# i.MX8X 板级开发板 5.4.24 内核定制

by John Li (NXA08200)
GSM 现场支持工程师
恩智浦半导体(上海),

本文说明i.MX8X 板级开发包版本5.4.24的内核细节，以帮助客户了解i.MX8X的内核是如何运行的，以及如何修改到客户的新板上。

阅读本文之前请先阅读文档\imx-yocto-L5.4.24_2.1.0 \ i.MX_Yocto_Project_User's_Guide.pdf，i.MX_Linux_User's_Guide.pdf。预先熟悉一下i.MX8X的编译环境，本文部分内容与之重复。

另外请参考i.MX_Reference_Manual.pdf和i.MX_Linux_Release_Notes.pdf了解i.MX8X MEK板板级开发包的基本情况，本文中是对此文档的查缺补漏，并专注于bring up相关软件定制, 与产品化定制。i.MX_Porting_Guide.pdf为官方的porting guide，建议阅读。

注意本文是由i.MX6修改过来的，目前版本中关于linux/dts通用部分还没有完全修改过来。所以推荐阅读第1章，2.1节，第3,6章。

请注意本文为培训和辅助文档，本文不是官方文档的替代，请一切以官方文档为准。

| 历史 | 说明 | 作者 |
|---|---|---|
| V1 | ● 创建文档 | ● John.Li |
| V2 | ● 修改了一些错误<br>● 更新了第 6.12 章 | ● John.Li |
| V3 | ● 增加 nandflash 支持/DDR 修改/i.MX8DX 支持 | ● John.Li |
| V4 | ● 更新至 4.19.35 | ● John.Li |
| V5 | ● 更新至 5.4.24<br>● 增加 lvds/dsi serdas/ahd/digital sensor/tja1101 驱动支持<br>● 更新 DTS/SPI/i.MX 8DX/DXP 支持章节 | ● John.Li |
| V6 | ● 更新至 5.4.24 | ● John.Li |

目录

1 创建 i.MX8QXP Linux 5.4.24 板级开发包编译环境 **Error! Bookmark not defined.**
1.1 下载板级开发包..... **Error! Bookmark not defined.**
1.2 创建yocto编译环境:. **Error! Bookmark not defined.**
1.3 独立编译 ..... **Error! Bookmark not defined.**
2 Device Tree..... **Error! Bookmark not defined.**
2.1 恩智浦的device Tree结构**Error! Bookmark not defined.**
2.2 device Tree的由来(no updates)**Error! Bookmark not defined.**
2.3 device Tree的基础与语法(no updates)**Error! Bookmark not defined.**
2.4 device Tree的代码分析(no updates)**Error! Bookmark not defined.**
3 恩智浦i.MX8XBSP 包文件目录结构. **Error! Bookmark not defined.**
4 恩智浦i.MX8XBSP的编译(no updates)..... **Error! Bookmark not defined.**
4.1 需要编译哪些文件 ... **Error! Bookmark not defined.**
4.2 如何编译这些文件 ... **Error! Bookmark not defined.**
4.3 如何链接为目标文件及链接顺序**Error! Bookmark not defined.**
4.4 kernel Kconfig ..... **Error! Bookmark not defined.**
5 恩智浦BSP的内核初始化过程(no updates) ..... **Error! Bookmark not defined.**
5.1 初始化的汇编代码 ... **Error! Bookmark not defined.**
5.2 初始化的C代码..... **Error! Bookmark not defined.**
5.3 init_machine..... **Error! Bookmark not defined.**
6 恩智浦BSP的内核定制 .....3
6.1 DDR修改.....4
6.2 IO管脚配置与Pinctrl驱动.....5
6.3 新板bringup.....21
6.4 更改调试串口 .....30
6.5 uSDHC设备定制(eMMC flash,SDcard, SDIOcard)35
6.6 LVDS LCD 驱动定制**Error! Bookmark not defined.**
6.7 LVDS LDB SerDas驱动支持 .....53
6.8 MiPi DSI SerDas驱动支持 .....59
6.9 V4L2框架汽车级高清摄像头/桥驱动：数字/模拟 ...63
6.10 GPIO_Key 驱动定制 .....80
6.11 GPIO_LED 驱动定制 .....84
6.12 Fuse nvram驱动.....87
6.13 SPI与SPI Slave驱动 .....88
6.14 USB 3.0 TypeC 改成 USB 3.0 TypeA(未验证).....96
6.15 汽车级以太网驱动定制 .....96
6.16 i.MX8DX MEK支持.....115
6.17 i.MX8DXP MEK支持 .....115
6.18 NAND Flash支持与烧录 .....116

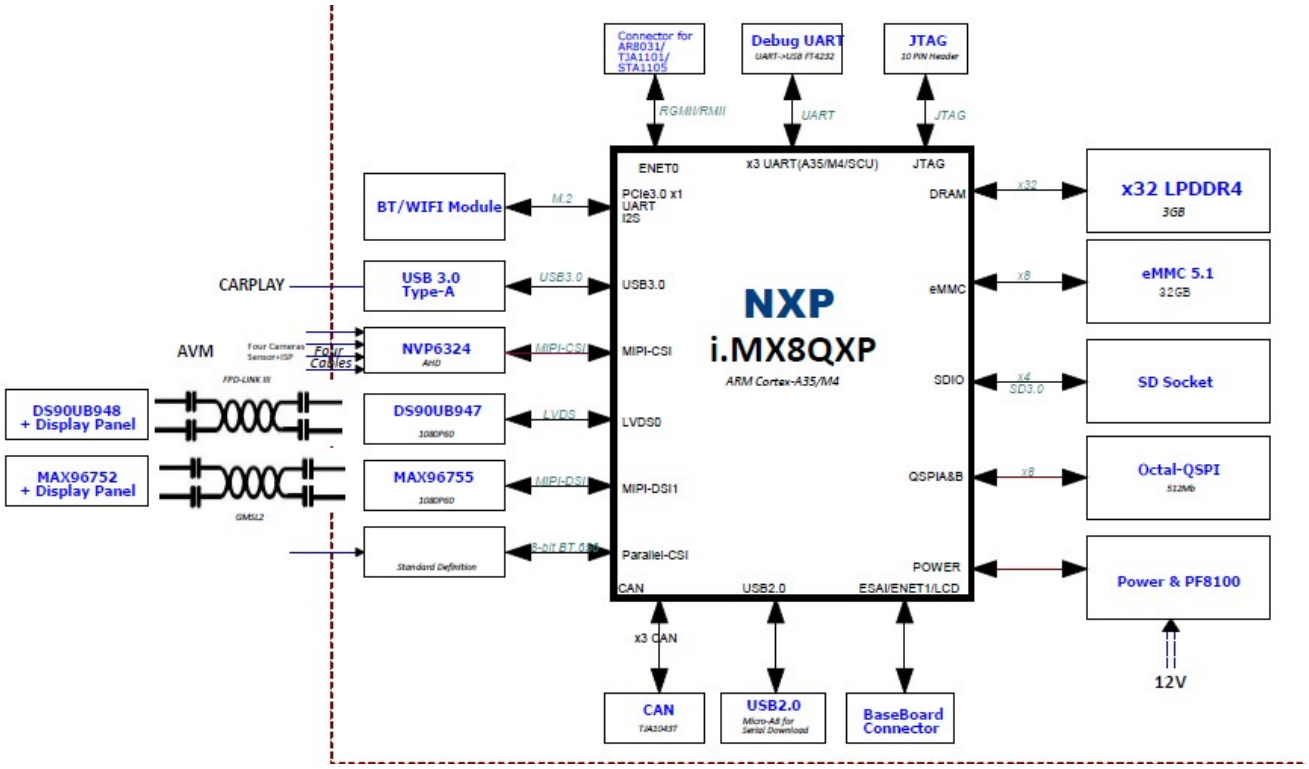
# 6 恩智浦 BSP 的内核定制

关于 i.MX8X MEK 板 BSP 的情况，请参考 BSP 用户手册” imx-yocto-L5.4.24_2.1.0 \i.MX_Reference_Manual.pdf”,关于一个基本定制和用户手册中没有涉及到的部分，请参考此章。

Porting 还可以参考文档：i.MX_Porting_Guide.pdf。

从 i.MX8X MEK 板的 BSP 修改为客户自己产品的板子，往往还需要一定的定制和一些驱动的再开发，比如说在汽车应用中的 serdas 屏，camera 等，所以本文主要关注这些接近产品级的定制和开发，i.MX8X MEK 的原生驱动情况请参考以上文档。

NXP i.MX AE 团队设计了一块接近与汽车产品的开发板，采用 6 层板降低成本，设计了 LVDS 的 Serdas 屏，MiPi DSI 的 Serdas 屏，AHD 的 camera bridge 输入，TJA1101 的汽车级以太网，本文会说明一下相关驱动的情况，建议通过 NXP 支持窗口获得此开发板相关硬件资料与驱动源代码（目前 patch 版本是 4.14.98_2.3.0,需要手动 porting 到 5.4.24_2.1.0）。框图如下：




另外通过 Max9286 子板，还可以连接各类使用 GMSL 同轴电缆连接的车用数字序列化摄像头，本文也有介绍。

i.MX8X 的集成度比较高，需要修改的驱动包括三类：

1. 一般如 GPU/VPU/ASRC/DSP 等是属于芯片内部模块，基本不需要修改。

2. i.MX8QXP 自己支持的一些外设驱动，比如说 usb/uart/display，一般只需要简单修改。
3. 另外需要调试的外设可能包括: 网口的 phy, audio codec(另有文档描述)，屏，电容或电阻式 i2C 接口触摸屏，i2c 的其它接口外设如 RTC 芯片等，此类驱动除了 i.MX8QXP MEK 板原来设计就含有的，就需要重新开发。

## 1.1 DDR 修改

根据文档《MX8X_5.4.24_ga_BootLoader_V*-*.pdf》所述，DDR 的修改主要是在 bootloader 中进行，而且 bootloader 会将 memory 的大小参数传递给内核，所以内核不需要去修改 memory 大小，如下：

\linux-imx\arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
memory@80000000 {
                device_type = "memory";
        reg = <0x00000000 0x80000000 0 0x10000000>; /*          reg = <0x00000000 0x80000000 0 0x40000000>;
johnli change to 256MB*/
                /* DRAM space - 1, size : 1 GB DRAM */ ////memory开始地址是0x80000000,大小为1GB,事实上内核会用uboot传过来的3GB的大小。
};
```

但是针对应用不同，对 reserved memory 和 cma 大小是可以调节的，这个主要是在内核 DTS 中配置，我们考虑一种极限情况，比如说 V2X 应用，没有 GPU/VPU/DSP/显示要求，那相关驱动的 reserved memory 可以完全去掉：

\linux-imx\arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
reserved-memory {…
                        /*
                 * reserved-memory layout
                 * 0x8800_0000 ~ 0x8FFF_FFFF is reserved for M4
                 * Shouldn't be used at A core and Linux side.
                 *
                 *///此段reserved内存是在bootloader中定义的，请参考MX8X_5.4.24_ga_BootLoader_V7-20200519.pdf文档如何去掉。
                /* //此处也去掉
                m4_reserved:
                */
    /*
                 decoder_boot:…
                 encoder_boot:…
              decoder_rpc:…
                 encoder_rpc:…
      encoder_reserved:… //VPU相关reserved memory,没有VPU要求可以注掉，注意一下调用处和VPU驱动最好也一起注掉。

                 rpmsg_reserved:…
                 rpmsg_dma_reserved:…//rpmsg使用，没有M4或PCIe(PCIe也用到了此段内存)可以注掉，注意一下调用处和PCIe驱动最好也一起注掉。
```

```
                */
```

如下为CMA的配置，如果没有比较大的CMA应用，比如说GPU/VPU/显示，可以缩小，或是内存比较小的，也要缩小，我们是定义为整个内存的1/3大小：

```
        linux,cma {
                        compatible = "shared-dma-pool";
                        reusable;
                        size = <0 0x5000000>;
                        alloc-ranges = <0 0x8fb00000 0 0x5000000>; /*johnli set the dma on high 256MB, size=80MB*///
考虑极限情况下只有256MB内存的情况下，把CMA放在最高位置，大小为80MB
                        /*
                        size = <0 0x3c000000>;
                        alloc-ranges = <0 0x96000000 0 0x3c000000>;//内存启始地址是0x80000000, i.MX8QXP MEK
板3GB LPPDR4，CMA定义的内存启始地址是0x96000000，大小是980MB
                        */
                        linux,cma-default;
                };
```

以下为 i.MX8QXP MEK 配置为 256MB 内存，去掉大部分驱动后启动的内存分配情况：

```
root@imx8qxpmek:~# cat /proc/meminfo
MemTotal:         221076 kB
MemFree:           67456 kB
MemAvailable:      53516 kB
…
```

## 1.2 IO 管脚配置与 Pinctrl 驱动

### 1.2.1 i.MX8QXP/DX MEK 板的管脚配置

i.MX8QXP/DX MEK 板的 IO 管脚配置大部分定义在:

arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;
        imx8qxp-mek {
                pinctrl_hog: hoggrp {
                        fsl,pins = < //此处定义通用的，一定会被调用的 iomux 设置，一般用于多个驱动共用的 pins，
如 MCLK(audio, camera 都会用)，或通用 GPIOs
                                IMX8QXP_MCLK_OUT0_ADMA_ACM_MCLK_OUT0      0x0600004c
                                IMX8QXP_COMP_CTL_GPIO_1V8_3V3_GPIORHB_PAD 0x000514a0
                        >;
                };
```

```
                pinctrl_cm40_i2c: cm40i2cgrp {
                fsl,pins = <
                        IMX8QXP_ADC_IN1_M40_I2C0_SDA                    0x0600004c
                        IMX8QXP_ADC_IN0_M40_I2C0_SCL                    0x0600004c
                >;
        };

        pinctrl_cm40_i2c_gpio: cm40i2cgrp-gpio {
                fsl,pins = <
                        IMX8QXP_ADC_IN1_LSIO_GPIO1_IO09         0xc600004c
                        IMX8QXP_ADC_IN0_LSIO_GPIO1_IO10         0xc600004c
                >;
        };

        pinctrl_i2c0_mipi_lvds0: mipi_lvds0_i2c0_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI0_I2C0_SCL_MIPI_DSI0_I2C0_SCL           0xc6000020
                        IMX8QXP_MIPI_DSI0_I2C0_SDA_MIPI_DSI0_I2C0_SDA           0xc6000020
                        IMX8QXP_MIPI_DSI0_GPIO0_01_LSIO_GPIO1_IO28              0x00000020
                >;
        };

        pinctrl_i2c0_mipi_lvds1: mipi_lvds1_i2c0_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI1_I2C0_SCL_MIPI_DSI1_I2C0_SCL           0xc6000020
                        IMX8QXP_MIPI_DSI1_I2C0_SDA_MIPI_DSI1_I2C0_SDA           0xc6000020
                        IMX8QXP_MIPI_DSI1_GPIO0_01_LSIO_GPIO2_IO00              0x00000020
                >;
        };

        pinctrl_esai0: esai0grp {
                fsl,pins = <
                        IMX8QXP_ESAI0_FSR_ADMA_ESAI0_FSR        0xc6000040
…
                >;
        };
```

```
        pinctrl_fec1: fec1grp {
                fsl,pins = <
                        IMX8QXP_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB0_PAD        0x000014a0
                        IMX8QXP_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB1_PAD        0x000014a0
                        IMX8QXP_ENET0_MDC_CONN_ENET0_MDC                     0x06000020
                        IMX8QXP_ENET0_MDIO_CONN_ENET0_MDIO                   0x06000020
…
                >;
        };

        pinctrl_fec2: fec2grp {
                fsl,pins = <
                        IMX8QXP_ESAI0_SCKR_CONN_ENET1_RGMII_TX_CTL           0x00000060
                        IMX8QXP_ESAI0_FSR_CONN_ENET1_RGMII_TXC               0x00000060
…
                >;
        };

        pinctrl_flexspi0: flexspi0grp {
                fsl,pins = <
                        IMX8QXP_QSPI0A_DATA0_LSIO_QSPI0A_DATA0     0x06000021
…
                        IMX8QXP_QSPI0B_SS1_B_LSIO_QSPI0B_SS1_B     0x06000021
                >;
        };

        pinctrl_ioexp_rst: ioexp_rst_grp {
                fsl,pins = <
                        IMX8QXP_SPI2_SDO_LSIO_GPIO1_IO01                     0x06000021
                >;
        };

        pinctrl_isl29023: isl29023grp {
                fsl,pins = <
                        IMX8QXP_SPI2_SDI_LSIO_GPIO1_IO02                     0x00000021
```

```
                >;
        };

        pinctrl_lpi2c1: lpi2c1grp {
                fsl,pins = <
                        IMX8QXP_USB_SS3_TC1_ADMA_I2C1_SCL                0x06000021
                        IMX8QXP_USB_SS3_TC3_ADMA_I2C1_SDA                0x06000021
                >;
        };

        pinctrl_flexcan1: flexcan0grp {
                fsl,pins = <
                        IMX8QXP_FLEXCAN0_TX_ADMA_FLEXCAN0_TX        0x21
                        IMX8QXP_FLEXCAN0_RX_ADMA_FLEXCAN0_RX        0x21
                >;
        };

        pinctrl_flexcan2: flexcan1grp {
                fsl,pins = <
                        IMX8QXP_FLEXCAN1_TX_ADMA_FLEXCAN1_TX        0x21
                        IMX8QXP_FLEXCAN1_RX_ADMA_FLEXCAN1_RX        0x21
                        >;
        };

        pinctrl_lpuart0: lpuart0grp {
                fsl,pins = <
                        IMX8QXP_UART0_RX_ADMA_UART0_RX                0x06000020
                        IMX8QXP_UART0_TX_ADMA_UART0_TX                0x06000020
                >;
        };

        pinctrl_lpuart1: lpuart1grp {
                fsl,pins = <
                        IMX8QXP_UART1_TX_ADMA_UART1_TX        0x06000020
                        IMX8QXP_UART1_RX_ADMA_UART1_RX        0x06000020
                        IMX8QXP_UART1_RTS_B_ADMA_UART1_RTS_B        0x06000020
```

```
                        IMX8QXP_UART1_CTS_B_ADMA_UART1_CTS_B        0x06000020
                >;
        };

        pinctrl_lpuart2: lpuart2grp {
                fsl,pins = <
                        IMX8QXP_UART2_TX_ADMA_UART2_TX              0x06000020
                        IMX8QXP_UART2_RX_ADMA_UART2_RX              0x06000020
                >;
        };

        pinctrl_lpuart3: lpuart3grp {
                fsl,pins = <
                        IMX8QXP_FLEXCAN2_TX_ADMA_UART3_TX  0x06000020
                        IMX8QXP_FLEXCAN2_RX_ADMA_UART3_RX  0x06000020
                >;
        };

        pinctrl_pcieb: pcieagrp{
                fsl,pins = <
                        IMX8QXP_PCIE_CTRL0_PERST_B_LSIO_GPIO4_IO00          0x06000021
                        IMX8QXP_PCIE_CTRL0_CLKREQ_B_LSIO_GPIO4_IO01         0x06000021
                        IMX8QXP_PCIE_CTRL0_WAKE_B_LSIO_GPIO4_IO02           0x04000021
                >;
        };

        pinctrl_pwm_mipi_lvds0: mipi_lvds0_pwm_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI0_GPIO0_00_MIPI_DSI0_PWM0_OUT               0x00000020
                >;
        };

        pinctrl_pwm_mipi_lvds1: mipi_lvds1_pwm_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI1_GPIO0_00_MIPI_DSI1_PWM0_OUT               0x00000020
                >;
```

```
        };

        pinctrl_sai1: sai1grp {
                fsl,pins = <
                        IMX8QXP_SAI1_RXD_ADMA_SAI1_RXD    0x06000040
…
                >;
        };

        pinctrl_typec: typecgrp {
                fsl,pins = <
                        IMX8QXP_SPI2_SCK_LSIO_GPIO1_IO03                0x06000021
                >;
        };

        pinctrl_typec_mux: typecmuxgrp {
                fsl,pins = <
                        IMX8QXP_ENET0_REFCLK_125M_25M_LSIO_GPIO5_IO09                0x60
                >;
        };

        pinctrl_usdhc1: usdhc1grp {
                fsl,pins = <
                        IMX8QXP_EMMC0_CLK_CONN_EMMC0_CLK                0x06000041
                        IMX8QXP_EMMC0_CMD_CONN_EMMC0_CMD                0x00000021
…
                        IMX8QXP_EMMC0_STROBE_CONN_EMMC0_STROBE                0x00000041
                >;
        };

        pinctrl_usdhc2_gpio: usdhc2gpiogrp {
                fsl,pins = <
                        IMX8QXP_USDHC1_RESET_B_LSIO_GPIO4_IO19    0x00000021
                        IMX8QXP_USDHC1_WP_LSIO_GPIO4_IO21        0x00000021
                        IMX8QXP_USDHC1_CD_B_LSIO_GPIO4_IO22       0x00000021
                >;
```

```
        };


        pinctrl_usdhc2: usdhc2grp {
                fsl,pins = <
                        IMX8QXP_USDHC1_CLK_CONN_USDHC1_CLK                  0x06000041
                        IMX8QXP_USDHC1_CMD_CONN_USDHC1_CMD                      0x00000021
…
                >;
        };

        pinctrl_i2c_mipi_csi0: i2c_mipi_csi0 {
                fsl,pins = <
                        IMX8QXP_MIPI_CSI0_I2C0_SCL_MIPI_CSI0_I2C0_SCL       0xc2000020
                        IMX8QXP_MIPI_CSI0_I2C0_SDA_MIPI_CSI0_I2C0_SDA       0xc2000020
                >;
        };

        pinctrl_mipi_csi0: mipi_csi0 {
                fsl,pins = <
                        IMX8QXP_MIPI_CSI0_GPIO0_01_LSIO_GPIO3_IO07          0xC0000041
                        IMX8QXP_MIPI_CSI0_GPIO0_00_LSIO_GPIO3_IO08          0xC0000041
                        IMX8QXP_MIPI_CSI0_MCLK_OUT_MIPI_CSI0_ACM_MCLK_OUT   0xC0000041
                >;
        };

        pinctrl_parallel_csi: parallelcsigrp {
                fsl,pins = <
                        IMX8QXP_CSI_D00_CI_PI_D02           0xC0000041
…
                >;
        };

        pinctrl_wifi: wifigrp{
                fsl,pins = <
                        IMX8QXP_SCU_BOOT_MODE3_SCU_DSC_RTC_CLOCK_OUTPUT_32K  0x20
```

```
                >;
        };

        pinctrl_wifi_init: wifi_initgrp{
                fsl,pins = <
                        /* reserve pin init/idle_state to support multiple wlan cards */
                >;
        };

};
```

恩智浦已经把每一个 IO 管脚可能使用到的 IOmux 功能，以及与此功能相对应的 IOPAD 属性都已经准备好，定义在文件 include\dt-bingings\pinctrl\pads-imx8qxp.h 中，所以只需要直接在数组中包括我们想使用的 IO 管脚及功能的宏就可以了，比如说:

```
IMX8QXP_MCLK_OUT0_ADMA_ACM_MCLK_OUT0      0x0600004c

#define IMX8QXP_MCLK_OUT0_ADMA_ACM_MCLK_OUT0                IMX8QXP_MCLK_OUT0                0

#define IMX8QXP_MCLK_OUT0                76   /* ADMA.ACM.MCLK_OUT0, ADMA.ESAI0.TX_HF_CLK,
ADMA.LCDIF.CLK, ADMA.SPI2.SDO, LSIO.GPIO0.IO20 */
```

其中 IMX8QXP _MCLK_OUT0_ADMA 表示这个管脚名，MCLK_OUT0 表示这个管脚 IOmux 的功能，然后此功能附带的 IOpad 属性使用 0x0600004c 宏来决定。

IOmux 对应寄存器定义为：

| | |
|---|---|
| 29-27<br>mux_mode | mux_mode<br>mux_mode<br>000b - ADMA_ACM_MCLK_OUT0<br>001b - ADMA_ESAI0_TX_HF_CLK<br>010b - ADMA_LCDIF_CLK<br>011b - ADMA_SPI2_SDO<br>100b - LSIO_GPIO0_IO20 |

IOpad 对应寄存器定义为：

| | |
|---|---|
| 26-25<br>sw_config | output and input configuration<br>output and input configuration<br>00b - DEFAULT<br>01b - OPEN_DRAIN<br>10b - OPEN_DRAIN_INPUT<br>11b - INOUT |

=0b11: INOUT.

| | |
|---|---|
| 6-5<br>PULL | Pull Down Pull Up<br>Pull Down Pull Up<br>00b - Prohibited<br>01b - pull up<br>10b - pull down<br>11b - pull disabled |

=0b10: pull down

| | |
|---|---|
| 0<br>PDRV | Drive<br>Drive<br>0b - Output is configured in High Drive mode both in 1.8 V and 3.3 V applications<br>1b - Output is configured in Low Drive mode both in 1.8 V and 3.3 V applications |

=0b0: High Drive mode.

GPIO 示例如下：

```
IMX8QXP_USDHC1_RESET_B_LSIO_GPIO4_IO19      0x00000021
```

| | |
|---|---|
| 6-5<br>PULL | Pull Down Pull Up<br>Pull Down Pull Up<br>00b - Prohibited<br>01b - pull up<br>10b - pull down<br>11b - pull disabled |

=0b01: pull up

| | |
|---|---|
| 0<br>PDRV | Drive<br>Drive<br>0b - Output is configured in High Drive mode both in 1.8 V and 3.3 V applications<br>1b - Output is configured in Low Drive mode both in 1.8 V and 3.3 V applications |

=0b0: Low Drive mode.

pin control subsystem 的文件列表

1、源文件列表

我们整理 linux/drivers/pinctrl 目录下 pin control subsystem 的源文件列表如下：

| 文件名 | 描述 |
|---|---|
| core.c core.h | pin control subsystem 的 core driver |
| pinctrl-utils.c pinctrl-utils.h | pin control subsystem 的一些 utility 接口函数 |
| pinmux.c pinmux.h | pin control subsystem 的 core driver(pin muxing 部分的代码，也称为 |

| | |
|---|---|
| | pinmux driver) |
| pinconf.c pinconf.h | pin control subsystem 的 core driver(pin config 部分的代码，也称为 pin config driver) |
| devicetree.c devicetree.h | pin control subsystem 的 device tree 代码 |
| pinctrl-imx6q.c/ pinctrl-imx6dl.c<br>pinctrl-imx.c | i.mx6q/dl pin controller 的 low level driver。 |

2、和其他内核模块接口头文件

很多内核的其他模块需要用到 pin control subsystem 的服务，这些头文件就定义了 pin control subsystem 的外部接口以及相关的数据结构。我们整理 linux/include/linux/pinctrl 目录下 pin control subsystem 的外部接口头文件列表如下：

| 文件名 | 描述 |
|---|---|
| consumer.h | 其他的 driver 要使用 pin control subsystem 的下列接口：<br>a、设置引脚复用功能<br>b、配置引脚的电气特性<br>这时候需要 include 这个头文件 |
| devinfo.h | 这是 for linux 内核的驱动模型模块（driver model）使用的接口。struct device 中包括了一个 struct dev_pin_info *pins 的成员，这个成员描述了该设备的引脚的初始状态信息，在 probe 之前，driver model 中的 core driver 在调用 driver 的 probe 函数之前会先设定 pin state |
| machine.h | 和 machine 模块的接口。 |

3、Low level pin controller driver 接口

我们整理 linux/include/linux/pinctrl 目录下 pin control subsystem 提供给底层 specific pin controller driver 的头文件列表如下：

| 文件名 | 描述 |
|---|---|
| pinconf-generic.h | 这个接口主要是提供给各种 pin controller driver 使用的，不是外部接口。 |
| pinconf.h | pin configuration 接口 |
| pinctrl-state.h | pin control state 状态定义 |
| pinmux.h | pin mux function 接口 |

## 1.2.2 pin control driver 代码分析

pin control 驱动主要功能有：设置 iomux，设置 iopad,设置 input daisy chain

#### 1.2.2.1 pin controller 相关的 DTS 描述

类似其他的硬件，pin controller 这个 HW block 需要是 device tree 中的一个节点。此外，各个其他的 HW block 在驱动之前也需要先配置其引脚复用功能，因此，这些 device（我们称 pin controller 是 host，那么这些使用 pin controller 进行引脚配置的 device 叫做 client device）也需要在它自己的 device tree node 中描述 pin control 的相关内容

```
        //arch/arm/boot/dts/imx6qdl.dtsi
                        iomuxc: iomuxc@020e0000 {
                                compatible = "fsl,imx6dl-iomuxc", "fsl,imx6q-iomuxc";
                                reg = <0x020e0000 0x4000>;
                        };
// arch/arm/boot/dts/imx6qdl-sabresd.dtsi
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;
        imx6qdl-sabresd {
                pinctrl_hog: hoggrp {
                        fsl,pins = <……
                >;
                };
…
};
```

每个 pin configuration 都是 pin controller 的 child node，描述了 client device 要使用到的一组 pin 的配置信息。具体如何定义 pin configuration 是和具体的 pin controller 相关的。

在 pin controller node 中定义 pin configuration 其目的是为了让 client device 引用。所谓 client device 其实就是使用 pin control subsystem 提供服务的那些设备，例如串口设备。在使用之前，我们一般会在初始化代码中配置相关的引脚功能是串口功能。有了 device tree，我们可以通过 device tree 来传递这样的信息。也就是说，各个 device 可以通过自己节点的属性来指向 pin controller 的某个 child node，也就是 pin configuration 了

一个典型的 device tree 中的外设 node 定义如下：

```
device-node-name {
    定义该 device 自己的属性
    pinctrl-names = "sleep", "active";－－－－－－（1）
    pinctrl-0 = <pin-config-0-a>;－－－－－－－－－－－－－－－－（2）
    pinctrl-1 = <pin-config-1-a pin-config-1-b>;
  };
```

（1）pinctrl-names 定义了一个 state 列表。那么什么是 state 呢？具体说应该是 pin state，对于一个 client device，它使用了一组 pin，这一组 pin 应该同时处于某种状态，毕竟这些 pin 是属于一个具体的设备功能。state 的定义和电源管理关系比较紧密，例如当设备 active 的时候，我们需要 pin controller 将相关的一组 pin 设定为具体的设备功能，而当设备进入 sleep 状态的时候，需要 pin controller 将相关的一组 pin 设定为普通 GPIO，并精确的控制 GPIO 状态以便节省系统的功耗。state 有两种，标识，一种就是 pinctrl-names 定义的字符串列表，另外一种就是 ID。ID 从 0 开始，依次加一。根据例子中的定义，state ID 等于 0（名字是 active）的 state 对应 pinctrl-0 属性，state ID 等于 1（名字是 idle）的 state 对应 pinctrl-1 属性。具体设备 state 的定义和各个设备相关，具体参考在自己的 device bind。

（2）pinctrl-x 的定义。pinctrl-x 是一个句柄（phandle）列表，每个句柄指向一个 pin configuration。有时候，一个 state 对应多个 pin configure。例如在 active 的时候，I2C 功能有两种配置，可以从不同的 pad iomux 出来

以 uart 为例:

```
&uart1 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_uart1>;
        status = "okay";
};
```

该 serial device 只定义了一个 state 就是 default，对应 pinctrl-0 属性定义。pinctrl-0 是一个句柄（phandle）列表，每个句柄指向一个 pin configuration, 这儿用到的是 pinctrl_uart1.

#### 1.2.2.2 pin controller 驱动的初始化

根据device tree代码分析，我们知道，在系统初始化的时候，dts 描述的 device node 会形成一个树状结构，在 machine 初始化的过程中，会 scan device node 的树状结构，将真正的硬件 device node 变成一个个的设备模型中的 device 结构（比如 struct platform_device）并加入到系统中。我们看看具体 imx6qdl 描述 pin controller 的 dts code，如下：

```
        //arch/arm/boot/dts/imx6qdl.dtsi
                        iomuxc: iomuxc@020e0000 {
                                compatible = "fsl,imx6dl-iomuxc", "fsl,imx6q-iomuxc";
                                reg = <0x020e0000 0x4000>;
                        };
```

reg 属性描述 pin controller 硬件的地址信息，开始地址是 0x020e0000 ，地址长度是 0x4000。compatible 属性用来描述 pin controller 的 programming model。该属性的值是 string list，定义了一系列的 modle（每个 string 是一个 model）。这些字符串列表被操作系统用来选择用哪一个 pin controller driver 来驱动该设备，后面的代码会更详细的描述。 pin control subsystem 要想进行控

制，必须首先了解自己控制的对象，也就是说软件需要提供一个方法将各种硬件信息（total 有多少可控的 pin， pin 的复用情况以及 pin 的配置情况）注册到 pin control subsystem 中，这也是 pin controller driver 的初始化的主要内容。这些信息当然可以通过定义静态的表格（参考 linux/drivers/pinctrl 目录下的 pinctrl-imx6q.c 文件，该文件定义了一个大数组 imx6q_pinctrl_pads 来描述每一个 pin），也可以通过 dts 加上静态表格的方式.

当然，: iomuxc@020e0000 这个 device node 也会变成一个 platform device 加入系统。有了 device，那么对应的 platform driver 是如何注册到系统中的呢？代码如下：

```
static struct of_device_id imx6q_pinctrl_of_match[] = {
        { .compatible = "fsl,imx6q-iomuxc", },
        { /* sentinel */ }
};
static int imx6q_pinctrl_probe(struct platform_device *pdev)
{
        return imx_pinctrl_probe(pdev, &imx6q_pinctrl_info);
}
static struct platform_driver imx6q_pinctrl_driver = {
        .driver = {
                .name = "imx6q-pinctrl",
                .owner = THIS_MODULE,
                .of_match_table = imx6q_pinctrl_of_match, //  匹配列表
        },
        .probe = imx6q_pinctrl_probe,  //该 driver 的初始化函数
        .remove = imx_pinctrl_remove,
};
```

probe 过程（driver 初始化过程）：

```
int imx_pinctrl_probe(struct platform_device *pdev,
                        struct imx_pinctrl_soc_info *info)
{
        struct imx_pinctrl *ipctl;
        struct resource *res;
        int ret;

        if (!info || !info->pins || !info->npins) {
                dev_err(&pdev->dev, "wrong pinctrl info\n");
                return -EINVAL;
        }
```

```
        info->dev = &pdev->dev;


        /* Create state holders etc for this driver */
        ipctl = devm_kzalloc(&pdev->dev, sizeof(*ipctl), GFP_KERNEL);
        if (!ipctl)
                return -ENOMEM;
/*
```

devm_kzalloc 函数是为 struct imx_pinctrl 数据结构分配内存。每当 driver probe 一个具体的 device 实例的时候，都需要建立一些私有的数据结构来保存该 device 的一些具体的硬件信息（本场景中，这个数据结构就是 struct imx_pinctr）。在过去，驱动工程师多半使用 kmalloc 或者 kzalloc 来分配内存，但这会带来一些潜在的问题。例如：在初始化过程中，有各种各样可能的失败情况，这时候就依靠 driver 工程师小心的撰写代码，释放之前分配的内存。当然，初始化过程中，除了 memory，driver 会为 probe 的 device 分配各种资源，例如 IRQ 号，io memory map、DMA 等等。当初始化需要管理这么多的资源分配和释放的时候，很多驱动程序都出现了资源管理的 issue。而且，由于这些 issue 是异常路径上的 issue，不是那么容易测试出来，更加重了解决这个 issue 的必要性。内核解决这个问题的模式（所谓解决一类问题的设计方法就叫做设计模式）是 Devres，即 device resource management 软件模块。更细节的内容就不介绍了，其核心思想就是资源是设备的资源，那么资源的管理归于 device，也就是说不需要 driver 过多的参与。当 device 和 driver detach 的时候，device 会自动的释放其所有的资源。

```
*/
        info->pin_regs = devm_kzalloc(&pdev->dev, sizeof(*info->pin_regs) *
                                      info->npins, GFP_KERNEL);
        if (!info->pin_regs)
                return -ENOMEM;


        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);// 分配 memory 资源
        ipctl->base = devm_ioremap_resource(&pdev->dev, res);
        if (IS_ERR(ipctl->base))
                return PTR_ERR(ipctl->base);
/*
```

分配了 struct imx_pinctrl 数据结构的内存，当然下一步就是初始化这个数据结构了。我们先看看 imx 的 pin controller driver 是如何定义该数据结构的

```
struct imx_pinctrl {
        struct device *dev;// 和 platform device 建立联系
        struct pinctrl_dev *pctl;// 向 core driver 的 pin controller class device
        void __iomem *base;// 访问硬件寄存器的基地址
        const struct imx_pinctrl_soc_info *info;
};
struct imx_pinctrl_soc_info {
```

```
        struct device *dev;
        const struct pinctrl_pin_desc *pins;// 指向 pin control subsystem 中 core driver 中抽象的
        unsigned int npins;
        struct imx_pin_reg *pin_regs;
        struct imx_pin_group *groups;// 描述 imx pin controller 中 pin groups 的信息
        unsigned int ngroups;// 描述 imx pin controller 中 pin groups 的数目
        struct imx_pmx_func *functions;// 描述 imx pin controller 中 function 信息
        unsigned int nfunctions;// 描述 imx pin controller 中 function 的数目
        unsigned int flags;
    };
```

struct pinctrl_desc 和 struct pinctrl_dev 都是 pin control subsystem 中 core driver 的概念。各个具体硬件的 pin controller 可能会各不相同，但是可以抽取其共同的部分来形成一个 HW independent 的数据结构，这个数据就是 pin controller 描述符，在 core driver 中用 struct pinctrl_desc 表示，具体该数据结构的定义如下：

```
    struct pinctrl_desc {
    const char *name;
    struct pinctrl_pin_desc const *pins;－－－指向 npins 个 pin 描述符，每个描述符描述一个 pin
    unsigned int npins;－－－－－－－－－－－－－－－该 pin controller 中有多少个可控的 pin
    const struct pinctrl_ops *pctlops;－－－－－－－callback 函数，是 core driver 和底层 driver 的接口
    const struct pinmux_ops *pmxops;－－－－－－callback 函数，是 core driver 和底层 driver 的接口
    const struct pinconf_ops *confops;－－－－－－callback 函数，是 core driver 和底层 driver 的接口
    struct module *owner;
};
```

其实整个初始化过程的核心思想就是 low level 的 driver 定义一个 pinctrl_desc ，设定 pin 相关的定义和 callback 函数，注册到 pin control subsystem 中。我们用引脚描述符（pin descriptor）来描述一个 pin。在 pin control subsystem 中，struct pinctrl_pin_desc 用来描述一个可以控制的引脚，我们称之引脚描述符，代码如下：

```
    struct pinctrl_pin_desc {
    unsigned number;－－－－－－－－ID，在本 pin controller 中唯一标识该引脚
    const char *name;－－－－－－－－user friedly name
    void *drv_data;
};


    */
        imx_pinctrl_desc.name = dev_name(&pdev->dev);
        imx_pinctrl_desc.pins = info->pins;
        imx_pinctrl_desc.npins = info->npins;


        ret = imx_pinctrl_probe_dt(pdev, info);
```

```
/*
|-> imx_pinctrl_parse_functions
|   |-> imx_pinctrl_parse_groups
|   |->
/*
         * the binding format is fsl,pins = <PIN_FUNC_ID CONFIG ...>,
         * do sanity check and calculate pins number
         */
        list = of_get_property(np, "fsl,pins", &size);
以下代码具体设置 mux/conf/input 寄存器
        for (i = 0; i < grp->npins; i++) {
                u32 mux_reg = be32_to_cpu(*list++);
                u32 conf_reg;
                unsigned int pin_id;
                struct imx_pin_reg *pin_reg;
                struct imx_pin *pin = &grp->pins[i];

                if (info->flags & SHARE_MUX_CONF_REG)
                        conf_reg = mux_reg;
                else
                        conf_reg = be32_to_cpu(*list++);

                pin_id = mux_reg ? mux_reg / 4 : conf_reg / 4;
                pin_reg = &info->pin_regs[pin_id];
                pin->pin = pin_id;
                grp->pin_ids[i] = pin_id;
                pin_reg->mux_reg = mux_reg;
                pin_reg->conf_reg = conf_reg;
                pin->input_reg = be32_to_cpu(*list++);
                pin->mux_mode = be32_to_cpu(*list++);
                pin->input_val = be32_to_cpu(*list++);

                /* SION bit is in mux register */
                config = be32_to_cpu(*list++);
                if (config & IMX_PAD_SION)
```

```
                        pin->mux_mode |= IOMUXC_CONFIG_SION;
                pin->config = config & ~IMX_PAD_SION;

                dev_dbg(info->dev, "%s: %d 0x%08lx", info->pins[i].name,
                                pin->mux_mode, pin->config);
        }
*/
        if (ret) {
                dev_err(&pdev->dev, "fail to probe dt properties\n");
                return ret;
        }

        ipctl->info = info;
        ipctl->dev = info->dev;
        platform_set_drvdata(pdev, ipctl);
        ipctl->pctl = pinctrl_register(&imx_pinctrl_desc, &pdev->dev, ipctl);
        if (!ipctl->pctl) {
                dev_err(&pdev->dev, "could not register IMX pinctrl driver\n");
                return -EINVAL;
        }

        dev_info(&pdev->dev, "initialized IMX pinctrl driver\n");

        return 0;
}
```

## 1.3 新板 bringup

移植 BSP 时，在修改完 Iomux 与 GPIO 之后，就需要先了解自己的板子与恩智浦 i.MX8QXPMEK 板的区别，如果只是修改或减少驱动，只需要把不需要的驱动在 dts 中删除掉或是设置为 disabled，然后需要修改的驱动，修改其驱动资源参数中的值，比如说 GPIO，I2C 地址等内容就可以了。

比如如果是使用 UUU 下载，只保留调试串口，eMMC/SD 驱动与 USB 相关驱动就可以了，其它的涉及的外设部分的都可以暂时先移掉，来保证初始化通过：

//arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
brcmfmac: brcmfmac { //disable wifi/bt/modem
```

```
status = "disabled";};

lvds_backlight0/1{status = "disabled";};

  modem_reset: modem-reset { status = "disabled";};

cbtl04gp {status = "disabled";};//如果不是使用 typec 下载，disable

regulators { //remove the gpio regulators, except sdcard and usbotg1
…
            reg_can_en: regulator-can-gen { status = "disabled";};
            reg_can_stby: regulator-can-stby {status = "disabled";};
            reg_fec2_supply: fec2_nvcc {status = "disabled";};
            reg_usdhc2_vmmc: usdhc2_vmmc {…};
            epdev_on: fixedregulator@100 {status = "disabled";};
            reg_usb_otg1_vbus: regulator@0 {…};
            reg_audio: fixedregulator@2 {status = "disabled";};
  };

sound-cs42888/wm8960 {//remove all sound device
status = "disabled"
}

&cm40_i2c /cm40_intmux{ status = "disabled"}
sound-amix-sai { status = "disabled"}

&dc0_pc/prg1~9/dpr1_channel1~3//dpr2_channel1~3 {status = "disabled";};

&pwm_mipi_lvds0 /1{        status = " disabled ";};

&i2c0_mipi_lvds0 /1{status = "disabled";};

&ldb1/2_phy/&ldb1/2{status = "disabled";};
```

```
&mipi0/1_dphy/&mipi0/1_dsi_host{status = "disabled";};

//uart 保留 debug uart0 和底板上的 uart2,其它的去掉
&lpuart1/3 { status = "disabled"}
&lpuart3 { status = "disabled"}

//can 去掉
&flexcan1/2 { status = "disabled"}

//audio 相关去掉
&amix/&asrc0/&dsp/&esai0/&sai1/4/5{ status = "disabled"}

//网口去掉
&fec1/2{ status = "disabled"}

//spi 去掉
&flexspi0 { status = "disabled"}

//i2c 去掉
&i2c1 { status = "disabled"}

//pcie 去掉
&pcieb{ status = "disabled"}

&rpmsg{ status = "disabled"}

//vpu/gpu 为内部模块，可以去掉，也可以不去
&vpu_decoder/encoder /&jpegdec/&jpegenc{ status = "disabled"}

//gpu 为内部模块，可以去掉，也可以不去, 注意对与无屏应用，如 V2X，如果 GPU 电源没有连接，一定要去掉
&gpu_3d0 /&imx8_gpu_ss{ status = "disabled"}

//camera 去掉,相关 I2C 控制口也去掉
&isi_0~4/&irqsteer_csi0/&mipi_csi_0/&cameradev/&parallel_csi/&i2c_mipi_csi0/{ status = "disabled"}
```

//thermal 为内部模块,可以不去掉

&thermal_zones {…}

如果使用 sdcard 启动，则把编译出来的最小系统的 fsl-imx8qxp-mek.dtb 替换掉 Boot imx8qx 分区中的 fsl-imx8qxp-mek.dtb,插上 sdcard，即可以启动最小系统。

如果没有设计 sdcard，直接使用 UUU 下载运行结果如下：

1. download i.mx8qxp mek image from nxpwebsite  Unzip it。
2. download uuu 1.2.130 from

https://github.com/NXPmicro/mfgtools/releases

put the uuu.exe in the same folder with demo image.

3. Copy the demo image\samples\example_kernel_emmc.uuu to beyond folder rename to example_kernel_emmc_John.uuu, same with demo image, and modify it as follow(just Change _flash.bin, _Image, _board.dtb, _initramfs.cpio.gz.uboot, emmc boot device number, remove optee, rootfs),to the right one in the demo image.

example_kernel_emmc_John.uuu:

```
uuu_version 1.0.1
# Please Replace below items with actually file names
# @_flash.bin                    | boot loader
# @_Image                        | kernel image, arm64 is Image, arm32 it is zImage
# @_board.dtb                    | board dtb file
# @_initramfs.cpio.gz.uboot      | mfgtool init ramfs
# @_rootfs.tar.bz2               | rootfs
# @_uTee.tar                     | optee image

SDP: boot -f imx-boot-imx8qxpmek-sd.bin-flash
# This command will be run when use SPL
SDPU: write -f imx-boot-imx8qxpmek-sd.bin-flash -offset 0x57c00
SDPU: jump
# This command will be run when ROM support stream mode
SDPS: boot -f imx-boot-imx8qxpmek-sd.bin-flash

# use uboot burn bootloader to eMMC
# becaue difference chip, offset is difference
```

```
# you can use kernel to do that for specific boards
FB: ucmd setenv fastboot_dev mmc
FB: ucmd setenv mmcdev ${emmc_dev}
FB: flash bootloader imx-boot-imx8qxpmek-sd.bin-flash
FB: ucmd setenv emmc_cmd mmc partconf ${emmc_dev} 0 1 0;
FB: ucmd if test "${emmc_skip_fb}" != "yes"; then run emmc_cmd; fi
FB: ucmd setenv emmc_cmd mmc bootbus ${emmc_dev} 2 2 1;
FB: ucmd if test "${emmc_skip_fb}" != "yes"; then run emmc_cmd; fi

FB: ucmd setenv fastboot_buffer ${loadaddr}
FB: download -f Image-imx8qxpmek.bin
FB: ucmd setenv fastboot_buffer ${fdt_addr}
FB: download -f Image-fsl-imx8qxp-mek.dtb
FB: ucmd setenv fastboot_buffer ${initrd_addr}
FB: download -f fsl-image-mfgtool-initramfs-imx_mfgtools.cpio.gz.u-boot
#FB: ucmd setenv bootargs console=${console},${baudrate} earlycon=${earlycon},${baudrate}
FB: acmd ${kboot} ${loadaddr} ${initrd_addr} ${fdt_addr}

# get mmc dev number from kernel command line
# Wait for emmc
FBK: ucmd while [ ! -e /dev/mmcblk0boot0 ]; do sleep 1; echo "wait for /dev/mmcblk*boot* appear"; done;

# serach emmc device number, if your platform have more than two emmc chip, please echo dev number >/tmp/mmcdev
FBK: ucmd dev=`ls /dev/mmcblk0boot0`; dev=($dev); dev=${dev[0]}; dev=${dev#/dev/mmcblk}; dev=${dev%boot0}; echo
$dev > /tmp/mmcdev;

# create partition
FBK: ucmd mmc=`cat /tmp/mmcdev`; PARTSTR=$'10M,500M,0c\n600M,,83\n'; echo "$PARTSTR" | sfdisk --force
/dev/mmcblk${mmc}

FBK: ucmd mmc=`cat /tmp/mmcdev`; dd if=/dev/zero of=/dev/mmcblk${mmc} bs=1k seek=4096 count=1
FBK: ucmd sync
# you can enable below command to write boot partition. but offset is difference at difference platform
#FBK: ucmd mmc=`cat /tmp/mmcdev`; echo 0 > /sys/block/mmcblk${mmc}boot0/force_ro
#FBK: ucp  _flash.bin t:/tmp
#FBK: ucmd mmc=`cat /tmp/mmcdev`; dd if=/tmp/_flash.bin of=/dev/mmc${mmc}boot0 bs=1K seek=32
```

```
#FBK: ucmd mmc=`cat /tmp/mmcdev`; echo 1 > /sys/block/mmcblk${mmc}boot0/force_ro
FBK: ucmd mmc=`cat /tmp/mmcdev`; while [ ! -e /dev/mmcblk${mmc}p1 ]; do sleep 1; done
FBK: ucmd mmc=`cat /tmp/mmcdev`; mkfs.vfat /dev/mmcblk${mmc}p1
FBK: ucmd mmc=`cat /tmp/mmcdev`; mkdir -p /mnt/fat
FBK: ucmd mmc=`cat /tmp/mmcdev`; mount -t vfat /dev/mmcblk${mmc}p1 /mnt/fat
FBK: ucp  Image-imx8qxpmek.bin t:/mnt/fat
FBK: ucp  Image-fsl-imx8qxp-mek.dtb t:/mnt/fat
#FBK: ucp  _uTee.tar t:/tmp/op.tar
#FBK: ucmd tar -xf /tmp/op.tar -C /mnt/fat
FBK: ucmd umount /mnt/fat
FBK: ucmd mmc=`cat /tmp/mmcdev`; mkfs.ext3 -F -E nodiscard /dev/mmcblk${mmc}p2
FBK: ucmd mkdir -p /mnt/ext3
FBK: ucmd mmc=`cat /tmp/mmcdev`; mount /dev/mmcblk${mmc}p2 /mnt/ext3
FBK: acmd export EXTRACT_UNSAFE_SYMLINKS=1; tar -jx -C /mnt/ext3
FBK: ucp  fsl-image-validation-imx-imx8qxpmek.tar.bz2 t:-
FBK: Sync
FBK: ucmd umount /mnt/ext3
FBK: DONE
```

4. 将编译出来的 fsl-imx8qxp-mek.dtb 文件替换掉 demo image 中的 Image-fsl-imx8qxp-mek.dtb 文件。
5. jump the i.mx8qxp mek board to download mode. 0001.
6. link the usb type c to pc, link the usb serial port to pc.
7. run command in command window: uuu.exe example_kernel_emmc_John.uuu
8. usb port information as follows:

```
C:\D\imx\imx8x\nxpwebsite\5.4.24\L5.4.24_2.0.0_images_MX8QXPMEK>uuu.exe
example_kernel_emmc_John.uuu

uuu (Universal Update Utility) for nxp imx chips -- libuuu_1.2.135-0-gacaf035

Success 1    Failure 0

1:18    20/20 [Done                           ] FBK: DONE

1:9      1/ 1 [==================100%==================] SDPS: boot -f
imx-boot-imx8qxpmek-sd.bin-flash

C:\D\imx\imx8x\nxpwebsite\5.4.24\L5.4.24_2.0_images_MX8QXPMEK>
```

9. serial port information as follows:

```
Detect USB boot. Will enter fastboot mode!

Fastboot: Normal
```

```
Boot from USB for mfgtools
Use default environment for                    mfgtools
Run bootcmd_mfg
…
# Checking Image at 83100000 ...
Unknown image format!
Run fastboot ...
1 setufp mode 0
1 cdns3_uboot_initmode 0
Detect USB boot. Will enter fastboot mode!
flash target is MMC:1
MMC: no card present
MMC card init failed!
MMC: no card present
** Block device MMC 1 not supported
Detect USB boot. Will enter fastboot mode!
flash target is MMC:0
status: -104 ep 'ep1in' trans: 0
Starting download of 932864 bytes
.......
downloading of 932864 bytes finished
writing to partition 'bootloader'
support sparse flash partition for bootloader
Initializing 'bootloader'
switch to partitions #1, OK
mmc0(part 1) is current device
Writing 'bootloader'

MMC write: dev # 0, block # 64, count 1882 ... 1882 blocks written: OK
Writing 'bootloader' DONE!
status: -104 ep 'ep1in' trans: 0
Detect USB boot. Will enter fastboot mode!
status: -104 ep 'ep1in' trans: 0
Detect USB boot. Will enter fastboot mode!
Detect USB boot. Will enter fastboot mode!
Detect USB boot. Will enter fastboot mode!
```

```
Detect USB boot. Will enter fastboot mode!
Starting download of 23163392 bytes
.....
downloading of 23163392 bytes finished
status: -104 ep 'ep1in' trans: 0
Detect USB boot. Will enter fastboot mode!
Starting download of 84164 bytes

downloading of 84164 bytes finished
Detect USB boot. Will enter fastboot mode!
Starting download of 10798655 bytes
status: -104 ep 'ep1in' trans: 0
.....................................................................
........
downloading of 10798655 bytes finished…
[    5.257452] Freeing unused kernel memory: 1280K
Found New UDC: ci_hdrc.0
ci_hdrc.0 0
Found New UDC: gadget-cdns3
gadget-cdns3 1
ffs.utp0
[    5.319329] file system registered
ffs.utp1
[    5.338016] Mass Storage Function, version: 2009/09/11
[    5.343262] LUN: removable file: (no medium)
[    5.347577] Mass Storage Function, version: 2009/09/11
run utp at /dev/usb-utp0/ep0[    5.352844] LUN: removable file: (no medium)

.
uuu fastboot client 1.0.0 [built Dec  4 2018 12:07:26]
Start[    5.359527] read descriptors
init usb
[    5.368194] read descriptors
run utp at /dev/usb-utp1/ep0
uuu fastboot client 1.0.0 [built De[    5.368220] read strings
c  4 2018 12:07:26]
```

```
Start init usb
write string
Start handle c[    5.380206] read strings
ommand
uuc /dev/utp1
write string
uuc 0.5 [built Dec  4 2018 12:07:26]
Start handle command
UTP: Waiting for /dev/utp1 to appear
[    5.503521] configfs-gadget gadget: super-speed config #1: c
[    5.535063] random: fast init done
run shell cmd: while [ ! -e /dev/mmcblk0boot0 ]; do sleep 1; echo "wait for /dev/mmcblk*boot* appear
"; done;
run shell cmd: dev=`ls /dev/mmcblk0boot0`; dev=($dev); dev=${dev[0]}; dev=${dev#/dev/mmcblk}; dev=${
dev%boot0}; echo $dev > /tmp/mmcdev;
run shell cmd: mmc=`cat /tmp/mmcdev`; PARTSTR=$'10M,500M,0c\n600M,,83\n'; echo "$PARTSTR" | sfdisk -
-force /dev/mmcblk${mmc}
[    6.015705]  mmcblk0: p1 p2
uuc /dev/utp
uuc 0.5 [built Dec  4 2018 12:07:26]
UTP: Waiting for /dev/utp to appear
Partition #1 contains a vfat signature.
Partition #2 contains a ext3 signature.
[    7.295661]  mmcblk0: p1 p2
run shell cmd: mmc=`cat /tmp/mmcdev`; dd if=/dev/zero of=/dev/mmcblk${mmc} bs=1k seek=4096 count=1
1+0 records in
1+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000997625 s, 1.0 MB/s
run shell cmd: sync
run shell cmd: mmc=`cat /tmp/mmcdev`; while [ ! -e /dev/mmcblk${mmc}p1 ]; do sleep 1; done
run shell cmd: mmc=`cat /tmp/mmcdev`; mkfs.vfat /dev/mmcblk${mmc}p1
run shell cmd: mmc=`cat /tmp/mmcdev`; mkdir -p /mnt/fat
run shell cmd: mmc=`cat /tmp/mmcdev`; mount -t vfat /dev/mmcblk${mmc}p1 /mnt/fat
WOpen:/mnt/fat
WOpen:/mnt/fat/Image-imx8qxpmek.bin
WOpen:/mnt/fat
```

WOpen:/mnt/fat/Image-fsl-imx8qxp-mek.dtb

run shell cmd: umount /mnt/fat

run shell cmd: mmc=`cat /tmp/mmcdev`; mkfs.ext3 -F -E nodiscard /dev/mmcblk${mmc}p2

mke2fs 1.43.8 (1-Jan-2018)

[ 12.222155] random: crng init done

run shell cmd: mkdir -p /mnt/ext3

run shell cmd: mmc=`cat /tmp/mmcdev`; mount /dev/mmcblk${mmc}p2 /mnt/ext3

[ 22.243037] EXT4-fs (mmcblk0p2): mounting ext3 file system using the ext4 subsystem

[ 22.255624] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)

run shell cmd: export EXTRACT_UNSAFE_SYMLINKS=1; tar -jx -C /mnt/ext3

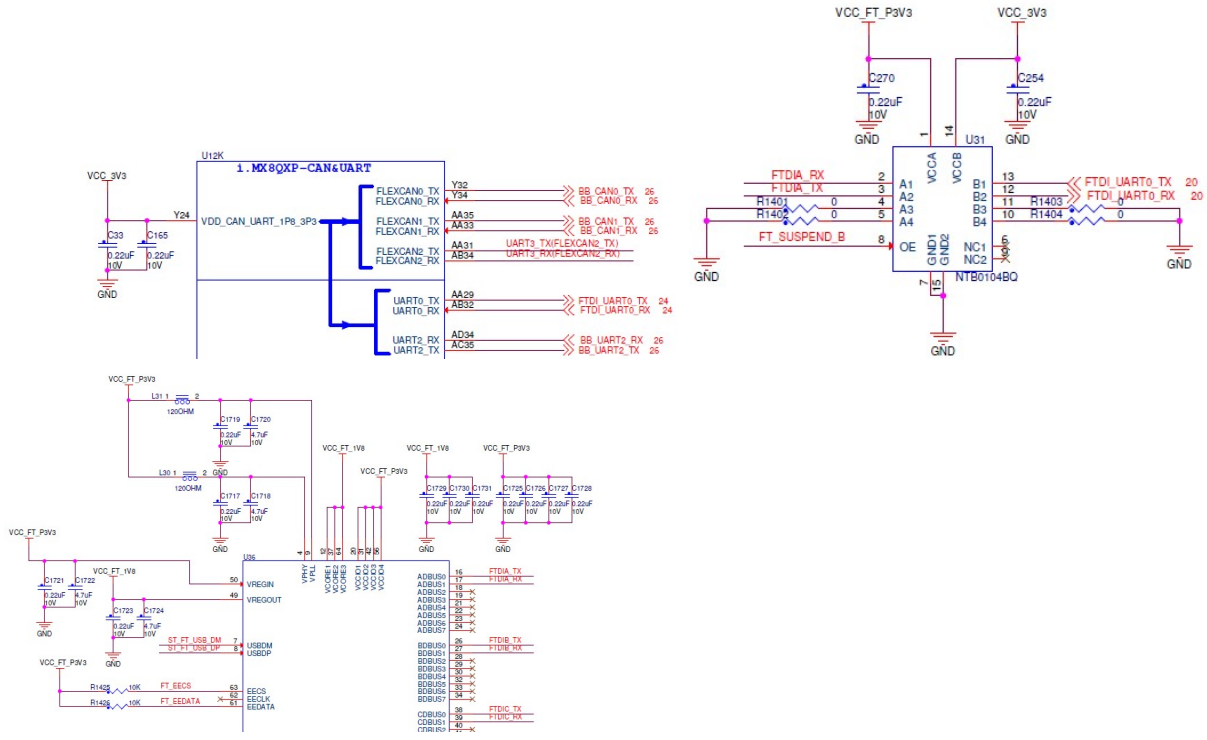
WOpen:-

wait for async proccess finish

run shell cmd: umount /mnt/ext3

## 1.4 更改调试串口

i.MX8QXP MEK 板默认的调试串口设计如下：从 UART0_TX/RX 3.3V 管脚，经过一个带开关功能的电平转换器(没有做电平转换)，然后连接到 UART to USB 桥上。



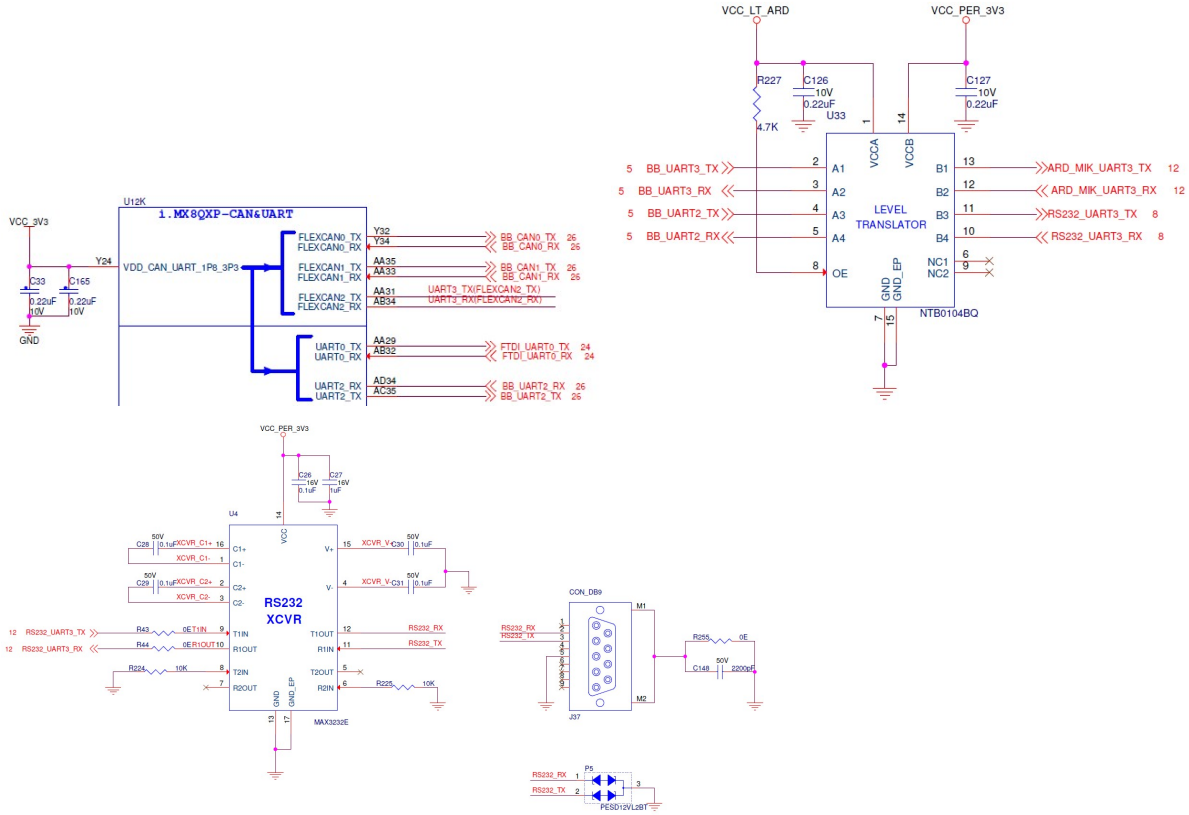
相关软件 DTB 设置为：

```
        //arch\arm64\boot\dts\freescale\imx8x-mek.dtsi
chosen {
                stdout-path = &lpuart0;
        };
…
        &lpuart0 {
          pinctrl-names = "default";
          pinctrl-0 = <&pinctrl_lpuart0>;
          status = "okay";
        };
          pinctrl_lpuart0: lpuart0grp {
                fsl,pins = <
                        IMX8QXP_UART0_RX_ADMA_UART0_RX                  0x06000020
                        IMX8QXP_UART0_TX_ADMA_UART0_TX                  0x06000020
                >;
          };

        //arch\arm64\boot\dts\freescale\imx8-ss-dma.dtsi
          lpuart0: serial@5a060000 {
                reg = <0x5a060000 0x1000>;
                interrupts = <GIC_SPI 345 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-parent = <&gic>;
                clocks = <&uart0_lpcg 1>, <&uart0_lpcg 0>;
                clock-names = "ipg", "baud";
                assigned-clocks = <&clk IMX_SC_R_UART_0 IMX_IMX8QXPM_CLK_PER>;
                assigned-clock-rates = <80000000>;
                power-domains = <&pd IMX_SC_R_UART_0>;
                status = "disabled";
          };
…
          uart0_lpcg: clock-controller@5a460000 {
                compatible = "fsl,imx8qxp-lpcg";
                reg = <0x5a460000 0x10000>;
                #clock-cells = <1>;
```

```
                clocks = <&clk IMX_SC_R_UART_0 IMX_IMX8QXPM_CLK_PER>,
                         <&dma_ipg_clk>;
                bit-offset = <0 16>;
                clock-output-names = "uart0_lpcg_baud_clk",
                                     "uart0_lpcg_ipg_clk";
                power-domains = <&pd IMX_SC_R_UART_0>;
        };
…
        //arch\arm64\boot\dts\freescale\imx8qxp-ss-adma.dtsi
&lpuart0 {
        compatible = "fsl,imx8qxp-lpuart", "fsl,imx7ulp-lpuart";
};
```

如果我们的设计不是使用这个串口做为调试串口，比如以下，我们以 i.MX8QXP MEK 板的底板上的 J37 为调试串口的，他的硬件设计为：




则此串口为 UART2，软件按照以下方法设置 UART2 为调试串口：

```
        //arch\arm64\boot\dts\freescale\ imx8x-mek.dtsi
```

```
chosen {
                stdout-path = &lpuart2;
        };
…
                pinctrl_lpuart2: lpuart2grp {
                        fsl,pins = <
                                IMX8QXP _UART2_TX_ADMA_UART2_TX        0x06000020
                                IMX8QXP _UART2_RX_ADMA_UART2_RX        0x06000020
                        >;
                };
&pd_dma_lpuart2 {
        debug_console;
};
&lpuart2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpuart2>;
        status = "okay";
};
        //arch\arm64\boot\dts\freescale\imx8-ss-dma.dtsi
          lpuart2: serial@5a080000 {
                reg = <0x5a080000 0x1000>;
                interrupts = <GIC_SPI 347 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-parent = <&gic>;
                clocks = <&uart2_lpcg 1>, <&uart2_lpcg 0>;
                clock-names = "ipg", "baud";
                assigned-clocks = <&clk IMX_SC_R_UART_2 IMX_IMX8QXPM_CLK_PER>;
                assigned-clock-rates = <80000000>;
                power-domains = <&pd IMX_SC_R_UART_2>;
/*调试串口去掉 PM*/
        /*
                power-domain-names = "uart";
        */
        /*调试串口去掉 DMA 功能*/
        /*
                dma-names = "tx","rx";
```

```
                dmas = <&edma2 13 0 0>,
                        <&edma2 12 0 1>;
        */
                status = "disabled";
          };
```

如下将 UART0 改为普通串口：增加 DMA 功能：

```
        //arch\arm64\boot\dts\freescale\imx8-ss-dma.dtsi
          lpuart0: serial@5a060000 {
                reg = <0x5a060000 0x1000>;
                interrupts = <GIC_SPI 345 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-parent = <&gic>;
                clocks = <&uart0_lpcg 1>, <&uart0_lpcg 0>;
                clock-names = "ipg", "baud";
                assigned-clocks = <&clk IMX_SC_R_UART_0 IMX_IMX8QXPM_CLK_PER>;
                assigned-clock-rates = <80000000>;
                power-domains = <&pd IMX_SC_R_UART_0>;
        //增加 power domain
                        power-domain-names = "uart";
        //增加 dma 支持
                dma-names = "tx","rx";
                dmas = <&edma2 9 0 0>,
                        <&edma2 8 0 1>;
        //end
                status = "disabled";
          };
//edma2 channel map 参考芯片手册如下：
```

Table 16-4. eDMA2 Channel Map

| Channel Number | Module | DMA Request Description |
|---|---|---|
| 0 | LPSPI0 | LPSPI0 receive request |
| 1 | LPSPI0 | LPSPI0 transmit request |
| 2 | LPSPI1 | LPSPI1 receive request |
| 3 | LPSPI1 | LPSPI1 transmit request |
| 4 | LPSPI2 | LPSPI2 receive request |
| 5 | LPSPI2 | LPSPI2 transmit request |
| 6 | LPSPI3 | LPSPI3 receive request |
| 7 | LPSPI3 | LPSPI3 transmit request |
| 8 | LPUART0 | LPUART0 receive request |
| 9 | LPUART0 | LPUART0 transmit request |
| 10 | LPUART1 | LPUART1 receive request |
| 11 | LPUART1 | LPUART1 transmit request |
| 12 | LPUART2 | LPUART2 receive request |
| 13 | LPUART2 | LPUART2 transmit request |

然后启动后停下 uboot,将 uboot 参数变量修改为：(一般 uboot 已经修改)

```
setenv console 'ttyLP2'
setenv earlycon 'lpuart32,0x5a080000'
sav
pri
reset
```

测试结果如下：

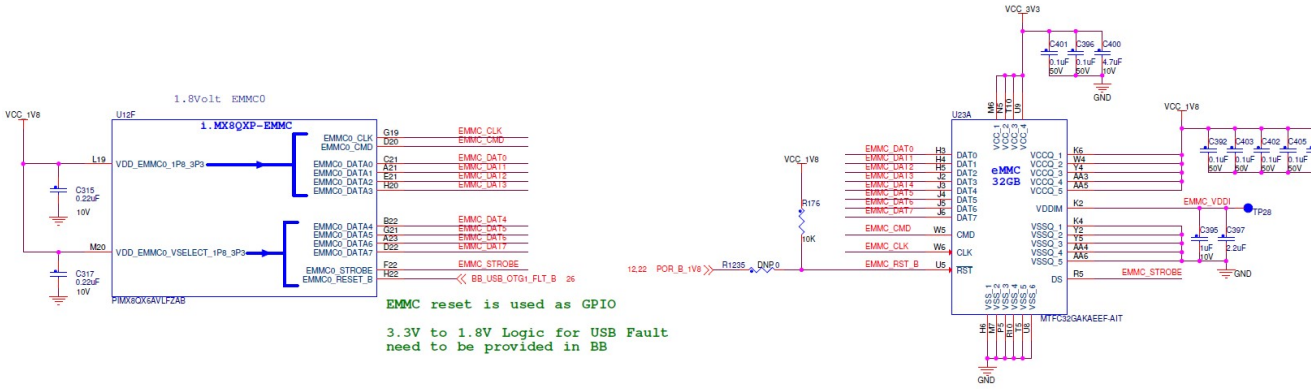
```
cat /proc/cmdline
console=ttyLP2,115200 earlycon=lpuart32,0x5a080000,115200 root=/dev/mmcblk1p2 rootwait rw
```

可以操作 ttyLP0 如下：

```
echo aaaaa > /dev/ttyLP0
```

## 1.5 uSDHC 设备定制(eMMC flash,SDcard, SDIOcard)

i.MX8QXP MEK 板连接的 emmc 是：




使用 1.8V IO 电压。

i.MX8QXP host controller 的功能有：

- Compatible with the MMC System Specification version 4.2/4.3/4.4/4.41/5.0/5.1
- Supports 1-bit/4-bit SD and SDIO modes, and 1-bit/4-bit/8-bit MMC modes

Up to 3200 Mbps of data transfer for MMC cards using eight parallel data lines

- in the Dual Data Rate (DDR) mode
- MMC HS400 mode (200 MHz both edges)
- VDD_EMMC0_1P8_3P3/ VDD_EMMC0_VSELECT_1P8_3P3

软件配置为：

Documentation\devicetree\binding\mmc\fsl-imx-esdhc.txt

```
Compatible :
  "fsl,imx8qxp-usdhc" "fsl,imx8mm-usdhc"
```

Arch\arm64\boot\dts\freescale\imx8qxp-ss-conn.dtsi

```
&usdhc1/2 {
        compatible = "fsl,imx8qxp-usdhc", "fsl,imx7d-usdhc";
};
```

Drivers/mmc/host/sdhci-esdhc-imx.c

```
static const struct of_device_id imx_esdhc_dt_ids[] = {
…
        { .compatible = "fsl,imx8qxp-usdhc", .data = &usdhc_imx8qxp_data, },
        { .compatible = "fsl,imx8mm-usdhc", .data = &usdhc_imx8mm_data, },
…}

static struct esdhc_soc_data usdhc_imx8qxp_data = {
```

```
/*
 * The flag tells that the ESDHC controller is an USDHC block that is
 * integrated on the i.MX6 series.
        .flags = ESDHC_FLAG_USDHC |
/* The IP supports standard tuning process */
                ESDHC_FLAG_STD_TUNING|
                /* The IP has SDHCI_CAPABILITIES_1 register */
                ESDHC_FLAG_HAVE_CAP1 |
                /* The IP supports HS200 mode */
                ESDHC_FLAG_HS200|
                /* The IP supports HS400 mode */
                ESDHC_FLAG_HS400 |
                /* The IP supports HS400ES mode */
                ESDHC_FLAG_HS400_ES
                /* The IP has Host Controller Interface for Command Queuing */
                        | ESDHC_FLAG_CQHCI
                /* The IP state got lost in low power mode */
                        | ESDHC_FLAG_STATE_LOST_IN_LPMODE
                /* The IP lost clock rate in PM_RUNTIME */
                        | ESDHC_FLAG_CLK_RATE_LOST_IN_PM_RUNTIME,
};
```

连接的emmc功能如下：(以下请参考emmc datasheet与JESD84-B51: Embedded Multi-Media Card (*e*•MMC) Electrical Standard (5.1))

- JEDEC/MMC standard version 5.0-compliant
- VCCQ (dual voltage): 1.65–1.95V; 2.7–3.6V

| | | |
|---|---|---|
| [23:15] | 1 1111 1111b | 2.7–3.6V voltage range |
| [14:8] | 000 0000b | 2.0–2.7V voltage range |
| [7] | 1b | 1.70–1.95V voltage range |

- HS200/HS400 mode with 1.8V IO

#### 7.4.59 DEVICE_TYPE [196]

This field defines the type of the Device.

**Table 137 — Device types**

| Bit | Device Type |
|---|---|
| 7 | HS400 Dual Data Rate *e*•MMC at 200 MHz – 1.2 V I/O |
| 6 | HS400 Dual Data Rate *e*•MMC at 200 MHz – 1.8 V I/O |
| 5 | HS200 Single Data Rate *e*•MMC at 200 MHz - 1.2 V I/O |
| 4 | HS200 Single Data Rate *e*•MMC at 200 MHz - 1.8 V I/O |
| 3 | High-Speed Dual Data Rate *e*•MMC at 52 MHz - 1.2 V I/O |
| 2 | High-Speed Dual Data Rate *e*•MMC at 52 MHz - 1.8 V or 3 V I/O |
| 1 | High-Speed *e*•MMC at 52 MHz - at rated device voltage(s) |
| 0 | High-Speed *e*•MMC at 26 MHz - at rated device voltage(s) |

| | | | | | | |
|---|---|---|---|---|---|---|
| Device type | DEVICE_TYPE | – | 1 | R | [196] | 57h |

| 5 | | | | 7 | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **0** | **1** | **0** | **1** | **0** | **1** | **1** | **1** |

- Data strobe pin（HS400 need it but do not support HS400ES mode）

#### 7.4.66 STROBE_SUPPORT [184]

This register indicates whether a device supports Enhanced Strobe mode for operation modes that STROBE is used (ie HS400).

Value "0x0" indicates No support of Enhanced Strobe mode
Value "0x1" indicates device supports Enhanced Strobe mode

#### 7.4.67 BUS_WIDTH [183]

It is set to '0' (1 bit data bus) after power up and can be changed by a SWITCH command.
Bus Width, Normal or DDR mode and Strobe mode (for HS400) are defined through BUS_WIDTH register.

**Table 144 — BUS_WIDTH**

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Enhanced Strobe | Reserved | Reserved | Reserved | Bus Mode Selection | | | |

Bit 7:

0x0: Strobe is provided only during Data Out and CRC response [Default]
0x1: Strobe is provided during Data Out, CRC response and CMD Response

| | | | | | | |
|---|---|---|---|---|---|---|
| Reserved | – | – | 1 | TBD | [184] | – |

| | | | | | | |
|---|---|---|---|---|---|---|
| Bus width mode | BUS_WIDTH | – | 1 | W/E_P | [183] | 00h |

- ECSD 寄存器无 cmdq 支持

| | | | | | |
|---|---|---|---|---|---|
| CMD Queuing Support | CMDQ_SUPPORT | 1 | R | [308] | |
| Reserved | – | – | 181 | TBD | [486:306] | – |

Bindings 文档:\Documentation\devicetree\bindings\mmc\mmc.txt 中相关信息：

1. bus-width: Number of data lines, can be <1>, <4>, or <8>
2. cd-gpios: Specify GPIOs for card detection, see gpio binding
3. wp-gpios: Specify GPIOs for write protection
4. no-1-8-v: when present, denotes that 1.8v card voltage is not supported on this system, even if the controller claims it is.

eMMC 的驱动除了 IOMUX 外，一般是不需要去定制的，eMMC 初始化的过程，就是发送 CMD，读取 eMMC 内部的 4 大寄存器：OCR,CID,CSD,ExtCSD 来获得 eMMC 的能力(支持电压，总线宽度，频率，时序等)，然后结合 host controller 的能力，再结合板级设计时 eMMC 供电电源的情况，来自动配置 host 电压和宽度，频率，时序等，就可以开始访问 eMMC 了。一般不需要手动设置，除了以下例外：

- 如果硬件是设计为3.3V了，那就算是host/emmc支持1.8V,也要在DTS中加上no-1-8-v,这样就变成了 ddr52 了: MMC_CAP2_DDR52_3_3V， hs200/hs400 disabled。
- 如果硬件没有连接 strobe pin,理论上应该协商为 hs200,如果仍然是 hs400,把 host 的 hs400 功能去掉。可以先从 eMMC 的 datasheet 中确认一下 ext_csd[196]:Device Type 寄存器，看看是否仅支持 hs200.另外如果内核可以启动，初始化信息中也会有相应消息：

  mmc0: new HS400 MMC card at address 0001

  host controller 如下去掉 HS400 支持：

```
Drivers/mmc/host/sdhci-esdhc-imx.c
static struct esdhc_soc_data usdhc_imx8qm_data = {
        …
        // ESDHC_FLAG_HS400 |
        // ESDHC_FLAG_HS400_ES
…
};
```

- 调试过程中如果需要降频测试，可以把 host controller 的 flag 去掉，比如说去 HS400,不行再去掉 HS200。
- 调试过程中如果怀疑信号线等长设计有问题，可以在 dts 中设置 bus-width = <4>;强制 host controller 使用 4bit 方式访问，减小信号线等长区别对时序的影响。

- Cmdq 5.4.24 BSP 是支持的，但是可以看到 i.MX8QXP MEK 板上的 eMMC 并不支持 cmdq,所以这个功能可以说没有充分验证过，如果客户自己选择的 eMMC 支持这个功能，使用中不稳定的话，可以把 host controller 的 flag 去掉。

```
Drivers/mmc/host/sdhci-esdhc-imx.c
static struct esdhc_soc_data usdhc_imx8qm_data = {
                …
                // | ESDHC_FLAG_CQHCI…
};
```

以下，为 i.MX8QXP MEK 板的 uSDHC 的 IOmux 设置，需要与自己板子硬件相匹配，注意一个 CD/WP 的 GPIO 可能要修改一下。

```
/arch/arm64/boot/dts/freescale/imx8x-mek.dtsi
    pinctrl_usdhc1: usdhc1grp {
            fsl,pins = <
                    IMX8QXP_EMMC0_CLK_CONN_EMMC0_CLK                0x06000041
                    IMX8QXP_EMMC0_CMD_CONN_EMMC0_CMD                0x00000021
…
                    IMX8QXP_EMMC0_STROBE_CONN_EMMC0_STROBE                  0x00000041
            >;
    };

    pinctrl_usdhc2_gpio: usdhc2gpiogrp {
            fsl,pins = <
                    IMX8QXP_USDHC1_RESET_B_LSIO_GPIO4_IO19    0x00000021
                    IMX8QXP_USDHC1_WP_LSIO_GPIO4_IO21       0x00000021
                    IMX8QXP_USDHC1_CD_B_LSIO_GPIO4_IO22       0x00000021
            >;
    };


    pinctrl_usdhc2: usdhc2grp {
            fsl,pins = <
                    IMX8QXP_USDHC1_CLK_CONN_USDHC1_CLK              0x06000041
                    IMX8QXP_USDHC1_CMD_CONN_USDHC1_CMD                  0x00000021
…
```

```
                >;
    }; //usdhc2 为 sdslot,定义如下:
&usdhc2 {
    pinctrl-names = "default", "state_100mhz", "state_200mhz";
    pinctrl-0 = <&pinctrl_usdhc2>, <&pinctrl_usdhc2_gpio>;
    pinctrl-1 = <&pinctrl_usdhc2>, <&pinctrl_usdhc2_gpio>;
    pinctrl-2 = <&pinctrl_usdhc2>, <&pinctrl_usdhc2_gpio>;
    bus-width = <4>;
    vmmc-supply = <&reg_usdhc2_vmmc>;//电源需要确认设置好了电压
    cd-gpios = <&lsio_gpio4 22 GPIO_ACTIVE_LOW>;//cd gpio 需要确认 iomux 已经设置为 gpio
    wp-gpios = <&lsio_gpio4 21 GPIO_ACTIVE_HIGH>;//wp gpio 需要确认 iomux 已经设置为 gpio
    status = "okay";
};
//usdhc1 为 emmc 定义如下：
&usdhc1 {
    pinctrl-names = "default", "state_100mhz", "state_200mhz";
    pinctrl-0 = <&pinctrl_usdhc1>;
    pinctrl-1 = <&pinctrl_usdhc1>;
    pinctrl-2 = <&pinctrl_usdhc1>;
    bus-width = <8>;
    no-sd;
    no-sdio;
    non-removable; //不可移除，所以没有 cd 的配置
    status = "okay";
};
```

进入内核后可以使用 mmc utility 工具检查，比如说如下命令：

```
mmc extcsd read /dev/mmcblk0
```

Mmc 驱动的详细内容如下：

### 1.5.1 Menuconfig

- CONFIG_MMC：增加对 MMC 总线协议支持：Device Drivers->MMC/SD/SDIO Card support
- CONFIG_MMC_BLOCK：增加对 MMC block 设备的支持，可用于支持文件系统挂接:Device Drivers->MMC/SD Card Support->MMC block device driver

- CONFIG_MMC_SDHCI:增加对 SDHC host controller 的支持: Device Drivers->MMC/SD Card Support-> Secure  Digital Host Controller Interface support
- CONFIG_MMC_SDHCI_PLTFM:增加对 SDHCI on the platform specific bus 的支持: Device Drivers->MMC/SD Card Support-> Secure  Digital Host Controller Interface support->SDHCI support on the platform specific bus
- CONFIG_MMC_ESDHC_IMX：增加对 i.MX USDHC 接口的支持：Device Drivers->MMC/SD Card Support->Secure Digital Host Controller Interface support->SDHCI support on the platform specific bus->SDHCI platform support for NXP eSDHC i.MX controller
- CONFIG_MMC_UNSAFE_RESUME:增加对使用 MMC/SD/SDIO 卡做根文件系统时的不可移除的支持：Device Drivers->MMC/SD/SDIO Card support->Assume MMC/SD cards care non-removable.

### 1.5.2 对应源代码

Linux-5.4.24/drivers/mmc

|->card \\对 mmc block 设备的支持驱动  存放闪存卡(块设备)的相关驱动，如 MMC/SD 卡设备驱动

|  |->Makefile:  obj-$(CONFIG_MMC_BLOCK)  += mmc_block.o

mmc_block-objs  := block.o queue.o

|  |->block.c\\对 mmc block 设备的支持驱动

|  |->queue.c

|->core\\mmc 协议支持

|  |->Makefile: obj-$(CONFIG_MMC)  += mmc_core.o

```
mmc_core-y          := core.o bus.o host.o \
                       mmc.o mmc_ops.o sd.o sd_ops.o \
                       sdio.o sdio_ops.o sdio_bus.o \
                       sdio_cis.o sdio_io.o sdio_irq.o \
                       quirks.o slot-gpio.o
```

|  |->core.c\\mmc 协议支持驱动整个 MMC 的核心层，这部分完成不同协议和规范的实现，为 host 层和设备驱动层提供接口函数

|  |->….

|->host\\针对不同主机端的 SDHC、MMC 控制器的驱动，这部分需要由驱动工程师来完成；

|  |-> Makefile:

```
obj-$(CONFIG_MMC_SDHCI)             += sdhci.o
obj-$(CONFIG_MMC_SDHCI_PLTFM)       += sdhci-pltfm.o
obj-$(CONFIG_MMC_SDHCI_ESDHC_IMX)   += sdhci-esdhc-imx.o
```

| |->sdhci.c:sdhci 标准 stack 代码

| |->sdhci-pltfm.c: sdhci 平台层

| |->sdhci-esdhc-imx.c/h :uSDHC 驱动层代码

### 1.5.3 MMC 驱动分层图

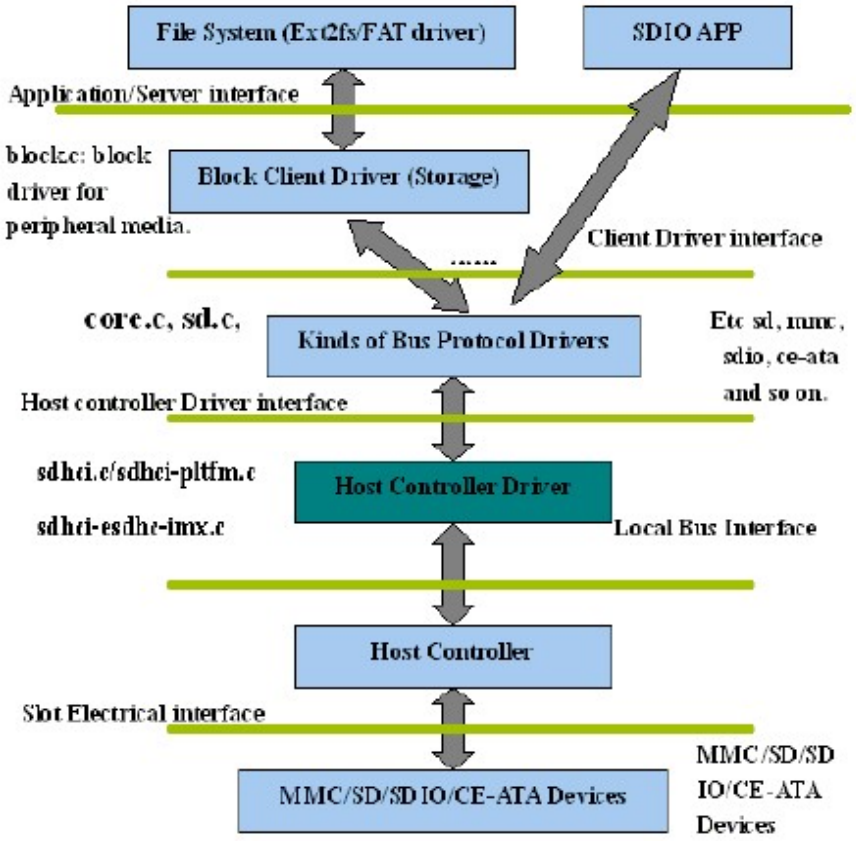



Figure 34-1. MMC Drivers Layering

### 1.5.4 MMC 初始化

MMC 驱动分为主设备驱动和从设备驱动，以下说明主机控制器端的初始化过程

系统在初始化时，在 device tree 扫描时，会将

```
usdhc1: mmc@5b010000 {
        interrupt-parent = <&gic>;
```

```
                interrupts = <GIC_SPI 232 IRQ_TYPE_LEVEL_HIGH>;
                reg = <0x5b010000 0x10000>;
                clocks = <&sdhc0_lpcg 1>,
                         <&sdhc0_lpcg 0>,
                         <&sdhc0_lpcg 2>;
                clock-names = "ipg", "per", "ahb";
                assigned-clocks = <&clk IMX_SC_R_SDHC_0 IMX_IMX8QXPM_CLK_PER>;
                assigned-clock-rates = <400000000>;
                power-domains = <&pd IMX_SC_R_SDHC_0>;
                fsl,tuning-start-tap = <20>;
                fsl,tuning-step= <2>;
                status = "disabled";
        };
        &usdhc1 {
        pinctrl-names = "default", "state_100mhz", "state_200mhz";
        pinctrl-0 = <&pinctrl_usdhc1>;
        pinctrl-1 = <&pinctrl_usdhc1>;
        pinctrl-2 = <&pinctrl_usdhc1>;
        bus-width = <8>;
        no-sd;
        no-sdio;
        non-removable; //不可移除，所以没有 cd 的配置
        status = "okay";
        };
```

注册进 platform 总线，此时仅注册了 platform_device。

注册 platform_driver 在\linux-5.4.24\drivers\mmc\host\sdhci-esdhc-imx.c

```
module_platform_driver(sdhci_esdhc_imx_driver);
static struct platform_driver sdhci_esdhc_imx_driver = {
        .driver                 = {
                .name   = "sdhci-esdhc-imx",
                .owner  = THIS_MODULE,
                .of_match_table = imx_esdhc_dt_ids,
                .pm     = &sdhci_esdhc_pmops,
        },
        .id_table= imx_esdhc_devtype,
```

```
        .probe          = sdhci_esdhc_imx_probe,
        .remove         = sdhci_esdhc_imx_remove,
};
sdhci_esdhc_imx_probe
|-> sdhci_pltfm_init
```

## 1.6 触摸屏驱动

i.MX8QXP 可以连接 rm67191 MiPi DSI 屏，这块屏在所有 i.MX8/8X/8M 系列中都可以使用，但是屏附带的触摸屏在 i.MX8QXP MEK 上没有实现驱动，可以从 i.MX8M 平台移植过来。

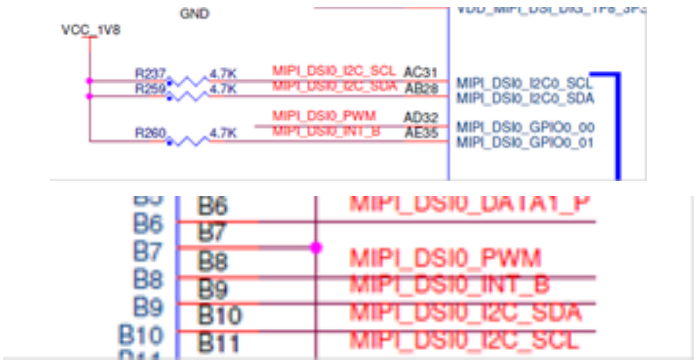
首先：确认触摸屏驱动已经编译到 i.MX8 系列的内核中：

```
  \linux-imx\arch\arm64\configs\ defconfig
  CONFIG_TOUCHSCREEN_SYNAPTICS_DSX_I2C=y
  drivers\input\touchscreen\ Makefile
  obj-$(CONFIG_TOUCHSCREEN_SYNAPTICS_DSX)  += synaptics_dsx/
  synaptics_dsx\ Synaptics_dsx_i2c.c
```

所以只要在 DTS 中使能触摸屏就可以加载了，参考 i.MX8MM 的驱动 DTS 示例：

```
Arch\arm64\boot\dts\freecale\fsl-imx8mm-evk-rm67191.dts:
        pinctrl_i2c2_synaptics_dsx_io: synaptics_dsx_iogrp {
            fsl,pins = <
                MX8MM_IOMUXC_GPIO1_IO09_GPIO1_IO9       0x19    /* Touch int */
            >;
        };
&i2c2 {
    synaptics_dsx_ts@20 {
        compatible = "synaptics_dsx";
        reg = <0x20>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c2_synaptics_dsx_io>;
        interrupt-parent = <&gpio1>;
        interrupts = <9 IRQ_TYPE_LEVEL_LOW>;
        synaptics,diagonal-rotation;
        status = "okay";
    };
```

和 i.MX8QXP MEK 板的设计：




由于 i.MX8QXP MEK 板的同一个口可以连接 LVDS 转 HDMI 的桥， 所以相应 i2C 和 GPIO 中断已经有 IOMUX 了如下：

```
        //MiPi DSI0
         pinctrl_i2c0_mipi_lvds0: mipi_lvds0_i2c0_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI0_I2C0_SCL_MIPI_DSI0_I2C0_SCL                0xc6000020
                        IMX8QXP_MIPI_DSI0_I2C0_SDA_MIPI_DSI0_I2C0_SDA                0xc6000020
                        IMX8QXP_MIPI_DSI0_GPIO0_01_LSIO_GPIO1_IO28          0x00000020
                >;
        };

&i2c0_mipi_lvds0 {
        #address-cells = <1>;
        #size-cells = <0>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c0_mipi_lvds0>;
        clock-frequency = <100000>;
        status = "okay";
        …
        //MiPi DSI1
pinctrl_i2c0_mipi_lvds1: mipi_lvds1_i2c0_grp {
                fsl,pins = <
                        IMX8QXP_MIPI_DSI1_I2C0_SCL_MIPI_DSI1_I2C0_SCL                0xc6000020
                        IMX8QXP_MIPI_DSI1_I2C0_SDA_MIPI_DSI1_I2C0_SDA                0xc6000020
                        IMX8QXP_MIPI_DSI1_GPIO0_01_LSIO_GPIO2_IO00          0x00000020
                >;
```

```
        };
&i2c0_mipi_lvds1 {
        #address-cells = <1>;
        #size-cells = <0>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c0_mipi_lvds1>;
        clock-frequency = <100000>;
        status = "okay";
…
```

所以按照以下步骤增加触摸屏驱动并测试：

1: add patch

```
--- a/arch/arm64/boot/dts/freescale/imx8qxp-mek-dsi-rm67191.dts
+++ b/arch/arm64/boot/dts/freescale/imx8qxp-mek-dsi-rm67191.dts
@@ -113,3 +113,15 @@
            };
    };
};
+&i2c0_mipi_lvds0{
+    synaptics_dsx_ts@20 {//i2c 地址
+        compatible = "synaptics_dsx";
+        reg = <0x20>;
+        interrupt-parent = <&lsio_gpio1>;
+        interrupts = <28 IRQ_TYPE_LEVEL_LOW>; //MiPi DSI 0 的 GPIO 中断连接在 GPIO1_IO28 上
+        synaptics,diagonal-rotation;
+        status = "okay";
+    };
+};
+&i2c0_mipi_lvds1{
+    synaptics_dsx_ts@20 {
+        compatible = "synaptics_dsx";
+        reg = <0x20>;
+        interrupt-parent = <&lsio_gpio2>;
+        interrupts = <0 IRQ_TYPE_LEVEL_LOW>;// MiPi DSI 1 的 GPIO 中断连接在 GPIO2_IO0 上
+        synaptics,diagonal-rotation;
```

```
+                status = "okay";
+        };
+};
make dtbs
```

to get imx8qxp-mek-dsi-rm67191-rpmsg.dtb and replace the SDcard fat partition imx8qxp-mek-dsi-rm67191-rpmsg.dtb,注意，有 M4 镜像时要使用 rpmsg.dtb，没有时使用一般的 dtb，我们这儿使用默认镜像是带 M4 的，所以加载 rpmsg.dtb。

2: boot and stop in uboot:

```
setenv fdt_file “imx8qxp-mek-dsi-rm67191-rpmsg.dtb” //替换 dtb 文件，改成显示接口为 MiPi DSI 屏
sav
reset
```

3: boot to shell run evtest, touch the screen and will have output.

```
root@imx8qxpc0mek:~# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      sc-powerkey
/dev/input/event1:      fxos8700
/dev/input/event2:      fxas2100x
/dev/input/event3:      mpl3115
/dev/input/event4:      isl29023 light sensor
/dev/input/event5:      synaptics_dsx_i2c
Select the device event number [0-5]: 5
Input driver version is 1.0.1
Input device ID: bus 0x18 vendor 0x0 product 0x3 version 0x2000
Input device name: "synaptics_dsx_i2c"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 116 (KEY_POWER)
    Event code 158 (KEY_BACK)
    Event code 172 (KEY_HOMEPAGE)
    Event code 325 (BTN_TOOL_FINGER)
    Event code 330 (BTN_TOUCH)
    Event code 580 (KEY_APPSELECT)
```

```
  Event type 3 (EV_ABS)
    Event code 47 (ABS_MT_SLOT)
      Value      0
      Min        0
      Max        9
    Event code 48 (ABS_MT_TOUCH_MAJOR)
      Value      0
      Min        0
      Max       30
    Event code 49 (ABS_MT_TOUCH_MINOR)
      Value      0
      Min        0
      Max       30
    Event code 53 (ABS_MT_POSITION_X)
      Value      0
      Min        0
      Max     1080
    Event code 54 (ABS_MT_POSITION_Y)
      Value      0
      Min        0
      Max     1920
    Event code 57 (ABS_MT_TRACKING_ID)
      Value      0
      Min        0
      Max    65535
Properties:
  Property type 1 (INPUT_PROP_DIRECT)
Testing ... (interrupt to exit)
Event: time 1590536892.172957, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value 3
Event: time 1590536892.172957, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1590536892.172957, type 1 (EV_KEY), code 325 (BTN_TOOL_FINGER), value 1
Event: time 1590536892.172957, type 3 (EV_ABS), code 53 (ABS_MT_POSITION_X), value 628
Event: time 1590536892.172957, type 3 (EV_ABS), code 54 (ABS_MT_POSITION_Y), value 1353
Event: time 1590536892.172957, type 3 (EV_ABS), code 49 (ABS_MT_TOUCH_MINOR), value 2
```

```
Event: time 1590536892.172957, ---------- SYN_REPORT ------------
Event: time 1590536892.257715, type 3 (EV_ABS), code 57 (ABS_MT_TRACKING_ID), value -1
Event: time 1590536892.257715, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1590536892.257715, type 1 (EV_KEY), code 325 (BTN_TOOL_FINGER), value 0
Event: time 1590536892.257715, ---------- SYN_REPORT ------------
…
```

## 1.7 LVDS LCD 驱动定制

i.MX8QXP 大部分在汽车上的显示设计有两种，一种是使用 MiPI DSI 或 LVDS 连接序列器，然后通过 LVDS 或 FPD-Link 等 串行线连接到解串器，然后再驱动屏。

另外一种是直接驱动屏，由于目前大部分的高清屏是奇偶分开的使用两路 LVDS 接口的屏，比如我们默认 BSP 中使用的是：

arch/arm64/boot/dts/freescale/imx8qxp-mek-jdi-wuxga-lvds0/1-panel.dts (0 或 1 的区别只是由那一路 LVDS 来做 Master 输出奇行)。

```
        {
  lvds0_panel {
                compatible = "jdi,tx26d202vm0bwa";
        …
};
&ldb1 {
  fsl,dual-channel;
…
  lvds-channel@0 {
                fsl,data-mapping = "spwg";
…
  };
};
&ldb2 {
  status = "disabled";
};
```

Uboot 设置 fdt_file 参数来使能此屏的驱动

```
=> setenv fdt_file 'imx8qxp-mek-jdi-wuxga-lvds0-panel.dtb'
=> sav
=> pri
```

```
fdt_file=imx8qxp-mek-jdi-wuxga-lvds0-panel.dtb
=> reset
```

源代码如下：

```
drivers/gpu/drm/panel/panel-simple.c
  {
                .compatible = "jdi,tx26d202vm0bwa",
                .data = &jdi_tx26d202vm0bwa,
    },
static const struct display_timing jdi_tx26d202vm0bwa_timing = {
    …
};

static const struct panel_desc jdi_tx26d202vm0bwa = {
    …
};
```

相关的 binding doc 在

\devicetree\bindings\video\display-timing.txt

\devicetree\bindings\video\fsl,ldb.txt

显示效果如下：

所以默认 BSP 是支持的一款 1920X1200 的双路 LVDS 的 LCD。由于在汽车上，比如说连接汽车仪表，大部分的高清屏是使用 1920X720 的分辨率，以下以中华印馆的 CLAA123FBA1XN 的 1920X720 的双路高清汽车级显示屏来说明如何修改 video mode 来支持一款新的 LVDS 屏：

根据该屏的 datasheet，其 timing 如下：

Timing Specification

| Item | | | | Symbol | Min | Typ | Max | Unit |
|---|---|---|---|---|---|---|---|---|
| LVDS input signal sequence | CLK Frequency | | | fCLKin | 40 | 52.3 | 66.12 | MHz |
| LCD input signal sequence （Input LVDS Transmitter） | DENA | Horizontal | Horizontal total Time | $t_H$ | 1070 | 1150 | 1230 | tCLK |
| | | | Horizontal effective Time | $t_{HA}$ | | 960 | | tCLK |
| | | | Horizontal Blank Time | $t_{HB}$ | 110 | 190 | 270 | tCLK |
| | | | Vertical total Time | $t_V$ | 748 | 758 | 768 | $t_H$ |
| | | | Vertical effectiveTime | $t_{VA}$ | | 720 | | $t_H$ |
| | | | Vertical Blank Time | $t_{VB}$ | 28 | 38 | 48 | $t_H$ |

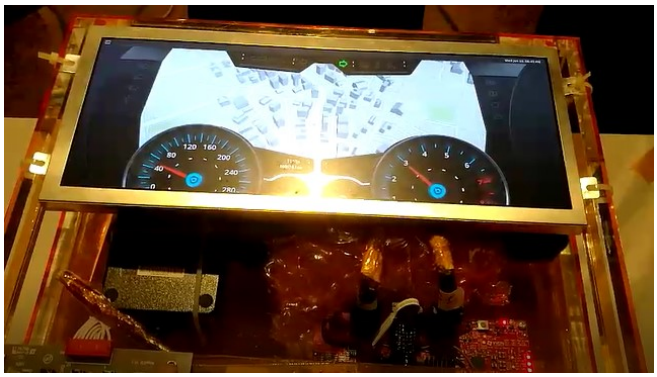
所以修改 vidoe mode 如下：

drivers/gpu/drm/panel/panel-simple.c

```
static const struct display_timing jdi_tx26d202vm0bwa_timing = {
```

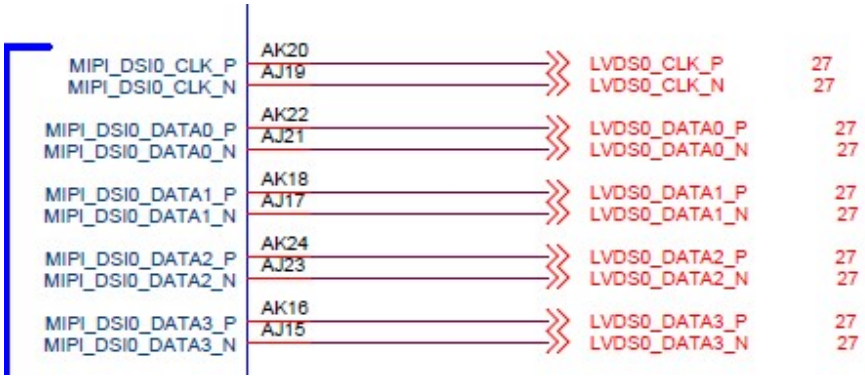
```
        .pixelclock = { 104604000, 104604000, 104604000 },//pixel clock=52.3MhzX2=(1920+200+180)X(720+30+8)X60=104604000
        .hactive = { 1920, 1920, 1920 }, //horizontal effective time =960X2=1920
        .hfront_porch = { 200, 200, 200 },//horizontal porch time=200+180=(1150-960)x2=380
        .hback_porch = { 180, 180, 190 },
        .hsync_len = { 10, 10, 10 }, //sync_len<front_porch
        .vactive = { 720, 720, 720}, //vertical effective time=720
        .vfront_porch = { 30, 30, 30 },//vertical porch time =30+8=758-720=38
        .vback_porch = { 8, 8, 8 },
        .vsync_len = { 2, 2, 2 },//sync_len< front_porch
        .flags = DISPLAY_FLAGS_DE_HIGH,
};
```

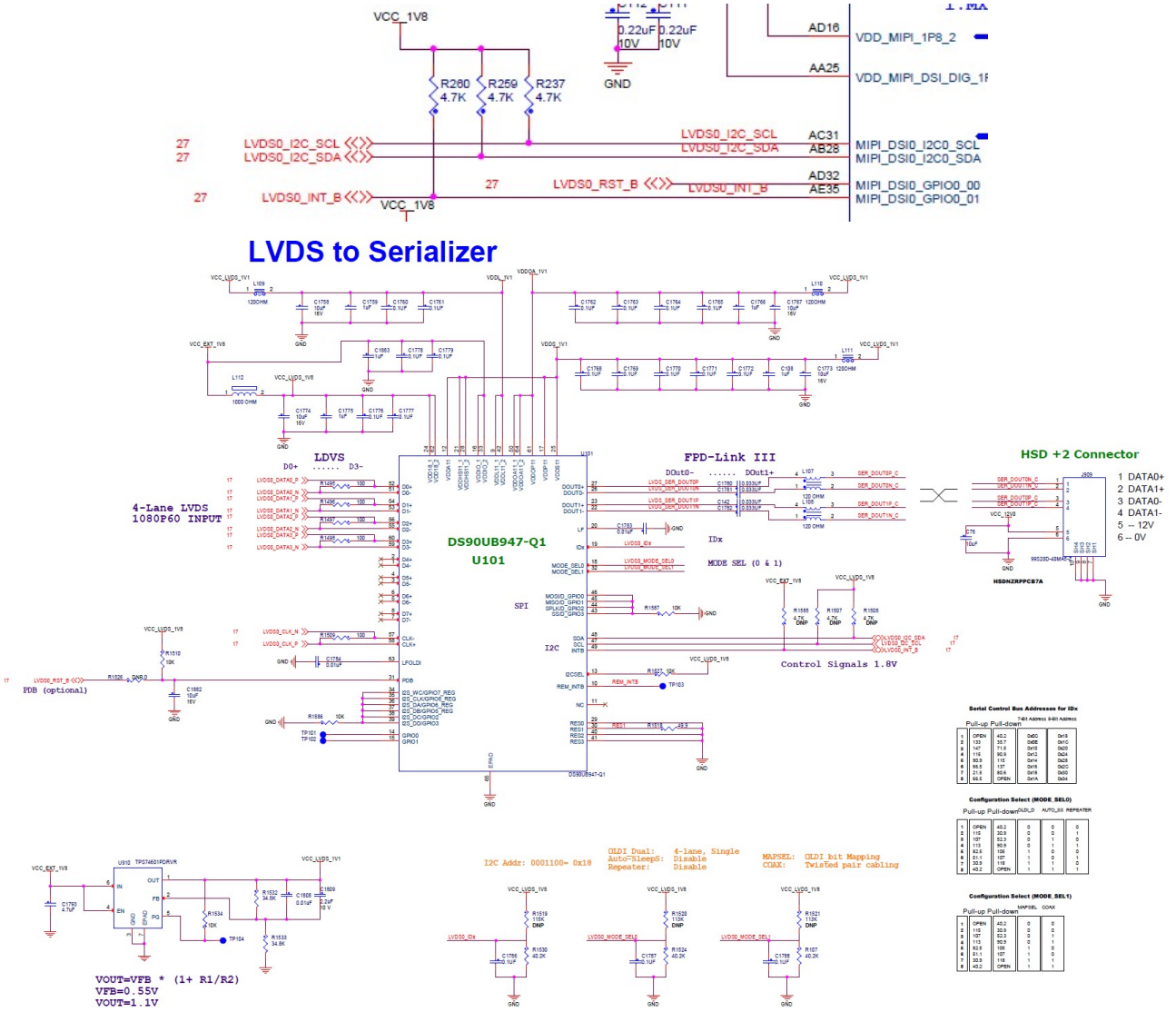
在我们默认开发板的屏和 1920X720 屏上的显示效果分别如下：



## 1.8 LVDS LDB SerDas 驱动支持

如前文所述，i.MX8QXP 6 layer 开发板支持 LVDS SerDas 屏，使用 TI DS90UB947 序列器，与 DS90UB948 解序器，硬件设计如下：






此设计最大的亮点在于 i.MX8QXP 使用 4 对 LVDS 线连接 TI947，输出 1080P, TI947 使用两个 channels 的 FPD-Link 连接到 TI948, TI948 再拆成奇偶各 4 对（共 8 对）LVDS 线，连接 1080P 的 LVDS 屏（目前主流的高清 LVDS 屏都是分成奇偶的，所以需要奇偶共 8 对 LVDS 线，比如说在 i.MX6 上就需要两个 LVDS 接口来驱动一块 LVDS 屏，但是在 i.MX8QXP 上，单个 4 对 LVDS 线的频率足够高，与 TI947/948 结合可以做到用一个 LVDS 接口来驱动高清屏，还可以空出一个 LVDS/MiPi DSI 接口来再接一个屏（i.MX8QXP 6layer 板使用了一个 MiPi DSI Serdas））。注意一下原理图中的硬件配置：

- OLDI_Dual: 4-lane, Single//硬件配置为OpenLDI(jeida),非Vesa(spwg)模式，一组pixel input.

Auto-SleepS: Disable

Repeater: Disable

- MAPSEL: OLDI_bit Mapping//硬件配置为OpenLDI(jeida),非Vesa(spwg)模式。

- COAX: Twisted pair cabling//硬件配置为双绞线，非同轴电缆。

**Table 6. MODE_SEL[1:0] Settings**

| Mode | Setting | Function |
|---|---|---|
| OLDI_DUAL: OpenLDI Interface | 0 | Single-pixel OpenLDI interface. |
| | 1 | Dual-pixel OpenLDI interface. |
| AUTO_SS: Auto Sleep-State | 0 | Disable. |
| | 1 | Enable. |
| REPEATER: Configure Repeater | 0 | Disable repeater mode. |
| | 1 | Enable repeater mode. |

**Table 6. MODE_SEL[1:0] Settings (continued)**

| Mode | Setting | Function |
|---|---|---|
| MAPSEL: OpenLDI Bit Mapping | 0 | OpenLDI bit mapping. |
| | 1 | SPWG bit mapping. |
| COAX: Cable Type | 0 | Enable FPD-Link III for twisted pair cabling. |
| | 1 | Enable FPD-Link III for coaxial cabling. |

以下简单说明驱动的情况，请与 NXP 支持窗口联系获得相关驱动源代码。

Drivers\gpu\drm\bridge\Kconfig:

```
config DRM_TI_DS90UB94x
        tristate "TI DS90UB947/948 FPD-Link serializer/deserializer chip"
    default y //默认在内核中使能，所以不用在 defconfig 中增加定义
    depends on I2C
    help
      if you say yes here you get support for the DS90UB947/948 series of
      chips.
```

Drivers\gpu\drm\bridge\Makefile:

```
obj-$(CONFIG_DRM_TI_DS90UB94x) += ds90ub94x.o
```

DTS 配置如下：arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
&i2c0_mipi_lvds0 {
…
/*
        lvds_bridge0: lvds-to-hdmi-bridge@4c {
                …
        };
*/
//add
        lvds_bridge0: ds90ub94x@0c {
```

```
                compatible = "ti,ds90ub94x";
                reg = <0x0c>;
                clock-frequency = <400000>;

                port@0 {
                        ds90ub94x_in: endpoint {
                                remote-endpoint = <&lvds0_out>;
                        };
                };
        };
//end
…
&ldb1_phy {
        status = "okay";
};
&ldb1 {
        status = "okay";
        lvds-channel@0 {
                fsl,data-mapping = "jeida";//bit mapping 软件配置为 OpenLDI(JEIDA)，非 VESA(SPWG)模式，与以上硬件配置相同
                fsl,data-width = <24>;//24bit
                status = "okay";
                port@1 {
                        reg = <1>;
                        lvds0_out: endpoint {
                                remote-endpoint = <&ds90ub94x_in>;// <&it6263_0_in>;将原本连接到 lvds0 上的 lvds to hdmi 桥改成 lvds to serdas 桥
                        };
                };
        };
};
```

源代码中主要关注一下对 TI947/948 的 I2C 配置：

Drivers\gpu\drm\bridge\ds90ub94x.c

```
static int ds90ub94x_probe(struct i2c_client *client,
```

```
                                const struct i2c_device_id *id)
{…
        /*OLDI_DUAL:Single-pixel mode,MAPSEL: OpenLDI Bit Mapping*/
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0x4f, 0xc0);//此处已经被硬件配置 strap 掉了，所以不起作用，以上注释为硬件配置
        if (ret)
                goto unregister_ds90ub948_i2c;

        /*AUTO_SS:Auto Sleep-State*/
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0x01, 0x00);
//和硬件配置一下，关闭 auto sleep
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0x01 | Reset | 7 | RW | 0x00 | SOFT_SLEEP | 0: Do not power down when no Bidirectional Control Channel link is detected (default).<br>1: Power down when no Bidirectional Control Channel link is detected. |

```
        if (ret)
                goto unregister_ds90ub948_i2c;
        /*REPEATER:Disable repeater mode*/
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0xc2, 0x98);
//和硬件配置一下，不使用 repeater 模式
```

| ADD (dec) | ADD (hex) | Register Name | Bit(s) | Register Type | Default (hex) | Function | Description |
|---|---|---|---|---|---|---|---|
| 194 | 0xC2 | CFG | 7 | RW | 0x80 | ENH_LV | Enable Enhanced Link Verification: Enables enhanced link verification. Allows checking of the encryption Pj value on every 16th frame.<br>1 = Enhanced Link Verification enabled.<br>0 = Enhanced Link Verification disabled. |
| | | | 6 | | | | Reserved. |
| | | | 5 | RW | | TX_RPTR | Transmit Repeater Enable: Enables the transmitter to act as a repeater.<br>1 = Transmit Repeater mode enabled.<br>0 = Transmit Repeater mode disabled. |
| | | | 4:3 | RW | | ENC_MODE | Encryption Control Mode: Determines mode for controlling whether encryption is required for video frames.<br>00 = Enc_Authenticated.<br>01 = Enc_Reg_Control.<br>10 = Enc_Always.<br>11 = Enc_InBand_Control (per frame).<br>If the Repeater strap option is set at power-up, Enc_InBand_Control (ENC_MODE == 11) will be se-lected. Otherwise, the default will be Enc_Authenticated mode (ENC_MODE == 00). |

```
        if (ret)
                goto unregister_ds90ub948_i2c;
        /*COAX:Enable FPD-Link III for STP*/
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0x5b, 0x20);
//硬件已经 strap 为双铰线模式
```

| ADD (dec) | ADD (hex) | Register Name | Bit(s) | Register Type | Default (hex) | Function | Description |
|---|---|---|---|---|---|---|---|
| 91 | 0x5B | DUAL_CTL1 | 7 | RW | Strap | FPD3_COAX_MODE | FPD-Link III Coax Mode: Enables configuration for the FPD-Link III Interface cabling type:<br>0 : Twisted Pair.<br>1 : Coax.<br>This bit is loaded from the MODE_SEL1 pin at power-up. |
| | | | 6 | RW | 0x20 | DUAL_SWAP | Dual Swap Control:<br>Indicates current status of the Dual Swap control. If automatic correction of Dual Swap is disabled via the DISABLE_DUAL_SWAP control, this bit may be modified by software. |

```
        if (ret)
                goto unregister_ds90ub948_i2c;

        /*PASS-THROUGH:Enable pass-through*/
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0x03, 0xDA);
//与默认值相比，主要是设置了 i2c 访问解序器的 pass-through 模式。
```

**Table 10. Serial Control Bus Registers (continued)**

| ADD (dec) | ADD (hex) | Register Name | Bit(s) | Register Type | Default (hex) | Function | Description |
|---|---|---|---|---|---|---|---|
| 3 | 0x03 | General Configuration | 7 | RW | 0xD2 | Back channel CRC Checker Enable | Enable/disable back channel CRC Checker.<br>0: Disable.<br>1: Enable (default). |
| | | | 6 | | | | *Reserved.* |
| | | | 5 | RW | | I2C Remote Write Auto Acknowledge Port0/Port1 | Automatically acknowledge I2C remote writes. When enabled, I2C writes to the Deserializer (or any remote I2C Slave, if I2C PASS ALL is enabled) are immediately acknowledged without waiting for the Deserializer to acknowledge the write. This allows higher throughput on the I2C bus. Note: this mode will prevent any NACK from a remote device from reaching the I2C master.<br>0: Disable (default).<br>1: Enable.<br>If PORT1_SEL is set, this field refers to Port1 operation. |
| | | | 4 | RW | | Filter Enable | HS, VS, DE two-clock filter. When enabled, pulses less than two full PCLK cycles on the DE, HS, and VS inputs will be rejected.<br>0: Filtering disable.<br>1: Filtering enable (default). |
| | | | 3 | RW | | I2C Pass-through Port0/Port1 | I2C pass-through mode. Read/Write transactions matching any entry in the Slave Alias registers will be passed through to the remote Deserializer.<br>0: Pass-through disabled (default).<br>1: Pass-through enabled.<br>If PORT1_SEL is set, this field refers to Port1 operation. |
| | | | 2 | | | | *Reserved.* |
| | | | 1 | RW | | PCLK Auto | Switch over to internal oscillator in the absence of PCLK.<br>0: Disable auto-switch.<br>1: Enable auto-switch (default). |

```
        if (ret)
                goto unregister_ds90ub948_i2c;
        ret = regmap_write(ds90ub94x->ds90ub947_regmap, 0x17, 0x9E);
```

**Table 10. Serial Control Bus Registers (continued)**

| ADD (dec) | ADD (hex) | Register Name | Bit(s) | Register Type | Default (hex) | Function | Description |
|---|---|---|---|---|---|---|---|
| 23 | 0x17 | I2C Control | 7 | RW | 0x1E | I2C Pass All | 0: Enable Forward Control Channel pass-through only of I2C accesses to I2C Slave IDs matching either the remote Deserializer Slave ID or the remote Slave ID (default).<br>1: Enable Forward Control Channel pass-through of all I2C accesses to I2C Slave IDs that do not match the Serializer I2C Slave ID. |
| | | | 6:4 | RW | | SDA Hold Time | Internal SDA hold time:<br>Configures the amount of internal hold time provided for the SDA input relative to the SCL input. Units are 40 nanoseconds. |
| | | | 3:0 | RW | | I2C Filter Depth | Configures the maximum width of glitch pulses on the SCL and SDA inputs that will be rejected. Units are 5 nanoseconds. |

```
        if (ret)
```

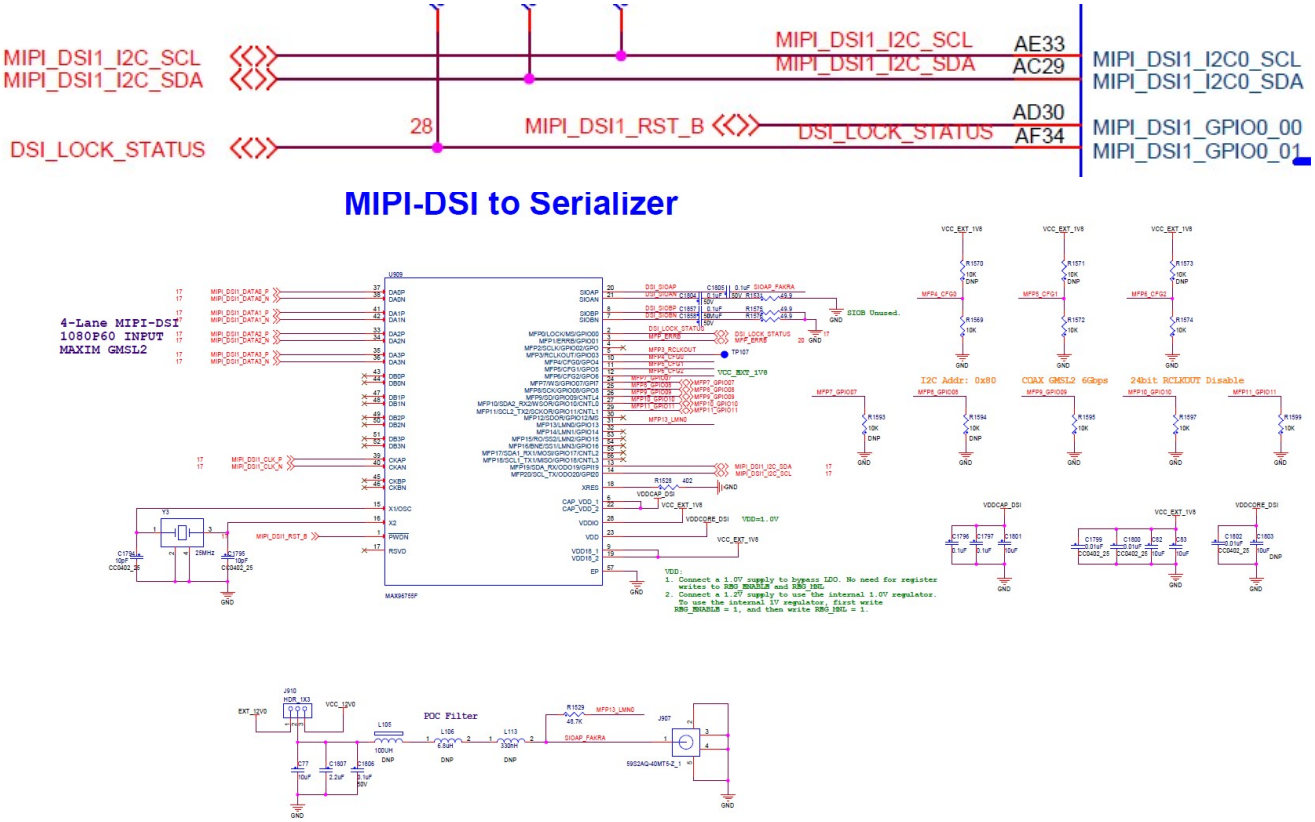
```
                goto unregister_ds90ub948_i2c;

        /*check if ds90ub948 device connected*///读 TI948 寄存器 0 看是否可以访问。
        ret = regmap_read(ds90ub94x->ds90ub948_regmap, 0x00, &data);
        if ((data >> 1) != DS90UB948_I2C_ADDR) {
                dev_err(dev, "No ds90ub948 device connected");
                goto unregister_ds90ub948_i2c;
…}
…
static void ds90ub94x_bridge_enable(struct drm_bridge *bridge)
{…
//reset TI948
map_write(ds90ub94x->ds90ub948_regmap, 0x01, 0x01);
                msleep(10);
…}
```

如上所述 TI947/948 Serdas 驱动还是比较简单的，大部分配置硬件已经完成，软件主要设置一下 slave i2c 的 pass thougth 模式。

## 1.9 MiPi DSI SerDas 驱动支持

如前文所述，i.MX8QXP 6 layer 开发板支持 MiPi SerDas 桥连接屏，使用 Maxim MAX96755 序列器，与 MAX96752 解序器，硬件设计如下：




MiPi DSI 本身支持高清，本设计中用了四 Lane 支持 1080P60,。注意一下原理图中的硬件配置：

以下简单说明驱动的情况，请与 NXP 支持窗口联系获得相关驱动源代码。

Drivers\gpu\drm\bridge\Kconfig:

```
config DRM_MAXIM_MAX9675x
        tristate "MAXIM MAX9675x bridge"
        select DRM_KMS_HELPER
        select REGMAP_I2C
        default y //默认在内核中使能，所以不用在 defconfig 中增加定义
        ---help---
          This is a driver for the display bridges of
          MAX9675x serder that convert MIPI_DSI to
          OLDI.
```

Drivers\gpu\drm\bridge\Makefile:

```
obj-$(CONFIG_DRM_MAXIM_MAX9675x) += mx9675x.o
```

DTS 配置如下：arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
&i2c0_mipi_lvds1 {
…
/* 注掉
    lvds_bridge1: lvds-to-hdmi-bridge@4c {
…
    };
    adv_bridge1: adv7535@3d {
…
    };
*/
//add
    max_serdeser: max9675x@40 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "max,max9675x", "max,max96755";
            reg = <0x40>;
            max,dsi-lanes = <4>;
            max,dsi-channel = <1>;
            reset-gpio = <&gpio1 31 GPIO_ACTIVE_LOW>;
            status = "okay";
            panel@0 {
                    compatible = "max,max96752-1080p";
                    reg = <0>;
                    panel-width-mm = <68>;
                    panel-height-mm = <121>;
                    port {
                            panel1_in: endpoint {
                                    remote-endpoint = <&max9675x_1_out>;
                            };
                    };
            };

            port@1 {
```

```
                        reg = <1>;
                        max9675x_1_in: endpoint {
                                remote-endpoint = <&mipi_dsi_bridge2_max>;
                        };
                };

                port@2 {
                        reg = <2>;
                        max9675x_1_out: endpoint {
                                remote-endpoint = <&panel1_in>;
                        };
                };
        };
//end
};
…
&mipi1_dphy {
    status = "okay";
};
&mipi1_dsi_host {
    status = "okay";
    ports {
                port@1 {
                        reg = <1>;
                        mipi1_adv_out: endpoint {
                                remote-endpoint = <&max9675x_1_in >;
                        };
                };
    };
};
```

源代码中主要关注一下对 serdas 的 I2C 配置：

Drivers\gpu\drm\bridge\Mx9675x.c

```
max9675x_probe
|-> max9675x_hw_reset//使用 DTS 中 parser 过来的 gpio reset 桥芯片
```

```
|-> max9675x_check_chipid// check both Serializer and DESerializer chip
|-> max9675x_ser_deser_init//初始化 ser/derser
        /* ser/deser AUDIO_RX1 close */
        /* Set transmitter bit rate to 6Gbps */
        /* Close video data before maxim init */
        /* Set EOM_PER_MODE and EOM_CHK_AMOUNT */
        /* Set AEQ_PER and AEQ_PER_MULT */
        /* Set DFEAdpDly */
        /* Set CTFAdpDly */
        /* Increase link robustness */
        /* oLDI Deserializers PHYA Optimize Settings */
        /* oLDI Deserializers PHYB Optimize Settings */
|-> max9675x_ser_sw_reset // link reset

max9675x_bridge_enable
|-> max9675x_config_front
|->max9675x_config_dsi
|->      /* Packets are not transmitted over GMSLB in splitter mode
         * Packets are transmitted over GMSLA in splitter mode
         */
|->max9675x_open_data_transfer
|->/* Set oLDI display */
```

## 1.10 V4L2 框架汽车级高清摄像头/桥驱动：数字/模拟

汽车级高清数字摄像头/ISP 芯片的主要供应商有 Onsemi 和 OV,序列器主要是 maxim 和 TI。

汽车级高清模拟(AHD)接收器/发射器+ISP 桥芯片主要供应商有 Nextchip 和 techpoint。目前均已经在 i.MX8X 平台上完成驱动开发和测试验证。

下表说明 i.MX8X 平台目前汽车级高清摄像头的支持情况：

| | 解序器 | 序列器 | ISP | Sensor | 驱动 | 应用 |
|---|---|---|---|---|---|---|
| 数字 | Max9286<br>(Maxim 4 channel MiPi CSI)<br>GMSL | Max9271<br>(Maxim) | N/A | ov10635(OV 彩色 sensor) | i.MX8QXP MEK 原生支持 | 环视(4 camera)<br>后视(1 camera) |
| | | Max96705<br>(Maxim)<br>GMSL | AP0101 | AR0132(Onsemi 彩色 sensor) | i.MX8QXP MEK Patch 支持 | 环视(4 camera)<br>后视(1 camera) |
| | | | N/A | AR0144(12 bitOnsemi<br>灰度 sensor) | i.MX8QXP MEK Patch 支持 | DMS(1 camera) |
| | | | N/A | Ov9284(OV10 bit 灰度 sensor) | i.MX8QXP MEK Patch 支持 | DMS(1 camera) |
| | ds90ub962<br>(TI 4 channel MiPi CSI)<br>FPDLink | ds90ub933<br>(TI)<br>FPDLink | N/A | ov10635(Onsemi 彩色 sensor) | i.MX8QXP MEK Patch 支持(相关驱动请与 TI 或其代理联系) | 环视(4 camera)<br>后视(1 camera) |
| | 接收器 | 发射器+ISP | | Sensor | 驱动 | 应用 |
| 高清模拟 (AHD) | nvp6324<br>(Nextchip) | nvp2431h/2631<br>(Nextchip) | | IMX225 | i.MX8QXP MEK Patch 支持<br>i.MX8QXP 6layer 板支持 | 环视(4 camera)<br>后视(1 camera) |
| | TP2855<br>(techpoint) | TP3812<br>(techpoint) | | IMX307<br>(Sony 彩色) | i.MX8QXP MEK Patch 支持(开发中) | 环视(4 camera)<br>后视(1 camera) |

注意：

- i.MX8QXP MEK 板可以通过 miniSAS 的 MiPi CSI 接口连接一个 miniSAS 的 MAX9286 的子板，所以序列化的数字摄像头模块，如 ov10635+ Max9271，或(AR0132+AP0101/OV9284/AR0144)+Max96705 数字摄像头模块,可以通过 GMSL 同轴电缆直接连接在 MAX9286 的子卡上，所以不用说明硬件连接情况。
- Nextchip 设计了一块 miniSAS 接口的 nvp6324 子板，可以连接到 i.MX8QXP MEK 板的 miniSAS 的 MiPi CSI 接口上，同时，i.MX8QXP 6layer 板上设计了 nvp6324 的接口。
- Techpoint 设计了一块 miniSAS 接口的 TP2855 子板，可以连接到 i.MX8QXP MEK 板的 miniSAS 的 MiPi CSI 接口上。
- 请与 NXP 支持窗口联系，获得其 Linux 驱动，以下简要说明 一下软件驱动：

### 1.10.1 V4L2 架构概述

从应用程序角度，通过 V4L2 接口操作一个 Camera 流程大概如下：

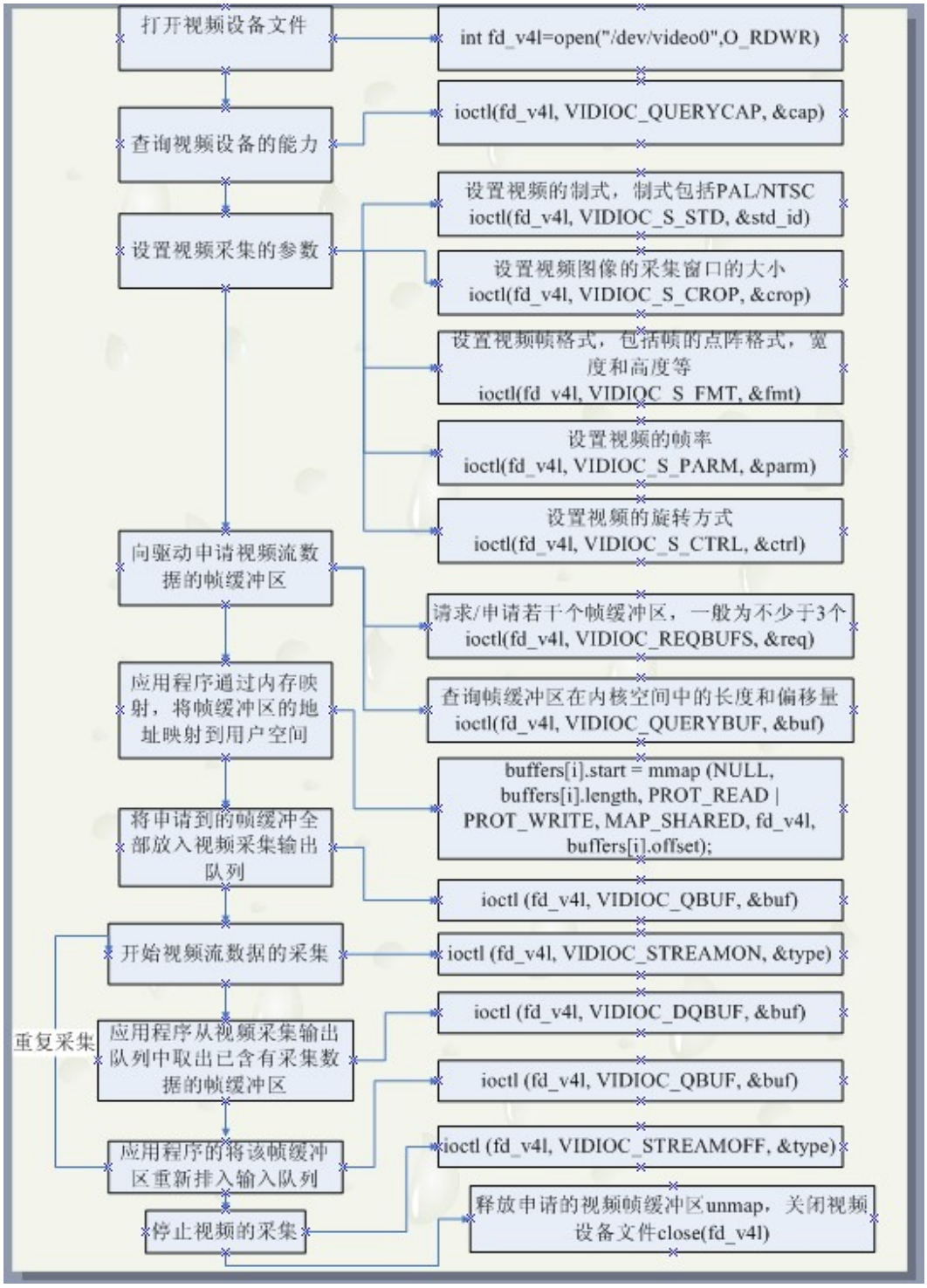
首先，打开视频设备文件，进行视频采集的参数初始化，通过 V4L2 接口设置视频图像的采集窗口、采集的点阵大小和格式；

其次，申请若干视频采集的帧缓冲区，并将这些帧缓冲区从内核空间映射到用户空间，便于应用程序读取/处理视频数据；

第三，将申请到的帧缓冲区在视频采集输入队列排队，并启动视频采集；

第四，驱动开始视频数据的采集，应用程序从视频采集输出队列取出帧缓冲区，处理完后，将帧缓冲区重新放入视频采集输入队列，循环往复采集连续的视频数据；

第五，停止视频采集。

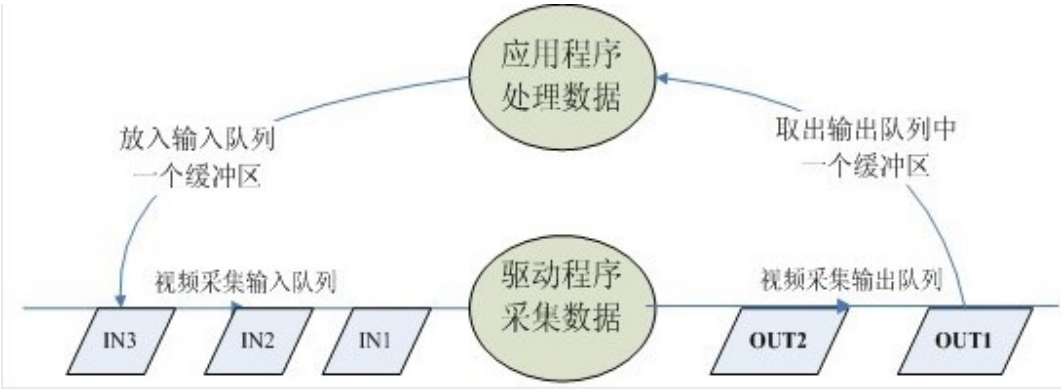



其实其他的都比较简单，就是通过 ioctl 这个接口去设置一些参数。最主要的就是 buf 管理。他有一个或者多个输入队列和输出队列。

启动视频采集后，驱动程序开始采集一帧数据，把采集的数据放入视频采集输入队列的第一个帧缓冲区，一帧数据采集完成，也就是第一个帧缓冲区存满一帧数据后，驱动程序将该帧缓冲区移至视频采集输出队列，等待应用程序从输出队列取出。驱动程序接下来采集下一帧数据，放入第二个帧缓冲区，同样帧缓冲区存满下一帧数据后，被放入视频采集输出队列。

应用程序从视频采集输出队列中取出含有视频数据的帧缓冲区，处理帧缓冲区中的视频数据，如存储或压缩。

最后，应用程序将处理完数据的帧缓冲区重新放入视频采集输入队列，这样可以循环采集，如图所示。




每一个帧缓冲区都有一个对应的状态标志变量，其中每一个比特代表一个状态

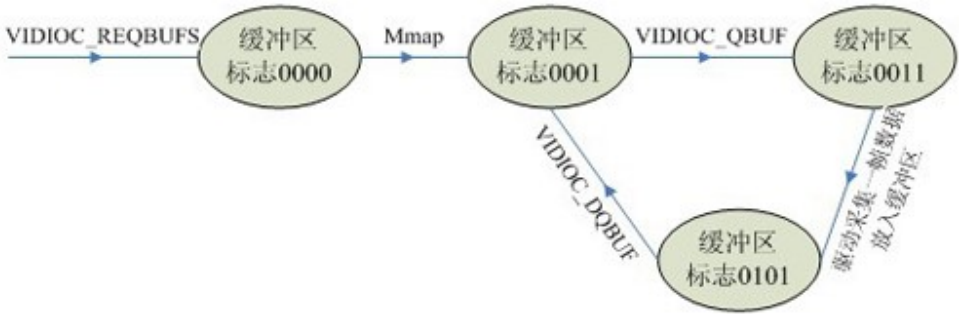
V4L2_BUF_FLAG_UNMAPPED 0B0000

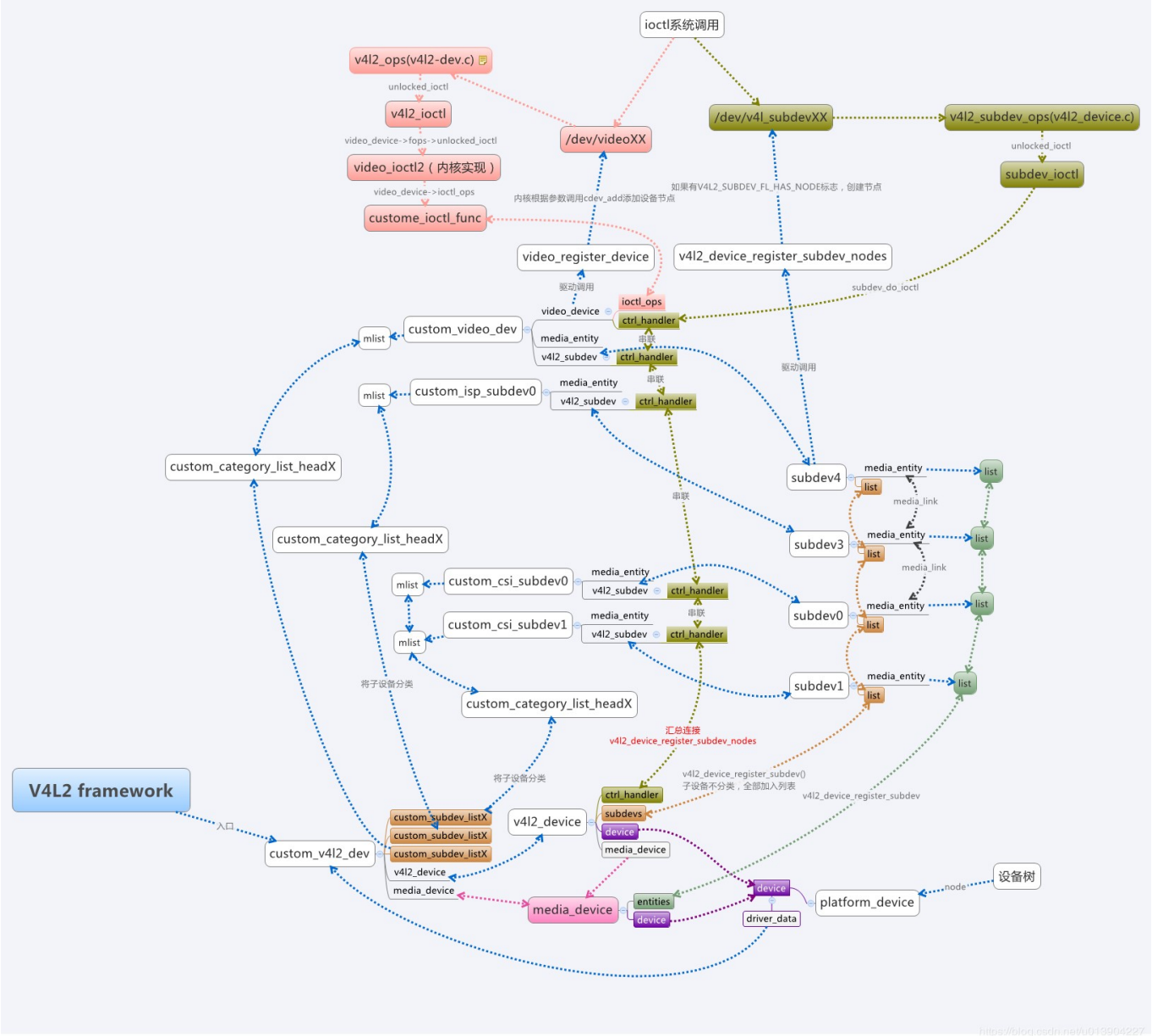
V4L2_BUF_FLAG_MAPPED 0B0001

V4L2_BUF_FLAG_ENQUEUED 0B0010

V4L2_BUF_FLAG_DONE 0B0100

缓冲区的状态转化如图所示。






## 1.10.2 OV9284(DMS)

drivers/staging/media/imx/Kconfig

```
config MAX9286_WISSEN
	tristate "Maxim max9286 wissen Deserializer Input support"
	  select SENSOR_OV10635
	depends on I2C
	---help---
	  If you plan to use the max9286 GMSL Deserializer with your capture system, say Y here.
```

所以在内核 config 时需要增加 MAX9286_WISSEN,去掉 CONFIG_GMSL_MAX9286 的支持

然后.config 如下：

```
# CONFIG_GMSL_MAX9286 is not set
CONFIG_MAX9286_WISSEN=y
```

drivers/staging/media/imx/Makefile

```
+obj-$(CONFIG_MAX9286_WISSEN) += max9286-wissen.o
```

源代码：

drivers/staging/media/imx/max9286-wissen.c, 此文件就是将 gmsl_max9286.c 修改过来的，将里面的 ov10635 的初始化和配置数组及访问函数改成了 ov9284，将 max9271 的序列器访问函数改成了 max96705 的序列器访问函数，然后数据格式从 MEDIA_BUS_FMT_YUYV8_1X16 改成了 MEDIA_BUS_FMT_Y10_1X10

```
static const struct of_device_id max9286_of_match[] = {
 { .compatible = "maxim,max9286_mipi" },//仍然是使用 maxim,max9286_mipi 的 compatible name,及对应的 DTS
 { /* sentinel */ }
};
…
max9286_probe
|->获得 mclk 和 gpio reset 保持不变
|->max9286_data->format.code = MEDIA_BUS_FMT_Y10_1X10;//DMS 使用的是 Y10 10bit 灰阶摄像头 MEDIA_BUS_FMT_YUYV8_1X16
|-> max9286_data->format.width/ height 从 ov9284_mode_info_data 获得
|-> max9286_hardware_preinit
|   |->      /* him high, bwl:1*/
|   |->      /* Set RAW 10 bits mode, Double Data Rate, 4 data lane */
|   |->      /* clinken */
|-> max9286_hardware_init
|   |->
…
```

ov9284_mode_info_data 数组包含了 ov9284_mode 数组仅支持 ov9284_mode_WXGA_1280_800，和配置数组 ov9284_setting_30fps_WXGA_1280_800，函数 max9286_s_parm/ max9286_enum_framesizes/ max9286_enum_frame_interval/ get_max_resolution/ match/ try_to_find_resolution/ max9286_probe 会调用到。

ov9284_frame_rate 数组仅支持 OV9284_30_FPS，函数 to_ov9284_frame_rate/max9286_s_parm/get_max_resolution/match/try_to_find_resolution/会调用到。

ov9284_init_data 初始化数组，函数 ov9284_initialize 会调用到。

Drivers\staging\media\imx\imx8-isi-cap.c ISI 使用 16bit 接收 12bit Y 值 raw 数据。

```
struct mxc_isi_fmt mxc_isi_out_formats[] = {
…
{
        .name           = "Y10",
        .fourcc         = V4L2_PIX_FMT_Y10, // #define V4L2_PIX_FMT_Y10    v4l2_fourcc('Y', '1', '0', '
') /* 10  Greyscale    */
        .depth          = { 16 },
        .color          = MXC_ISI_OUT_FMT_RAW16,
        .memplanes      = 1,
        .colplanes      = 1,
        .mbus_code      = MEDIA_BUS_FMT_Y10_1X10,
 }
```

### 1.10.3 AR0144(DMS)

drivers/staging/media/imx/Kconfig

```
config MAX9286_AR0144
    tristate "Maxim max9286 AR0144 Deserializer Input support"
    depends on I2C
    ---help---
      If you plan to use the max9286 GMSL Deserializer with your capture system, say Y here.
```

所以在内核 config 时需要增加 MAX9286_WISSEN,去掉 CONFIG_GMSL_MAX9286 的支持

然后.config 如下：

```
# CONFIG_GMSL_MAX9286 is not set
CONFIG_MAX9286_AR0144=y
```

drivers/staging/media/imx/Makefile

```
obj-$(CONFIG_MAX9286_AR0144) += max9286-ar0144.o
```

源代码：

drivers/staging/media/imx/ max9286-ar0144.c, 此文件就是将 gmsl_max9286.c 修改过来的，将里面的 ov10635 的初始化和配置数组及访问函数改成了 ar0144，将 max9271 的序列器访问函数改成了 max96705 的序列器访问函数，然后数据格式从 MEDIA_BUS_FMT_YUYV8_1X16 改成了 MEDIA_BUS_FMT_Y12_1X12

```
static const struct of_device_id max9286_of_match[] = {
 { .compatible = "maxim,max9286_mipi" },//仍然是使用 maxim,max9286_mipi 的 compatible name,及对应的 DTS
```

```
 { /* sentinel */ }
};
…
max9286_probe
|->获得 mclk 和 gpio reset 保持不变
|-> max9286_data->format.code = MEDIA_BUS_FMT_Y12_1X12;;//DMS 使用的是 Y12 12bit 灰阶摄像头 MEDIA_BUS_FMT_YUYV8_1X16
|-> max9286_data->format.width/ height 从 ar0144_mode_info_data 获得
|-> max9286_hardware_preinit
|   |->     /* him high, bwl:1*/
|   |->     /* Set RAW 12 bits mode, Double Data Rate, 4 data lane */
|   |->     /* clinken */
|-> max9286_hardware_init
|   |->
…
```

ar0144_mode_info_data 数组包含了 ar0144_mode 数组:仅支持 ar0144_mode_WXGA_1280_800，和配置数组 ar0144_setting_30fps_WXGA_1280_800，函数 max9286_s_parm/ max9286_enum_framesizes/ max9286_enum_frame_interval/ get_max_resolution/ match/ try_to_find_resolution/ max9286_probe 会调用到。

ar0144_frame_rate 数组仅支持 AR0144_30_FPS，函数 to_ ar0144_frame_rate/max9286_s_parm/get_max_resolution/match/try_to_find_resolution/会调用到。

ar0144_init_data 初始化数组，函数 ar0144_initialize 会调用到。

Drivers\staging\media\imx\imx8-isi-cap.c ISI 使用 16bit 接收 12bit Y 值 raw 数据。

```
struct mxc_isi_fmt mxc_isi_out_formats[] = {
…
{
        .name           = "BYR2",
        .fourcc         = V4L2_PIX_FMT_SBGGR16, // #define V4L2_PIX_FMT_SBGGR16 v4l2_fourcc('B', 'Y', 'R', '2') /* 16  BGBG.. GRGR.. */
        .depth          = { 16 },
        .color          = MXC_ISI_OUT_FMT_RAW16,
        .memplanes      = 1,
        .colplanes      = 1,
        .mbus_code      = MEDIA_BUS_FMT_SBGGR16_1X16,
 }
}
```

### 1.10.4 AR0132(彩色)

arch/arm64/boot/dts/freescale/imx8x-mek.dtsi //AR0132 需要修改 Max9286 的 mclk

```
            mclk = <24000000>;//<27000000>;
            mclk = <24000000>;
```

drivers/staging/media/imx/Kconfig

```
config MAX9286_AR0132
 tristate "Maxim max9286 AR0132 Deserializer Input support"
 depends on I2C
 ---help---
  If you plan to use the max9286 GMSL Deserializer with your capture system, say Y here.
```

所以在内核 config 时需要增加 CONFIG_MAX9286_AR0132,去掉 CONFIG_GMSL_MAX9286 的支持

然后.config 如下：

```
 # CONFIG_GMSL_MAX9286 is not set
 CONFIG_MAX9286_AR0132=y
```

drivers/staging/media/imx/Makefile

```
obj-$(CONFIG_MAX9286_AR0132) += max9286-ar0132.o
```

源代码：

drivers/staging/media/imx/ max9286-ar0132.c, 此文件就是将 gmsl_max9286.c 修改过来的，将里面的 ov10635 的初始化和配置数组及访问函数改成了 ar0132，将 max9271 的序列器访问函数改成了 max96705 的序列器访问函数，然后数据格式从 MEDIA_BUS_FMT_YUYV8_1X16 改成了 MEDIA_BUS_FMT_Y12_1X12

```
static const struct of_device_id max9286_of_match[] = {
 { .compatible = "maxim,max9286_mipi" },//仍然是使用 maxim,max9286_mipi 的 compatible name,及对应的 DTS
 { /* sentinel */ }
};
…
max9286_probe
|->获得 mclk 和 gpio reset 保持不变
|-> max9286_data->format.code = MEDIA_BUS_FMT_YUYV8_1X16;//camera 格式与 ov10635 一样
|-> max9286_data->format.width/ height 从 ar0132_mode_info_data 获得
|-> max9286_hardware_preinit
|   |->      /* him high, bwl:1*/
```

```
|   |->      /* Set YUV422 8 bits mode, Double Data Rate, 4 data lane */
|   |->      /* clinken */
|-> max9286_hardware_init
|   |->
…
```

ar0132_mode_info_data 数组包含了 ar0132_mode 数组:仅支持 ar0132_mode_720P_1280_720，和配置数组 ar0132_setting_60fps_720P_1280_720，函数 max9286_s_parm/ max9286_enum_framesizes/ max9286_enum_frame_interval/ get_max_resolution/ match/ try_to_find_resolution/ max9286_probe 会调用到。

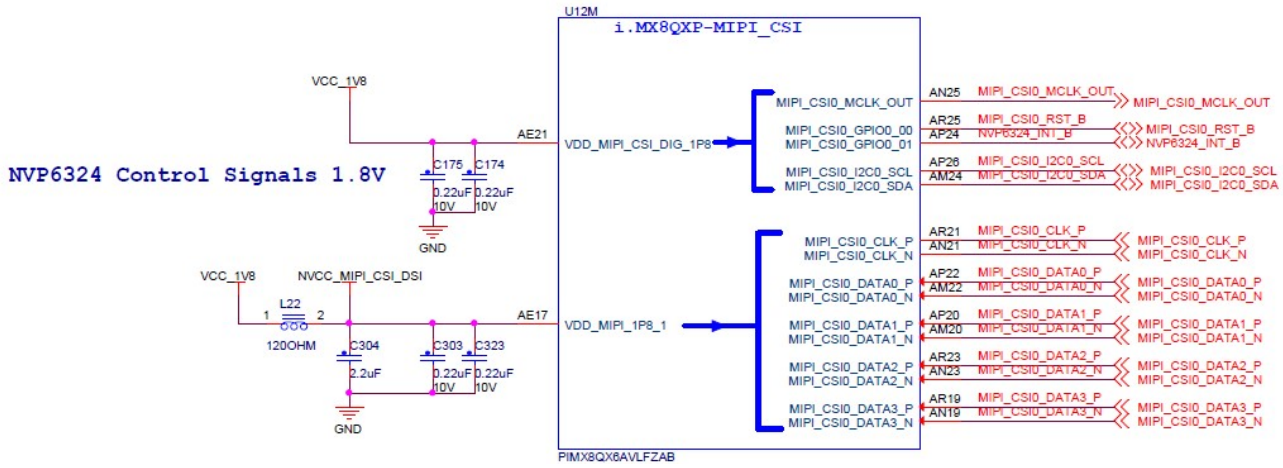
ar0132_frame_rate 数组仅支持 AR0132_30_FPS，函数 to_ ar0132_frame_rate/max9286_s_parm/get_max_resolution/match/try_to_find_resolution/会调用到。

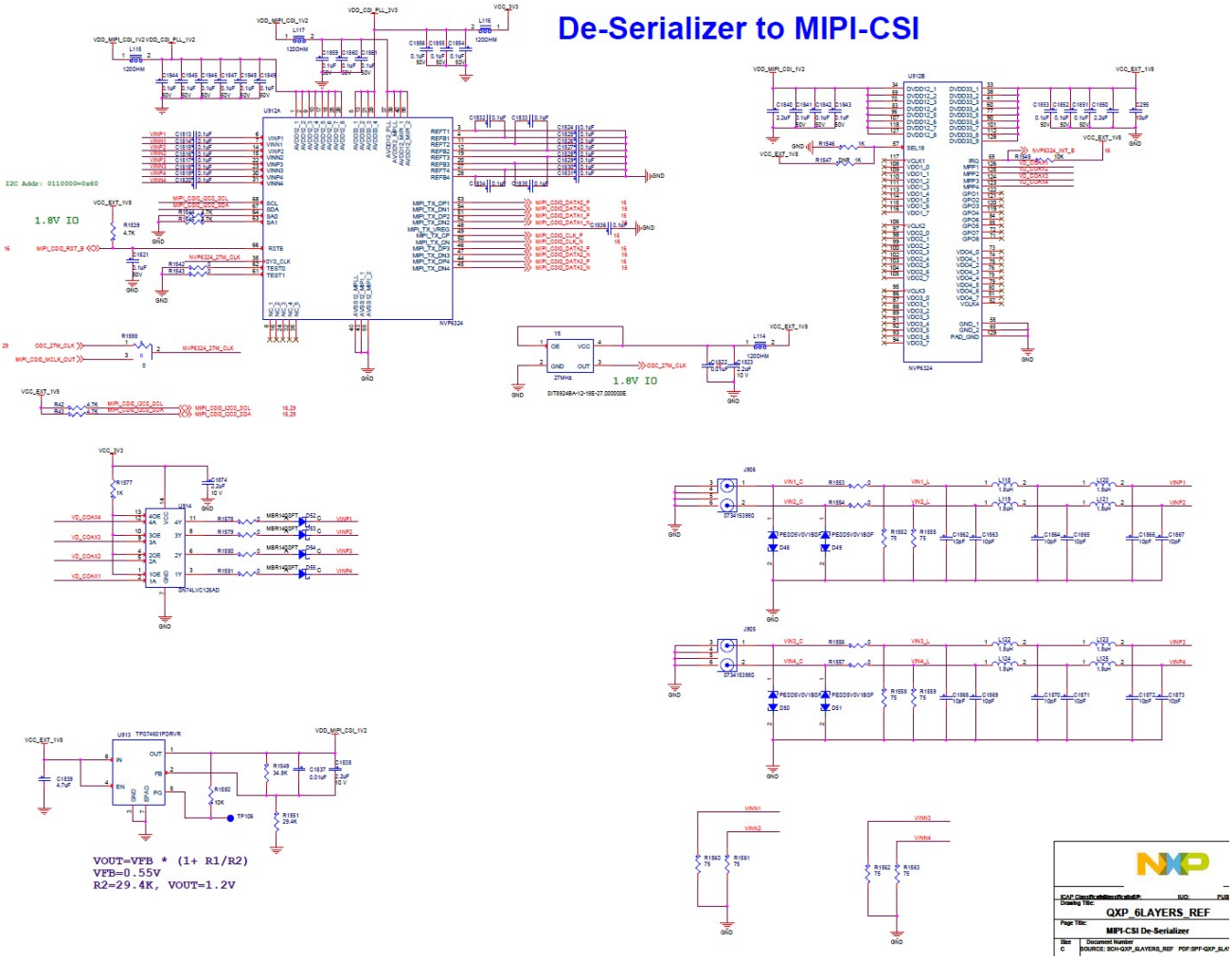
ar0132_init_data 初始化数组，函数 ar0132_initialize 会调用到。

Drivers\staging\media\imx\imx8-isi-cap.c ar0132 与 ov10635 一致，不需要修改

### 1.10.5 AHD Camera In bridge NVP6324 驱动支持

如前文所述，i.MX8QXP 6 layer 开发板支持 MiPi CSI 桥连接 AHD Camera(i.MX8QXP MEK 板原生支持数字高清 MAX9286)，使用 Nextchip NVP6324 四个 AHD 输入桥，硬件设计如下：






以下简单说明驱动的情况，请与 NXP 支持窗口联系获得相关驱动源代码。

Drivers\media\platform\imx8\Kconfig

```
source "drivers/media/platform/imx8/nvp6324/Kconfig"
```

drivers/media/platform/imx8/nvp6324/Kconfig

```
config IMX8_NVP6324
        tristate "IMX8 NVP6324 Driver"
        depends on I2C
        default y //默认在内核中使能，所以不用在 defconfig 中增加定义
```

Drivers\media\platform\imx8\Makefile

```
obj-$(CONFIG_IMX8_NVP6324) += nvp6324/
```

drivers/media/platform/imx8/nvp6324/Makefile

```
obj-$(CONFIG_IMX8_NVP6324) += nvp6324.o
```

```
nvp6324-objs := nvp6324_core.o nvp6324_video.o nvp6324_video_eq.o nvp6324_mipi.o
```

DTS 配置如下：arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
&i2c_mipi_csi0 {
…
        /* 注掉
           max9286_mipi@6a {…
           };
       */
       //add
          nvp6324_mipi@30          {
                    compatible = "nextchip,nvp6324_mipi";
                    reg = <0x30>;
                    pinctrl-names = "default";
                    pinctrl-0 = <&pinctrl_mipi_csi0_gpio>;
                    clocks = <&clk IMX8QXP_CLK_DUMMY>;
                    clock-names = "capture_mclk";
                    mclk = <27000000>;
                    mclk_source = <0>;
                    pwn-gpios = <&gpio3 7 GPIO_ACTIVE_HIGH>;
                    virtual-channel;
                    status = "okay";
                    port {
                              nvp6324_0_ep: endpoint {
                              remote-endpoint = <&mipi_csi0_ep>;
                              data-lanes = <1 2 3 4>;
                              };
                    };
          };
       //end
};
…
&mipi_csi_0 {
…
          /* Camera 0  MIPI CSI-2 (CSIS0) */
```

```
            port@0 {
                        reg = <0>;
                        mipi_csi0_ep: endpoint {
                                    remote-endpoint = <&nvp6324_0_ep>;//改为 nvp6324 <&max9286_0_ep>;
                                    data-lanes = <1 2 3 4>;
                        };
            };
};dr
```

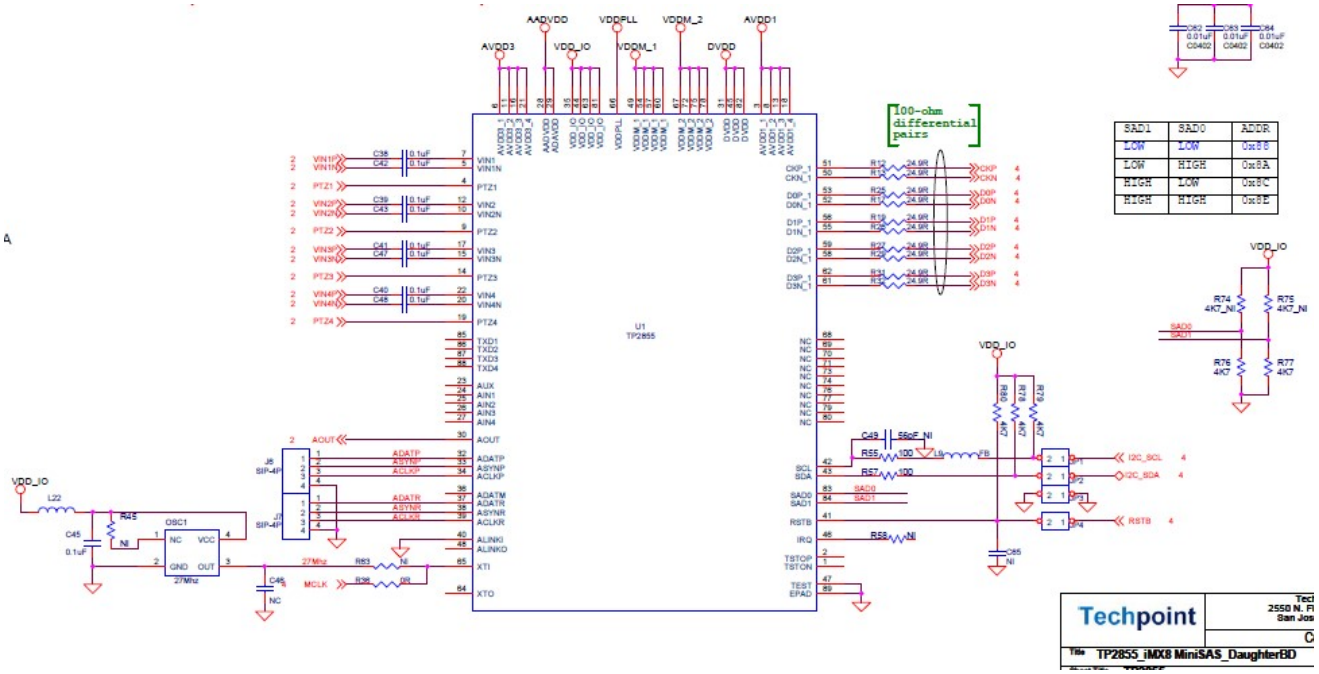
源代码中主要关注一下对桥芯片的 I2C 配置：

Drivers\media\platform\imx8\nvp6324\nvp6324_core.c

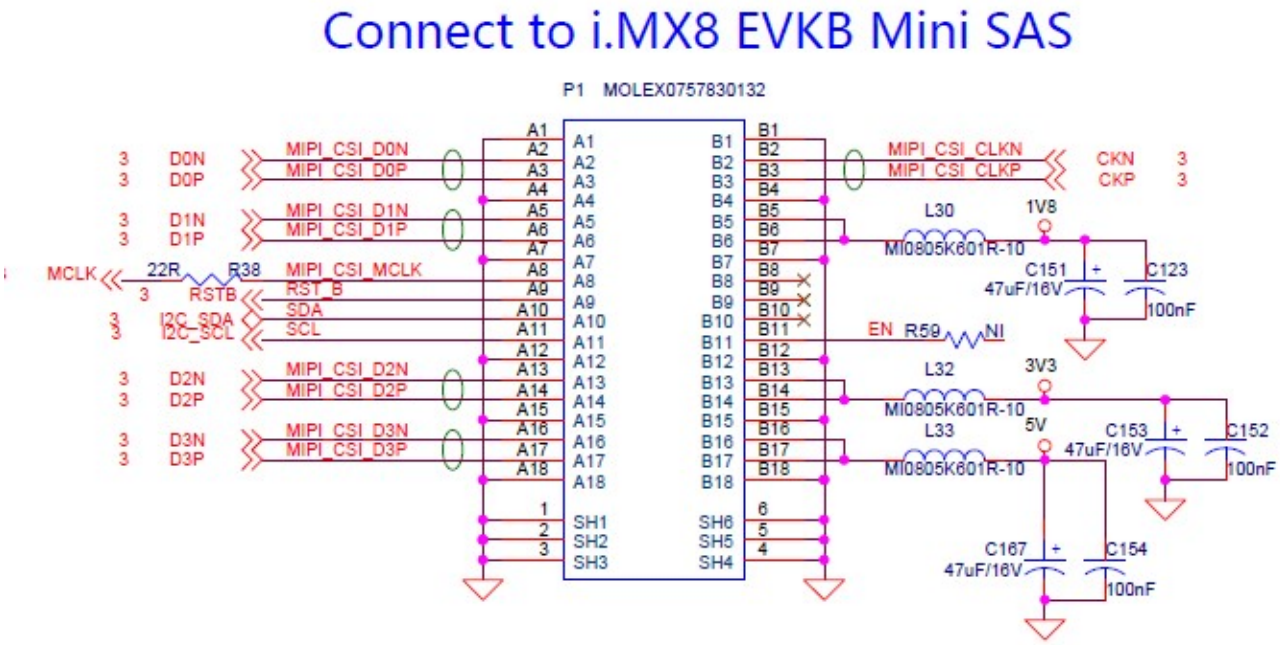
```
nvp6324_probe
|->nvp6324 使用外部 27Mhz 晶体，先从 DTS 中获得 MCLK 时钟值
|-> nvp6324_hw_reset //使用从 DTS 中 parser 过来的 gpio 来 reset 桥芯片
|->设置寄存器 bank,读 chip id
nvp6324_set_reg_bank(nvp6324, 0);
            ret = nvp6324_read_reg(nvp6324, 0xf4);
rev = nvp6324_read_reg(nvp6324, 0xf5);
|-> nvp6324_video_init
|    |-> nvp6324_hardware_preinit
//Pad Control Setting
// Clock Delay Setting
// MIPI_V_REG_OFF
// AGC_OFF  08.31
|    |->nvp6324_hardware_init
    /* configure differ or single input */
   /* vafe fsc common set */
```

### 1.10.6 AHD Camera In bridge TP2855 驱动支持

Techpoint 设计了一块 TP2855 子卡，可以用 miniSAS 线通过 MiPi CSI 桥连接到 i.MX8QXP MEK 板上(i.MX8QXP MEK 板原生支持数字高清 MAX9286)，硬件设计如下：




I2C 地址为 0x88 右移一位为 0x44。




以下简单说明驱动的情况，请与 NXP 支持窗口联系获得相关驱动源代码。

Drivers\media\platform\imx8\Kconfig

```
config Decoder  TP2855
        tristate "Techpoint tp2855 Video Decoder Input support"
```

```
          depends on I2C
          ---help---
        If you plan to use the tp2855 video decoder with your capture system, say Y here.
```

所以需要在 make menuconfig 中增加支持：

```
reconfig the kernel setting, you can run menuconfig to achieve this.

          Symbol: Decoder_TP2855 [=y]
          Type  : tristate
          Prompt: Techpoint tp2855 Video Decoder Input support
            Location:
      -> Device Drivers
        -> Multimedia support (MEDIA_SUPPORT [=y])
          -> V4L platform devices (V4L_PLATFORM_DRIVERS [=y])
            -> MX8 Video For Linux Video Capture (VIDEO_MX8_CAPTURE [=y])
  (1)         -> IMX8 Camera ISI/MIPI Features support
    Defined at drivers/media/platform/imx8/Kconfig:50
    Depends on: MEDIA_SUPPORT [=y] && V4L_PLATFORM_DRIVERS [=y] && VIDEO_MX8_CAPTURE [=y]
&& I2C [=y]
  set 'y' to Decoder_TP2855
  recompile the kernel and push it to the board.
```

Drivers\media\platform\imx8\Makefile

```
obj-$(CONFIG_Decoder_TP2855) += tp2855_mipi.o
```

DTS 配置如下：arch\arm64\boot\dts\freescale\imx8x-mek.dtsi

```
&i2c_mipi_csi0 {
…
        /* 注掉
          max9286_mipi@6a {…
          };
      */
      //add
          tp2855_mipi@44  {
                    compatible = "techpoint,tp2855_mipi";
                    reg = <0x44>;
                    pinctrl-names = "default";
                    pinctrl-0 = <&pinctrl_mipi_csi0>;
                    clocks = <&clk IMX8QXP_CLK_DUMMY>;
                    clock-names = "capture_mclk";
                    mclk = <27000000>;
                    mclk_source = <0>;
                    rst-gpios = <&gpio3 8 GPIO_ACTIVE_HIGH>;
                    virtual-channel;
                    status = "okay";
                    port {
```

```
                        tp2855_0_ep: endpoint {
                        remote-endpoint = <&mipi_csi0_ep>;
                        data-lanes = <1 2 3 4>;
                        };
                };
        };
    //end
};
…
&mipi_csi_0 {
…
        /* Camera 0  MIPI CSI-2 (CSIS0) */
        port@0 {
                reg = <0>;
                mipi_csi0_ep: endpoint {
                        remote-endpoint = <& tp2855_0_ep>;//改为 tp2855_0_ep <&max9286_0_ep>;
                        data-lanes = <1 2 3 4>;
                };
        };
};dr
```

源代码中主要关注一下对桥芯片的 I2C 配置：

Drivers\media\platform\imx8\tp2855.c

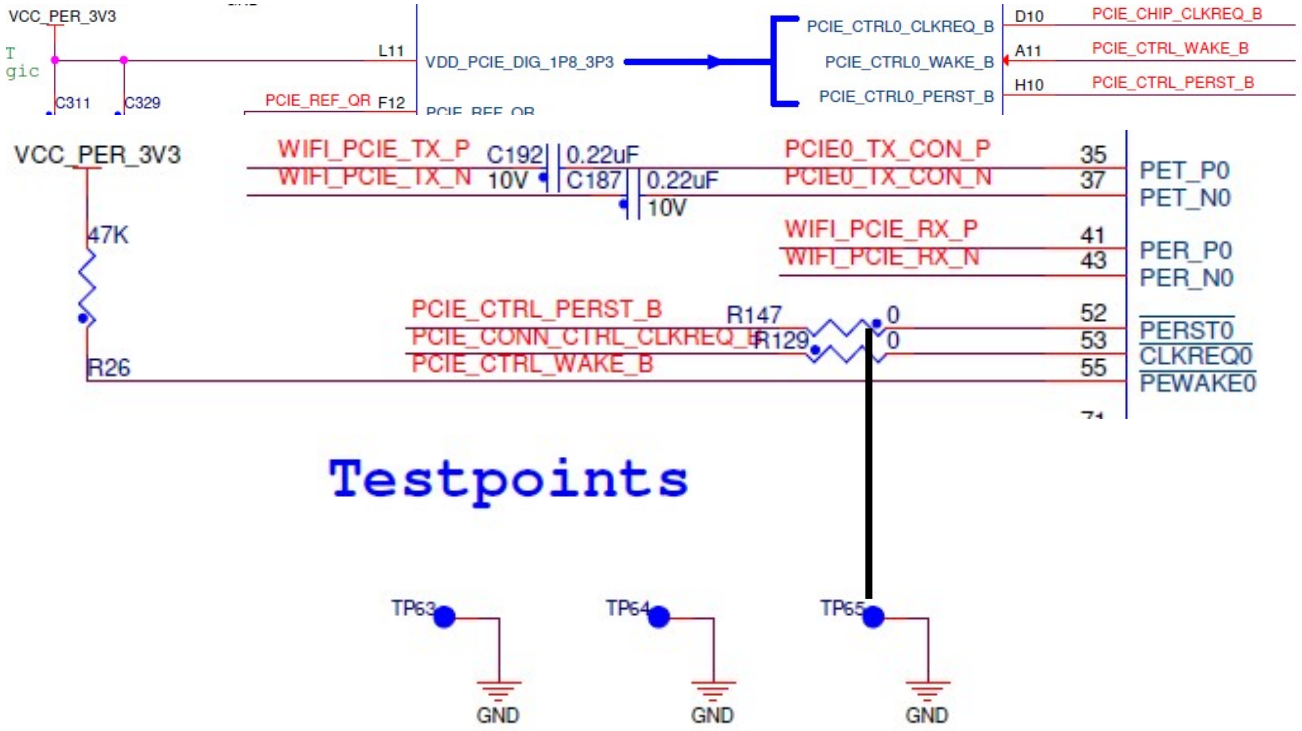
```
tp2855_probe
|-> tp2855 使用外部 27Mhz 晶体，先从 DTS 中获得 MCLK 时钟值
|-> tp2855_hw_reset //使用从 DTS 中 parser 过来的 gpio 来 reset 桥芯片
|->读 chip id
        chip_id_high = tp2855_read_reg(tp2855_data, 0xfe);
        chip_id_low = tp2855_read_reg(tp2855_data, 0xff);

        if (chip_id_high != 0x28 || chip_id_low != 0x55) {
                pr_warn("tp2855 is not found.\n");
                clk_disable_unprepare(tp2855_data->sensor_clk);
                devm_gpio_free(dev, tp2855_data->rst_gpio);
                return -ENODEV;
        }
|-> tp2855_common_init(tp2855_data);
|   |-> /* Disable MIPI CSI2 output */
|   |->tp2855_decoder_init
|   |->tp2855_mipi_out
```

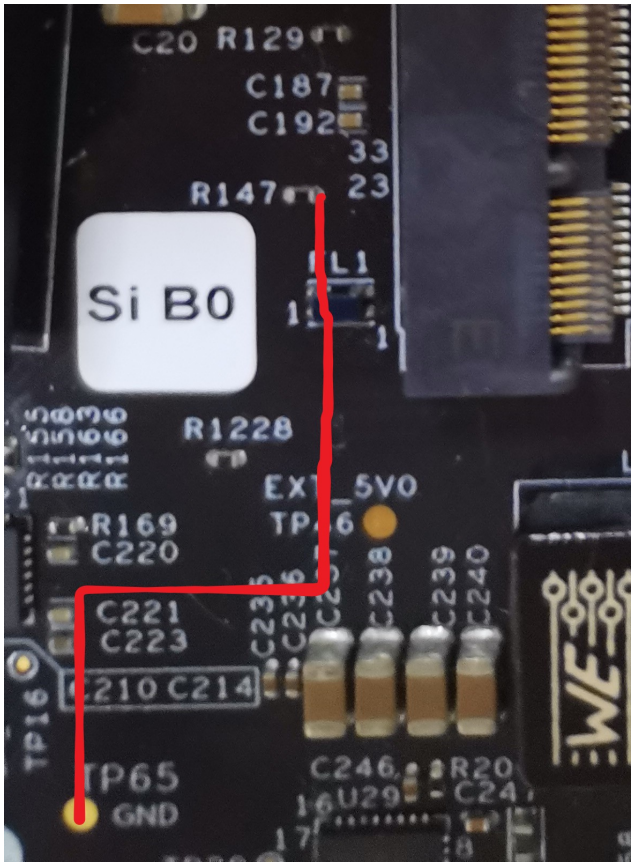
## 1.11 GPIO_Key 驱动定制

i.MX8QXP 的 BSP 目前并不支持 GPIO KEY,不过已经从 i.MX2X~i.MX6X 的 BSP 中有很多示例，可以参考加入支持 GPIO_Key 的驱动：GPIO_Key 驱动也可以用于应用程序监控某些 GPIO 信号，比如电源变化，外设变化等。

硬件 rework 如下，考虑将 PCIE_CTRL_PERST_B R147 的一端在按下按键是用模拟连接到地来实现：




Rework 照片如下：(注意这个测试电路没有限流电阻，请轻触既放)。




GPIO_KEY 驱动默认已经编译到内核中了，如下内核配置文件：

```
vi arch/arm/configs/imx_v7_defconfig
CONFIG_KEYBOARD_GPIO=y
```

GPIO_KEY 的驱动 Makefile 文件

```
vi drivers/input/keyboard/Makefile
obj-$(CONFIG_KEYBOARD_GPIO)             += gpio_keys.o
```

GPIO_LED 的驱动源代码文件

```
vi drivers/leds/gpio-keys.c
static const struct of_device_id gpio_keys_of_match[] = {
    { .compatible = "gpio-keys", },
    { },
};
```

GPIO_KEY 的驱动 binding 文件说明了如何在 DTS 中加上 GPIO_KEY 支持

```
vi Documentation/devicetree/bindings/input/gpio-keys.txt
```

i.MX8QXP MEK 板的支持如下：其中 linux,code 项为实际上报的 key_event 的键值。

```
gpio-keys {
        compatible = "gpio-keys";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_ pcieb>;
        home {
                label = "Home Button";
                gpios = <&gpio4 0 GPIO_ACTIVE_LOW>;
                gpio-key,wakeup;
                linux,code = <KEY_HOME>;
        };
  };
```

IOMUX 配置在仍保持 pcieb 的设置

```
pinctrl_pcieb: pcieagrp{
                fsl,pins = <
                        IMX8QXP_PCIE _CTRL0_PERST_B_LSIO_GPIO4_IO00          0x06000021
                        …
                >;
        };
```

然后去掉 PCIE 驱动：

```
&pcieb{
…
        status = "disabled";
};
```

源代码初始化过程如下：

```
gpio_keys_probe
|-> gpio_keys_get_devtree_pdata
|-> gpio_keys_setup_key
|    |-> gpio_request_one//申请 gpio
|    |-> gpio_to_irq//申请 gpio 中断，
//对于支持唤醒的 gpio,增加 IRQF_NO_SUSPENDflags
        if (bdata->button->wakeup)
                irqflags |= IRQF_NO_SUSPEND;
|-> input_register_device //注册 gpio 输入设备
|->
```

```
        for (i = 0; i < pdata->nbuttons; i++)
                gpio_keys_report_event(&ddata->data[i]); //调用 input_event 上报 gpio 事件
        input_sync(input);
```

所以目前的 bsp 对 gpio 按键已经做了很好的封装，增加一个 gpio key 只需要 2 步。

1. 在 dts 中设置 pin 的 iomux
2. 在 dts 中加入 key 的配置

驱动测试：

- 启动信息

```
input: gpio-keys as /devices/platform/gpio-keys/input/input6
```

- 测试的硬件连接

由于 GPIO_KEY 管脚是默认上拉，然后下拉触发，所以我们只要从相应 GPIO_KEY 管脚跳线到地，就可以测试 GPIO_KEY。

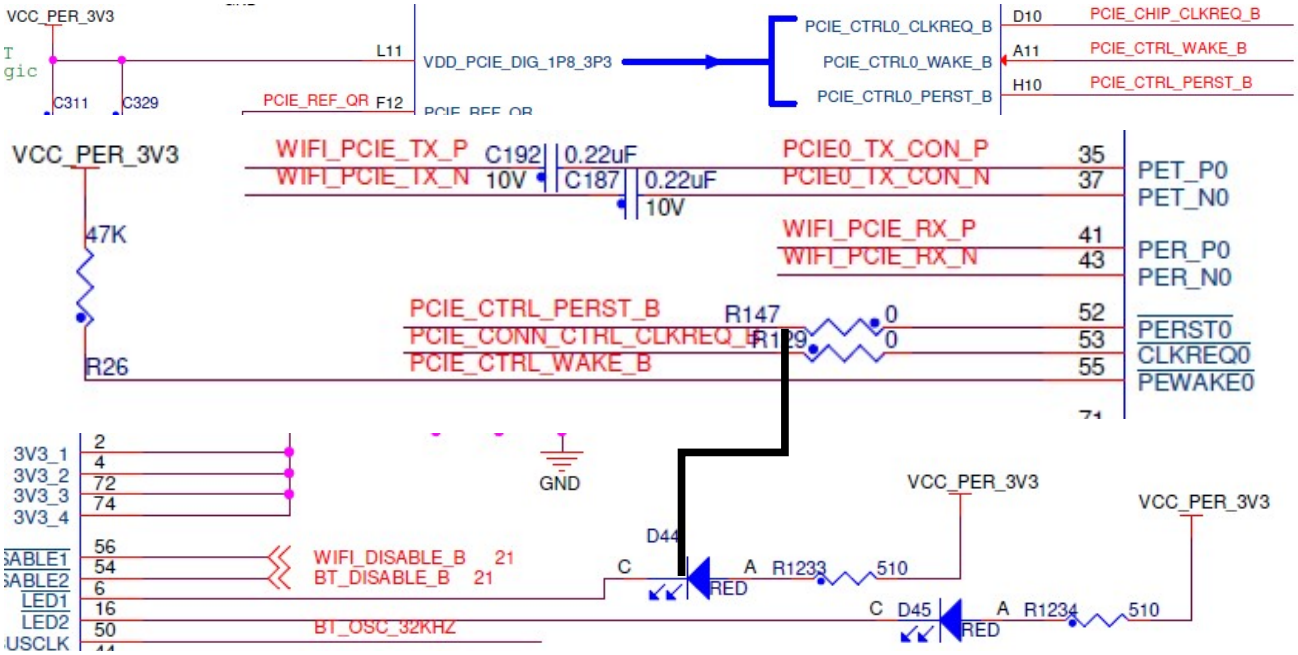
- 测试命令

```
root@imx8qxpmek:~# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
…
/dev/input/event5:      gpio-keys
Select the device event number [0-5]: 5
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "gpio-keys"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 102 (KEY_HOME)
Properties:
Testing ... (interrupt to exit)
    Event: time 1550694706.256007, type 1 (EV_KEY), code 102 (KEY_HOME), value 1
    Event: time 1550694706.256007, ---------- SYN_REPORT ------------
    Event: time 1550694706.275977, type 1 (EV_KEY), code 102 (KEY_HOME), value 0
    …
```

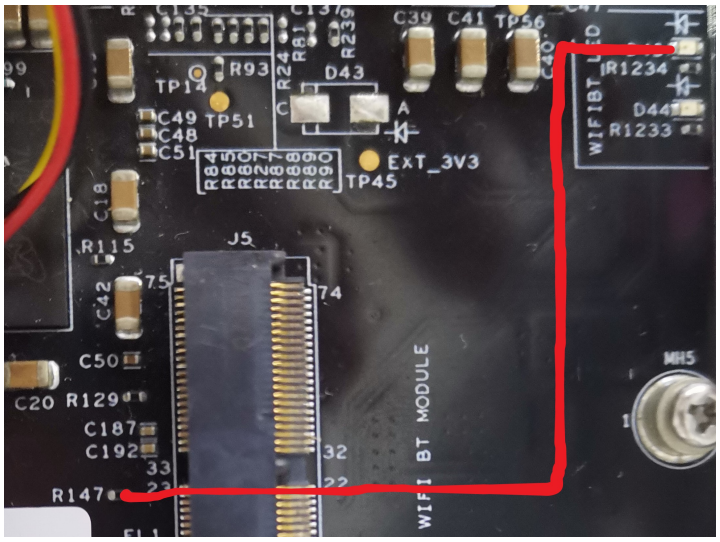
## 1.12 GPIO_LED 驱动定制

i.MX8QXP 的 BSP 本身不支持 GPIO_LED 的驱动，不过以前的 i.MX6X 支持，我们可以参考加入，硬件上考虑使用 PCIE 的 RESET GPIO 管脚来做为输出脚。GPIO_LED 驱动也可以用于应用程序控制某些 GPIO 输出信号。

硬件 rework 如下，考虑将 R147 的一端连接到 D44 的非 VCC_PER_3V3 一端：




Rework 照片如下：




GPIO_LED 驱动默认已经编译到内核中了，如下内核配置文件：

```
vi arch/arm64/configs/defconfig
CONFIG_LEDS_GPIO=y
```

GPIO_LED 的驱动 Makefile 文件

```
vi drivers/leds/Makefile
obj-$(CONFIG_LEDS_GPIO)                 += leds-gpio.o
```

GPIO_LED 的驱动源代码文件

```
vi drivers/leds/leds-gpio.c
static const struct of_device_id of_gpio_leds_match[] = {
    { .compatible = "gpio-leds", },
    {},
    };
```

GPIO_LED 的驱动 binding 文件说明了如何在 DTS 中加上 GPIO_LED 支持

```
vi Documentation/devicetree/bindings/leds/leds-gpio.txt
```

修改 i.MX8QXP MEK 板 DTS 如下：

```
        leds {
                compatible = "gpio-leds";
                pinctrl-names = "default";
                pinctrl-0 = <&pinctrl_pcieb>;
                led1: user1 {
                        label = "user1";
                        gpios = <&gpio4 0 GPIO_ACTIVE_LOW >;
                        default-state i2= "on";
                        linux,default-trigger = "heartbeat";
                };
        };
```

IOMUX 配置在仍保持 pcieb 的设置

```
pinctrl_pcieb: pcieagrp{
                        fsl,pins = <
                                IMX8QXP_PCIE_CTRL0_PERST_B_LSIO_GPIO4_IO00              0x06000021
                                …
                        >;
                };
```

然后去掉 PCIE 驱动：

```
&pcieb{
```

```
…
        status = "disabled";
};
```

源代码初始化过程如下：

```
gpio_led_probe
|-> create_gpio_led
|    |-> devm_gpio_request//申请 gpio
|    |-> gpio_direction_output//设置 gpio 为输出
```

所以目前的 bsp 对 gpio led 已经做了很好的封装，增加一个 gpio led 或 gpio 控制脚只需要 2 步。

1. 在 dts 中设置 pin 的 iomux
2. 在 dts 中加入 led 的配置

驱动测试：

- SYS 文件系统节点

```
root@imx8qxpmek:~# cd /sys/class/leds/
root@imx8qxpmek:/sys/class/leds# ls
mmc0::  mmc1::  user1
root@imx8qxpmek:/sys/class/leds# cd user1
root@imx8qxpmek:/sys/class/leds/user1# ls
brightness  device  invert  max_brightness  power  subsystem  trigger  uevent
```

- 测试命令

1. 测试开关

   LED 我们默认设置为 linux,default-trigger = "heartbeat"; 所以连接好后，LED 会使用 heatbeat 模式闪烁

   如果要测试开关，我们需要先将 trigger 设置为 default-on:

```
root@imx8qxpmek:/sys/class/leds/user2# cat trigger
none rc-feedback bluetooth-power kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shiftllock kbd-shiftrlock kbd-ctrllock kbd-ctrlrlock mmc0 mmc1 [heartbeat] cpu cpu0 cpu1 cpu2 cpu3 default-on
echo default-on > trigger
root@imx8qxpmek:/sys/class/leds/user2# cat trigger
none rc-feedback bluetooth-power kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shiftllock kbd-shiftrlock kbd-ctrllock kbd-ctrlrlock mmc0 mmc1 heartbeat cpu cpu0 cpu1 cpu2 cpu3 [default-on]
```

   然后设置其 brightess

```
root@imx8qxpmek:/sys/class/leds/user2# cat max_brightness
255
root@imx8qxpmek:/sys/class/leds/user2# cat brightness
255
root@imx8qxpmek:/sys/class/leds/user2# echo 0 > brightness
root@imx8qxpmek:/sys/class/leds/user2# cat brightness
0
```

就会关闭 LED。

## 1.13 Fuse nvram 驱动

与 i.MX6 不同，i.MX8 的 ocotp 驱动位于 nvmem 之下：

```
arch\arm64\boot\dts\freescale\fsl-imx8dx.dtsi
ocotp: ocotp {
…
                compatible = "fsl,imx8qxp-ocotp", "syscon";
        };
 drivers/nvmem/imx-scu-ocotp.c
 static const struct of_device_id imx_scu_ocotp_dt_ids[] = {
        { .compatible = "fsl,imx8qm-ocotp", (void *)&imx8qm_data },
        { .compatible = "fsl,imx8qxp-ocotp", (void *)&imx8qxp_data },
        { },
};
static struct nvmem_config imx_scu_ocotp_nvmem_config = {
        .name = "imx-ocotp",
        .read_only = true,
        .word_size = 4,
        .stride = 1,
        .owner = THIS_MODULE,
        .reg_read = imx_scu_ocotp_read,
};
static int imx_scu_ocotp_read(void *context, unsigned int offset,
                              void *val, size_t bytes)
{
  sciErr = sc_misc_otp_fuse_read(priv->nvmem_ipc, i, (u32 *)buf);  //通过 scu 读取 fuse.
```

}

所以只能通过 nvmem 接口访问 fuse,而且只能是只读，如下示例访问芯片 UID:

| | | | |
|---|---|---|---|
| 0x0900[7:0] | 16 | 0x000 | LOT_NO_ENC[7:0] ( SJC_CHALL[7:0] / UNIQUE_ID[7:0] ) |
| 0x0900[15:8] | 16 | 0x000 | LOT_NO_ENC[15:8] ( SJC_CHALL[15:8] / UNIQUE_ID[15:8] ) |
| 0x0900[23:16] | 16 | 0x000 | LOT_NO_ENC[23:16] ( SJC_CHALL[23:16] / UNIQUE_ID[23:16] ) |
| 0x0900[31:24] | 16 | 0x000 | LOT_NO_ENC[31:24] ( SJC_CHALL[31:24] / UNIQUE_ID[31:24] ) |
| 0x0910[7:0] | 17 | 0x001 | LOT_NO_ENC[39:32]( SJC_CHALL[39:32] / UNIQUE_ID[39:32] ) |
| 0x0910[15:8] | 17 | 0x001 | WAFER_NO[4:0] ( SJC_CHALL[47:43] / UNIQUE_ID[47:43] )/<br>LOT_NO_ENC[42:40]( SJC_CHALL/UNIQUE_ID[42:40] ) |
| 0x0910[23:16] | 17 | 0x001 | DIE-Y-CORDINATE[7:0] ( SJC_CHALL[55:48] / UNIQUE_ID[55:48] ) |
| 0x0910[31:24] | 17 | 0x001 | DIE-X-CORDINATE[7:0] ( SJC_CHALL[63:56] / UNIQUE_ID[63:56] ) |

16X4=64=0x40

```
hexdump /sys/bus/nvmem/devices/imx-ocotp0/nvmem
0000040 1a17 57ac 200b f568 0000 0000 0000 0000
```

这儿就是 UID.

简单的方法可以从下节点得到 UID:

```
p508:/sys/devices/soc0 # cat soc_uid
0c10100e828962c1
```

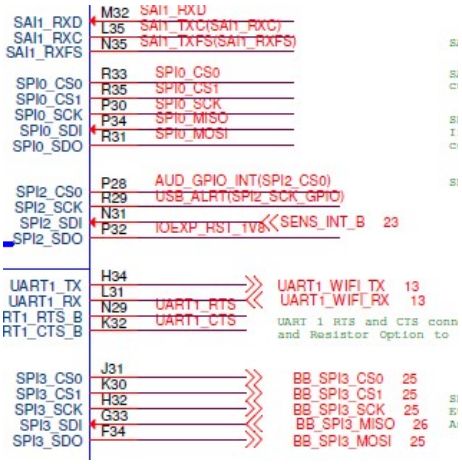
## 1.14 SPI 与 SPI Slave 驱动

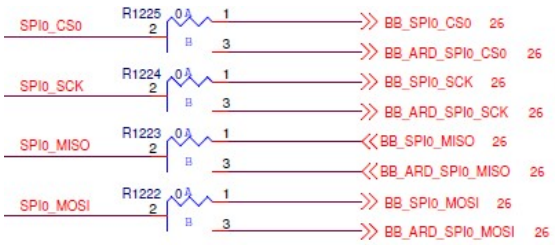
i.MX8QXP MEK 板 BSP 默认没有打开 SPI 接口，事实上在汽车的应用中，是有可能使用 SPI 接口来连接外设的，甚至有可能是 MCU 做主通过 SPI 接口来访问 i.MX8X,所以 i.MX8X 是做从的。所以如下说明在 i.MX8QXP MEK 板上如何集成 SPI Master 和 SPI Slave 的。

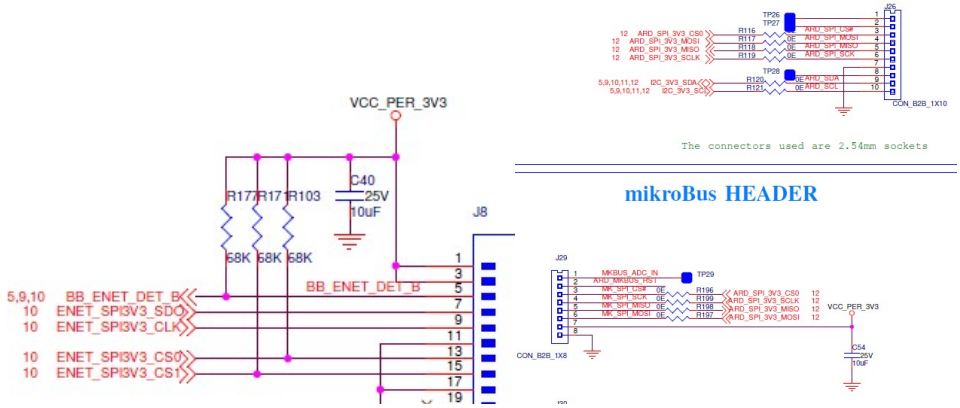
如下硬件设计，i.MX8QXP MEK 板+底板在 ARD/MKBus 上支持 SPI0(需要在 CPU 上将 BB_SPI0 跳线为 BB_ARD_SPI0), 我们将这个接口设计为 SPI Master。在 ENET CONN 上有 SPI3 接口，我们将这个接口设计为 SPI Slave。(注意因为 ENET CONN 跳线困难，所以 V1 完成截至硬件 rework 并没有做)。

**SPI TABLE**

| DEVICE | SPI(QM) | SPI(QXP) | I/O LEVEL FOR DEVICE |
|---|---|---|---|
| ENET CONN | SPI 1(1.8V) | SPI 3(1.8V) | 3.3V |
| ARD/MKBUS | SPI 2(1.8V) | SPI 0(1.8V) | 3.3V |
| AUD CONN | SPI 3(1.8V) | SPI 3(1.8V) | 1.8V |
| | SPI 0(1.8V) | SPI 0(1.8V) | 1.8V |




软件上可以参考 imx8qxp-lpddr4-val-lpspi.dts/imx8qxp-lpddr4-val-lpspi-slave.dts，如何集成 SPI:

arch/arm64/boot/dts/freescale/imx8x-mek.dtsi

```
pinctrl_lpspi0: lpspi0grp {
                        fsl,pins = <
                                IMX8QXP_SPI0_SCK_ADMA_SPI0_SCK          0x0600004c
                                IMX8QXP_SPI0_SDO_ADMA_SPI0_SDO          0x0600004c
                                IMX8QXP_SPI0_SDI_ADMA_SPI0_SDI          0x0600004c
                        >;
                };
```

```
//spi0 配置为使用 gpio 管脚来接 CS 的模式，注意有一个 SPI 外设，比如 SPI-NOR，它要求在整个 message 传输过程中 CS 都是低有效的，这种情况下使用 IP 的 CS 管脚不支持，所以必须要用 gpio 来做片选。后文详细解释
                pinctrl_lpspi0_cs: lpspi0cs {
                        fsl,pins = <
                                IMX8QXP_SPI0_CS0_LSIO_GPIO1_IO08        0x21
                        >;
                };
        pinctrl_lpspi2: lpspi2grp {
                fsl,pins = <
                        IMX8QXP_SPI2_SCK_ADMA_SPI2_SCK          0x600004c
                        IMX8QXP_SPI2_SDO_ADMA_SPI2_SDO          0x600004c
                        IMX8QXP_SPI2_SDI_ADMA_SPI2_SDI          0x600004c
                        IMX8QXP_SPI2_CS0_ADMA_SPI2_CS0          0x600004c
                >;
        };
                pinctrl_lpspi3: lpspi3grp {
                        fsl,pins = <
                                IMX8QXP_SPI3_SCK_ADMA_SPI3_SCK          0x0600004c
                                IMX8QXP_SPI3_SDO_ADMA_SPI3_SDO          0x0600004c
                                IMX8QXP_SPI3_SDI_ADMA_SPI3_SDI          0x0600004c
                                IMX8QXP_SPI3_CS0_ADMA_SPI3_CS0          0x0600004c
                        >;
                };
…

&lpspi0 {
        fsl,spi-num-chipselects = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpspi0 &pinctrl_lpspi0_cs>;
        cs-gpios = <&lsio_gpio1 8 GPIO_ACTIVE_LOW>; //使用 gpio 做片选
        status = "okay";

        spidev0: spi@0 { //设置一个 spidev 用户空间设备，方便访问调试
                reg = <0>;
                compatible = "rohm,dh2228fv";
                spi-max-frequency = <30000000>;
```

```
        };

/*
        flash: at45db041e@0 {
…
        };
*/
};
//spi2 不用 gpio 作片选
&lpspi2 {
        fsl,spi-num-chipselects = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpspi2>;
        status = "okay";

        spidev0: spi@0 {
                reg = <0>;
                compatible = "rohm,dh2228fv";
                spi-max-frequency = <10000000>;
        };
};

&lpspi3 {
        fsl,spi-num-chipselects = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpspi3>;
        spi-slave; //spi3 工作在 slave 模式
        status = "okay";
};
```

设备节点：

```
root@imx8qxpmek:/dev# pwd
/dev
root@imx8qxpmek:/dev# ls *spi*
spidev0.0 //这个是 spi 主
```

echo spidev > /sys/class/spi_slave/spi1/slave //执行此命令后，出现了 spidev1.0 从

```
root@imx8qxpmek:/dev# ls *spi*
spidev0.0  spidev1.0
```

编译测试程序：

```
source ~ /imx-yocto-bsp/imx8qxpmek_xwayland/sdk/environment-setup-aarch64-poky-linux
pwd
~/imx-yocto-bsp/standalone/linux-imx/tools/spi
make
…
LINK     spidev_test
```

测试方法：(待验证)

硬件连接 SPI0 主/SPI3 从.

- 运行程序：

slave 发：./spidev_test -D /dev/spidev1.0 -p slave-hello-to-master -v //这个时候 spi slave 会阻塞在这儿，等待 spi master 提供时钟过去

```
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
…
```

master 发：./spidev_test -D /dev/spidev0.0 -p master-hello-to-slave -v //spl master 发送数据给 spl slave，同时提供了时钟给 spl slave, spl slave 就将数据送过来。

```
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
TX | 6D 61 73 74 65 72 2D 68 65 6C 6C 6F 2D 74 6F 2D 73 6C 61 76 65 __ __ __ __ __ __ __ __ __ __ __ |
master-hello-to-slave
RX | …
```

最后解释一下 CS 的配置情况：

- 如果要求在每个 transfer 的多个 word 发送过程中，要求 CS 保持低有效，这个是目前 BSP 的默认做法。
- 如果要求在每个 transfer 的多个 word 发送过程中，没有发送 word 时 CS 为高，需要加如下 patch 去掉 CONT 功能：

```
diff --git a/drivers/spi/spi-fsl-lpspi.c b/drivers/spi/spi-fsl-lpspi.c
index 7d10e566911f..65ca0a5d3d04 100644
--- a/drivers/spi/spi-fsl-lpspi.c
+++ b/drivers/spi/spi-fsl-lpspi.c
```

```
@@ -188,7 +188,6 @@ static int lpspi_unprepare_xfer_hardware(struct spi_controller *controller)
 static void fsl_lpspi_write_tx_fifo(struct fsl_lpspi_data *fsl_lpspi)
 {
         u8 txfifo_cnt;
-        u32 temp;

         txfifo_cnt = readl(fsl_lpspi->base + IMX7ULP_FSR) & 0xff;

@@ -200,12 +199,6 @@ static void fsl_lpspi_write_tx_fifo(struct fsl_lpspi_data *fsl_lpspi)
         }

         if (txfifo_cnt < fsl_lpspi->txfifosize) {
-                if (!fsl_lpspi->is_slave) {
-                        temp = readl(fsl_lpspi->base + IMX7ULP_TCR);
-                        temp &= ~TCR_CONTC;
-                        writel(temp, fsl_lpspi->base + IMX7ULP_TCR);
-                }
-
                 fsl_lpspi_intctrl(fsl_lpspi, IER_FCIE);
         } else
                 fsl_lpspi_intctrl(fsl_lpspi, IER_TDIE);
@@ -227,17 +220,6 @@ static void fsl_lpspi_set_cmd(struct fsl_lpspi_data *fsl_lpspi,
                 temp |= fsl_lpspi->config.prescale << 27;
                 temp |= (fsl_lpspi->config.mode & 0x3) << 30;
                 temp |= (fsl_lpspi->config.chip_select & 0x3) << 24;
-
-                /*
-                 * Set TCR_CONT will keep SS asserted after current transfer.
-                 * For the first transfer, clear TCR_CONTC to assert SS.
-                 * For subsequent transfer, set TCR_CONTC to keep SS asserted.
-                 */
-                if (!fsl_lpspi->usedma) {
-                        temp |= TCR_CONT;
-                        if (fsl_lpspi->is_first_byte)
-                                temp &= ~TCR_CONTC;
-                        else
-                                temp |= TCR_CONTC;
-                }
```

如果如同SPI-NOR一样，要求整个message发送过程中，有多个transfer的情况下,CS要一直保持低有效，则使用IP的CS管脚没有办法做到，只能修改为使用GPIO来做CS管脚。

- 目前 SPI0 速度限制在 10M,如果要升高，注意下这儿要改（SPI2/3 为 30M）：

Arch\arm64\boot\dts\freescale\ imx8-ss-dma.dtsi

```
    lpspi0: lpspi@5a000000 {
        compatible = "fsl,imx7ulp-spi";
…
        assigned-clock-rates = <20000000>;
…
    };
```

assigned-clock-rates = <20000000>;

最大可以改到 120Mhz.

spi-max-frequency 没有用，驱动没有 parser.

这儿是实际使用速度的 2 倍，lpspi 可以到 60Mhz.

总结就是需要：

1: 这儿的 assigned-clock-rates = <20000000>;改成需要速度的两倍，最高可以到 120Mhz.

2: 调用 spi_dev 设备节点时设置速度参数，最大可以到 60Mhz.

- 如果是使用 GPIO 支持多片选，需要增加以下 patch:

```
Why it does not work at the moment:
- num_chipselect sets the number of cs-gpios that are in the DT.
  This comes from drivers/spi/spi.c
- num_chipselect gets set with devm_spi_register_controller, that is
  called in drivers/spi/spi.c
- devm_spi_register_controller got called after num_chipselect has
  been used.
How this commit fixes the issue:
- devm_spi_register_controller gets called before num_chipselect is
  being used.
diff --git a/drivers/spi/spi-fsl-lpspi.c b/drivers/spi/spi-fsl-lpspi.c
index 2cc0ddb4a988..1375bdfc587b 100644
--- a/drivers/spi/spi-fsl-lpspi.c
+++ b/drivers/spi/spi-fsl-lpspi.c
@@ -862,6 +862,22 @@  static int fsl_lpspi_probe(struct platform_device *pdev)
         fsl_lpspi->dev = &pdev->dev;
         fsl_lpspi->is_slave = is_slave;

+       controller->bits_per_word_mask = SPI_BPW_RANGE_MASK(8, 32);
+       controller->transfer_one = fsl_lpspi_transfer_one;
+       controller->prepare_transfer_hardware = lpspi_prepare_xfer_hardware;
+       controller->unprepare_transfer_hardware = lpspi_unprepare_xfer_hardware;
+       controller->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
+       controller->flags = SPI_MASTER_MUST_RX | SPI_MASTER_MUST_TX;
```

```
+	controller->dev.of_node = pdev->dev.of_node;
+	controller->bus_num = pdev->id;
+	controller->slave_abort = fsl_lpspi_slave_abort;
+
+	ret = devm_spi_register_controller(&pdev->dev, controller);
+	if (ret < 0) {
+		dev_err(&pdev->dev, "spi_register_controller error.\n");
+		goto out_controller_put;
+	}
+
	if (!fsl_lpspi->is_slave) {
		for (i = 0; i < controller->num_chipselect; i++) {
			int cs_gpio = of_get_named_gpio(np, "cs-gpios", i);
@@ -885,16 +901,6 @@  static int fsl_lpspi_probe(struct platform_device *pdev)
		controller->prepare_message = fsl_lpspi_prepare_message;
	}

-	controller->bits_per_word_mask = SPI_BPW_RANGE_MASK(8, 32);
-	controller->transfer_one = fsl_lpspi_transfer_one;
-	controller->prepare_transfer_hardware = lpspi_prepare_xfer_hardware;
-	controller->unprepare_transfer_hardware = lpspi_unprepare_xfer_hardware;
-	controller->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
-	controller->flags = SPI_MASTER_MUST_RX | SPI_MASTER_MUST_TX;
-	controller->dev.of_node = pdev->dev.of_node;
-	controller->bus_num = pdev->id;
-	controller->slave_abort = fsl_lpspi_slave_abort;
-
	init_completion(&fsl_lpspi->xfer_done);

	res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
@@ -952,12 +958,6 @@  static int fsl_lpspi_probe(struct platform_device *pdev)
	if (ret < 0)
		dev_err(&pdev->dev, "dma setup error %d, use pio\n", ret);
```

```
-       ret = devm_spi_register_controller(&pdev->dev, controller);
-       if (ret < 0) {
-               dev_err(&pdev->dev, "spi_register_controller error.\n");
-               goto out_controller_put;
-       }
-
        return 0;
```

## 1.15 USB 3.0 TypeC 改成 USB 3.0 TypeA(未验证)

在我们 i.MX8QXP MEK 开发板设计中。USB_OTG1 是一个 USB2.0,连接到底板上的 USB OTG 口上，而 USB_OTG2/SS3 是一个 USB3.0 口，是连接到一个 Type C 接口上的，而在大多数汽车产品设计中，可能不会使用 Type C 接口中，而还是连接为 Type A 接口的。所以如下软件修改：

Arch/arm64/boot/dts/freescale/imx8x-mek.dtsi:

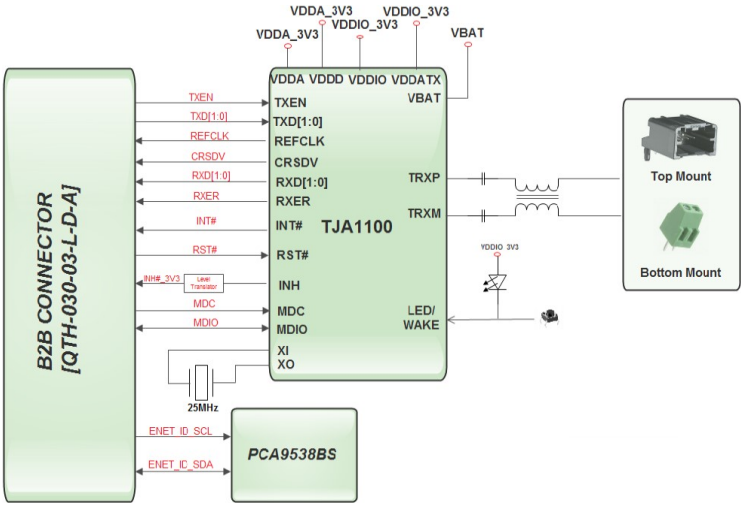
```
&usbotg3 {
        dr_mode ="peripheral"; // "otg"; 修改为 peripheral 模式，可用于 UUU 下载
        // extcon = <&typec_ptn5110>; 去掉 PTN5110 支持
        status = "okay";
};
        typec_ptn5110: typec@50 {
        …
                status = "disabled";
        };
```

注意因为 i.MX8QXP MEK 板设计中，USB_OTG2_ID 并没有连接出来，所以 dr_mode 不能设置为 otg;如果是 peripheral 可以用于 UUU 下载，如果是 host 请确认 USB Host 有供电，如果要使用 otg 请确认如同 USB_OTG1 一样，有 ID 管脚切换电源的设计。

## 1.16 汽车级以太网驱动定制

### 1.16.1 TJA1100

i.MX8QXP MEK 板的 CPU 板上通过 ENET0 连接的是一个 1G 的通用以太网 PHY，AR8031。另外，通过 ENET1 连接到底板上，再连接到一个 NXP 的 TJA1100 汽车级以太网 PHY。




DTS 配置为：

```
//板级层，从 imx8x-mek.dtsi/ imx8qxp-mek.dts/imx8qxp-mek-enet2.dts/
imx8qxp-enet2-tja1100.dtsi/imx8qxp-mek-enet2-tja1100.dts/中得到：
//iomux 冲突，去掉
&fec1 {
    status = "disabled";
};
&fec2 {
    pinctrl-0 = <&pinctrl_fec2_rmii>;
    clocks = <&enet1_lpcg 4>, \\ 100Mhz 网的 clock tree
             <&enet1_lpcg 2>,
             <&clk IMX_SC_R_ENET_1 IMX_SC_C_DISABLE_50>,
             <&enet1_lpcg 0>,
             <&enet1_lpcg 1>;
    phy-mode = "rmii"; //这个表示是一个 100M 网
    phy-handle = <&ethphy2>;  //以太网接口
    /delete-property/ phy-supply; //没有电源控制
    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy2: ethernet-phy@5 {
            compatible = "ethernet-phy-ieee802.3-c22";
```

```
                        reg = <5>;
                        tja110x,refclk_in;
                };
        };
};

&iomuxc {
                pinctrl_fec2_rmii: fec2rmiigrp {
                        fsl,pins = <
                                IMX8QXP_ _ENET0_MDC_CONN_ENET1_MDC                0x06000020
…//使用 RMII 接口 PIN，收发各两根数据线
                                IMX8QXP_ _ESAI0_SCKR_CONN_ENET1_RGMII_TX_CTL      0x06000020
                        >;
                };
};
```

源代码是：

```
Arch/arm64/configs/defconfg
CONFIG_NXP_TJA110X_PHY=y
/drivers/net/phy
Makefile
obj-$(CONFIG_NXP_TJA110X_PHY) += tja110x.o
tja110x.c/h
/* PHY IDs */
#define NXP_PHY_ID_TJA1100      (0x0180DC40U)
static struct phy_driver nxp_drivers[] = {
{
    .phy_id = NXP_PHY_ID_TJA1100,
    .name = "TJA1100",
    .phy_id_mask = NXP_PHY_ID_MASK,
    .features = (SUPPORTED_TP | SUPPORTED_MII | SUPPORTED_100BASET1_FULL),
    .flags = 0,
    .probe = &nxp_probe,
    …
    .read_status = &genphy_read_status,
…
```

```
}
static int nxp_probe(struct phy_device *phydev)
{
…
    if (of_property_read_bool(dev->of_node, "tja110x,refclk_in"))
            nxp_specific->quirks |= TJA110X_REFCLK_IN;
```

所以如果是连接 100Mhz 的汽车级以太网 PHY，建议选用 NXP TJA11XX 系列，已经有 Linux 驱动支持并且在 i.MX 平台上验证过，会持续维护。

**注意一下：**

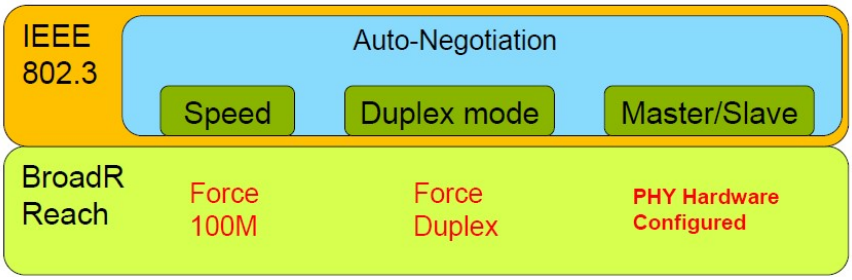
1. drivers/net/phy/tja110x.h 中定义了：#define NXP_PHY_ID_TJA1102P1 (0x00000000U)，这个 TJA1102 是一个双路 PHY 芯片，但是他的第二个 PHY 的 ID 默认设置成了 0x00。这样会导致一个问题，就是如果因为某些硬件原因导致了 mdio 访问 PHY 失败，get_phy_id 函数会返回 phy_id=0x00。这个时候理论上应该是找不到 PHY，从而不能加载相应 PHY 驱动，但是因为 TJA1102P1 的 PHY ID 刚好设置成了 0x00。所以错误的加载了此驱动：

```
24.045570] TJA1102_p1 5b050000.ethernet-1:02: power mode transition failed
[   24.052818] TJA1102_p1 5b050000.ethernet-1:02: attached PHY driver [TJA1102_p1]
(mii_bus:phy_addr=5b050000.ethernet-1:02, irq=POLL)
```

   这个会导致误认为 mdio 已经工作，实际上是一个误会，此点需要注意。

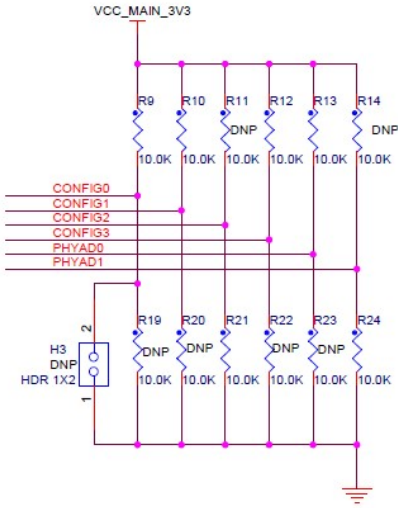
2. tja1101 是一颗 100BASE_T1 规范的汽车级以太网 PHY，与普通以太网相比，除了使用的是汽车级双绞线之外，一个重要的区别是没有自协商功能，所以需要硬件配置，如果是两块板互连，则要一个 master 一个 slave，如下：

Does not use IEEE 802.3 style auto-negotiation due to associated latency that does not meet start-up time requirements of automotive networks.

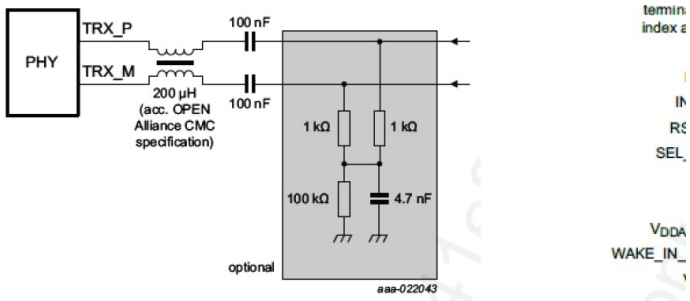



**Hardware Strapping Options**

| Pin Name | Config | Selected Mode |
|---|---|---|
| CONFIG0 | HIGH | PHY configured as Master |
| CONFIG1 | HIGH | Autonomous operation |
| CONFIG2 | LOW | RMII mode (25 MHz crystal) |
| CONFIG3 | HIGH | |
| PHYAD0 | HIGH | PHY address bits 0 & 1 |
| PHYAD1 | LOW | |




3. 100Base-T1 本身的 MDI 接口是高速差分信号线，所以线上的容性负载不能太大，要求如下：




Common mode choke： 请选择适用于100Base-T1的choke

100 nF AC-coupling 电容： **Tolerance** ≤ 10% ； **Voltage range** ≥ 50V

1K Resistor: Tolerance ≤ 1% ；Power rating ≥ 0.4W
100K Resistor: Tolerance ≤ 1% ；Power rating ≥ 0.1W
4.7nF Cap: **Tolerance** ≤ 10% ； **Voltage range** ≥ 50V

4. i.MX8QXP MEK+TJA1101 目前是只支持 fec2, fec1 disabled 的，因为它使用 fec2 的 mdio 来访问 PHY，而这个 mdio 与 fec1 的 mdio 共用一个管脚，所以有 iomux 冲突，如果是想两个 PHY 都可以工作，解决方式可以参考 i.MX8QM MEK+TJA1101 的方法，就是用 fec1 的 mdio 接口来访问两个 PHY，如下：

```
                pinctrl_fec2_rmii: fec2rmiigrp {
                        fsl,pins = <
..
                //IMX8QXP _ENET0_MDC_CONN_ENET1_MDC                0x06000020
                //IMX8QXP__ENET0_MDIO_CONN_ENET1_MDIO
        0x06000020
..
                        >;
                };
&fec1 {
        pinctrl-names = "default";
…
        phy-handle = <&ethphy0>;
…
        status = "okay";
        mdio {
                #address-cells = <1>;
                #size-cells = <0>;
                ethphy0: ethernet-phy@3 {//keep fec1 phy mdio handler
                        compatible = "ethernet-phy-ieee802.3-c22";
                        reg = <3>;
                };
                ethphy2: ethernet-phy@2 {//add fec2 phy mdio handler
                        compatible = "ethernet-phy-ieee802.3-c22";
                        reg = <2>;
                };
        };
};

&fec2 {
        pinctrl-names = "default";
…
```

```
            phy-handle = <&ethphy2>;
…
            phy-reset-duration = <100>;
//remove which mdio setting.
            status = "okay";
};
```

5. 如以下特殊情况，marvell C45 PHY 使用了 C22 接口来模拟 C45 PHY,从而修改了 Linux 内核原生驱动：drivers/net/phy/phy_device.c get_phy_id 函数，所以如果是即要支持标准 C22 PHY，又要支持这种特殊的 C22 模拟 C45 PHY，那对这个函数就要做修改，比较粗糙直接的办法是可以用 PHY_ADD 来区别两个 PHY，分别调用不同的 get_phy_id 实现，具体 marvell C45 PHY 使用了 C22 接口来模拟 C45 PHY 的修改方法请参考下文。

### 1.16.2 Marvell 88Q2110/2

以下以 Marvell 88Q2110/2 为例说明如果开发一颗新的 PHY 芯片，这是一颗 1G 的汽车以太网 PHY,目前并没有驱动支持。

支持一颗新的以太网 PHY 的步骤是：

1. 确认一下硬件连接，配置相应的 IOMUX。
2. 确认电源情况，配置相应的 regulator。
3. 确认 reset pin，配置相应的 gpio reset 接口。
4. 确认一下 clock source, RGMII 应该是谁发谁送 clock。rgmii tx clock 是 controller 发， rx clock 是 phy 发。
5. 如果 iomux,电源，reset, clock 正常，这个时候 PHY 应该可以工作了，就可能通过 MDIO 接口读到 PHY 的 ID 号(读到 PHY 的 ID 号是判断 PHY 是否工作的重要标志)。
6. 到这个时候，对于 C22 的 PHY，会已经找到一个 generic 的以太网 PHY 驱动，接下来一步就是通过厂商名在/driver/net/phy 下面找到此厂商的标准 PHY 驱动，然后在此驱动文件中通过 PHY ID 来注册 phy_driver 函数数组，最后再根据 PHY 芯片数据手册，完善此函数数组中的每个函数。一般来讲，C22 的 PHY 的寄存器定义是相同或相似的，C22 的驱动文件也相对比较全。

fsl-fec 的驱动 binding 文件在：documentation/devicetree/bindings/net/fsl-fec.txt

- phy-mode: operation mode of the PHY interface

* "rmii"

* "rgmii" (RX and TX delays are added by the MAC when required)

* "rgmii-id" (RGMII with internal RX and TX delays provided by the PHY, the MAC should not add the RX or TX delays in this case)

* "rgmii-rxid" (RGMII with internal RX delay provided by the PHY, the MAC should not add an RX delay in this case)

* "rgmii-txid" (RGMII with internal TX delay provided by the PHY, the MAC should not add an TX delay in this case)

- phy-reset-gpios : Should specify the gpio for phy reset

phy-reset-duration : Reset duration in milliseconds

phy-reset-active-high:注意驱动 PHY 是 reset 电路是低 reset 还是高 reset

- phy-supply : regulator that powers the Ethernet PHY.

phy-handle : phandle to the PHY device connected to this device.

fixed-link: //一些以太网的交换机配置可能用到, PHY 都是自动协商的。

fixed-link { speed = <1000>;

full-duplex; };

fsl,num-tx-queues : The property is valid for enet-avb IP, which supports hw multi queues. Should specify the tx queue number, otherwise set tx queue number to 1.

fsl,num-rx-queues : The property is valid for enet-avb IP, which supports hw multi queues. Should specify the rx queue number, otherwise set rx queue number to 1.

fsl,magic-packet : If present, indicates that the hardware supports waking up via magic packet.

fsl,wakeup_irq : The property define the wakeup irq index in enet irq source

Optional subnodes:

- mdio : specifies the mdio bus in the FEC, used as a container for phy nodes according to phy.txt in the same directory

1. 如下为 iomux 配置：

```
pinctrl_fec1: fec1grp {
                        fsl,pins = <
                    IMX8QXP_MIPI_CSI0_GPIO0_00_MIPI_CSI0_GPIO0_IO00                 0x21 //reset 使用此GPIO来连接PHY的Reset管脚。
                    IMX8QXP_MIPI_CSI0_I2C0_SDA_LSIO_GPIO3_IO06                  0x21 //power 使用此GPIO来给PHY芯片的3.3V供电
                                IMX8QXP_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB0_PAD   0x000014a0
                                IMX8QXP_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB1_PAD   0x000014a0
                                IMX8QXP_ ENET0_MDC_CONN_ENET0_MDC               0x06000020
….//此处为标准ENET0的IOMUX, 1G收发各四根数据线
                                IMX8QXP_ENET0_RGMII_RXD3_CONN_ENET0_RGMII_RXD3  0x00000061
                        >;
                };
```

2. 供电控制 GPIO 电源设置如下：

```
        reg_fec_supply: fec_nvcc {
                compatible = "regulator-fixed";
                regulator-name = "fec_nvcc";
                regulator-min-microvolt = <3300000>;
                regulator-max-microvolt = <3300000>;
                gpio = <&gpio3 6 GPIO_ACTIVE_HIGH>;
                enable-active-high;
                vin-supply = <&reg_fec_wake>;
        };
```

3. fec1 的板级 dts 设置如下：

```
&fec1 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_fec1>;
    phy-mode = "rgmii-txid";
    phy-handle = <&ethphy0>;
    phy-supply = <&reg_fec_supply>; //电源gpio
     phy-reset-gpios =<&gpio0 0 GPIO_ACTIVE_LOW>; //reset gpio
     phy-reset-duration = <100>;
    fsl,magic-packet;
     fsl,rgmii_rxc_dly;
    status = "okay";
    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy0: ethernet-phy@3 { //phy address硬件配置为3
                compatible = "ethernet-phy-ieee802.3-c22"; //表明此是一颗C22的PHY
                reg = <3>;
        };
    };
};
```

4. fec/phy 驱动初始化流程如下：

```
drivers/net/ethernet/freescale/fec_main.c
fec_probe
|->
        if (of_get_property(np, "fsl,magic-packet", NULL))
                fep->wol_flag |= FEC_WOL_HAS_MAGIC_PACKET;
        if (of_get_property(np, "fsl,rgmii_txc_dly", NULL))
                fep->rgmii_txc_dly = true;
        if (of_get_property(np, "fsl,rgmii_rxc_dly", NULL))
                fep->rgmii_rxc_dly = true;
        phy_node = of_parse_phandle(np, "phy-handle", 0);
of_phy_is_fixed_link
of_phy_register_fixed_link
of_get_phy_mode
```

```
|->fec_reset_phy //use gpio to reset the phy
 of_property_read_u32(np, "phy-reset-duration", &msec); //sample phy-reset-duration = <10>; //从 dts 中读出 reset gpio 并 reset phy
 of_get_named_gpio(np, "phy-reset-gpios", 0) //sample:phy-reset-gpios = <&gpio1 25 GPIO_ACTIVE_LOW>;
|->fec_enet_init
| |->fec_get_mac //此处是如何获得 MAC 地址 的方式，我们的 i.MX8QXP MEK 开发板是从 fuse 中读取的，客户在量产是需要烧写 MAC 地址 fuse
1) module parameter via kernel command line in form
        *    fec.macaddr=0x00,0x04,0x9f,0x01,0x30,0xe0
2) from device tree data
3) from flash or fuse (via platform data)
4) FEC mac registers set by bootloader
5) random mac address
netdev_err(ndev, "Invalid MAC address: %pM\n", iap);
netdev_info(ndev, "Using random MAC address: %pM\n",
                                ndev->dev_addr);
| |->fec_set_mac_address
|->fec_enet_mii_init
| |->of_mdiobus_register
| | |->of_mdiobus_register_phy
| | | |->get_phy_device
| | | | |->get_phy_id
//加打印看看是否读到正确的 phyid
| | | | |->phy_device_create
| | | |->phy_device_register
//如果有 fixup 函数，则执行
        /* Run all of the fixups for this PHY */
        err = phy_scan_fixups(phydev);
        if (err) {
                pr_err("PHY %d failed to initialize\n", phydev->mdio.addr);
                goto out;
        }
dev_dbg(&mdio->dev, "registered phy %s at address %i\n",
                child->name, addr);
```

在 get_phy_id 函数中增加打印，确认一下读出的 PHY ID 是否正确，如果与 datasheet 对比正确，则表示 PHY 已经正常工作了。

5. 通常如果得到了 PHY ID，驱动会根据这个 ID 号去找 PHY 的驱动函数，执行其 probe 来初始化 PHY 驱动，所以对于 C22 的 PHY，就可以如上所说在对应厂商的驱动文件中确认或加入新的 PHY 支持。
6. 不过对于这一颗 Marvell 的以太网 PHY，有一个例外，它其实是一个 C45 的 PHY，但是又提供 C22 的访问方法，如下文档 Marvell 详细说明了这样做的目的和方法。

**Overview**

- Clause 45 defined a new register accessmethod with a larger address space.
- Clause 45 was 1st used for new 10 GigPHYs and MACs (802.3ae).Since both 10 Gig PHYs and MACs were new designs this approach worked well.
- Clause 45 appeared to solve 802.3's register space problem forever (It didn't).

**The Problem**

- Most 802.3ah PHYs need to use the larger address space defined by Clause 45
- Most 802.3ah PHYs want to work with existing 10/100 MACs using MII for frame data & MDC/MDIO for register access
  - Most Existing 10/100 MACs can't do Clause 45! They can only do Clause 22!

**The Solution**

- We need to define a standard way to access Clause 45 registers using Clause 22
- Using a standard 'backwards compatible' way to access Clause 45 registers WILL solve 802.3's register access problems for 802.3ah and beyond
- This must be defined NOW since there are only 2 unused Clause 22 registers left

**The Implementation**

- Use Clause 22 Register 13 as a Clause 45 Command register
- Use Clause 22 Register 14 as a Clause 45 Address/Data register

**Clause 22 vs. Clause 45**

| Clause 22: | Clause 45: |
|---|---|
| 2 Opcodes: Read & Write | 4 Opcodes: Address, Read, Write &Read Increment |
| 32 Ports | 32 Ports<br>32 Devices per Port |
| 32 Registers per Port | 64K Registers per Device |
| **Operation of Clause 22**<br>● To Read a Clause 22 Register Perform:Read Register RRRRR from PHY AAAAA<br>● To Write a Clause 22 Register Perform:Write Register RRRRR to PHY AAAAA<br>● Each Operation Takes 1 Step | **Operation of Clause 45**<br>● To Read a Clause 45 Register Perform:Write Address AAAAAAAAAAAAAAAA to Device EEEEE on Port PPPPP; Read Register From Device EEEEE on Port PPPPP<br>● To Write a Clause 45 Register Perform:Write Address AAAAAAAAAAAAAAAA to Device EEEEE on Port PPPPP; Write Register To Device EEEEE on Port PPPPP<br>● Each Operation Takes 2 Steps |

**Clause 22 vs. Clause 45:**

| | Management Frame Fields - Clause 22 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | PRE | ST | OP | PHYAD | REGAD | TA | DATA | IDLE |
| Read | 1 . . .1 | 01 | 10 | AAAAA | RRRRR | Z0 | DDDDDDDDDDDDDDDD | Z |
| Write | 1 . . .1 | 01 | 01 | AAAAA | RRRRR | 10 | DDDDDDDDDDDDDDDD | Z |

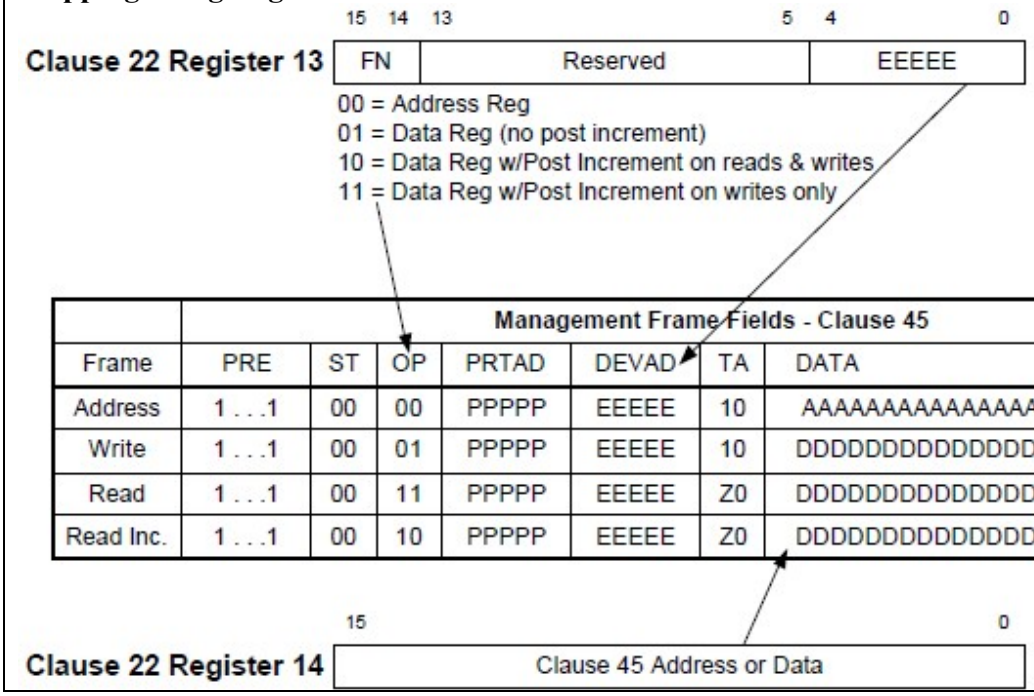
Same

Only 13 & 14 are Left

Need to Map

| | Management Frame Fields - Clause 45 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Frame | PRE | ST | OP | PRTAD | DEVAD | TA | DATA | IDLE |
| Address | 1 . . .1 | 00 | 00 | PPPPP | EEEEE | 10 | AAAAAAAAAAAAAAAA | Z |
| Write | 1 . . .1 | 00 | 01 | PPPPP | EEEEE | 10 | DDDDDDDDDDDDDDDD | Z |
| Read | 1 . . .1 | 00 | 11 | PPPPP | EEEEE | Z0 | DDDDDDDDDDDDDDDD | Z |
| Read Inc. | 1 . . .1 | 00 | 10 | PPPPP | EEEEE | Z0 | DDDDDDDDDDDDDDDD | Z |

**Mapping Using Reg 13 & 14:**




| | Management Frame Fields - Clause 45 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Frame | PRE | ST | OP | PRTAD | DEVAD | TA | DATA | IDLE |
| Address | 1 . . .1 | 00 | 00 | PPPPP | EEEEE | 10 | AAAAAAAAAAAAAAAA | Z |
| Write | 1 . . .1 | 00 | 01 | PPPPP | EEEEE | 10 | DDDDDDDDDDDDDDDD | Z |
| Read | 1 . . .1 | 00 | 11 | PPPPP | EEEEE | Z0 | DDDDDDDDDDDDDDDD | Z |
| Read Inc. | 1 . . .1 | 00 | 10 | PPPPP | EEEEE | Z0 | DDDDDDDDDDDDDDDD | Z |

**Operation of C22 to C45**

- To Read a C45 Register C22 Perform:
  1. Write FN = Address & EEEEE to C22 Register 13 on Port PPPPP
  2. Write Address AAAAAAAAAAAAAAAA to C22 Register 14 on Port PPPPP
  3. Write FN = Data & EEEEE to C22 Register 13 on Port PPPPP
  4. Read Register From C22 Register 14 on Port PPPPP

Read Operation Takes 4 Steps

- To Write a C45 Register C22 Perform:
  1. Write FN = Address & EEEEE to C22 Register 13 on Port PPPPP
  2. Write Address AAAAAAAAAAAAAAA to C22 Register 14 on Port PPPPP
  3. Write FN = Data & EEEEE to C22 Register 13 on Port PPPPP
  4. Write Register To C22 Register 14 on Port PPPPP

Only the Last Step is Different from Read

Marvell 88Q2110/2 数据手册说明如下：

### 2.18.2.2 Clause 22 MDIO Register Access Method

The 88Q2110/88Q2112 device supports Clause 22 MDIO manageable devices (MMD) extension registers to access Clause 45 MMD registers, using register 13 and 14 as specified in the IEEE Annex 22D.

**Table 27: XMDIO MMD Control Register**
**Device 0, Register 13**

| Bits | Field | Mode | HW Rst | SW Rst | Description |
|---|---|---|---|---|---|
| 15:14 | Function | R/W | 0 | 0 | 11 = Data, post increments on writes only<br>10 = Data, post increment on reads and writes<br>01 = Data, no post increment<br>00 = Address |
| 13:5 | Reserved | RO | 0 | 0 | Reserved |
| 4:0 | DEVAD | R/W | 0 | 0 | Device Address |

**Table 28: XMDIO MMD Address Data Register**
**Device 0, Register 14**

| Bits | Field | Mode | HW Rst | SW Rst | Description |
|---|---|---|---|---|---|
| 15:0 | Address Data | R/W | 0 | 0 | If 13.15:14 = 00, then MMD DEVAD's address register. Otherwise, MMD DEVAD's data register as indicated by the contents of its address register. |

i.MX8QXP 本身的 MAC 是可以支持 C45 的 PHY 的，但是由于 i.MX8QXP 目前使用的开发板并没有设计连接 C45 的 PHY，所以默认驱动中并没有 C45 PHY 的 MDIO 驱动支持。 所以此处需要修改 C22 的 MDIO 访问 PHY 的方式，来支持这种 PHY。以下以函数 get_phy_id 为例：

```
//drivers/net/phy/phy_device.c
static int get_phy_id(struct mii_bus *bus, int addr, u32 *phy_id,
                      bool is_c45, struct phy_c45_device_ids *c45_ids)
{
    int phy_reg;
   int ret;//johnli add
```

…

//如下为 Marvell 88Q2110/2 的 phy id 寄存器，位与 device 1 的第 2/3 个寄存器

**Table 45: PMA/PMD Device Identifier Register 1**
**Device 1, Register 0x0002**

| Bits | Field | Mode | HW Rst | SW Rst | Description |
|---|---|---|---|---|---|
| 15:0 | Organizationally Unique Identifier Bits 3:18 | RO | 0x002B | 0x002B | Bits 3 to 18 of the Marvell OUI |

**Table 46: PMA/PMD Device Identifier Register 2**
**Device 1, Register 0x0003**

| Bits | Field | Mode | HW Rst | SW Rst | Description |
|---|---|---|---|---|---|
| 15:10 | Organizationally Unique Identifier Bits 19:24 | RO | 0x02 | 0x02 | Bits 19:24 of the Marvell OUI |
| 9:4 | Model Number | RO | 0x18 | 0x18 | The model number is 011000. |
| 3:0 | Revision Number | RO | See Description. | See Description. | This is the revision number. Contact Marvell FAEs for information on the device revision number. |

```
#if 1 //johnli
        /* Grab the bits from PHYIR1, and put them in the upper half */
ret = mdiobus_write_nested(bus, addr, 0xD, 0x0001); //To Register 13, write 00(address) to bit 15:14 and the device address value to bit 4:0=device 1. 表示将在 Register 14 中放 device 1 的寄存器地址
if (ret)  return ret;
ret = mdiobus_write_nested(bus, addr, 0xE, 0x0002); //To Register 14, write address value=0x2, access register 2 放入寄存器地址
if (ret) return ret;
ret = mdiobus_write_nested(bus, addr, 0xD, 0x4001);// //To Register 13, write 01(Data, no post increment) to bit 15:14 and the same device address value to bit 4:0=device 1 表示从 register 14 中得到的会是 device 1 的一个数据
if (ret) return ret;
phy_reg = mdiobus_read(bus, addr, 0xE); //To Register 14, read the content of the MMD's selected register: read out the device 1/register 2 value 读出数据。
if (phy_reg < 0)
        return -EIO;
*phy_id = (phy_reg & 0xffff) << 16;
/* Grab the bits from PHYIR2, and put them in the lower half */
//same way to read out read out the device 1/register 3 value
```

```
    ret = mdiobus_write(bus, addr, 0xD, 0x0001);
    if (ret)  return ret;
    ret = mdiobus_write(bus, addr,0xE, 0x0003);
    if (ret)  return ret;
    ret = mdiobus_write(bus, addr, 0xD, 0x4001);
    if (ret) return ret;
    phy_reg = mdiobus_read(bus, addr, 0xE);
    if (phy_reg < 0)
        return -EIO;
    #else
…
#endif
    *phy_id |= (phy_reg & 0xffff);
    return 0;
}
```

修改此 Linux 原生函数后，可以得到 Marvell 88Q2110/2 PHY_ID= 0x2b0980.

7. 得到 PHY ID 后，证明 PHY 已经正常工作了，接下来要根据 PHY_ID 开发此 PHY 的 Linux 驱动，首先，Marvell phy 的标准驱动文件在：

```
Arch/arm64/configs/defconfg
CONFIG_MARVELL_PHY=m //reconfigure it to y
/drivers/net/phy
Makefile
obj-$(CONFIG_MARVELL_PHY)+= marvell.o
marvell.c/marvell_phy.h
#define MARVELL_PHY_ID_88Q2110          0x002b0980 //头文件中增加
// 然后在 C 文件中增加函数数组：
static struct mdio_device_id __maybe_unused marvell_tbl[] = {
    { MARVELL_PHY_ID_88Q2110, MARVELL_PHY_ID_MASK },
…
}
    {
        .phy_id = MARVELL_PHY_ID_88Q2110,
        .phy_id_mask = MARVELL_PHY_ID_MASK,
        .name = "Marvell 88Q2110",
```

```
                .features = PHY_GBIT_FEATURES,
                .flags = PHY_HAS_INTERRUPT,
                .probe = 88q2110_probe,
                .config_init = &88q2110_config_init,
                .config_aneg = &88q2110_config_aneg,
                .read_status = &88q2110_read_status,
                .ack_interrupt = &88q2110l_ack_interrupt,
                .config_intr = &88q2110_config_intr,
                .did_interrupt = &88q2110_did_interrupt,
                .resume = &88q2110_resume,
                .suspend = &88q2110_suspend,
                .get_sset_count = 88q2110l_get_sset_count,
                .get_strings = 88q2110_get_strings,
                .get_stats = 88q2110_get_stats,
    },
```

其中的每个函数都需要根据 Marvell 提供的数据手册，和他们的编程指南，使用 C22 接口，来模拟访问这个 C45 PHY 的寄存器。具体不再详述。刚刚在 get_phy_id 中有读 PHY 寄存器的方法，以下为写 PHY 寄存器的方法：

```
/**
* regWrite - Convenience function for writing a given PHY register(through clause 22 to access clause 45 for Marvell 88Q2110)
* @bus: the mii bus struct
* @addr: phy address
* @devAddr: device address
* @regAddr: register address
* @data: value to write to @regnum
*
* NOTE: MUST NOT be called from interrupt context,
* because the bus read/write functions may wait for an interrupt
* to conclude the operation.
*/
void regwrite_Marvell_88Q2110(struct mii_bus *bus, int addr, uint16_t devAddr, uint16_t regAddr, uint16_t data)
{
   int err;
       uint16_t reg_MMD_CTRL = 0x0000;
```

```
        reg_MMD_CTRL |= devAddr;
        err = mdiobus_write(bus, addr, 0xD, reg_MMD_CTRL);  //To Register 13, write 00(address) to bit 15:14 and the device address value to bit 4:0;
        if(err)
                return;
        err = mdiobus_write(bus, addr, 0xE, regAddr);       //To Register 14, write address value;
        if(err)
                return;
        reg_MMD_CTRL |= 0x4000;
        err = mdiobus_write(bus, addr, 0xD, reg_MMD_CTRL);  //To Register 13, write 01(Data, no post increment) to bit 15:14 and the same device address value to bit 4:0;
        if(err)
                return;
        err = mdiobus_write(bus, addr, 0xE, data);          // To Register 14, write the content of the MMD's selected register.
        if(err)
                return;
}
```

最后说明如何烧写 MAC 地址到内部 fuse,目前 i.MX8QXP 的 BSP 仅支持在 scfw 和 uboot 中烧写 fuse,内核不支持 fuse 写操作，如下 uboot 中的操作：

```
=> fuse
fuse - Fuse sub-system

Usage:
fuse read <bank> <word> [<cnt>] - read 1 or 'cnt' fuse words,
  starting at 'word'
fuse sense <bank> <word> [<cnt>] - sense 1 or 'cnt' fuse words,
  starting at 'word'
fuse prog [-y] <bank> <word> <hexval> [<hexval>...] - program 1 or
  several fuse words, starting at 'word' (PERMANENT)
fuse override <bank> <word> <hexval> [<hexval>...] - override 1 or
  several fuse words, starting at 'word'
```

芯片手册的 MAC 地址是：

| | | |
|---|---|---|
| 0x2340 [7:0] | 708 | MAC1_ADDR [7:0] |
| 0x2340 [15:8] | 708 | MAC1_ADDR [15:8] |
| 0x2340 [23:16] | 708 | MAC1_ADDR [23:16] |
| 0x2340 [31:24] | 708 | MAC1_ADDR [31:24] |
| 0x2350 [7:0] | 709 | MAC1_ADDR [39:32] |
| 0x2350 [15:8] | 709 | MAC1_ADDR [47:40] |

| | | |
|---|---|---|
| 0x2360 [7:0] | 710 | MAC2_ADDR [7:0] |
| 0x2360 [15:8] | 710 | MAC2_ADDR [15:8] |
| 0x2360 [23:16] | 710 | MAC2_ADDR [23:16] |
| 0x2360 [31:24] | 710 | MAC2_ADDR [31:24] |
| 0x2370 [7:0] | 711 | MAC2_ADDR [39:32] |
| 0x2370 [15:8] | 711 | MAC2_ADDR [47:40] |

读MAC地址方法：

```
=> fuse read 0 708 1
Reading bank 0:
Word 0x000002c4: 059f0400

=> fuse read 0 709 1
Reading bank 0:
Word 0x000002c5: 0000fdc7
```

内核中：

```
root@imx8qxpmek:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:04:9f:05:c7:fd
```

On i.MX8/8x the fuses are organized in fuse arrays instead of fuse banks and words, in this case the bank parameter should be set to zero and the word should match the "Fuse row Index"，The command line below can be used as an example to program the MAC1_ADDR[31:00] fuses in i.MX8QXP:

注意空fuse才能写， 一次性的。

```
=> fuse prog 0 708 0xa295fc11
Programming bank 0 word 0x000002c4 to 0xa295fc11...
Warning: Programming fuses is an irreversible operation!
This may brick your system.
Use this command only if you are sure of what you are doing!
Really perform this fuse programming? <y/N>

y
```

然后在 UUU 的 script 中增加以下命令就可以用 uboot 命令来烧写 MAC 地址了：

```
SDPS: boot -f imx-boot-imx8qxpmek-sd.bin-flash
+FB: ucmd fuse prog -y 0 708 0xa295fc11
+FB: ucmd fuse prog -y 0 709 0x000017b4
FB: ucmd setenv fastboot_dev mmc
```

### 1.16.3 TJA1101

NXP 推荐使用 TJA1101 代替 TJA1100, i.MX8QXP 6 layer 板测试过 TJA1101。

TJA1101 与 TJA1100 复用同一个驱动，所以 DTS 与源代码一样，注意以下区别：

Arch\arm64\boot\dts\freescale\imx8qxp-enet2-tja1100.dtsi

```
                ethphy2: ethernet-phy@5 { //应该根据实际硬件配置的 phy 地址来配置
                        compatible = "ethernet-phy-ieee802.3-c22";
                        reg = <5>;
                        tja110x,refclk_in;
                };
```

Drivers\net\phy\nxp-tja11xx.c 中已经支持 tja1101,如下：

```
#define PHY_ID_TJA1101                          0x0180dd00
..
    static struct mdio_device_id __maybe_unused tja11xx_tbl[] = {
        { PHY_ID_MATCH_MODEL(PHY_ID_TJA1100) },
        { PHY_ID_MATCH_MODEL(PHY_ID_TJA1101) },
        { }
};
…
static struct phy_driver tja11xx_driver[] = {
        {…
                PHY_ID_MATCH_MODEL(PHY_ID_TJA1101),
                .name                = "NXP TJA1101",
                …
        }
…}
    static int tja11xx_config_init(struct phy_device *phydev)
{..
        case PHY_ID_TJA1101:
                ret = phy_set_bits(phydev, MII_COMMCFG, MII_COMMCFG_AUTO_OP);
```

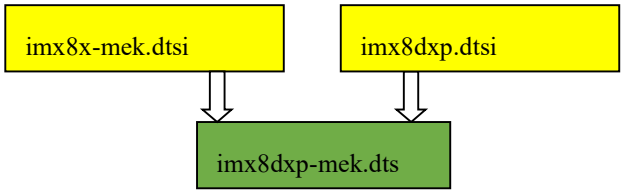
```
                if (ret)
                        return ret;
                break;
…}
```

## 1.17 i.MX8DX MEK 支持

在一些汽车应用，比如仪表和 V2X 中，使用 i.MX8DX 的机会比较多，目前 5.4.24 中已经支持 i.MX8DX MEK 板，只要 uboot 使用 i.mx8dx mek 的 config 编译，传过来的 dtb 正确，就会正确加载 i.mx8dx-mek.dtb。

## 1.18 i.MX8DXP MEK 支持

i.MX8DXP 和 i.MX8QXP 使用同一个晶圆，只是 fuse 掉了两个 A35 核，所以只需要修改加载的 DTB 就可以支持，如下设计 i.MX8DXP MEK 的 DTS:




如下：arch\arm64\boot\dts\freescale\imx8dxp.dtsi

```
#include "imx8qxp.dtsi"
&thermal_zones { //重新覆盖了 thermal 驱动，只用 A35_0/1 device
        cpu-thermal0 {
                cooling-maps {
                        map0 {
                                cooling-device =
                                <&A35_0 THERMAL_NO_LIMIT THERMAL_NO_LIMIT>,
                                <&A35_1 THERMAL_NO_LIMIT THERMAL_NO_LIMIT>;
                        };
                };
        };
};

&cpus {
```

```
        /delete-node/ cpu@2; //去掉了 cpu2/3 节点
        /delete-node/ cpu@3;
};
```

所以 imx8dxp-mek.dtsi 可以从 imx8qxp-mek.dtsi 拷贝一份，然后头部修改为：

```
//#include "imx8qxp.dtsi"
#include "imx8dxp.dtsi"
#include "imx8x-mek.dtsi"…
```

然后多余的驱动可以去掉。

然后修改 Makefile 将 fsl-imx8dx-mek.dts 增加进去：

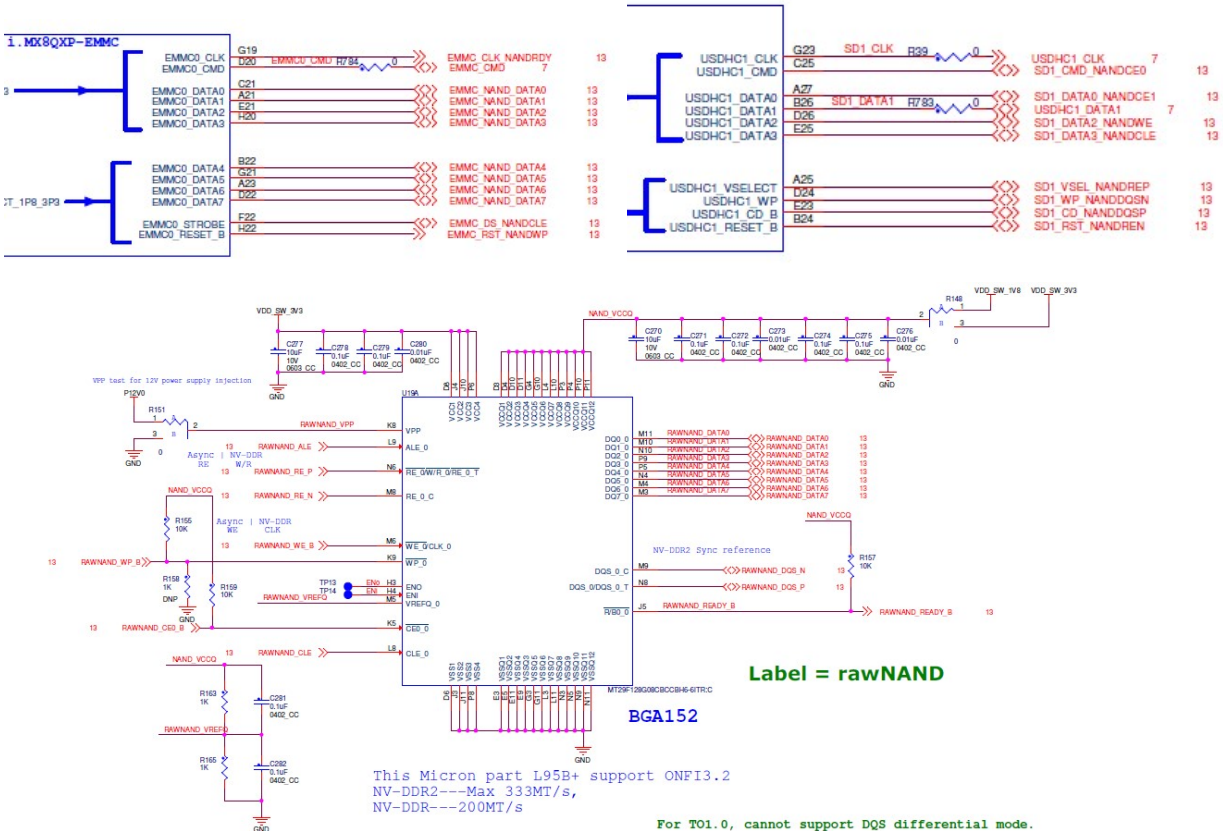
\linux-imx\arch\arm64\boot\dts\freescale\Makefile

```
dtb-$(CONFIG_ARCH_MXC) += imx8qxp-mek.dtb imx8dxp-mek.dtb…
```

最后 修改 uboot 参数中的 fdt_file= imx8dxp-mek.dtb。

## 1.19 NAND Flash 支持与烧录

i.MX8QXP MEK 板没有设计 Nand flash，但是 i.MX8QXP ARM2 板有设计，如下：




**i.MX8X 内核驱动代码与定制**

硬件设计上要注意：

1. i.MX8QXP Nandflash 与 usdhc 0/1 有 iomux 冲突，所以设计为 nandflash 后，usdhc0/1 就不能工作了。
2. nandflash 的 Ready_B 为开漏电路，设计中要求上拉。

软件上可以参考 ARM2 板子的 DTS 配置 porting 过来：

\linux-imx\arch\arm64\boot\dts\freescale\ fsl-imx8qxp-mek.dtsi

```
pinctrl_gpmi_nand_1: gpmi-nand-1 {
                        fsl,pins = <
                                IMX8QXP_EMMC0_CLK_CONN_NAND_READY_B      0x0e00004c
                                IMX8QXP _EMMC0_DATA0_CONN_NAND_DATA00    0x0e00004c
                                …
                                IMX8QXP _EMMC0_DATA7_CONN_NAND_DATA07    0x0e00004c
                                IMX8QXP _EMMC0_STROBE_CONN_NAND_CLE              0x0e00004c
                                IMX8QXP _EMMC0_RESET_B_CONN_NAND_WP_B    0x0e00004c

                                IMX8QXP_USDHC1_DATA0_CONN_NAND_CE1_B     0x0e00004c
                                IMX8QXP_USDHC1_DATA2_CONN_NAND_WE_B      0x0e00004c
                                IMX8QXP_USDHC1_DATA3_CONN_NAND_ALE               0x0e00004c
                                IMX8QXP_USDHC1_CMD_CONN_NAND_CE0_B               0x0e00004c

                                /* i.MX8QXP NAND use nand_re_dqs_pins */
                                IMX8QXP_USDHC1_CD_B_CONN_NAND_DQS     0x0e00004c
                                IMX8QXP_USDHC1_VSELECT_CONN_NAND_RE_B  0x0e00004c
//注意一下ready_b要求外部上拉，而我们所有nandflash pin是设置为下拉的，所以如果外部没有设计上拉，可以把此处配置为上拉 IMX8QXP_USDHC1_VSELECT_CONN_NAND_RE_B  0x0e00002c
```

| 4 | | | | e | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** |

| | |
|---|---|
| 6-5<br>PULL | Pull Down Pull Up<br>Pull Down Pull Up<br>00b - Prohibited<br>01b - pull up<br>10b - pull down<br>11b - pull disabled |

```
                        >;
                };
```

增加 gpmi 支持：

```
&gpmi {
```

```
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_gpmi_nand_1>;
        status = "okay";
        nand-on-flash-bbt;
};
```

因为 iomux 冲突，去掉 usdhc0/1 支持：

```
 &usdhc1 {…
        status = "disabled";
};

&usdhc2 {…
status = "disabled";
};
```

Nandflash 的 uuu 烧写脚本如下：

\L4.14.98_2.0.0_ga_images_MX8QXPMEK\samples\ example_kernel_nand.uuu

将其和 uuu.exe 工具放在与镜像同级目录，修改：

- _flash_fw.bin 为 flash_fw.bin(请参考 bootlaoder 文档，由 imx-mkimage 工具，采用 ARM2 板的 uboot 导出)
- _flash.bin 为 flash.bin(请参考 bootlaoder 文档，由 imx-mkimage 工具，采用 ARM2 板的 uboot 导出)
- _Image 为编译出来的内核
- _board.dtb 为编译出来的 fsl-imx8qxp-mek.dtb
- _initramfs.cpio.gz.uboot 为下载 demo 镜像中的 fsl-image-mfgtool-initramfs-imx_mfgtools.cpio.gz.u-boot

然后运行 uuu.exe example_kernel_nand.uuu 就可以开始烧写 nandflash 了。

脚本会根据 uboot 传递过来的 mtdparts 信息来创建 mtd 设备 partition

```
FBK: ucmd cat /proc/mtd | while read dev size erase name; do mtd=${dev:3}; mtd=${mtd%:}; name=${name%\"}; name=${name#\"}; echo export $name=$mtd >> /tmp/mtd.sh; done;
```

使用 kobs-ng 来生成坏块表和烧写 bootloader:

```
 FBK: ucmd source /tmp/mtd.sh; cd /tmp; if ! [[ `cat /sys/devices/soc0/soc_id` = *"MX8Q"* ]]; then pad="-x"; fi; kobs-ng init $pad -v --chip_0_device_path=/dev/mtd${nandboot} /tmp/boot
```

使用 nandwrite 来烧写内核等，注意以下 TEE 部分如果不使用可以去掉：

```
# burn uTee
# FBK: ucmd source /tmp/mtd.sh; flash_erase /dev/mtd${nandtee} 0 0
# FBK: ucp _tee t:/tmp/tee
# FBK: ucmd source /tmp/mtd.sh; nandwrite -p /dev/mtd${nandtee} -p /tmp/tee
```

使用 ubi 命令来创建 rootfs 文件系统，然后使用 tar 命令来写入 rootfs:

```
# burn rootfs
FBK: ucmd source /tmp/mtd.sh; flash_erase /dev/mtd${nandrootfs} 0 0
FBK: ucmd source /tmp/mtd.sh; ubiattach /dev/ubi_ctrl -m ${nandrootfs}
FBK: ucmd source /tmp/mtd.sh; ubimkvol /dev/ubi0 -Nnandrootfs -m
FBK: ucmd source /tmp/mtd.sh; mkdir -p /mnt/mtd
FBK: ucmd source /tmp/mtd.sh; mount -t ubifs ubi0:nandrootfs /mnt/mtd
FBK: acmd export EXTRACT_UNSAFE_SYMLINKS=1; tar -jx -C /mnt/mtd
FBK: ucp _rootfs.tar.bz2 t:-
FBK: sync
```

最后 umount 掉 mtd，done

```
FBK: ucmd umount /mnt/mtd
FBK: done
```