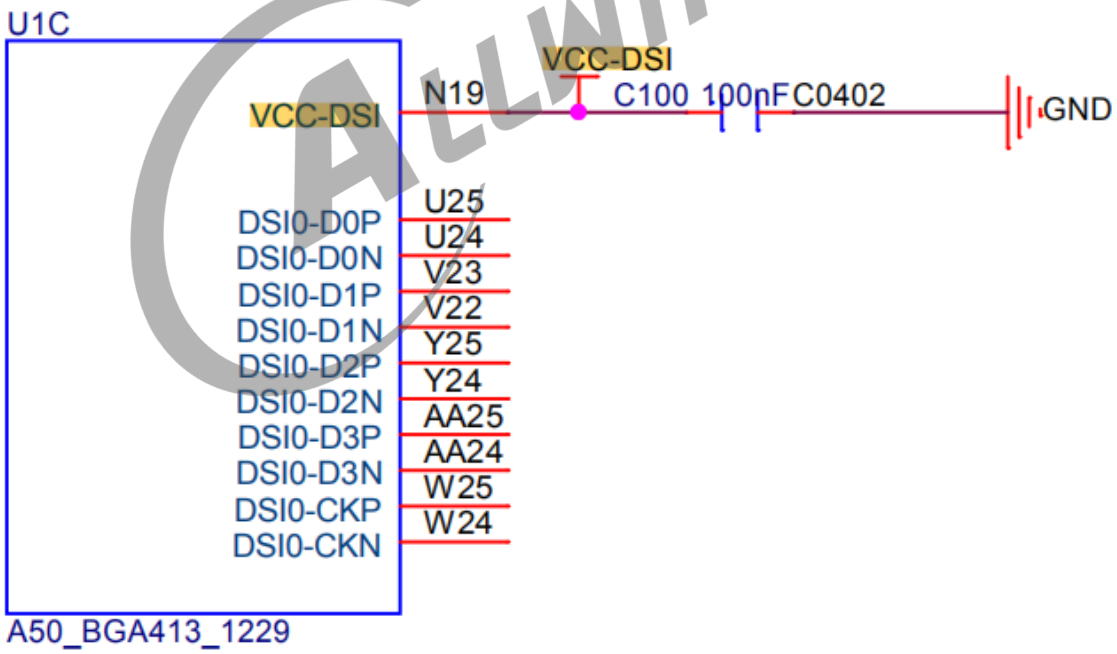

图 4-8: pinmux

4.4.4 判断是否支持某款 MIPI-DSI 屏

1. 分辨率限制。有 lane 的速度限制，我们可以得到最大分辨率的限制，计算公式如下，只要 lane_speed 不超过上面 IC 规格规定的速度，那么理论上是支持的，请查看IC 规格。

lane_speed=lcd_vt * lcd_ht * fps * bit_per_pixel / lane_num / 1e9

- 单位：Gbps。
- fps: 期望刷新率，通过屏手册可知道，一般是 60。请看lcd_dclk_freq。
- bit_per_pixel: 每个像素包含的比特数量，一般是 24 或者 18，通过lcd_dsi_format来设置。
- lane_num:lane 数量，通过lcd_dsi_lane来设置。
- 1e9:1000000000 的科学计数写法。

2. 选择分辨率的同时需要考虑系统带宽，DE 能力，所以即使接口方面支持这个分辨率，对于整个系统来说不一定支持，比如说硬件为了节省成本选择了一款速度很慢的 DDR 内存然后同时又想选择高分辨率的屏幕，很明显这是不现实的。
3. lane 数量限制。绝大部分全志科技 IC 最大支持 4 lane 的 MIPI-DSI，如果你看到该款屏超过 4 lane 就肯定不支持了。少数 IC 最大支持 8 lane，应该选择该款 IC。
4. MIPI-DSI 标准不兼容。请查看IC 规格。

4.4.5 计算 MIPI-DSI 时钟 lane 频率

使用示波器测量 MIPI-DSI 的时钟信号，确定其频率是否满足屏的需求。

首先，我们由给定的像素时钟和 lane 数量，可以计算出理论 CLK 信号的频率，如下公式：

```
Freq_dsi_clk = (Dclk * colordepth * 3 / lane ) / 2
```

1. Freq_dsi_clk：我们要测量的 dsi 时钟脚的频率。单位 MHz。
2. Dclk：像素时钟。由lcd_ht*lcd_vt*fps/1e6公式算出来。
3. Colordepth：颜色深度，一般是 8 或者 6。
4. 乘以 3 表示 RGB 分量 3 个。
5. Lane：dsi 的 lane 数量。
6. 除以 2：是因为 dsi 时钟是双沿采样。

4.4.6 MIPI-DSI Video mode 屏配置示例

绝大多数 MIPI-DSI 屏的配置都是用 video mode。

下面是典型 MIPI-DSI video mode 的 board.dts 配置示例，其中用空行把配置分成几个部分

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。
2. 第二部分，决定该配置是 dsi 接口，而且 dsi 接口使用的是 video mode。
3. 第三部分，决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。
4. 第四部分，背光相关的设置。请看背光相关参数。
5. 第五部分，dsi 接口的详细设置。
6. 第六部分，显示效果相关的设置。
7. 第七部分，管脚和电源设置。请看电源和管脚参数。

```
&lcd0 {
    /* part 1 */
    lcd_used            = <1>;
    lcd_driver_name     = "k101im2qa04";

    /* part 2 */
    lcd_if              = <4>;
    lcd_dsi_if          = <0>;

    /* part 3 */
    lcd_x               = <800>;
    lcd_y               = <1280>;
    lcd_width           = <135>;
    lcd_height          = <216>;
    lcd_dclk_freq       = <68>;
    lcd_hbp             = <36>;
    lcd_ht              = <854>;
    lcd_hspw            = <18>;
    lcd_vbp             = <12>;
    lcd_vt              = <1320>;
    lcd_vspw            = <4>;

    /* part 4 */
    lcd_backlight       = <50>;
    lcd_pwm_used        = <1>;
    lcd_pwm_ch          = <0>;
    lcd_pwm_freq        = <50000>;
    lcd_pwm_pol         = <1>;
    lcd_pwm_max_limit   = <255>;
    lcd_bl_en = <&pio PB 8 1 0 3 1>;
    lcd_bright_curve_en = <0>;

    /* part 5 */
    lcd_dsi_lane        = <4>;
    lcd_dsi_format      = <0>;
    lcd_dsi_te          = <0>;

    /* part 6 */
    lcd_frm             = <0>;
    lcd_gamma_en        = <0>;
    lcd_cmap_en         = <0>;

    /* part 7 */
    lcd_pin_power = "dcdc1";
    lcd_pin_power1 = "eldo3";
    lcd_power = "dc1sw";
    lcd_gpio_0 = <&pio PD 22 1 0 3 1>;
    pinctrl-0 = <&dsi4lane_pins_a>;
```

```
    pinctrl-1 = <&dsi4lane_pins_b>;
};
```

4.4.7 MIPI-DSI 超高分辨率屏配置示例

根据分辨率的高低通常分为几种模式来配置。1080p 分辨率及其以下：只需要设置 lcd_dsi_if 来控制就行。Command mode 一般是低分辨率屏，而 video mode 和 burst mode 则是用于高分辨率的。如果分辨率达到 2k，则需要额外的设置。

分辨率达到 2k 以上的屏，实际上需要多达 8 条数据 lane 才能正常显示，其中四条 lane 发送一副图像中的奇像素，另外一副图像发送偶像素。

说明

注意只有部分 IC 支持超高分辨率，具体查看芯片规格中的 MIPI-DSI 部分

下面是 MIPI-DSI 高分辨超高分辨率（大于 2k）board.dts 配置示例，其中用空行把配置分成几个部分

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。
2. 第二部分，决定该配置是 dsi 接口，而且 dsi 接口使用的是 video mode。
3. 第三部分，决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。
4. 第四部分，背光相关的设置，请看背光相关参数。
5. 第五部分，dsi 接口的详细设置。

说明

lcd_dsi_lane 依旧设置成 4 条 lane 的原因，是因为这个是设置一个 dsi 的 lane 数量，这个屏要用两个 dsi。加起来就是 8 条 lane。
此时 lcd_tcon_mode, lcd_dsi_port_num 和 lcd_tcon_en_odd_even_div 三个选项需要特别设置，点击查看具体含义，如果是 1080p 及其以下分辨率的屏 (只用 4lane 或者以下的)，那么这三个配置默认 0 即可。

6. 第六部分，显示效果部分的设置。
7. 第七部分，是管脚和电源的配置。请根据电路图来配置。请看电源和管脚参数。

```
&lcd0 {

    /* part 1 */
    lcd_used            = <1>;
    lcd_driver_name     = "lq101r1sx03";

    /* part 2 */
    lcd_if              = <4>;
    lcd_dsi_if          = <0>;

    /* part 3 */
    lcd_x               = <2560>;
    lcd_y               = <1600>;
    lcd_width           = <216>;
```

```
    lcd_height      = <135>;
    lcd_dclk_freq   = <268>;
    lcd_hbp         = <80>;
    lcd_ht          = <2720>;
    lcd_hspw        = <32>;
    lcd_vbp         = <37>;
    lcd_vt          = <1646>;
    lcd_vspw        = <6>;

    /* part 4 */
    lcd_backlight   = <50>;
    lcd_pwm_used    = <1>;
    lcd_pwm_ch      = <0>;
    lcd_pwm_freq    = <50000>;
    lcd_pwm_pol     = <1>;
    lcd_pwm_max_limit = <255>;
    lcd_bl_en       = <&pio PH 10 1 0 3 1>;

    /* part 5 */
    lcd_dsi_lane    = <4>;
    lcd_dsi_format  = <0>;
    lcd_dsi_te      = <0>;
    lcd_dsi_port_num    = <1>;
    lcd_tcon_mode       = <4>;
    lcd_tcon_en_odd_even_div = <1>;

    /* part 6 */
    lcd_frm         = <0>;
    lcd_io_phase    = <0x0000>;
    lcd_gamma_en    = <0>;
    lcd_bright_curve_en = <0>;
    lcd_cmap_en     = <0>;

    /* part 7 */
    lcd_power       = "vcc18-lcd";
    lcd_power1      = "vcc33-lcd";
    lcd_pin_power   = "vcc-pd";
    lcd_gpio_0      = <&pio PH 11 1 0 3 1>;
    lcd_gpio_1      = <&pio PH 12 1 0 3 1>;
};
```

4.4.8 MIPI-DSI Command mode 屏配置示例

Command mode 下的 DSI 屏类似与 I8080 接口，屏内部带 RAM 用于缓冲和图像处理，这种情况一般都需要用屏的 te 脚来触发 vsync 中断，所以与其它类型的 DSI 屏不同的是，这里需要设置 lcd_vsync 脚，屏的 te 脚就连到 lcd_vsync 上，并且 lcd_dsi_te 设置成 1。

te 脚的设置非常关键，一般来说如果屏有 te 脚，则必须连上，否则在显示动态画面的时候会画面会撕裂，而且软件无法解决，直接造成最终硬件无法量产的结果。

这里只列举出与 MIPI-DSI video mode 不同的关键之处，其它参考上一小节。

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。

2. 第二部分，决定该配置是 dsi 接口，而且lcd_dsi_if设置成 1 表明 command mode。
3. 第三部分，决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。
4. 第四部分，背光相关的设置。请看背光相关参数。
5. 第五部分，dsi 接口的详细设置。lcd_dsi_te，这里设置为 1 表示使能 te 触发。
6. 第六部分，显示效果相关的设置。
7. 第七部分，管脚和电源设置。lcd_vsync，这里是 te 脚，硬件上需要将这根脚连接到屏的 te 脚，软件上需要将其设置为 vsync 功能。请看电源和管脚参数。

```
&lcd0 {
    /* part 1 */
    lcd_used        = <1>;
    lcd_driver_name    = "h245qbn02";

    /* part 2 */
    lcd_if          = <4>;
    lcd_dsi_if      = <1>;

    /* part 3 */
    lcd_x           = <240>;
    lcd_y           = <432>;
    lcd_width       = <52>;
    lcd_height      = <52>;
    lcd_dclk_freq   = <18>;
    lcd_hbp         = <96>;
    lcd_ht          = <480>;
    lcd_hspw        = <2>;
    lcd_vbp         = <21>;
    lcd_vt          = <514>;
    lcd_vspw        = <2>;

    /* part 4 */
    lcd_backlight   = <100>;
    lcd_pwm_used    = <1>;
    lcd_pwm_ch      = <0>;
    lcd_pwm_freq    = <50000>;
    lcd_pwm_pol     = <1>;
    lcd_pwm_max_limit  = <255>;
    lcd_bright_curve_en = <0>;
    lcd_bl_en       = <&pio PB 3 1 0 3 1>;

    /* part 5 */
    lcd_dsi_lane    = <1>;
    lcd_dsi_format  = <0>;
    lcd_dsi_te      = <1>;

    lcd_frm         = <0>;
    lcd_io_phase    = <0x0000>;
    lcd_gamma_en    = <0>;
    lcd_cmap_en     = <0>;

    /* part 7 */
    lcd_power       = "axp233_dc1sw"
    lcd_power1      = "axp233_eldo1"
    lcd_gpio_0      = <&pio PB 2 1 0 3 0>;
    lcd_vsync       = <&pio PD 21 2 0 3 0>;
};
```

4.4.9 MIPI-DSI VR 双屏配置示例

实际场景是两个物理屏，每个屏是 1080p，每个屏都是 4 条 lane，要求的是两个屏各自显示一帧图像的左右一半，由于宽高比和横竖屏以及 DE 处理能力的因素，一个 DE+ 一个 tcon+ 两个 DSI 已经无法满足，必须用两个 tcon 各自驱动一个 dsi，但是两路显示必须要同步，这就需要用到两个 tcon 的同步模式。

1. LCD0 标记为 slave tcon，它由 master tcon 来驱动（设置lcd_tcon_mode）。
2. LCD1 标记为 master tcon，并且负责两个屏的所有电源，背光，管脚的开关。
3. 把管脚，电源等都放到 LCD1 开，LCD0 先开，对应模块寄存器都初始化，但是电源不开，然后开 LCD1，LCD1 使能就会触发 LCD0 一起发数据。这样做到同时亮灭。

说明

注意：仅有极少 IC 支持该模式

```
;slave
&lcd0 {
    lcd_used        = <1>;

    lcd_driver_name    = "lpm025m475a";
    ;lcd_bl_0_percent   = <0>;
    ;lcd_bl_40_percent  = <23>;
    ;lcd_bl_100_percent = <100>;

    lcd_backlight     = <50>;
    lcd_if          = <4>;
    lcd_x           = <1080>;
    lcd_y           = <1920>;
    lcd_width       = <31>;
    lcd_height      = <56>;
    lcd_dclk_freq     = <141>;

    lcd_pwm_used      = <0>;
    lcd_pwm_ch        = <0>;
    lcd_pwm_freq      = <20000>;
    lcd_pwm_pol       = <0>;
    lcd_pwm_max_limit  = <255>;

    lcd_hbp         = <100>;
    lcd_ht          = <1212>;
    lcd_hspw        = <5>;
    lcd_vbp         = <8>;
    lcd_vt          = <1936>;
    lcd_vspw        = <2>;

    lcd_dsi_if       = <0>;
    lcd_dsi_lane      = <4>;
    lcd_dsi_format     = <0>;
    lcd_dsi_te       = <0>;
    lcd_dsi_eotp       = <0>;

    lcd_frm         = <0>;
    lcd_io_phase      = <0x0000>;
    lcd_hv_clk_phase   = <0>;
    lcd_hv_sync_polarity= <0>;
```

```
    lcd_gamma_en      = <0>;
    lcd_bright_curve_en = <0>;
    lcd_cmap_en       = <0>;


    lcd_dsi_port_num   = <0>;
    lcd_tcon_mode      = <3>;
    lcd_slave_stop_pos = <0>;
    lcd_sync_pixel_num = <0>;
    lcd_sync_line_num  = <0>;
};


&lcd1 {
    lcd_used        = <1>;

    lcd_driver_name    = "lpm025m475a";
    ;lcd_bl_0_percent   = <0>;
    ;lcd_bl_40_percent  = <23>;
    ;lcd_bl_100_percent = <100>;

    lcd_backlight     = <50>;
    lcd_if          = <4>;
    lcd_x           = <1080>;
    lcd_y           = <1920>;
    lcd_width       = <31>;
    lcd_height      = <56>;
    lcd_dclk_freq     = <141>;

    lcd_pwm_used      = <1>;
    lcd_pwm_ch        = <0>;
    lcd_pwm_freq      = <20000>;
    lcd_pwm_pol       = <0>;
    lcd_pwm_max_limit  = <255>;

    lcd_hbp         = <100>;
    lcd_ht          = <1212>;
    lcd_hspw        = <5>;
    lcd_vbp         = <8>;
    lcd_vt          = <1936>;
    lcd_vspw        = <2>;

    lcd_dsi_if       = <0>;
    lcd_dsi_lane      = <4>;
    lcd_dsi_format    = <0>;
    lcd_dsi_te        = <0>;
    lcd_dsi_eotp      = <0>;

    lcd_frm         = <0>;
    lcd_io_phase      = <0x0000>;
    lcd_hv_clk_phase   = <0>;
    lcd_hv_sync_polarity= <0>;
    lcd_gamma_en      = <0>;
    lcd_bright_curve_en = <0>;
    lcd_cmap_en       = <0>;


    lcd_dsi_port_num   = <0>;
    lcd_tcon_mode      = <1>;
    lcd_tcon_slave_num = <0>;
```

```
    lcd_slave_stop_pos = <0>;
    lcd_sync_pixel_num = <0>;
    lcd_sync_line_num  = <0>;

    lcd_bl_en       = <&pio PH 10 1 0 3 1>;
    lcd_power       = "vcc-dsi";
    lcd_power1      = "vcc18-lcd";
    lcd_power2      = "vcc33-lcd";

    lcd_gpio_0      = <&pio PH 8 1 0 3 1>;
    lcd_gpio_1      = <&pio PH 11 1 0 3 1>;
    lcd_gpio_2      = <&pio PH 12 1 0 3 1>;
    lcd_pin_power   = "vcc-ph"
};
```

4.5 I8080 接口

4.5.1 概述

Intel 8080 接口屏 (又称 MCU 接口) 很老的协议，一般用在分辨率很小的屏上。

信号线：

- CS 片选信号，决定该芯片是否工作。
- RS 寄存器选择信号，低表示选择 index 或者 status 寄存器，高表示选择控制寄存器。实际场景中一般接 SoC 的 LCD_DE 脚（数据使能脚）。
- /WR （低表示写数据) 数据命令区分信号，也就是写时钟信号，一般接 SoC 的 LCD_CLK 脚。
- /RD （低表示读数据）数据读信号，也就是读时钟信号，一般接 SoC 的 LCD_HSYNC 脚。
- RESET 复位 LCD（用固定命令系列 0 1 0 来复位)。
- Data 双向传输的数据总线。

I8080 根据的数据位宽接口有 8/9/16/18，连哪些脚参考，即使位宽一样，连的管脚也不一样，还要考虑的因素是 rgb 格式。

1. RGB565，总共有 65K 这么多种颜色。
2. RGB666，总共有 262K 那么多种颜色。
3. 9bit 固定为 262K。

从屏手册得知：数据位宽，颜色数量之和，参考RGB 和 I8080 管脚配置示意图，进行硬件连接。

4.5.2 I8080 接口屏典型配置示例

下面是典型是一个 RGB565 的，位宽为 8 位的 I8080 接口的屏的 board.dts 配置示例

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。
2. 第二部分，决定该配置是 I8080 接口，而且是 8bit/2cycle 格式 RGB565。

技巧

为什么叫做 8bit/2cycle RGB565 呢，首先它的格式是 RGB565，也就是一个像素是 16bit，然后它是 8bit 的位宽，就需要两个时钟周期才能发完一个像素，所以才叫 2 cycle。

3. 第三部分，决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。这里比较特殊的是设置像素时钟要满足以下公式：lcd_dclk_freq*2>=lcd_ht*lcd_vt*fps，或者lcd_dclk_freq=lcd_ht*2*lcd_vt*60，也就是要么双倍lcd_ht要么双倍lcd_dclk_freq。
4. 第四部分，背光相关的设置。请看背光相关参数。
5. 第五部分，cpu 接口的详细设置。这里使能了lcd_cpu_te和lcd_cpu_mode，意思是使用 te 触发和规定了触发间隔。这是非常关键的设置。
6. 第六部分，显示效果相关的设置。这里使能了lcd_frm也是比较关键的设置，详细意思点击查看。
7. 第七部分，管脚和电源设置。这里为了用 te 触发，同样需要设置 lcd_vsync，该脚功能定义已经包括在 pinctrl-0 中。这里自定义了一组管脚。参考RGB 和 I8080 管脚配置示意图，通过确定 I8080 的位宽，像素格式（颜色数量），在表中确定需要连接哪些管脚。请看电源和管脚参数。

```
&pio {

  I8080_8bit_pins_a: I8080_8bit@0 {
    allwinner,pins = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,pname = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,function = "I8080_8bit";
    allwinner,muxsel = <2>;
    allwinner,drive = <3>;
    allwinner,pull = <0>;
  };
  I8080_8bit_pins_b: I8080_8bit@1 {
    allwinner,pins = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,pname = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,function = "I8080_8bit_suspend";
    allwinner,muxsel = <7>;
    allwinner,drive = <3>;
    allwinner,pull = <0>;
  };

};

&lcd0 {

  /* part 1 */
  lcd_used        = <1>;
  lcd_driver_name    = "s2003t46g";

  /* part 2 */
  lcd_if          = <1>;
  lcd_cpu_if       = <14>;
```

```
    /* part 3 */
    lcd_x           = <240>;
    lcd_y           = <320>;
    lcd_width       = <108>;
    lcd_height      = <64>;
    lcd_dclk_freq   = <16>;
    lcd_hbp         = <20>;
    lcd_ht          = <298>;
    lcd_hspw        = <10>;
    lcd_vbp         = <8>;
    lcd_vt          = <336>;
    lcd_vspw        = <4>;

    /* part 4 */
    lcd_pwm_used    = <1>;
    lcd_pwm_ch      = <8>;
    lcd_pwm_freq    = <50000>;
    lcd_pwm_pol     = <1>;
    lcd_pwm_max_limit  = <255>;
    lcd_bright_curve_en = <1>;


    /* part 5 */
    lcd_cpu_mode    = <1>;
    lcd_cpu_te      = <1>;

    /* part 6 */
    lcd_frm         = <1>;
    lcd_gamma_en    = <0>;
    lcd_cmap_en     = <0>;
    lcd_rb_swap     = <0>;

    /* part 7 */
    lcd_power       = "vcc-lcd";
    lcd_pin_power   = "vcc-pd";
    ;reset pin
    lcd_gpio_0      = <&pio PD 9 1 0 3 1>;
    ;cs pin
    lcd_gpio_1      = <&pio PD 10 1 0 3 0>;
    pinctrl-0 = <&I8080_8bit_pins_a>;
    pinctrl-1 = <&I8080_8bit_pins_a>;

};
```

4.6 LVDS 接口

4.6.1 概述

LVDS 即 Low Voltage Differential Signaling 是一种低压差分信号接口。

4.6.2 LVDS Single link 典型配置

LVDS 接口，lcd0 对应的 lvds 管脚和 lcd1 对应的 lvds 管脚是固定而且不一样。

由于 lvds 协议不具备传输数据之外的能力，一般屏端不需要任何初始化，只需要初始化 SoC 端即可。所以这里的 lcd_driver_name 依旧是”default_lcd”，当然你可以为初始化的启动延时做专门的优化。

下面是典型是 single link lvds 屏的 board.dts 配置示例，其中用空行把配置分成几个部分

1. 第一部分，决定该配置是否使用，以及使用哪个屏驱动，lcd_driver_name 决定了用哪个屏驱动来初始化。
2. 第二部分，决定该配置是 lvds 接口，而且是 single link。

技巧

如果 Dual Link 的屏，那么除了要改 lcd_lvds_if 为 1 之外，管脚方面还要把 lcd1 的管脚一起搬到下面去，也就是总共需要配置 PD0 到 PD9，和配置 PD10 到 PD19 总共二十根脚为 lvds 管脚功能（功能 3）。当然屏的 timing 也是要根据屏来改的。

3. 第三部分，决定了 SoC 中的 LCD 模块发送时序，请查看屏时序参数说明。
4. 第四部分，背光相关的设置。请看背光相关参数。
5. 第五部分，lvds 接口的详细设置。
6. 第六部分，显示效果相关的设置。
7. 第七部分，管脚和电源设置。请看电源和管脚参数。

```
&lcd0 {
    /* part 1 */
    lcd_used            = 1
    lcd_driver_name     = "default_lcd";

    /* part 2 */
    lcd_if              = 3
    lcd_lvds_if         = 0

    /* part 3 */
    lcd_x               = 1280
    lcd_y               = 800
    lcd_width           = 150
    lcd_height          = 94
    lcd_dclk_freq       = 70
    lcd_hbp             = 20
    lcd_ht              = 1418
    lcd_hspw            = 10
    lcd_vbp             = 10
    lcd_vt              = 814
    lcd_vspw            = 5

    /* part 4 */
    lcd_pwm_used        = 1
    lcd_pwm_ch          = 0
    lcd_pwm_freq        = 50000
    lcd_pwm_pol         = 0
```

```
    lcd_pwm_max_limit  = 255
    lcd_backlight      = 50
    lcd_bright_curve_en = 0
    lcd_bl_en          = <&pio PD 21 1 0 3 1>;

    /* part 5 */
    lcd_lvds_colordepth = 1
    lcd_lvds_mode      = 0

    /* part 6 */
    lcd_frm            = 1
    lcd_hv_clk_phase   = 0
    lcd_hv_sync_polarity= 0
    lcd_gamma_en       = 0
    lcd_cmap_en        = 0

    /* part 7 */
    lcd_power          = "vcc-lcd"
    pinctrl-0 = <&lvds0_pins_a>;
    pinctrl-1 = <&lvds0_pins_b>;
};
```

4.6.3 LVDS dual link 典型配置

如果 Dual Link 的屏:

1. lcd_lvds_if设置为 1（场景 1）或者 2（场景 2）。
2. 管脚配置方面，也从 4 data lane 变成 8 data lane，包括 clk lane 总共 20 根管脚。

场景 1，物理上连接一个屏，8 data lane，SoC 向每 4 条 lane 传输一半的像素，奇数像素或者偶数像素。

```
&lcd1 {
        lcd_used          = <1>;

        lcd_driver_name   = "bp101wx1";
        lcd_backlight     = <50>;
        lcd_if            = <3>;

        lcd_x             = <2560>;
        lcd_y             = <800>;
        lcd_width         = <150>;
        lcd_height        = <94>;
        lcd_dclk_freq     = <138>;

        lcd_pwm_used      = <0>;
        lcd_pwm_ch        = <2>;
        lcd_pwm_freq      = <50000>;
        lcd_pwm_pol       = <1>;
        lcd_pwm_max_limit  = <255>;


        lcd_hbp           = <40>;
        lcd_ht            = <2836>;
        lcd_hspw          = <20>;
```

```
        lcd_vbp         = <10>;
        lcd_vt          = <814>;
        lcd_vspw        = <5>;

        lcd_lvds_if     = <1>;
        lcd_lvds_colordepth = <0>;
        lcd_lvds_mode   = <0>;
        lcd_frm         = <0>;
        lcd_hv_clk_phase  = <0>;
        lcd_hv_sync_polarity= <0>;
        lcd_gamma_en    = <0>;
        lcd_bright_curve_en = <0>;
        lcd_cmap_en     = <0>;
        lcd_fsync_en    = <0>;
        lcd_fsync_act_time  = <1000>;
        lcd_fsync_dis_time  = <1000>;
        lcd_fsync_pol   = <0>;

        deu_mode        = <0>;
        lcdgamma4iep    = <22>;
        smart_color     = <90>;
        lcd_bl_en       = <&pio PJ 27 1 0 3 1>;
        lcd_gpio_0      = <&pio PI 1 1 0 3 1>;

        lcd_pin_power = "bldo5";

        lcd_power = "dc1sw";

        pinctrl-0 = <&lcd1_lvds2link_pins_a>;
        pinctrl-1 = <&lcd1_lvds2link_pins_b>;

    };
```

场景 2（部分 IC 支持），物理上连接两个屏，每个屏各自 4 条 lane，两个屏是一样型号，分辨率和 timing 一样，这时候部分 IC 支持将全部像素发到每个屏上，实现双显（信号上的双显），注意这时候 lcd timing 是一个屏的 timing, lcd_lvds_if 为 2。

```
lcd1: lcd1@01c0c001 {
        lcd_used        = <1>;

        lcd_driver_name     = "bp101wx1";
        lcd_backlight   = <50>;
        lcd_if          = <3>;

        lcd_x           = <1280>;
        lcd_y           = <800>;
        lcd_width       = <150>;
        lcd_height      = <94>;
        lcd_dclk_freq   = <70>;

        lcd_pwm_used    = <0>;
        lcd_pwm_ch      = <2>;
        lcd_pwm_freq    = <50000>;
        lcd_pwm_pol     = <1>;
        lcd_pwm_max_limit  = <255>;


        lcd_hbp         = <20>;
        lcd_ht          = <1418>;
```

```
        lcd_hspw        = <10>;
        lcd_vbp         = <10>;
        lcd_vt          = <814>;
        lcd_vspw        = <5>;

        lcd_lvds_if     = <2>;
        lcd_lvds_colordepth = <0>;
        lcd_lvds_mode   = <0>;
        lcd_frm         = <0>;
        lcd_hv_clk_phase  = <0>;
        lcd_hv_sync_polarity= <0>;
        lcd_gamma_en    = <0>;
        lcd_bright_curve_en = <0>;
        lcd_cmap_en     = <0>;
        lcd_fsync_en    = <0>;
        lcd_fsync_act_time  = <1000>;
        lcd_fsync_dis_time  = <1000>;
        lcd_fsync_pol   = <0>;

        deu_mode        = <0>;
        lcdgamma4iep    = <22>;
        smart_color     = <90>;
        lcd_bl_en     = <&pio PJ 27 1 0 3 1>;
        lcd_gpio_0    = <&pio PI 1 1 0 3 1>;

        lcd_pin_power = "bldo5";

        lcd_power = "dc1sw";

        pinctrl-0 = <&lcd1_lvds2link_pins_a>;
        pinctrl-1 = <&lcd1_lvds2link_pins_a>;

    };
```

4.7 RGB 和 I8080 管脚配置示意图

	SYNC RGB				CPU cmd	CPU 18bit	CPU 16bit								CPU 8bit					CPU 9bit				
I/O						256K	256K							65K	256K			65K		256K				
		1st	2nd	3rd			1st	2nd	3rd	1st	2nd	1st	2nd		1st	2nd	3rd	1st	2nd	1st	2nd			
IO0	VSYNC				CS																			
IO1	HSYNC				RD																			
IO2	DCLK				WR																			
IO3	DE				RS																			
D23	R7					D23	R5	R5	B5	G5	R5		R5	B5	R4										
D22	R6					D22	R4	R4	B4	G4	R4		R4	B4	R3										
D21	R5					D21	R3	R3	B3	G3	R3		R3	B3	R2										
D20	R4					D20	R2	R2	B2	G2	R2		R2	B2	R1										
D19	R3					D19	R1	R1	B1	G1	R1		R1	B1	R0										
D18	R2					D18	R0	R0	B0	G0	R0		R0	B0	G5										
D17	R1					D17																			
D16	R0					D16																			
D15	G7					D15	G5								G4										
D14	G6					D14	G4								G3										
D13	G5					D13	G3																		
D12	G4	D17	D27	D37	D7	D12	G2	G5	R5	B5	G5	B5	G5		G2	R5	G5	B5	R4	G2	R5	G2			
D11	G3	D16	D26	D36	D6	D11	G1	G4	R4	B4	G4	B4	G4		G1	R4	G4	B4	R3	G1	R4	G1			
D10	G2	D15	D25	D35	D5	D10	G0	G3	R3	B3	G3	B3	G3		G0	R3	G3	B3	R2	G0	R3	G0			
D9	G1					D9																			
D8	G0					D8																			
D7	B7	D14	D24	D34	D4	D7	B5	G2	R2	B2	G2	B2	G2		B4	R2	G2	B2	R1	B4	R2	B5			
D6	B6	D13	D23	D33	D3	D6	B4	G1	R1	B1	G1	B1	G1		B3	R1	G1	B1	R0	B3	R1	B4			
D5	B5	D12	D22	D32	D2	D5	B3	G0	R0	B0	G0	B0	G0		B2	R0	G0	B0	G5	B2	R0	B3			
D4	B4	D11	D21	D31	D1	D4	B2								B1				G4	B1	G5	B2			
D3	B3	D10	D20	D30	D0	D3	B1								B0				G3	B0	G4	B1			
D2	B2					D2	B0														G3	B0			
D1	B1					D1																			
D0	B0					D0																			

图 4-9: pinmux

4.8 单固件自适应多款 LCD 屏

4.8.1 功能说明

自适应多款 LCD 屏，其实就是单一固件同时支持多款接口、型号不一的 LCD 屏幕模组（主要特指屏驱动、时序不能共用），即支持动态识别 LCD 模组型号，并加载对应 LCD 驱动并以相应的 LCD 屏幕进行显示。

对于多屏兼容的实现，除了将需要兼容的屏全部在对应的平台调通点亮外，另一个关键是如何将当前正在使用的屏识别出来并告知驱动框架以加载对应的屏驱动，一般而言有如下几种方法：

- 当使用不同屏时，硬件上也将某些 io 同时拉高或拉低，用 io 电平甚至电压标识屏模组。
- 出厂时根据使用的屏型号，往 flash 中同时烧录一个屏模组型号标识。
- 对 mipi 等支持通信的屏，通过读屏 id 区分。

上述三种方法的缺点如下：

- 对硬件依赖高，需要占用 SOC 宝贵的 IO 口，屏端或者 SOC 的 PCB 端需要提前做好将 IO 电平拉高或拉低，当需要兼容屏较多时，不能简单通过高低电平区分。
- 需要额外烧录一个标志，产线成本有所增加；且一旦标识烧录完成，只支持对应的屏，不能混用其他屏，一定程度上不算多屏兼容。
- 限定屏的接口必须是 mipi-dsi，由于需要逐一 try，耗时较长。

上述三种方法的优点如下：

- 区分方法简单，读 io 软件实现简单可靠。短耗时。
- 不要求额外硬件资源，纯软实现，相对简单可靠，耗时可控。
- 使用上最灵活，不占用额外硬件资源，不需要额外产线操作。

说明

当且仅当使用屏为mipi-dsi接口时，才能同一板子烧录固件后不做任何修改换上任意兼容屏都可正常显示！
上述方法2，需要额外烧录一个表示屏物料的标识，烧录标识完成后，仅支持代表当前标识的屏，不能兼容其他屏。
上述方法1，若io的拉高拉低不是在lcd端完成而是在SOC端的PCB完成的，一旦对应引脚的电平定好，将不能点亮其他屏；但如果对应引脚是在屏端确定电平，SOC的PCB端是悬空的，实际也能实现同一块板子烧录完固件后不做任何修改换上任意兼容屏都可正常显示的效果，但硬件上屏端不一定可以实现将某IO电平拉高或拉低。

uboot 和 kernel 分别都有单固件自适应多款 LCD 屏的功能，如果方案支持 uboot-2018，则建议使用 uboot 实现的多款 LCD 屏自适应功能。如果方案不需要 uboot(极少)，比如某些快起项目会将 uboot 裁剪掉，那么就使用 kernel 的多屏自适应功能。

4.8.2 Uboot 单固件自适应多款 LCD 屏的使用方法

该功能目前只支持在单屏显示时兼容多屏（也就是 lcd0），并且使用前为了避免不必要的麻烦需要先确保兼容的屏在不使用屏兼容功能时能正常点亮。具体使用方法如下：

1. Uboot 及内核的 dts 中的 soc 节点下增加节点 lcd0_1（或者 lcd0_2、lcd0_3，根据需要按顺序增加, 参考 lcd0 节点确保编译通过），并填入需要对应的屏参数及配置。
2. 在默认的屏，也就是 lcd0 的屏驱动中添加读屏 id(或其他方法) 然后切换的逻辑，切换的接口为 sunxi_lcd_switch_compat_panel。具体可参照下图 4-8“屏驱动切换示例”。
3. 在调用 sunxi_lcd_switch_compat_panel 前，确保执行了之前调用的开屏流程对应的关屏流程，类似图 “屏驱动切换示例” 中的 LCD_power_off。
4. 确保在 LCD_open_flow 中，调用 sunxi_lcd_switch_compat_panel 所在的开屏函数以及之前的开屏函数的延时都是 0，具体参照下图 4-9“屏驱动切换延时”。

```
static void LCD_panel_try_switch(u32 sel)
{
        u8 result[16] = {0};
        u32 num = 0;
        sunxi_lcd_delay_ms(100);
        sunxi_lcd_dsi_dcs_read(sel, 0x04, result, &num);
        printf("get lcd id 0x%x\n", result[0]);

        if (result[0] == 0x93) {
                LCD_power_off(sel);
                sunxi_lcd_switch_compat_panel(sel, 1);/*switch to lcd0_1*/
                return;
        }
}
```

图 4-10: 屏驱动切换示例

```
static s32 LCD_open_flow(u32 sel)
{
        LCD_OPEN_FUNC(sel, LCD_power_on, 0);   //must delay 0
        LCD_OPEN_FUNC(sel, LCD_panel_try_switch, 0);   //must delay 0
        LCD_OPEN_FUNC(sel, LCD_panel_init, 200);   //open lcd power, than delay 200ms
        LCD_OPEN_FUNC(sel, sunxi_lcd_tcon_enable, 50);     //open lcd controller, and delay 50ms
        LCD_OPEN_FUNC(sel, LCD_bl_open, 0);     //open lcd backlight, and delay 0ms

        return 0;
}
```

图 4-11: 屏驱动切换延时

参考配置：

1. brandy/brandy-2.0/u-boot-2018/drivers/video/sunxi/disp2/disp/lcd/K080_IM2HYL802R_800X1280.c
2. brandy/brandy-2.0/u-boot-2018/arch/arm/dts/a133-b4-board.dts或device/config/chips/a133/configs/b4/uboot-board.dts
3. device/config/chips/a133/configs/b4/linux-5.4/board.dts

其他说明：

1. 由于实现原理是通过 uboot 替换 kernel 的 lcd 相关的 dts 配置，kernel 的屏驱动无须做任何兼容处理，屏兼容对内核是完全透明的。

2. 为了加快机器的启动速度，默认在第一次启动时记录当前使用的兼容屏，后续启动时将不执行 try 屏驱动的逻辑而直接使用之前记录的屏驱动。如不需要该功能，需要手动关闭 uboot 配置项 “COMPATIBLE_PANEL_RECORD”。
3. 多屏兼容的重点是识别当前使用的屏物料，功能说明章节中有常用的识别方法及限制说明，请仔细阅读该部分内容，综合评估做好取舍。

4.8.3 kernel 单固件自适应多款 LCD 屏的使用方法

这个功能使用的前提是屏端支持读写命令，可以通过读命令读取屏 ID，暂不支持通过前述其他方式区分屏。

目前在 linux-4.9 内核下有相关实现。使用方法：

步骤一首先打开如下内核配置

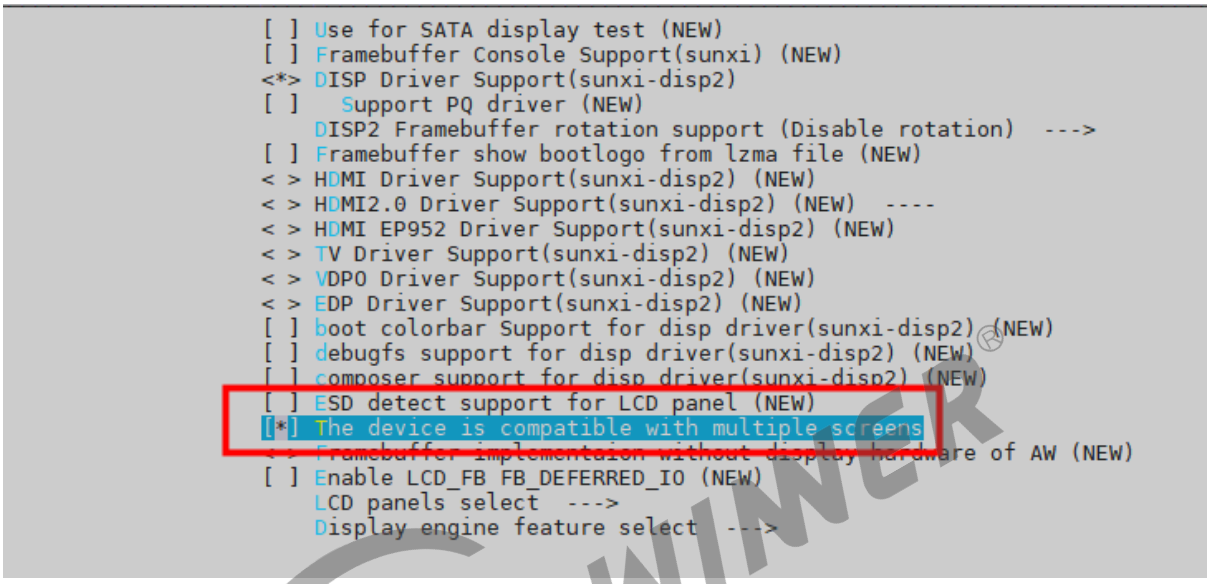

图 4-12: 单固件自适应多款 LCD 屏配置

步骤二、增加自适应屏同等数量的设备树节点：

需要增加自适应屏的设备树节点的命令格式是：lcd{显示通道}_ 屏子序号。比如要增加一个自适应屏，那么要新增的设备树节点如下：

```
soc设备树新增内容：
    lcd0_1: lcd0_1@05462000 {
        compatible = "allwinner,sunxi-lcd0_1"; // 注意这个的sunxi-lcd0_1的后半段 和节点名一样
        pinctrl-names = "active","sleep";

        status = "okay";
    };

板级设备树新增内容：
&lcd0_1 {
  base_config_start  = <1>;
  lcd_used           = <1>;

  lcd_driver_name    = "t050k589_1";

  lcd_bl_0_percent   = <0>;
  lcd_bl_40_percent  = <23>;
  lcd_bl_100_percent = <100>;
  lcd_backlight      = <150>;
  //lcd_backlight    = <80>;
```

```
    lcd_if          = <4>;
    lcd_x           = <720>;
    lcd_y           = <1280>;
    lcd_width       = <62>;
    lcd_height      = <110>;
    lcd_dclk_freq   = <60>;

    lcd_pwm_used    = <1>;
    lcd_pwm_ch      = <6>;
    lcd_pwm_freq    = <50000>;
    lcd_pwm_pol     = <1>;
    lcd_pwm_max_limit  = <255>;

    lcd_hbp         = <32>;
    lcd_ht          = <780>;
    lcd_hspw        = <8>;
    lcd_vbp         = <12>;
    lcd_vt          = <1304>;
    lcd_vspw        = <4>;

    lcd_dsi_if      = <0>;
    lcd_dsi_lane    = <2>;
    lcd_dsi_format  = <0>;
    lcd_dsi_te      = <0>;
    lcd_dsi_eotp    = <0>;

    lcd_frm         = <0>;
    lcd_io_phase    = <0x0000>;
    lcd_hv_clk_phase   = <0>;
    lcd_hv_sync_polarity= <0>;
    lcd_gamma_en    = <0>;
    lcd_bright_curve_en = <0>;
    lcd_cmap_en     = <0>;

    lcdgamma4iep    = <22>;

    lcd_gpio_0      = <&pio PD 19 1 0 3 1>;
    lcd_gpio_1      = <&pio PE 1 1 0 3 1>;
    //lcd_gpio_2      = <&pio PD 7 1 0 3 1>;
    pinctrl-0       = <&dsi4lane_pins_a>;
    pinctrl-1       = <&dsi4lane_pins_b>;
    base_config_end   = <1>;
    status      = "okay";
};
```

步骤三、修改屏驱动，实现检查屏 ID 是否正确的功能：

```
static s32 lcd_panel_id_check(u32 sel)
{
        s32 ret = 0;
        u32 num = 0;
        u8 pn[3];
        struct LCM_setting_table lst ={0x00,2,{0xE0,0x00}};
        sunxi_lcd_dsi_set_max_ret_size(sel,3);
        sunxi_lcd_dsi_dcs_write(sel, lst.cmd,lst.para_list,lst.count);
        ret=sunxi_lcd_dsi_dcs_read(sel, 0x04, pn, &num);
        printk("[%s] pn[0]:0x%x,pn1[1]:0x%x,pn1[2]:0x%x,num:0x%x,ret:%u\n", __FILE__, pn[0],pn[1],pn[2],num,ret);

        if(pn[0]!=0x93 || pn[1] != 0x65)
        {
                ret = -1;
        }

        return ret;


}
```

图 4-13: panel_id_check 函数实现

```
struct __lcd_panel t050k589_1_panel = {
        /* panel driver name, must mach the name of
         * lcd_drv_name in sys_config.fex
         */
        .name = "t050k589_1",
        .func = {
                .cfg_panel_info = lcd_cfg_panel_info,
                        .cfg_open_flow = lcd_open_flow,
                        .cfg_close_flow = lcd_close_flow,
                        .lcd_user_defined_func = lcd_user_defined_func,

                        .esd_check = lcd_esd_check,
                        .reset_panel = lcd_reset_panel,
                        .set_esd_info = lcd_set_esd_info,
                        .panel_id_check = lcd_panel_id_check,


        },
};
NORMAL  lcd/t050k589_1.c
9 disp.ko|1696 col 5711| <80> ^P ^T<80> ¹<9b>^A ^R ^R ^B )<80> ¡ö^A ^^ ^R ^B 5<80> öP^B 2 ^R ^B ?<80>
```

图 4-14: 设置 panel_id_checkf 方法

不同屏之间读 ID 的方法差异很大，需要根据具体的屏做实现。当这个函数返回-1，表示屏 ID 不对，显示驱动会自动切到下一个序号的 lcd 屏驱动的 panel_id_check 函数继续尝试读屏 ID，当屏驱动 panel_id_check 函数返回 0 时，表示读取的 ID 和屏匹配，显示驱动会加载这个驱动对应的参数信息，并最后点亮屏。

验证方法

可以在屏驱动的初始化函数中加入打印，在驱动加载后，只有 panel_id_check 函数返回 0 的屏驱动的初始化函数才会被执行。

4.9 从 sys_config.fex 到 board.dtsi 的迁移注意事项

为了规范等原因，部分平台将配置放在 board.dtsi 中实现。下面说明了修改 board.dtsi 的注意事项。

4.9.1 管脚定义

在配置 RGB 屏或者 LVDS 屏时，现在不再需要复杂的定义，也不需要了解到底哪些管脚需要配置，也不需要lcd0_suspend节点了。其中rgb24_pins_a这个名字是定义好的，直接用即可，一般 LCD 屏直接可用的配置会在注释中写明，你可以在内核目录下arch/arm/boot/dts或者arch/arm64/boot/dts下的**平台-pinctrl.dtsi** 文件中找。

例子:

```
pinctrl-0 = <&rgb24_pins_a>;
pinctrl-1 = <&rgb24_pins_b>;//休眠时候的定义，io_disable
```

当然，你也可以自定义一组脚，写在 board.dtsi 中，只要名字不要和现有名字重复就行。

为了规范，我们将在所有平台保持一致的名字，其中后缀为 a 即为管脚使能，b 的为 io_disable 用于设备关闭时。

目前有以下管脚定义可用：

表 4-2: 显示管脚名称表

管脚名称	描述
rgb24_pins_a 和 rgb24_pins_b	RGB 屏接口，而且数据位宽是 24，RGB888
rgb18_pins_a 和 rgb18_pins_b	RGB 屏接口，而且数据位宽是 16，RGB666
lvds0_pins_a 和 lvds0_pins_b	Single link LVDS 接口 0 管脚定义（主显 lcd0）
lvds1_pins_a 和 lvds1_pins_b	Single link LVDS 接口 1 管脚定义（主显 lcd0
lvds2link_pins_a 和 lvds2link_pins_b	Dual link LVDS 接口管脚定义（主显 lcd0）
lvds2_pins_a 和 lvds2_pins_b	Single link LVDS 接口 0 管脚定义（主显 lcd1）
lvds3_pins_a 和 lvds3_pins_b	Single link LVDS 接口 1 管脚定义（主显 lcd1）
lcd1_lvds2link_pins_a 和 lcd1_lvds2link_pins_b	Dual link LVDS 接口管脚定义（主显 lcd1）
dsi4lane_pins_a 和 dsi4lane_pins_b	DSI 屏接口管脚定义，4lane，如果是其它 lane 数量，只需要

4.9.2 电源定义

电源定义在旧的 SDK 中并不需要注意什么，还是直接把 axp 的别名字符串赋值给想 lcd_power 这样的属性上即可，但是新的 SDK 中，如果需要使用某路电源必须先在 disp 节点中定义，然后 lcd 部分使用的字符串则要和 disp 中定义的一致。比如下面的例子：

```
disp: disp@01000000 {
        disp_init_enable      = <1>;
        disp_mode             = <0>;

         /* VCC-LCD */
        dc1sw-supply = <&reg_sw>;
        /* VCC-LVDS and VCC-HDMI */
        bldo1-supply = <&reg_bldo1>;
```

```
        /* VCC-TV */
        cldo4-supply = <&reg_cldo4>;
};
```

其中”-supply” 是固定的，它之前的字符串则是随意的，不过建议取有意义的名字。而后面的像 <&reg_sw> 则必须在 board.dtsi 的 regulator0 节点中找到。

然后 lcd0 节点中，如果要使用 reg_sw，则类似下面这样写就行，dc1sw 对应 dc1sw-supply。

```
lcd_power=”dc1sw”
```

由于 u-boot 中也有 axp 驱动和 display 驱动，它们和内核一样，都是读取同份配置，为了能互相兼容，取名的时候，有以下限制。

在 u-boot 2018 中，axp 驱动只认类似 bldo1 这样从 axp 芯片中定义的名字，所以命名 xxx-supply 的时候最好按照这个 axp 芯片的定义来命名。

4.9.3 其它注意事项

board.dtsi 里面可能只有 lcd0 没有 lcd1，或只有 tv0 没有 tv1，这时候你要添加的话，需要参考内核目录 arch/arm/boot/dts 或者 arch/arm64/boot/dts 下对应的平台.dtsi 文件。其中最关键的是 @ 后面那串地址必须与内核中定义一致，比如：

```
lcd1: lcd1@01c0c000
```

5 硬件参数说明

5.1 LCD 接口参数说明

5.1.1 lcd_driver_name

Lcd 屏驱动的名字（字符串），必须与屏驱动的名字对应。

5.1.2 lcd_model_name

Lcd 屏模型名字，非必须，可以用于同个屏驱动中进一步区分不同屏。

5.1.3 lcd_if

Lcd Interface

设置相应值的对应含义为：

```
0：HV RGB接口
1：CPU/I80接口
2：Reserved
3：LVDS接口
4：DSI接口
```

5.1.4 lcd_hv_if

Lcd HV panel Interface

这个参数只有在 lcd_if=0 时才有效。定义 RGB 同步屏下的几种接口类型。

设置相应值的对应含义为：

```
0：Parallel RGB
8：Serial RGB
10：Dummy RGB
11：RGB Dummy
12：Serial YUV (CCIR656)
```

5.1.5 lcd_hv_clk_phase

Lcd HV panel Clock Phase

这个参数只有在 lcd_if=0 时才有效。定义 RGB 同步屏的 clock 与 data 之间的相位关系。总共有 4 个相位可供调节。

设置相应值的对应含义为：

```
0: 0 degree
1: 90 degree
2: 180 degree
3: 270 degree
```

5.1.6 lcd_hv_sync_polarity

Lcd HV panel Sync signals Polarity

这个参数只有在 lcd_if=0 时才有效。定义 RGB 同步屏的 hsync 和 vsync 的极性。

设置相应值的对应含义为：

```
0：vsync active low，hsync active low
1：vsync active high，hsync active low
2：vsync active low，hsync active high
3：vsync active high，hsync active high
```

5.1.7 lcd_hv_srgb_seq

Lcd HV panel Serial RGB output Sequence

这个参数只有在 lcd_if=0 且 lcd_hv_if=8（Serial RGB）时才有效。

定义奇数行 RGB 输出的顺序：

```
0:  Odd lines R-G-B; Even line R-G-B
1:  Odd lines B-R-G; Even line R-G-B
2:  Odd lines G-B-R; Even line R-G-B
4:  Odd lines R-G-B; Even line B-R-G
5:  Odd lines B-R-G; Even line B-R-G
6:  Odd lines G-B-R; Even line B-R-G
8:  Odd lines R-G-B; Even line B-R-G
9:  Odd lines B-R-G; Even line G-B-R
10: Odd lines G-B-R; Even line G-B-R
```

5.1.8 lcd_hv_syuv_seq

Lcd HV panel Serial YUV output Sequence

这个参数只有在 lcd_if=0 且 lcd_hv_if=12（Serial YUV）时才有效。

定义 YUV 输出格式：

```
0：YUYV
1：YVYU
2：UYVY
3：VYUY
```

5.1.9 lcd_hv_syuv_fdly

Lcd HV panel Serial YUV F line Delay

这个参数只有在 lcd_if=0 且 lcd_hv_if=12（Serial YUV）时才有效。

定义 CCIR656 编码时 F 相对有效行延迟的行数：

```
0：F toggle right after active video line
1：Delay 2 lines (CCIR PAL)
2：Delay 3 lines (CCIR NTSC)
```

5.1.10 lcd_cpu_if

Lcd CPU panel Interface

这个参数只有在 lcd_if=1 时才有效, 具体时序可参照RGB 和 I8080 管脚配置示意图中 CPU 那几列。

设置相应值的对应含义为：

```
0：18bit/1cycle (RGB666)
2: 16bit/3cycle (RGB666)
4：16bit/2cycle (RGB666)
6：16bit/2cycle (RGB666)
8：16bit/1cycle (RGB565)
10：9bit/1cycle (RGB666)
12：8bit/3cycle (RGB666)
14：8bit/2cycle (RGB565)
```

5.1.11 lcd_cpu_te

Lcd CPU panel tear effect

设置相应值的对应含义为，设置为 0 时，刷屏间隔时间为 lcd_ht × lcd_vt；设置为 1 或 2 时，刷屏间隔时间为两个 te 脉冲：

```
0：frame trigged automatically
1：frame trigged by te rising edge
2：frame trigged by te falling edge
```

5.1.12 lcd_lvds_if

Lcd LVDS panel Interface

设置相应值的对应含义为：

```
0：Single Link( 1 clock pair+3/4 data pair)
1：Dual Link(8 data lane，每4条lane接受一半像素，奇数像素或者偶数像素)
2: Dual Link (每4条lane接受全部像素，常用于物理双屏，且两个屏一样)
```

lcd_lvds_if 等于 2 的场景是，接两个一模一样的屏，然后两个屏显示同样的内容，此时 lcd 的其它 timing 只需要填写一个屏的 timing 即可。

5.1.13 lcd_lvds_colordepth

Lcd LVDS panel color depth

设置相应值对应含义为：

```
0：8bit per color(4 data pair)
1：6bit per color(3 data pair)
```

5.1.14 lcd_lvds_mode

Lcd LVDS Mode

这个参数只有在 lcd_lvds_bitwidth=0 时才有效。

设置相应值对应含义为 (见下图）：

```
0：NS mode
1：JEIDA mode
```

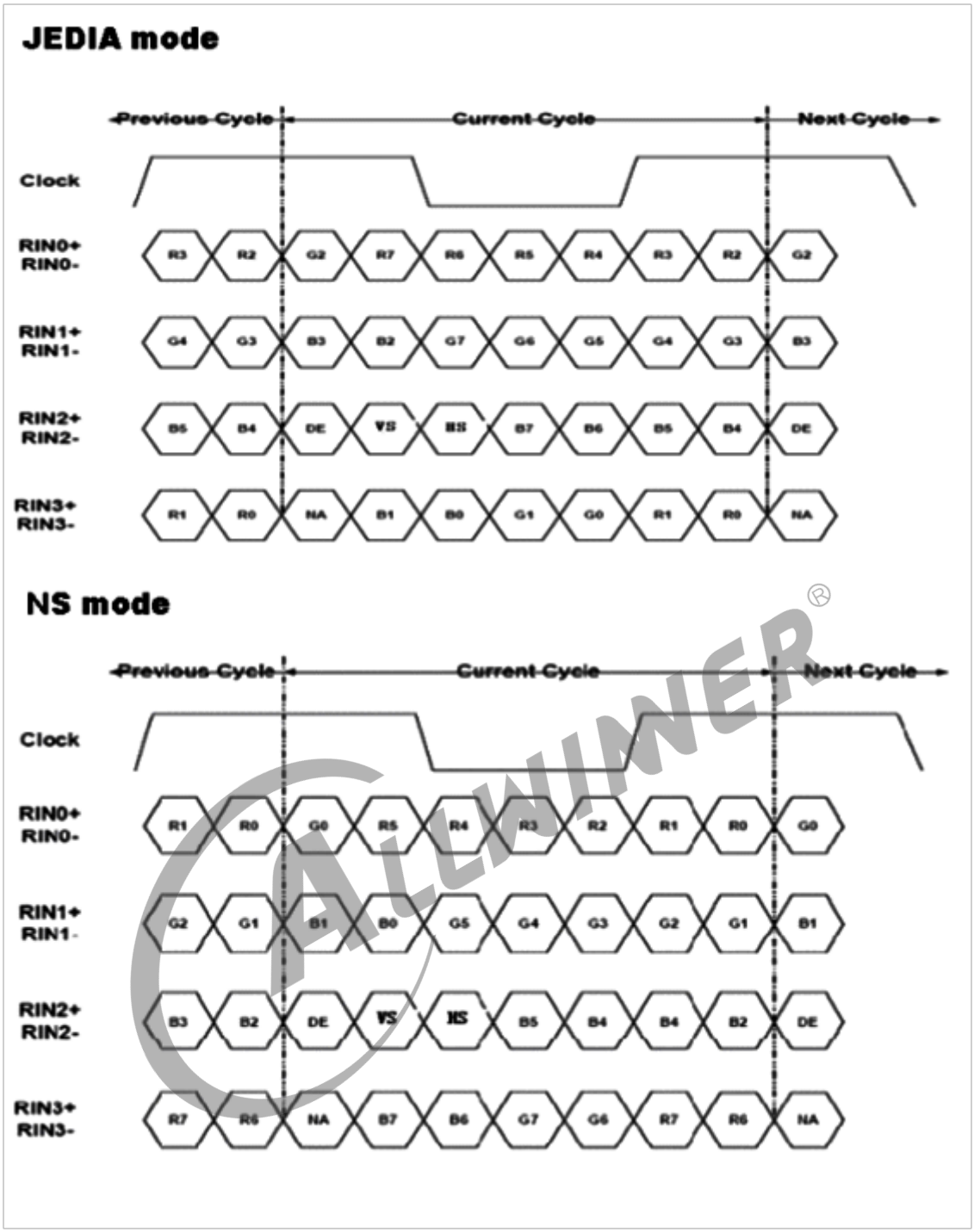

图 5-1: lvds mode

5.1.15 lcd_dsi_if

Lcd MIPI DSI panel Interface

这个参数只有在 lcd_if=4 时才有效。定义 MIPI DSI 屏的两种类型。

设置相应值的对应含义为：

```
0：Video mode
1：Command mode
2：video burst mode
```

注：Video mode 的 LCD 屏，是实时刷屏的，有 ht，hbp 等时序参数的定义；Command mode 的屏，屏上带有显示 Buffer，一般会有一个 TE 引脚。

5.1.16 lcd_dsi_lane

Lcd MIPI DSI panel Data Lane number

这个参数只有在 lcd_if=4 时才有效。

设置相应值的对应含义为：

```
1：1 data lane
2：2 data lane
3：3 data lane
4：4 data lane
```

5.1.17 lcd_dsi_format

Lcd MIPI DSI panel Data Pixel Format

这个参数只有在 lcd_if=4 时才有效。

设置相应值的对应含义为：

```
0：Package Pixel Stream, 24bit RGB
1：Loosely Package Pixel Stream, 18bit RGB
2：Package Pixel Stream, 18bit RGB
3：Package Pixel Stream, 16bit RGB
```

5.1.18 lcd_dsi_te

Lcd MIPI DSI panel Tear Effect

这个参数只有在 lcd_if=4 时才有效。

设置相应值的对应含义为：

```
0：frame trigged automatically
1：frame trigged by te rising edge
2：frame trigged by te falling edge
```

注：设置为 0 时，刷屏间隔时间为 lcd_ht × lcd_vt；设置为 1 或 2 时，刷屏间隔时间为两个 te 脉冲。

这个的作用就是屏一端发给 SoC 端的信号，用于同步信号，如果使能这个变量，那么 SoC 内部的显示中断将由这个外部脚来触发。

5.1.19 lcd_dsi_port_num

DSI 屏 port 数量

这个参数只有在 lcd_if=4 时才有效。

设置相应值的对应含义为：

```
0：一个port
1：两个port
```

这个选项的一个作用是，单 LCD 屏为 8 条 lane 的时候，如果只需要初始化其中一个 driver IC，则这个设置 1，如果两个 driver IC 都要初始化，这里设置成 0。并使用 lcd_source.c 定义的函数来进行初始化。

5.1.20 lcd_tcon_mode

Tcon 模式

这个参数只有在 lcd_if=4 时才有效。

设置相应值的对应含义为：

```
0：normal mode
1：tcon master mode（在第一次发送数据同步）
2:：tcon master mode（每一帧都同步）
3：tcon slave mode（依靠master mode来启动）
4：one tcon driver two dsi（8条lane）
```

5.1.21 lcd_slave_tcon_num

Slave Tcon 的序号

这个参数只有在 lcd_if=4 时而且 lcd_tcon_mode 等于 1 或者 2 才有效。用于告诉 master 模式下的 tcon，从 tcon 的序号是多少。

设置相应值的对应含义为：

```
0：tcon_lcd0
1：tcon_lcd1
```

5.1.22 lcd_tcon_en_odd_even_div

这个参数只有在 lcd_if=4 而且 lcd_tcon_mode=4 时才有效。

设置相应值的对应含义为：

```
0：tcon将一帧图像分左右两半来发送给两个DSI模块
1：tcon将一帧图像分奇偶像素来发给两个DSI模块
```

5.1.23 lcd_sync_pixel_num

这个参数只有在 lcd_if=4 而且 lcd_tcon_mode 等于 2 或者 3 时才有效。

设置同步从 tcon 的起始 pixel。

```
整数：不超过lcd_ht
```

5.1.24 lcd_sync_line_num

这个参数只有在 lcd_if=4 而且 lcd_tcon_mode 等于 2 或者 3 时才有效。

设置同步从 tcon 的起始行。

```
整数：不超过lcd_vt
```

5.1.25 lcd_cpu_mode

Lcd CPU 模式，控制。

设置相应值的对应含义为，设置为 0 时，刷屏间隔时间为 lcd_ht × lcd_vt；设置为 1 或 2 时，刷屏间隔时间为两个 te 脉冲：

```
0：中断自动根据时序，由场消隐信号内部触发。
1：中断根据数据Block的counter触发或者由外部te触发。
```

5.1.26 lcd_fsync_en

LCD 使能 fsync 功能，用于触发 sensor 出图, 目的是同步，部分 IC 支持。

```
0：disable
1：enable
```

5.1.27 lcd_fsync_act_time

LCD 的 fsync 功能，其中的有效电平时间长度，单位：像素时钟的个数。

0~lcd_ht-1

5.1.28 lcd_fsync_dis_time

LCD 的 fsync 功能，其中的无效电平时间长度，单位：像素时钟的个数。

0~lcd_ht-1

5.1.29 lcd_fsync_pol

LCD 的 fsync 功能的有效电平的极性。

0：有效电平为低
1：有效电平为高

5.1.30 lcd_start_delay

出现 LCD 显示前几行有闪条，或者 colorbar 1 ~ 7 可以显示，colorbar 8 显示不了的问题，可以调试一下这个参数试试。

整数：调试范围可以先从0 ~ 10调试，不可以再往上增加

5.2 屏时序参数说明

下面几个参数对于调屏非常关键，决定了发送端（SoC）发送数据时序。由于涉及到发送端和接收端的调试，除了分辨率和尺寸之外，其它几个数值都不是绝对不变的，两款一样分辨率，同种接口的屏，它们的数值也有可能不一样。

获取途径如下：

1. 询问 LCD 屏厂。
2. 从屏手册或者 Driver IC 手册中查找（向屏厂索要这些文档），如下图所示。

3. Mechanical Specification

Parameter		Specifications	Unit
Outline dimensions (typ)		125.55 (W) × 170.95 (H) × 1.95 (D) *1	mm
Main LCD Panel	Active area	119.808 (W) × 159.744(H)	mm
	Display format	1536(W) × RGB × 2048(H)	-
	Dot pitch	0.026 (W) × 0.078 (H)	mm
	Base color	Normally Black	-
	Illumination mode	Transmissive	
Mass		65(TYP.)	g

*1 The above-mentioned table indicates module sizes without some projections and FPC.

图 5-2: lcd_info1

Ta=25 °C

Item	Symbol	Min.	Typ.	Max.	Unit
Horizontal Frequency		-	124.32	-	kHz
Pixel Clock Frequency		-	118	-	MHz
Horizontal Total	THT	878	948	-	CK
Horizontal Synchronization	THS	1	24	-	CK
Horizontal Back Porch	THB	55	90	-	CK
Horizontal Address	THA	768	768	768	CK
Horizontal Front Porch	THF	55	90	-	CK
MIPI Port 1 & 2 Skew	SKEW	-THB	0	THF	A to B
Vertical Frequency		57	60	63	Hz
Vertical Total	TVT	2068	2072	-	THT
Vertical Synchronization	TVS	1	2	-	THT
Vertical Back Porch	TVB	8	10	-	THT
Vertical Address	TVA	2048	2048	2048	THT
Vertical Front Porch	TVF	12	14	-	THT
Mipi Clock Frequency		764	804	844	Mbps

IOVCC=1.8V.VSP=5.6V.VSN=-5.6V.GND=0V

图 5-3: lcd_info2

3. 在前面两步都搞不定的情况下，可以根据 vesa 标准来设置，主要是 DMT 和 CVT 标准。

其中 DMT，指的是《VESA and Industry Standards and Guidelines for Computer Display Monitor Timing(DMT)》，下载该标准，里面就有各种常用分辨率的 timing。

其中的 CVT，指的是《VESA Coordinated Video Timings(CVT) Standard》，该标准提供一种通用公式用于计算出指定分辨率，刷新率等参数的 timing。

可以下载这个 excel 表来计算VESA Coordinated Video Timing Generator。

由下面两条公式得知，我们不需要设置lcd_hfp和lcd_vfp参数，因为驱动会自动根据其它几个已知参数中算出lcd_hfp和lcd_vfp。

```
lcd_ht = lcd_x + lcd_hspw + lcd_hbp + lcd_hfp

lcd_vt = lcd_y + lcd_vspw + lcd_vbp + lcd_vfp
```

5.2.1 lcd_x

显示屏的水平像素数量，也就是屏分辨率中的宽。

5.2.2 lcd_y

显示屏的垂直行数，也就是屏分辨率中的高。

5.2.3 lcd_ht

Horizontal Total time

指一行总的 dclk 的 cycle 个数。见下图：

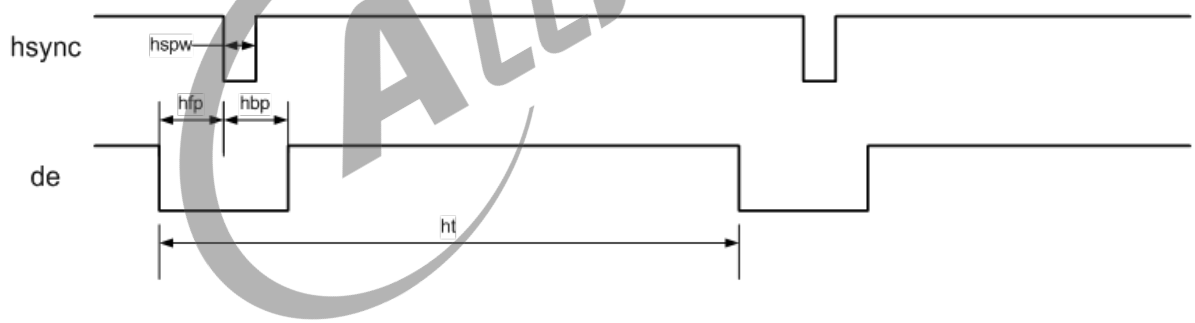

图 5-4: lcdht

5.2.4 lcd_hbp

Horizontal Back Porch

指有效行间，行同步信号（hsync）开始，到有效数据开始之间的 dclk 的 cycle 个数，包括同步信号区。见上图，注意的是包含了 hspw 段。

说明

是包含了 hspw 段，也就是
lcd_hbp= 实际的 hbp+ 实际的 hspw

5.2.5 lcd_hspw

Horizontal Sync Pulse Width

指行同步信号的宽度。单位为 1 个 dclk 的时间（即是 1 个 data cycle 的时间）。见上图。

5.2.6 lcd_vt

Vertical Total time

指一场的总行数。见下图：

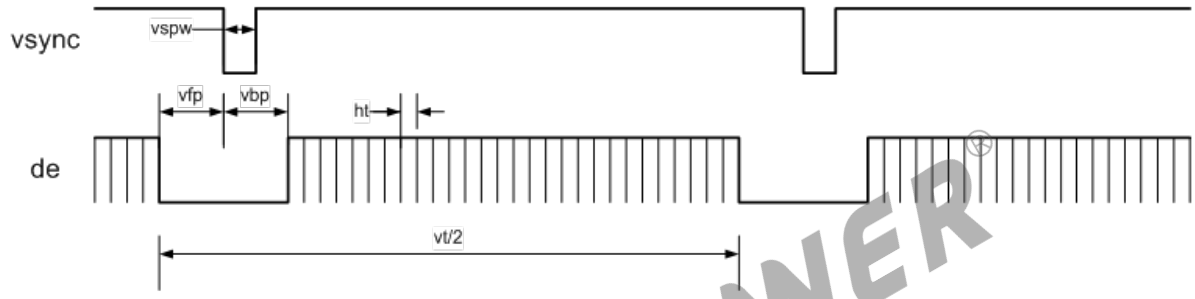

图 5-5: lcdvt

5.2.7 lcd_vbp

Vertical Back Porch

指场同步信号（vsync）开始，到有效数据行开始之间的行数，包括场同步信号区。

说明

是包含了 vspw 段，也就是
lcd_vbp= 实际的 vbp+ 实际的 vspw

5.2.8 lcd_vspw

Vertical Sync Pulse Width

指场同步信号的宽度。单位为行。见上图。

5.2.9 lcd_dclk_freq

Dot Clock Frequency

传输像素传送频率。单位为 MHz。

fps = (lcd_dclk_freq×1000×1000) / (ht×vt)。

这个值根据以下公式计算：

```
lcd_dclk_freq=lcd_ht*lcd_vt*fps
```

注意：

1. 后面的三个参数都是从屏手册中获得，fps 一般是 60。
2. 如果是串行接口，发完一个像素需要 2 到 3 个周期的，那么。

```
lcd_dclk_freq * cycles = lcd_ht*lcd_vt*fps
```

或者

```
lcd_dclk_freq = lcd_ht*cycles*lcd_vt*fps
```

5.2.10 lcd_width

Width of lcd panel in mm

此参数描述 lcd 屏幕的物理宽度，单位是 mm。用于计算 dpi。

5.2.11 lcd_height

height of lcd panel in mm

此参数描述 lcd 屏幕的物理高度，单位是 mm。用于计算 dpi。

5.3 背光相关参数

目前用得比较广泛的就是 pwm 背光调节，原理是利用 pwm 脉冲开关产生的高频率闪烁效应，通过调节占空比，达到欺骗人眼，调节亮暗的目的。

5.3.1 lcd_pwm_used

是否使用 pwm。

此参数标识是否使用 pwm 用以背光亮度的控制。

5.3.2 lcd_pwm_name

使用 pwm 通道的名称。

此参数用于指定已经申请好的 pwm 通道名称（需要在 disp 节点下填写 pwms 和 pwm-names）。如果该项未指定，驱动会申请 lcd_pwm_ch 对应的通道。

5.3.3 lcd_pwm_ch

Pwm channel used

此参数标识使用的 Pwm 通道，这里是指使用 SoC 哪个 pwm 通道，通过查看原理图连接可知。

5.3.4 lcd_pwm_freq

Lcd backlight PWM Frequency

这个参数配置 PWM 信号的频率，单位为 Hz。

说明

频率不宜过低否则很容易就会看到闪烁，频率不宜过快否则背光调节效果差。部分屏手册会标明所允许的 pwm 频率范围，请遵循屏手册固定范围进行设置。
在低亮度的时候容易看到闪烁，是正常现象，目前已知用上 pwm 的背光都是如此。

5.3.5 lcd_pwm_pol

Lcd backlight PWM Polarity

这个参数配置 PWM 信号的占空比的极性。设置相应值对应含义为：

```
0：active high
1：active low
```

5.3.6 lcd_pwm_max_limit

Lcd backlight PWM 最高限制，以亮度值表示

比如 150，则表示背光最高只能调到 150，0-255 范围内的亮度值将会被线性映射到 0-150 范围内。用于控制最高背光亮度，节省功耗。

5.3.7 lcd_bl_en

背光使能脚，非必须，看原理图是否有，用于使能或者禁止背光电路的电压。

示例：lcd_bl_en = port:PD24<1><2><default><1>

含义：PD24 输出高电平时打开 LCD 背光；下拉，默认高电平

- 第一个尖括号：功能分配；1 为输出；
- 第二个尖括号：内置电阻；使用 0 的话，标示内部电阻高阻态，如果是 1 则是内部电阻上拉，2 就代表内部电阻下拉。使用 default 的话代表默认状态，即电阻上拉。其它数据无效。
- 第三个尖括号：驱动能力；default 表驱动能力是等级 1
- 第四个尖括号：电平；0 为低电平，1 为高电平。

需要在屏驱动调用相应的接口进行开、关的控制。

说明

一般来说，高电平是使能，在这个前提下，建议将内阻电阻设置成下拉，防止硬件原因造成的上拉，导致背光提前亮。默认电平请填写高电平，这是 uboot 显示过度到内核显示，平滑无闪烁的需要。

5.3.8 lcd_bl_n_percent

背光映射值，n 为 (0-100)

此功能是针对亮度非线性的 LCD 屏的，按照配置的亮度曲线方式来调整亮度变化，以使亮度变化更线性。

比如 lcd_bl_50_percent = 60，表明将 50% 的亮度值调整成 60%，即亮度比原来提高 10%。

说明

修改此属性不当可能导致背光调节效果差。

5.3.9 lcd_backlight

背光默认值，0-255。

此属性决定在 uboot 显示 logo 阶段的亮度，进入都内核时则是读取保存的配置来决定亮度。

说明

显示 logo 阶段，一般来说需要比较亮的亮度，业内做法都是如此。

5.4 显示效果相关参数

5.4.1 lcd_frm

Lcd Frame Rate Modulator

FRM 是解决由于 PIN 减少导致的色深问题。

这个参数设置相应值对应含义为：

```
0：RGB888 -- RGB888 direct
1：RGB888 -- RGB666 dither
2：RGB888 -- RGB565 dither
```

有些 LCD 屏的像素格式是 18bit 色深（RGB666）或 16bit 色深（RGB565），建议打开 FRM 功能，通过 dither 的方式弥补色深，使显示达到 24bit 色深（RGB888）的效果。如下图所示，上图是色深为 RGB66 的 LCD 屏显示，下图是打开 dither 后的显示，打开 dither 后色彩渐变的地方过度平滑。

图 5-6: lcd_frm 打开

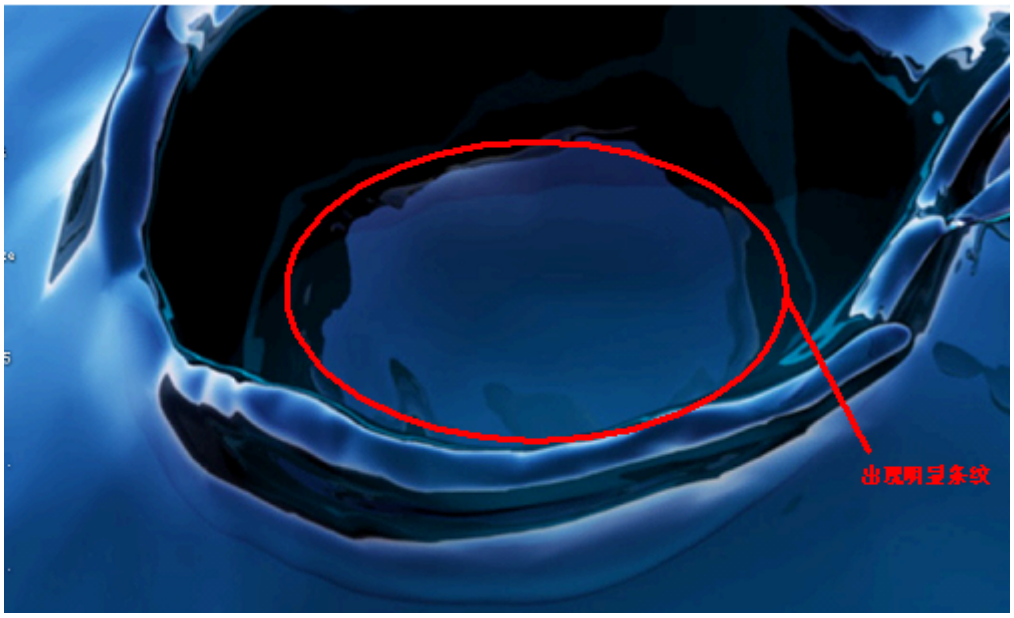

图 5-7: lcd_frm 关闭

5.4.2 lcd_gamma_en

Lcd Gamma Correction Enable

设置相应值的对应含义为：

```
0：Lcd的Gamma校正功能关闭
1：Lcd的Gamma校正功能开启
```

设置为 1 时，需要在屏驱动中对 lcd_gamma_tbl[256] 进行赋值。

5.4.3 lcd_cmap_en

Lcd Color Map Enable

设置相应值的对应含义为：

```
0：Lcd的色彩映射功能关闭
1：Lcd的色彩映射功能开启
```

设置为 1 时，需要对 lcd_cmap_tbl [2][3][4] 进行赋值 Lcd Color Map Table。

每个像素有 R、G、B 三个单元，每四个像素组成一个选择项，总共有 12 个可选。数组第一维表示奇偶行，第二维表示像素的 RGB，第三维表示第几个像素，数组的内容即表示该位置映射到的内容。

LCD CMAP 是对像素的映射输出功能，只有像素有特殊排布的 LCD 屏才需要配置。

LCD CMAP 定义每行的 4 个像素为一个总单元，每个像素分 R、G、B 3 个小单元，总共有 12 个小单元。通过 lcd_cmap_tbl 定义映射关系，输出的每个小单元可随意映射到 12 个小单元之一。

```
__u32 lcd_cmap_tbl[2][3][4] = {
  {
    {LCD_CMAP_G0,LCD_CMAP_B1,LCD_CMAP_G2,LCD_CMAP_B3},
    {LCD_CMAP_B0,LCD_CMAP_R1,LCD_CMAP_B2,LCD_CMAP_R3},
    {LCD_CMAP_R0,LCD_CMAP_G1,LCD_CMAP_R2,LCD_CMAP_G3},
  },
  {
    {LCD_CMAP_B3,LCD_CMAP_G2,LCD_CMAP_B1,LCD_CMAP_G0},
    {LCD_CMAP_R3,LCD_CMAP_B2,LCD_CMAP_R1,LCD_CMAP_B0},
    {LCD_CMAP_G3,LCD_CMAP_R2,LCD_CMAP_G1,LCD_CMAP_R0},
  },
};
```

如上，上三行代表奇数行的像素排布，下三行代表偶数行的像素排布。

每四个像素为一个单元，第一列代表每四个像素的第一个像素映射，第二列代表每四个像素的第二个像素映射，以此类推。

如上的定义，像素的输出格式如下图所示。

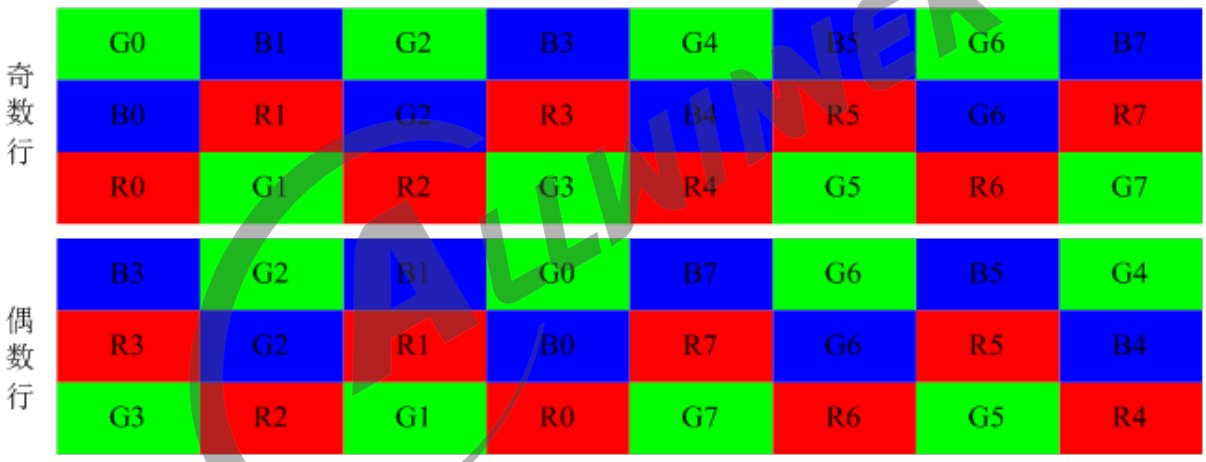

图 5-8: cmap

5.4.4 lcd_rb_swap

调换 tcon 模块 RGB 中的 R 分量和 B 分量。

```
0：不变
1：调换R分量和B分量
```

5.5 电源和管脚参数

5.5.1 概述

如果需要使用某路电源必须现在[disp]节点中定义，然后[lcd]部分使用的字符串则要和 disp 中定义的一致。比如下面的例子：

```
disp: disp@01000000 {
        disp_init_enable      = <1>;
        disp_mode             = <0>;

         /* VCC-LCD */
        dc1sw-supply = <&reg_sw>;
        /* VCC-LVDS and VCC-HDMI */
        bldo1-supply = <&reg_bldo1>;
        /* VCC-TV */
       cldo4-supply = <&reg_cldo4>;
};
```

其中-supply是固定的，在-supply之前的字符串则是随意的，不过建议取有意义的名字。在=后面的像<&reg_sw>则必须在 board.dtsi 的regulator0节点中找到。

然后 lcd0 节点中，如果要使用 reg_sw，则像下面这样写就行，dc1sw 对应 dc1sw-supply。

```
lcd_power=”dc1sw”
```

由于 u-boot 中也有 axp 驱动和 display 驱动，和内核，它们都是读取同份配置，为了能互相兼容，取名的时候，有以下限制：

在 u-boot 2018 中，axp 驱动只认类似 bldo1 这样从 axp 芯片中定义的名字，所以命名 xxx-supply 的时候最好按照这个 axp 芯片的定义来命名。

5.5.2 lcd_power

见上面概述的注意事项。

```
示例：lcd_power = “vcc-lcd”
```

配置 regulator 的名字。配置好之后，需要在屏驱动调用相应的接口进行开、关的控制。

注意：如果有多个电源需要打开，则定义 lcd_power1，lcd_power2 等。

5.5.3 lcd_pin_power

用法 lcd_power一致，区别是用户设置之后，不需要在屏驱动中去操作，而是驱动框架自行在屏驱动之前使能，在屏驱动之后禁止。

示例：lcd_pin_power = “vcc-pd”

注意：如果需要多组，则添加 lcd_pin_power1，lcd_pin_power2 等。除了 lcddx 之外，这里的电源还有可能是 pwm 所对应管脚的电源。

5.5.4 lcd_gpio_0

示例：lcd_gpio_0 = port:PD25<0><0><default><0>

含义：lcd_gpio_0 引脚为 PD25。

- 第一个尖括号：功能分配；0 为输入，1 为输出。
- 第二个尖括号：内置电阻；使用 0 的话，标示内部电阻高阻态，如果是 1 则是内部电阻上拉，2 就代表内部电阻下拉。使用 default 的话代表默认状态，即电阻上拉。其它数据无效。
- 第三个尖括号：驱动能力；default 表驱动能力是等级 1。
- 第四个尖括号：表示默认值；即是当设置为输出时，该引脚输出的电平，0 为低电平，1 为高电平。

需要在屏驱动调用相应的接口进行拉高，拉低的控制。请看管脚控制函数说明

注意：如果有多个 gpio 脚需要控制，则定义 lcd_gpio_0，lcd_gpio_1 等。

5.5.5 lcddx

示例：lcdd0 = port:PD00<3><0><default><default>

含义：lcdd0 这个引脚，即是 PD0，配置为 LVDS 输出。

- 第一个尖括号：功能分配；0 为输入，1 为输出，2 为 LCD 输出，3 为 LVDS 接口输出，7 为 disable。
- 第二个尖括号：内置电阻；使用 0 的话，标示内部电阻高阻态，如果是 1 则是内部电阻上拉，2 就代表内部电阻下拉。使用 default 的话代表默认状态，即电阻上拉。其它数据无效。
- 第三个尖括号：驱动能力；default 表驱动能力是等级 1。
- 第四个尖括号：表示默认值；即是当设置为输出时，该引脚输出的电平，0 为低电平，1 为高电平。

LCD PIN 的配置如下：

LCD 为 HV RGB 屏，CPU/I80 屏时，必须定义相应的 IO 口为 LCD 输出（如果是 0 路输出，第一个尖括号为 2；如果是 1 路输出，第一个尖括号为 3）。

具体的 IO 对应关系可参考 user manual 手册进行配置。

LCD PIN 的所有 IO，均可通过注释方式去掉其定义，显示驱动对注释 IO 不进行初始化操作。

需要在屏驱动调用相应的接口进行开、关的控制。

注意：不是一定要叫做 lcdd0 这个名字，你改成其它名字对驱动不会造成任何影响，这里只是为了方便记忆。

5.5.6 pinctrl-0 和 pinctrl-1

在配置lcd0节点时，当碰到需要配置管脚复用时，你只要把pinctrl-0和pinctrl-1赋值好就行，可以用提前定义好的，也可以用自己定义的，提前定义的管脚一般可以在内核目录下arch/arm/boot/dts或者arch/arm64/boot/dts下找：**平台-pinctrl.dtsi** 中找到定义。

例子:

```
pinctrl-0 = <&rgb24_pins_a>;
pinctrl-1 = <&rgb24_pins_b>;//休眠时候的定义，io_disable
```

表 5-1: 提前定义好的管脚名称

管脚名称	描述
rgb24_pins_a 和 rgb24_pins_b	RGB 屏接口，而且数据位宽是 24，RGB888
rgb18_pins_a 和 rgb18_pins_b	RGB 屏接口，而且数据位宽是 16，RGB666
lvds0_pins_a 和 lvds0_pins_b	Single link LVDS 接口 0 管脚定义（主显 lcd0）
lvds1_pins_a 和 lvds1_pins_b	Single link LVDS 接口 1 管脚定义（主显 lcd0）
lvds2link_pins_a 和 lvds2link_pins_b	Dual link LVDS 接口管脚定义（主显 lcd0）
lvds2_pins_a 和 lvds2_pins_b	Single link LVDS 接口 0 管脚定义（主显 lcd1）
lvds3_pins_a 和 lvds3_pins_b	Single link LVDS 接口 1 管脚定义（主显 lcd1）
lcd1_lvds2link_pins_a 和 lcd1_lvds2link_pins_b	Dual link LVDS 接口管脚定义（主显 lcd1）
dsi4lane_pins_a 和 dsi4lane_pins_b	4 lane DSI 屏接口管脚定义

自定义一组脚

写在 board.dtsi 中，只要名字不要和现有名字重复就行，首先判断自己需要用的管脚，属于大 cpu 域还是小 cpu 域，以此判断需要将管脚定义放在pio（大 cpu 域）下面还是r_pio（小 cpu 域）下面。

例子：

```
&pio {

  I8080_8bit_pins_a: I8080_8bit@0 {
    allwinner,pins = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,pname = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
    allwinner,function = "I8080_8bit";
    allwinner,muxsel = <2>;
    allwinner,drive = <3>;
    allwinner,pull = <0>;
```

```
    };
    I8080_8bit_pins_b: I8080_8bit@1 {
        allwinner,pins = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
        allwinner,pname = "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7", "PD8", "PD18", "PD19", "PD20", "PD21";
        allwinner,function = "I8080_8bit_suspend";
        allwinner,muxsel = <7>;
        allwinner,drive = <3>;
        allwinner,pull = <0>;
    };

};
```

- pins，具体管脚。
- pname，管脚名称，随便取。
- function，管脚功能名称，随便取。
- muxsel，管脚功能选择。根据 port spec 来选择对应功能。
- drive，驱动能力，数值越大驱动能力越大。
- pull，上下拉，使用 0 的话，标示内部电阻高阻态，如果是 1 则是内部电阻上拉，2 就代表内部电阻下拉。使用 default 的话代表默认状态，即电阻上拉。其它数据无效。

为了规范，我们将在所有平台保持一致的名字，其中后缀为 a 为管脚使能，b 的为 io_disable 用于设备关闭时。

有时候，你需要用两组不同功能的管脚，可以像下面这样定义即可。

```
pinctrl-0 = <&rgb24_pins_a>, <&xxx_pins_a>;
pinctrl-1 = <&rgb24_pins_b>, <&xxx_pins_b>;//休眠时候的定义，io_disable
```

5.6 ESD 静电检测自动恢复功能

这个功能在 linux4.9 以及 linux 3.10 sunxi-product 分支上实现了，如果需要这个功能，需要完成以下步骤。

首先打开如下内核配置：

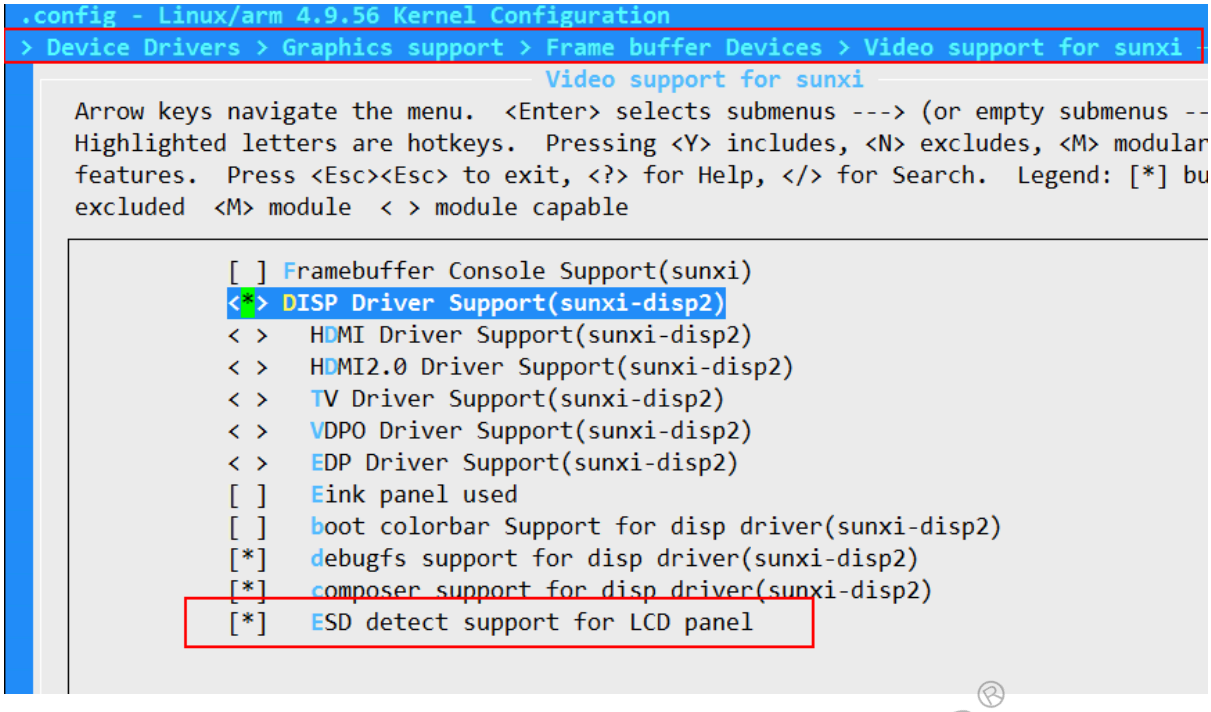

图 5-9: ESD 内核配置

修改屏驱动，实现三个回调函数：

如下示例，在屏 he0801a068 上添加 esd 相关的回调函数。

（linux-4.9/drivers/video/fbdev/sunxi/disp2/disp/lcd/he0801a068.c）。

```
struct __lcd_panel he0801a068_panel = {
        /* panel driver name, must mach the name of
         * lcd_drv_name in sys_config.fex
         */
        .name = "he0801a068",
        .func = {
                .cfg_panel_info = lcd_cfg_panel_info,
                        .cfg_open_flow = lcd_open_flow,
                        .cfg_close_flow = lcd_close_flow,
                        .lcd_user_defined_func = lcd_user_defined_fun
                        .esd_check = lcd_esd_check,
                        .reset_panel = lcd_reset_panel,
                        .set_esd_info = lcd_set_esd_info,
        },
};
```

图 5-10: ESD 屏驱动添加函数

esd_check 函数原型：

S32 esd_check(u32 sel)

作用：是给上层反馈当前屏的状态。

返回值：如果屏正常的话就返回 0，不正常的话就返回非 0。

sel：显示索引。

由于屏的类型接口众多，不同屏检测屏的状态各异，一般来说是通过驱动接口读取屏的内部信息（id 或者其它寄存器），如果获取正常则认为屏是正常的，获取失败则认为屏是异常的。比如下面 dsi 屏的做法：

```
static s32 lcd_esd_check(u32 sel)
{
    u8 result[32] = {0};
    u32 len = 0;
    s32 ret = 0;
    dsi_read_mode_en(sel, 1);
    dsi_dcs_rd(sel, 0x0a, result, &len);
    dsi_read_mode_en(sel, 0);
    if(result[0] == 0x9c)
        ret = 0;
    else
        ret = -1;
    return ret;
}
```

图 5-11: ESD 静电检测逻辑代码

此外，一般情况下，也会通过 dsi 接口读取 0x0A 命令（获取 power 模式）来判断屏是否正常。

sunxi_lcd_dsi_dcs_read(sel, 0x0A, result, &num)

5.1.4 Read Display Power Mode (0Ah)

0AH	RDDPM (Read Display Power Mode)											
	DCX	RDX	WRX	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	0	0	0	0	1	0	1	0	0A
1st Parameter	1	↑	1	D7	D6	D5	D4	D3	D2	0	0	xx

This command indicates the current status of the display as described in the table below:

Bit	Description	Comment
D7	Booster Voltage Status	–
D6	Idle Mode On/Off	–
D5	Not Defined	Set to "0"
D4	Sleep In/Out	–
D3	Display Normal Mode On/Off	–
D2	Display On/off	–
D1	Not Defined	Set to "0"
D0	Not Defined	Set to "0"

图 5-12: ESD MIPI 状态寄存器

reset_panel 函数原型：

```
s32 reset_panel(u32 sel)
```

作用：当屏幕异常的时候所需要的复位操作。

返回值：复位成功就是 0，复位失败非 0。

sel：显示索引。

每个屏的初始化都不同，顺序步骤都不一样，总的来说就是执行部分或者完整的屏驱动里面的 close_flow 和 open_flow 所定义的回调函数。根据实际情况灵活编写这个函数。

值得注意的是：某些 dsi 屏中，需要至少执行过一次 sunxi_lcd_dsi_clk_disable（dsi 时钟禁止）和 sunxi_lcd_dsi_clk_enable（dsi 时钟使能），否则可能导致 dsi 的读函数异常。

下图是复位函数示例：

```
static s32 lcd_reset_panel(u32 sel)
{
        sunxi_lcd_dsi_dcs_write_0para(sel, 0x28);
        sunxi_lcd_delay_ms(50);
        sunxi_lcd_dsi_dcs_write_0para(sel, 0x10);
        sunxi_lcd_delay_ms(200);

        sunxi_lcd_power_disable(sel, 1);
        sunxi_lcd_delay_ms(100);
        sunxi_lcd_power_enable(sel, 1);
        sunxi_lcd_delay_ms(260);

        lcd_panel_init(sel);
        sunxi_lcd_delay_ms(20);

        return 0;
}
```

图 5-13: ESD 复位函数

set_esd_info 函数原型：

```
s32 set_esd_info(struct disp_lcd_esd_info *p_info)
```

作用：控制 esd 检测的具体行为。比如间隔多长时间检测一次，复位的级别，以及检测函数被调用的位置。

返回值：成功设置返回 0，否则非 0。

p_info：需要设置的 esd 行为结构体。

示例：如下面图所示，每隔 60 次显示中断检测一次（调用 esd_check 函数，如果显示帧率是 60fps 的话，那么就是 1 秒一次），然后将在显示中断处理函数里面执行检测函数，由 esd_check_func_pos 成员决定调用 esd_check 函数的位置，如果是 0 则在中断之外执行检测函数，之所以有这个选项是因为显示中断资源（中断处理时间）是非常珍贵的资源，关系到显示帧率的问题。

下图中的 level 为 2 表示复位全志 SoC 的 DISP 相关模块，level 为 1 表示复位全志 SoC 的 LCD 相关模块以及 reset_panel 里面的操作，level 为 0 的时候表示仅仅执行 reset_panel 里面的操作。

```
static int lcd_set_esd_info(struct disp_lcd_esd_info *p_info)
{
        if (!p_info)
                return -1;
        p_info->level = 1;
        p_info->freq = 60;
        p_info->esd_check_func_pos = 1;
        return 0;
}
```

图 5-14: ESD 设置信息函数

可以通过 cat /sys/class/disp/disp/attr/sys 获取当前的 esd info。

```
screen 0:
de_rate 594000000 hz, ref_fps:60
mgr0: 2560x1600 fmt[rgb] cs[0x204] range[full] eotf[0x4] bits[8bits] unblank err[0] force_sync[0]
dmabuf: cache[0] cache max[0] umap skip[0] overflow[0]
capture: dis req[0] runing[0] done[0,0]
   lcd output(enable)  backlight( 50)  fps:60.9   esd level(1)   freq(300)   pos(1) reset(244)  2560x1600
   err:0   skip:0  skip T.O:50 irq:73424   vsync:0 vsync_skip:0
  BUF  en ch[1] lyr[0] z[0] prem[N] fbd[N] a[globl 255] fmt[  0] fb[2560,1600;2560,1600;2560,1600] crop[   0,   0,2560,1600]
       frame[   0,   0,2560,1600] addr[98100000,00000000,00000000] right[00000000,00000000,00000000] flags[0x00] trd
       [0,0] depth[ 0]
acquire: 0, 25.5 fps
release: 0, 25.5 fps
display: 0, 25.5 fps
```

esd level(1) freq(300) pos(1) reset(244)

esd levele 和 freq 和 pos 的意思请看上面 set_esd_info 函数原型的解释。

Reset 后面的数字表示屏复位的次数（也就是 esd 导致屏挂掉之后，并且成功检测到并复位的次数）。

此功能可能遇到的问题。

1. 打静电挂了，但是读出来的值仍然是正确的，此问题无解。
2. Dsi 读操作卡住了，卡在中断里面了。此问题可能和 DSI 的 lp 模式下的速率有关系，而 lp 的速率又和 dclk 的频率有关系。此时可以尝试修改 de_dsi_28.c() 文件中的 dsi_basic_cfg 函数，如

下图所示红框所示的寄存器值，这个寄存器是 Lp 模式时钟分频值，一般来说这个值越小，lp 速率越快，尝试改小看是否还会卡住。

3. 读寄存器返回的长度和屏手册介绍的不一致，需要调用 sunxi_lcd_dsi_set_max_ret_size 发送 lp 命令，告知屏幕返回参数的最大长度。

```
static __s32 dsi_basic_cfg(__u32 sel, struct disp_panel_para *panel)
{
        __u32 mode, lane, format, dclk;
        u32 timing_div = 1;

        if (panel->lcd_if == LCD_IF_DSI &&
            panel->lcd_tcon_mode == DISP_TCON_DUAL_DSI)
                timing_div = 2;

        mode = panel->lcd_dsi_if;
        lane = panel->lcd_dsi_lane;
        format = panel->lcd_dsi_format;
        dclk = panel->lcd_dclk_freq / timing_div;

        dsi_dev[sel]->dsi_ctl0.bits.module_en = 1;
        if (LCD_DSI_IF_VIDEO_MODE == mode || LCD_DSI_IF_BURST_MODE == mode) {
                dsi_dev[sel]->dsi_ctl2.dwval =
                    ((15 * dsi_bits_per_pixel[format] + 4 * lane * 10) /
                     (8 * lane * 10)) &
                    0xff;
                dsi_delay_ms(100);
                dsi_dev[sel]->dsi_to_ctl0.dwval =
                    ((dclk * dsi_bits_per_pixel[format] + 4 * lane * 10) /
                     (8 * lane * 10)) &
                    0xff;
        } else {
```

图 5-15: Lp 模式时钟分频值

6 调试方法

系统起来之后可以读取 sysfs 一些信息，来协助调试。

6.1 加快调试速度的方法

很明显，如果你在安卓上调试 LCD 屏会比较不方便，安卓编译时间和安卓固件都太过巨大，每次修改内核后，可能都要经过 10 几分钟都才能验证，这样效率就太低下了，可用采用如下方法：

1. 使用 linux 固件而不是安卓固件。SDK 是支持仅仅编译 linux 固件，一般是配置 lichee 或者 longan 的时候选择 linux，打包的时候，用 lichee 或者 longan 根目录下的 build.sh 来打包就行。因为 linux 内核小得多，编译更快，更方便调试。
2. 使用内核来调试 LCD 屏。我们知道 Uboot 和内核都需要添加 LCD 驱动，这样才能快速显示 logo，但是 uboot 并不方便调试，所以有时候我们需要把 uboot 的显示驱动关掉，专心调试内核的 LCD 驱动，调好之后才移植到 uboot，另外这样做的一个优点是，我可以非常方便的修改 lcd timing 而不需要重烧固件。就是利用 uboot 命令的 fdt 命令修改 device tree。

比如说：

```
fdt set lcd0 lcd_hbp <40>
```

更多命令见 fdt help。

如何关闭 uboot 显示呢，一般是在 uboot 源码路径下 inlcude/configs/平台.h 中，注释掉 CONFIG_SUNXI_MODULE_DISPLAY 即可，如果是 uboot 2018 则是注释掉 configs/平台 _defconfig 中 CONFIG_DISP2_SUNXI。

6.2 查看显示信息

以下信息是所有信息中最重要的。

```
cat /sys/class/disp/disp/attr/sys

screen 0:
de_rate 297000000 hz, ref_fps:60
mgr0: 1280x800 fmt[rgb] cs[0x204] range[full] eotf[0x4] bits[8bits] err[0] force_sync[0] unblank direct_show[false]
  lcd output  backlight( 50)  fps:60.9   1280x 800
  err:0   skip:31 irq:1942   vsync:0 vsync_skip:0
```

```
BUF   enable ch[1] lyr[0] z[0] prem[N] a[globl 255] fmt[  8] fb[1280, 800;1280, 800;1280, 800] crop[   0,   0,1280, 800]
      frame[   0,   0,1280, 800] addr[      0,      0,      0] flags[0x      0] trd[0,0]
```

lcd output

表示当前显示接口是 LCD 输出。

1280x800

表示当前 LCD 的分辨率，与 board.dts 中 lcd0 的设置一样。

ref_fps:60

是根据你在board.dts的lcd0填的时序算出来的理论值。

fps:60.9

后面的数值是实时统计的，正常来说应该是在 60(期望的 fps) 附近，如果差太多则不正常，重新检查屏时序，和在屏驱动的初始化序列是否有被调用到。

irq:1942

这是 vsync 中断的次数，每加 1 都代表刷新了一帧，正常来说是一秒 60（期望的 fps）次，重复 cat sys，如果无变化，则异常。

BUF

开头的表示图层信息，一行 BUF 表示一个图层，如果一个 BUF 都没有出现，那么将是黑屏，不过和屏驱动本身关系就不大了，应该查看应用层 & 框架层。

err:0

这个表示缺数，如果数字很大且一直变化，屏幕会花甚至全黑，全红等。

skip:31

这个表示跳帧的数量，如果这个数值很大且一直变化，有可能卡顿，如果数字与 irq 后面的数字一样，说明每一帧都跳，会黑屏（有背光）。

6.3 查看电源信息

查看 axp 某一路电源是否有 enable 可以通过下面命令查看。当然这个只是软件的，实际还是用万用表量为准。

```
cat /sys/class/regulator/dump


pmu1736_ldoio2 : disabled  0  700000   supply_name:
pmu1736_ldoio1 : disabled  0  700000   supply_name:
pmu1736_dc1sw : enabled  1  3300000   supply_name:  vcc-lcd
```

```
pmu1736_cpus : enabled  0  900000   supply_name:
pmu1736_cldo4 : disabled  0  700000   supply_name:
pmu1736_cldo3 : disabled  0  700000   supply_name:
pmu1736_cldo2 : enabled  1  3300000   supply_name:  vcc-pf
pmu1736_cldo1 : disabled  0  700000   supply_name:
pmu1736_bldo5 : enabled  2  1800000   supply_name:  vcc-cpvin  vcc-pc
pmu1736_bldo4 : disabled  0  700000   supply_name:
pmu1736_bldo3 : disabled  0  700000   supply_name:
pmu1736_bldo2 : disabled  0  700000   supply_name:
pmu1736_bldo1 : disabled  0  700000   supply_name:
pmu1736_aldo5 : enabled  0  2500000   supply_name:
pmu1736_aldo4 : enabled  0  3300000   supply_name:
pmu1736_aldo3 : enabled  1  1800000   supply_name:  avcc
pmu1736_aldo2 : enabled  0  1800000   supply_name:
pmu1736_aldo1 : disabled  0  700000   supply_name:
pmu1736_rtc : enabled  0  1800000   supply_name:
pmu1736_dcdc6 : disabled  0  500000   supply_name:
pmu1736_dcdc5 : enabled  0  1480000   supply_name:
pmu1736_dcdc4 : enabled  1  900000   supply_name:  vdd-sys
pmu1736_dcdc3 : enabled  0  900000   supply_name:
pmu1736_dcdc2 : enabled  0  1160000   supply_name:
pmu1736_dcdc1 : enabled  4  3300000   supply_name:  vcc-emmc  vcc-io  vcc-io  vcc-io
```

6.4 pwm 信息查询与背光调节

pwm 的用处这里是提供背光电源。

```
cat /sys/kernel/debug/pwm

platform/7020c00.s_pwm, 1 PWM device
 pwm-0   ((null)          ): period: 0 ns duty: 0 ns polarity: normal

platform/300a000.pwm, 2 PWM devices
 pwm-0   (lcd             ): requested enabled period: 20000 ns duty: 3984 ns polarity: normal
 pwm-1   ((null)          ): period: 0 ns duty: 0 ns polarity: normal
```

上面的 “requested enabled” 表示请求并且使能了，括号里面的 lcd 表示是由 lcd 申请的。

背光调节命令如下：

```
mount -t debugfs none /sys/kernel/debug
cd /sys/kernel/debug/dispdbg
// 获取当前背光值
echo getbl > command; echo lcd0 > name; echo 1 > start; cat info
// 设置当前背光值为255
echo setbl> command; echo lcd0 > name; echo 255 > param; echo 1 > start
```

6.5 查看管脚信息

```
cat /sys/kernel/debug/pinctrl/pio/pinmux-pins


pin 227 (PH3): twi1 (GPIO UNCLAIMED) function io_disabled group PH3
pin 228 (PH4): (MUX UNCLAIMED) (GPIO UNCLAIMED)
pin 229 (PH5): (MUX UNCLAIMED) pio:229
pin 230 (PH6): (MUX UNCLAIMED) pio:230
pin 231 (PH7): (MUX UNCLAIMED) pio:231
```

上面的信息我们知道 PH5，PH6 这些 IO 被申请为普通 GPIO 功能，而 PH3 被申请为 twi1。

6.6 查看时钟信息

```
cat /sys/kernel/debug/clk/clk_summary
```

这个命令可以看哪个时钟是否使能，然后频率是多少。

与显示相关的是 tcon，pll_video，mipi 等等。

```
cat /sys/kernel/debug/clk/clk_summary | grep tcon
cat /sys/kernel/debug/clk/clk_summary | grep pll_video
cat /sys/kernel/debug/clk/clk_summary | grep mipi
```

6.7 查看接口自带 colorbar

显示是一整条链路，中间任何一个环节出错，最终的表现都是显示异常，图像显示异常几个可能原因：

1. 图像本身异常。
2. 图像经过 DE（Display Engine）后异常。
3. 图像经过接口模块后异常。这是我们关注的点。

有一个简单的方法可以初步判断，接口模块（tcon 和 dsi 等）可以自己输出内置的一些 patten（比如说彩条、灰阶图、棋盘图等），当接口输出这些内置 patten 的时候，如果这时候显示就异常，这说明了：

1. LCD 的驱动或者配置有问题。
2. LCD 屏由于外部环境导致显示异常。

显示自带 patten 的方式：

在 linux-4.9 及其以上版本的内核，disp 的 sysfs 中有一个 attr 可以直接操作显示:

```
echo X > /sys/class/disp/disp/attr/colorbar
```

上面的操作是显示 colorbar，其中的 X 可以是 0 到 8，对应的含义如下图所示：

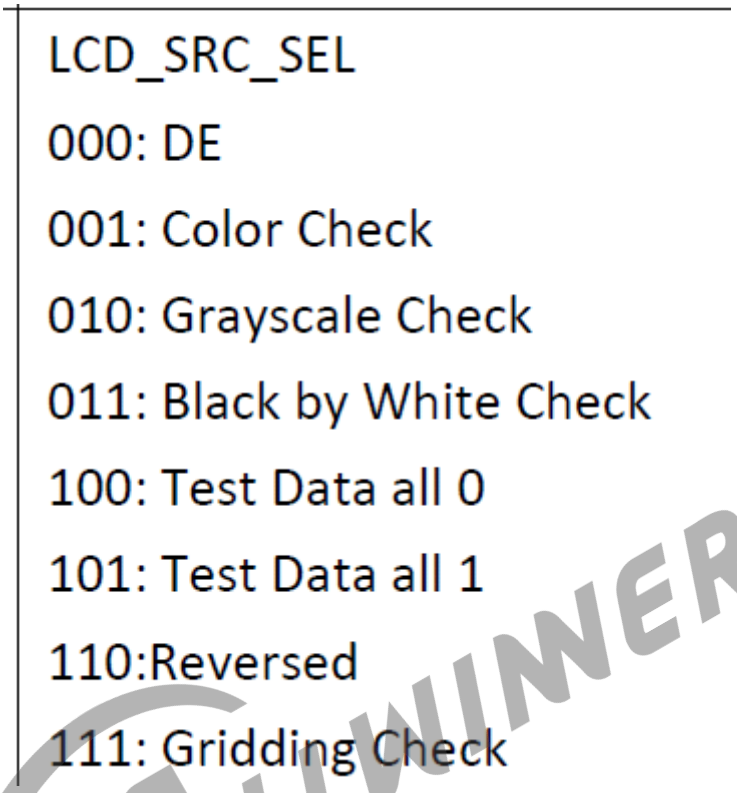

图 6-1: colorbar

如果有多个显示设备，想让第二个显示设备显示 colorbar 的话，那么先：

```
echo 1 > /sys/class/disp/disp/attr/disp
```

然后再执行上面操作。

如果没有这个 attr 的话，可以直接操作寄存器，也就是操作 tcon 寄存器的 040 偏移的最低 3 位。

在 linux 下，cd /sys/class/sunxi_dump 然后：

```
echo 0x06511040 > dump;cat dump
```

这样会打印当前 tcon 的 040 偏移寄存器的值，然后在上面值的基础上修改最低 3 位为上图的值即可，修改方式示例：

```
echo 0x06511040 0x800001f1 > write
```

注意 tcon 的基地址不一定是 0x06511000，不同平台不一样，请参考 SoC 文档获取 tcon 的基地址。

6.8 DE 截屏

显示出现异常的时候，有可能是下面三个原因：

1. SoC 端屏接口模块 +LCD 屏出现了问题。
2. 图像经过 SoC 端图像合成模块 (DE）处理后出现了问题。
3. 图像源本身就有问题。

本节介绍，确认图像源本身没问题的前提下，如何进一步确认是否经过 DE 处理之后图像是否有问题。

语法：

```
echo 屏幕索引 > /sys/class/disp/disp/attr/disp
echo 路径/bmp文件名 > /sys/class/disp/disp/attr/capture_dump
```

第一个 echo 的作用是确定捕捉哪个显示的，“屏幕索引” 可选取值是 0 或者 1。

第二个 echo 作用是生成截屏 bmp 文件，确保 “路径” 是可写的，有剩余空间的。

比如：

```
echo 0 > /sys/class/disp/disp/attr/disp
echo /data/xx.bmp > /sys/class/disp/disp/attr/capture_dump
```

这样就会在/data 目录下生成 screen 0 的 bmp 截图，文件名是 xx.bmp。

除了 bmp 之外，还支持保存 raw 数据，包括 RGB 和 YUV 颜色空间，它们以后缀来区分，用法如下：

```
# 截取yuv420p颜色空间的raw数据。
echo /data/xx.yuv420_p > /sys/class/disp/disp/attr/capture_dump
# 截取yuv420_sp_uvuv颜色空间的raw数据。
echo /data/xx.yuv420_sp_uvuv > /sys/class/disp/disp/attr/capture_dump
# 截取yuv420_sp_vuvu颜色空间的raw数据。
echo /data/xx.yuv420_sp_vuvu > /sys/class/disp/disp/attr/capture_dump
# 截取yuv420_sp_vuvu颜色空间的raw数据。
echo /data/xx.yuv420_sp_vuvu > /sys/class/disp/disp/attr/capture_dump
# 截取argb8888颜色空间的raw数据。
echo /data/xx.argb8888 > /sys/class/disp/disp/attr/capture_dump
# 截取abgr8888颜色空间的raw数据。
echo /data/xx.abgr8888 > /sys/class/disp/disp/attr/capture_dump
# 截取rgb888颜色空间的raw数据。
echo /data/xx.rgb888 > /sys/class/disp/disp/attr/capture_dump
# 截取bgr888颜色空间的raw数据。
echo /data/xx.bgr888 > /sys/class/disp/disp/attr/capture_dump
# 截取rgba8888颜色空间的raw数据。
echo /data/xx.rgba8888 > /sys/class/disp/disp/attr/capture_dump
# 截取bgra8888颜色空间的raw数据。
echo /data/xx.bgra8888 > /sys/class/disp/disp/attr/capture_dump
```

注意：这个功能只有 linux-4.9 以及后续的内核才支持这个功能。

6.9 fb 调试命令

```
// 查看fb分辨率
cat /sys/class/graphics/fb0/modes

// 查看fb是否有双缓冲
cat /sys/class/graphics/fb0/virtual_size

// 查看fb位深
cat /sys/class/graphics/fb0/bits_per_pixel

// 屏幕布满随机色点
cat /dev/urandom > /dev/fb0

// 清空屏幕
dd if=/dev/zero of=/dev/fb0

// 获取屏幕裸argb数据
cat /dev/fb0 > sreensnap

// 显示argb裸数据
dd if=sreensnap of=/dev/fb0
```

6.10 休眠、唤醒

```
mount -t debugfs none /sys/kernel/debug
cd /sys/kernel/debug/dispdbg
// 休眠
echo suspend > command; echo disp0 > name; echo 1 > start
// 唤醒
echo resume > command; echo disp0 > name; echo 1 > start
```

6.11 enhance 模块调试命令

```
mount -t debugfs none /sys/kernel/debug
cd /sys/kernel/debug/dispdbg
// 获取enhance模块信息
echo getinfo > command; echo enhance > name; echo 1 > start; cat info

// 设置disp0的色彩增强模式为 0：standard、1：enhance、2：soft、3：enhance+demo
echo 0 > /sys/class/disp/disp/attr/disp
echo 1 > /sys/class/disp/disp/attr/enhance_mode

// 使能丽色系统，色彩增强模式为 0：standard、1：enhance、2：soft、3：enhance+demo
echo 1 > /sys/class/disp/disp/attr/enhance_mode
// 设置色度，范围0~100
echo 9 > /sys/class/disp/disp/attr/enhance_bright
// 设置对比度，范围0~100
echo 9 > /sys/class/disp/disp/attr/enhance_contrast
// 设置饱和度，范围0~100
```

```
echo 9 > /sys/class/disp/disp/attr/enhance_saturation
```

6.12 色温调节

```
// 色温，范围[-256,256]
echo 0 > /sys/class/disp/disp/attr/color_temperature
```

7 FAQ

7.1 屏显示异常

总结过往经验，绝大部分屏显异常都是由于上下电时序和 timing 不合理导致。

请看屏时序参数说明和屏驱动分解。

7.2 黑屏-无背光

问题表现：完全黑屏，背光也没有。

有两种可能：

1. 屏驱动添加失败。驱动没有加载屏驱动，导致背光电源相关函数没有运行到。这个你可以通过调试方法定位下。
2. pwm 配置和背光电路的问题，pwm 的信息可以看pwm 信息和背光相关参数，另外就是直接测量下硬件测量下相关管脚和电压。

7.3 黑屏-有背光

黑屏但是有背光，可能有多种原因导致，请依次按以下步骤检查：

1. 没送图层。如果应用没有送任何图层那么表现的现象就是黑屏，通过查看显示信息一小节可以确定有没有送图层。如果确定没有图层，可以通过查看接口自带 colorbar，确认屏能否正常显示。
2. SoC 端的显示接口模块没有供电。SoC 端模块没有供电自然无法传输视频信号到屏上。一般 SoC 端模块供电的 axp 名字叫做 vcc-lcd，vcc-dsi，vcc33-lcd，vcc18-dsi 等。
3. 复位脚没有复位。如果有复位脚，请确保硬件连接正确，确保复位脚的复位操作有放到屏驱动中。
4. board.dts 中 lcd0 有严重错误。第一个是 lcd 的 timing 太离谱，请严格按照屏手册中的提示来写！参考屏时序参数说明。第二个就是，接口类型搞错，比如接的 DSI 屏，配置却写成 LVDS 的。
5. 屏的初始化命令不对。包括各个步骤先后顺序，延时等，这个时候请找屏厂确认初始化命令。

7.4 闪屏

分为几种：

1. 屏的整体在闪。

这个最大可能是背光电路的电压不稳定，检查电压。

2. 屏部分在闪，而且是概率性。

board.dts 中的时序填写不合理。

3. 屏上由一个矩形区域在闪。

屏极化导致，需要关机放一边再开机则不会。

7.5 条形波纹

有些 LCD 屏的像素格式是 18bit 色深（RGB666）或 16bit 色深（RGB565），建议打开 FRM 功能，通过 dither 的方式弥补色深，使显示达到 24bit 色深（RGB888）的效果。

设置 [lcd0] 的 lcd_frm 属性可以改善这种现象。请看lcd_frm解释。

7.6 背光太亮或者太暗

请看背光相关参数。

7.7 重启断电测试屏异常

花屏的第一个原因是 fps 过高，超过屏的限制：

FPS 异常是一件非常严重的事情，关系到整个操作系统的稳定，如果 fps 过高会造成系统带宽增加，送显流程异常，fps 过高还会造成 LCD 屏花屏不稳定，容易造成 LCD 屏损坏，FPS 过低则造成用户体验过差。

1. 通过查看查看显示信息一节，可以得知现在的实时统计的 fps。
2. 如果 fps 离正常值差很多，首先检查 board.dts 中 [lcd0] 节点，所填信息必须满足下面公式。

```
lcd_dclk_freq*num_of_pixel_clk=lcd_ht*lcd_vt*fps /1e9
```

其中，num_of_pixel_clk 通常为 1，表示发送一个像素所需要的时钟周期为 1 一个，低分辨率的 MCU 和串行接口通常需要 2 到 3 个时钟周期才能发送完一个像素。

如果上面填写没有错，通过查看查看时钟信息一节可以确认下几个主要时钟的频率信息，把这些信息和 board.dts 发给维护者进一步分析。

7.8 RGB 接口或者 I8080 接口显示抖动有花纹

1. 改大时钟管脚的管脚驱动能力

参考lcd_gpio_0一小节和pinctrl-0 和 pinctrl-1，修改驱动能力，改大。

还有另外一种写法，比如原来是：

```
lcdclk          = port:PD18<2><0><2><default>
```

可以改成：

```
lcdclk          = port:PD18<2><0><3><default>
```

2. 修改时钟相位，也就是修改 lcd_hv_clk_phase。由于发送端和接收端时钟相位的不同导致接收端解错若干位。

7.9 LCD 屏出现极化和残影

何谓液晶极化现象：实际上就是液晶电介质极化。就是在外界电场作用下，电介质内部沿电场方向产生感应偶极矩，在电解质表明出现极化电荷的现象叫做电介质的极化。

通俗的讲就是在液晶面板施加一定电压后，会聚集大量电荷，当电压消失的时候，这些聚集的电荷也要释放，但由于介电效应，这些聚集的电荷不会立刻释放消失，这些不会马上消失的惰性电荷造成了液晶的 DC 残留从而形成了极化现象。

几种常见的液晶极化现象

1. 液晶长期静止某个画面的时候，切换到灰阶画面的时候出现屏闪，屏闪一段时间后消失。这种现象属于残留电荷放电的过程。
2. 液晶长期静止某个画面的时候，出现四周发黑中间发白的现象，业内称为黑白电视框异常。
3. 非法关机的时候，重新上电会出现屏闪，屏闪一定时间后消失。与第一种原因相同。
4. 残影现象：当液晶静止在一个画面比较久的情况下，切换其他画面出现的镜像残留。残影的本质来说是液晶 DC 残留电荷导致，某种意义来说也属于液晶极化现象。

针对液晶屏出现极化和残影现象，有如下对策。

1. 调整 vcom 电压大小。

VCOM 是液晶分子偏转的参考电压，要求要稳定，对液晶显示有直接影响，具体的屏不同的话也是不同的。电压的具体值是根据输入的数据以及 Vcom 电压大小来确定的，用来显示各种不同灰阶，也就是实现彩色显示 GAMMA。Gamma 电压是用来控制显示器的灰阶的，一般情况下分为 G0~G14，不同的 Gamma 电压与 Vcom 电压之间的压差造成液晶旋转角度不同从而形成亮度的差异，Vcom 电压最好的状况是位于 G0 和 G14 的中间值，这样液晶屏的闪烁状况会最好。

调节 vcom 电压的方式，如果屏管脚有 vcom 管脚，直接调整相关电路，如果屏 driver IC 提供寄存器接口，可以通过寄存器接口来调整大小。

2. 严格按照屏规定的上下电时序来对屏进行开关屏。许多极化残影现象并非长时间显示静止显示某个画面导致的，而是由于关机或者关屏时没有严格按照下电时序导致的，比如该关的电没关，或者延时不够。

7.10 如何制作并替换开机 LOGO

调整完毕 LCD 后，一般需要替换开机 logo。

1. 准备一张 bmp 格式的图片，重命名为 bootlogo.bmp，需要分辨率小于等于 LCD 分辨率，bmp 颜色格式要和 sys_config.fex 或者 board.dts 中 fb0_format 配置的一致。

fb0_format=0 是 ARGB8888，fb0_format=8 是 RGB888，fb0_format=10 是 RGB565。

如果 fb0_format 配置的是 ARGB8888 32bit，logo 的是 RGB888 24bit，开机 logo 显示正常，但是运行 MiniGUI 等 UI 程序的时候可能会有异常。因为在部分平台，framebuffer 的像素格式会被改成和 logo 一样的位深，导致一些显示异常问题。

2. 建议准备好 bmp 后，在BMP 图片格式转换中转换颜色格式，fb0_format 配置为 0 时，转换参数设置为 Color:32(Ture color, RGB)，With rows direction: 保持默认。

如果不转换，有可能在 uboot 中看到正常的 logo，到 kernel 后，logo 透明了导致看不到 logo，因为 uboot 和 kernel 中显示通道的合成算法不同。

正常的 bootlogo.bmp，ARGB 中的 A 的值是 ff，如果不转换，可能 ARGB 中的 A 的值是 0，0 表示透明，ff 表示不透明。

3. 把 bootlogo.bmp 放到 tina/device/config/chips/芯片/configs/方案/configs/bootlogo.bmp，然后重新打包即可。

7.11 Checklist

调试 LCD 一般是用 Linux 固件，先把 Uboot 的显示关闭，点亮 Kernel 的显示。因为关闭 Uboot 显示会导致没有开机 logo，所以到内核后是黑屏，这个时候可以设置显示 colorbar，来检查 lcd 是否正常显示

1. 检查 Dts 中，屏的类型是否配置正确，如 mipi，rgb，lvds 的配置，各 gpio 是否与原理图一一对应，并且 gpio 与屏端的脚一一对应，不要接反了或接错了
2. 检查供电，电是最重要的，根据原理图看屏的 1.8V，3.3V 是否有供上，各数据和时钟脚是否有电，没有的话都需要配置上。有条件可以量屏端的偏置电压
3. 检查背光，黑屏要先判断有无背光，没有背光要先把背光点亮，配置好 pwm
4. 检查上电时序，最重要的包括屏的 reset 高低高控制，电源的控制
5. 如有有初始化代码的话，检查初始化代码，检查代码是否和屏厂给的一致，不要大意写错一个，有条件的可以用逻辑分析仪抓取屏端接受的参数，然后对比。要注意 uboot 开了显示后，kernel 中开机就不会走屏的初始化流程，休眠唤醒才会重新走初始化流程
6. 如果屏有测试模式的话，可以先进测试模式。测试模式可以亮屏，那么就可以排除电源，背光，上电时序，初始化代码下发等问题。如果不能亮，逻辑分析仪又抓到了初始化数据，那么可能屏，排线，转接板等损坏了
7. 如果测试模式能亮屏，取消测试模式后，只有背光黑屏，可以看/sys/class/disp/disp/attr/sys 节点，帧率是否接近 60fps，err 数有没有增长，可以尝试稍微调整下屏参数与 dclk
8. 如果还有多套硬件，尝试换套硬件测试，排除硬件损坏问题，如果还有问题，提个 case 给到专业的工程师吧

7.12 tina lcd 使用 runtime 功能

1. echo 1 > /sys/class/disp/disp/attr/runtime_enable
2. 打开/dev/disp 节点，调用 DISP_BLANK disable //开机后先执行 1、2 步，第 3 与 6 步后面循环调用。即调用 DISP_BLANK enable 后需要调用 DISP_BLANK disable
3. 调用 DISP_BLANK enable，驱动则会执行 disp_runtime_idle（此时屏上无显示背光亮，应用可以在调用 DISP_BLANK enable 前，调暗背光），5s 后会执行 disp_runtime_suspend //DE 模块关闭，背光关闭
4. echo mem > /sys/power/state，驱动会调用 disp_suspend //此处 LCD 会下电，DE 模块关闭
5. 按键唤醒，驱动会调用 disp_resume //此处 DE 模块打开，LCD 上电并初始化
6. 调用 DISP_BLANK disable，驱动会调用 disp_runtime_resume //此处背光打开

说明

1. **仅支持 linux-4.9 及以上内核。**
2. **注意第 3 步与第 6 步需要成对调用，因为内核维护的是设备引用计数。**
3. **注意打开/dev/disp 后不能关闭节点，否则会关闭图层，如果想在关闭节点时不关闭图层，可以把驱动 dev_disp.c 中 disp_release 函数里内容注释掉，已注释可忽略。**

8 总结

调试 LCD 显示屏实际上就是调试发送端芯片（全志 SoC）和接收端芯片（LCD 屏上的 driver IC）的一个过程：

1. 添加屏驱动请看添加屏驱动步骤和屏驱动说明。
2. 仔细阅读屏手册以及 driver IC 手册。
3. 仔细阅读硬件参数说明。
4. 确保 LCD 所需要的各路电源管脚正常。

著作权声明

商标声明

免责声明