NXP Semiconductor
Application Note

Document Number: ANnnnn
Rev. 3, 9/2019

i.MX8X 板级开发板 4.14.98_ga 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4.14.98_ga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-L4.14.98_2.0.0_ga\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了解i.MX8QXP 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。

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。

history	comment	Author
V1	● 创建文档	● John.Li
V2	● 修改了一些错误 ● 更新了第 6.12 章	● John.Li
V2	● 增加 nandflash 支持/DDR 修改/i.MX8DX 支持	● John.Li

Contents

1 创建 i.MX8QXP Linux 4.14.98_ga 板级开发包编译环境 2
1.1 下载板级开发包 ... 2
1.2 创建yocto编译环境: ... 3
2 Device Tree ... 15
2.1 恩智浦的device Tree结构 ... 15
2.2 device Tree的由来(no updates) ... 18
2.3 device Tree的基础与语法(no updates) ... 21
2.4 device Tree的代码分析(no updates) ... 42
3 恩智浦i.MX8XBSP 包文件目录结构 ... 75
4 恩智浦i.MX8XBSP的编译(no updates) ... 77
4.1 需要编译哪些文件 ... 77
4.2 如何编译这些文件 ... 78
4.3 如何链接为目标文件及链接顺序 ... 79
4.4 kernel Kconfig ... 81
5 恩智浦BSP的内核初始化过程(no updates) ... 81
5.1 初始化的汇编代码 ... 83
5.2 初始化的C代码 ... 87
5.3 init_machine ... 100
6 恩智浦BSP的内核定制 ... 103
6.1 DDR修改 ... 103
6.2 IO管脚配置与Pinctrl驱动 ... 105
6.3 新板bringup ... 120
6.4 更改调试串口 ... 128
6.5 uSDHC设备定制(eMMC flash,SDcard, SDIOcard) 135
6.6 LVDS LCD 驱动定制 ... 144
6.7 GPIO_Key 驱动定制 ... 147
6.8 GPIO_LED 驱动定制 ... 151
6.9 Fuse nvram驱动 ... 154
6.10 SPI与SPI Slave驱动 ... 155
6.11 USB 3.0 TypeC 改成 USB 3.0 TypeA(未验证) ... 162
6.12 汽车级以太网驱动定制 ... 162
6.13 i.MX8DX MEK支持 ... 180
6.14 NAND Flash支持与烧录 ... 181

1 创建 i.MX8QXP Linux 4.14.98_ga 板级开发包编译环境

1.1 下载板级开发包

i.MX8QXP_QM Linux 4.14.98_ga 板级开发包下载地址 如下:

https://www.nxp.com/support/developer-resources/run-time-software/i.mx-developer-resources/i.mx-software-and-development-tool:IMX_SW

i.MX BSP Updates and Releases

Linux

Linux 4.14.98_2.0.0

Source Code

Linux Binary Demo Files - i.MX 6QuadPlus, i.MX 6Quad, i.MX 6DualPlus, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo, i.MX 6SoloX

Linux Binary Demo Files - i.MX 6SoloLite EVK

Linux Binary Demo Files - i.MX 6SLL EVK

Linux Binary Demo Files - i.MX 6UltraLite, i.MX 6ULL, i.MX 7Dual

Linux Binary Demo Files - i.MX 7Dual SabreSD

Linux Binary Demo Files - i.MX 7ULP EVK

Linux Binary Demo Files - i.MX 8MMini EVK

Linux Binary Demo Files - i.MX 8MQuad EVK

Linux Binary Demo Files - i.MX 8QMax MEK

Linux Binary Demo Files - i.MX 8QXPlus MEK

SCFW Porting Kit

AACPlus Codec

Documentation

Linux

Linux L4.14.98_2.0.0 Documentation

可以首先下载:

1. Linux L4.14.98_2.0.0 Documentation

i.MX_BSP_Porting_Guide.pdf

i.MX_Graphics_User's_Guide.pdf

i.MX_Linux_Reference_Manual.pdf

i.MX_Linux_Release_Notes.pdf

i.MX_Linux_User's_Guide.pdf

i.MX_VPU_Application_Programming_Interface_Linux_Reference_Manual.pdf

i.MX_Yocto_Project_User's_Guide.pdf

2. SCFW Porting Kit

imx-scfw-porting-kit-1.2.tar.gz

i.MX8QXP MEK Linux Demo 镜像也可以下载。

1.2 创建 yocto 编译环境:

Ubuntu 14.04 编译主机需要事先执行以下命令安装编译所需包：

```
sudo apt-get update
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat libsdl1.2-dev
sudo apt-get install libsdl1.2-dev xterm sed cvs subversion coreutils texi2html docbook-utils python-pysqlite2 help2man make gcc g++ desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf automake groff curl lzop asciidoc
sudo apt-get install u-boot-tools

git config --global user.name xxx
git config --global user.email xxx@xxx.com
git config –list
```

根据文档 imx-yocto-L4.14.98_2.0.0_ga\i.MX_Yocto_Project_User's_Guide.pdf 创建 yocto 编译环境时，需要注意以下几点：

1. 中国大陆地区无法从 google 的 git 服务器下载 repo 工具，可以改成如下清华的 git 服务器：

```
mkdir bin
 cd bin
 curl https://mirrors.tuna.tsinghua.edu.cn/git/git-repo -o repo
 chmod a+x repo
 export PATH=~/bin:$PATH
```

2. 可以通过如下地址检查目前可以下载的 manifest：

   https://source.codeaurora.org/external/imx/imx-manifest/tree/?h=imx-linux-sumo

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp
repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-linux-sumo -m imx-4.14.98-2.0.0_ga.xml  --repo-url=https://mirrors.tuna.tsinghua.edu.cn/git/git-repo
repo sync
```

3. 建议编译 wayland 后端，而不是编译 fb 后端(对于没有屏的应用如 Tbox，可以使用 fb 后端)。

DISTRO=fsl-imx-wayland MACHINE=imx8qxpmek source fsl-setup-release.sh -b imx8qxpmek_wayland

或者支持 QT 的话编译 xwayland 后端

DISTRO=fsl-imx-xwayland MACHINE=imx8qxpmek source fsl-setup-release.sh -b imx8qxpmek_xwayland

4. 如下命令编译基于 wayland 的支持 GUI 的测试镜像：

bitbake fsl-image-validation-imx

或者编译 QT

bitbake fsl-image-qt5-validation-imx

5. 编译成功后的输出是：

Wayland：

/imx-yocto-bsp/imx8qxpmek_wayland/tmp/deploy/images/imx8qxpmek/
fsl-image-validation-imx-imx8qxpmek.sdcard.bz2 ->
fsl-image-validation-imx-imx8qxpmek-20190114071748.rootfs.sdcard.bz2 (sdcard 镜像，解压后可以直接烧写)

fsl-image-validation-imx-imx8qxpmek.tar.bz2 ->
fsl-image-validation-imx-imx8qxpmek-20190114071748.rootfs.tar.bz2 (rootfs 压缩包)

u-boot-sd-2018.03-r0.bin (uboot)

imx-boot-tools/ mx8qx-mek-scfw-tcm.bin (sc firmware)

imx-boot-tools/ bl31-imx8qxp.bin (ATF)

imx-boot-tools/mx8qx-ahab-container.img (seco HAB container 仅有镜像)

xwayland+QT:

pwd

~/imx-yocto-bsp/imx8qxpmek_xwayland/tmp/deploy/images/imx8qxpmek/

fsl-image-validation-imx-imx8qxpmek.sdcard.bz2 ->
fsl-image-qt5-validation-imx-imx8qxpmek-20190731045445.rootfs.sdcard.bz2 (sdcard 镜像，解压后可以直接烧写)

fsl-image-validation-imx-imx8qxpmek.tar.bz2 ->
fsl-image-qt5-validation-imx-imx8qxpmek-20190731045445.rootfs.tar.bz2 (rootfs 压缩包)

u-boot-sd-2018.03-r0.bin (uboot)

imx-boot-tools/ mx8qx-mek-scfw-tcm.bin (sc firmware)

imx-boot-tools/ bl31-imx8qxp.bin (ATF)

imx-boot-tools/mx8qx-ahab-container.img (seco HAB container 仅有镜像)

6. 烧写 SD 卡镜像

Validation image:

bunzip2 -dk -f fsl-image-validation-imx-imx8qxpmek-20190114071748.rootfs.sdcard.bz2

```
sudo dd if= fsl-image-validation-imx-imx8qxpmek-20190114071748.rootfs.sdcard of=/dev/sd<partition> bs=1M conv=fsync
```

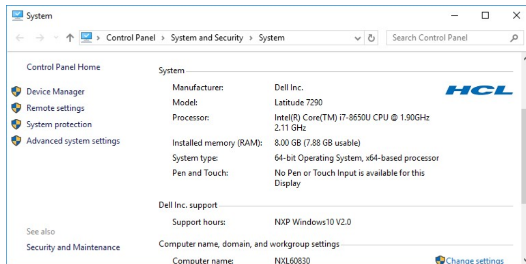
QT image:

```
bunzip2 -dk -f  fsl-image-qt5-validation-imx-imx8qxpmek-20190731045445.rootfs.sdcard.bz2
```

```
sudo dd if= fsl-image-qt5-validation-imx-imx8qxpmek-20190731045445.rootfs.sdcard of=/dev/sd<partition> bs=1M conv=fsync
```

注意：以上 yocto 编译测试环境为：

- 编译主机与主机操作系统：



- 虚拟机工具，版本，设置与 ubuntu 镜像版本：

VMware® Workstation 14 Player 14.1.1 build-7528167

虚拟机设置

硬件 选项

设备	摘要
内存	2 GB
处理器	2
硬盘 (SCSI)	200 GB
硬盘 2 (SCSI)	64 GB
CD/DVD (SATA)	自动检测
CD/DVD 2 (SATA)	自动检测
网络适配器	NAT
USB 控制器	存在
声卡	自动检测
打印机	存在
显示器	自动检测

```
uname -a
```

```
Linux ubuntu 3.16.0-77-generic #99~14.04.1-Ubuntu SMP Tue Jun 28 19:17:10 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

- 测试网络环境：

家用上海电信 1G 光纤宽带。

- 可能遇到的问题及解决办法：(gpu sdk 报错)

  如下缺少 typing 库错误

```
…
o run this tool with Python 3.4 please install the 'typing' library with 'sudo pip3 install typing' or upgrade to Python 3.5+. If you dont have 'pip3' then you need to install it with 'sudo apt-get install python3-pip'
…
sudo apt-get install python3-pip
sudo pip3 install typing
```

如果出现缺少 imx-gpu-sdk 相关库的错误(本次测试并没有遇到过); 则手工 wget 下载相关包，需要改名的改名，然后再次运行 bitbake

```
cd tmp/work/aarch64-mx8-poky-linux/imx-gpu-sdk/5.2.0-r0/git/.Thirdparty/.DownloadCache
wget https://source.codeaurora.org/mirrored_source/external/imx/tclap-1.2.2.tar.gz
wget https://github.com/Unarmed1000/RapidVulkan/archive/1.0.68.0a.tar.gz
wget https://github.com/g-truc/gli/archive/0.8.2.0.tar.gz
wget https://github.com/Unarmed1000/RapidOpenCL/archive/1.1.0.1.tar.gz
wget https://github.com/Tencent/rapidjson/archive/v1.1.0.tar.gz
mv 1.0.68.0a.tar.gz RapidVulkan-1.0.68.0a.tar.gz
mv 0.8.2.0.tar.gz gli-0.8.2.0.tar.gz
mv 1.1.0.1.tar.gz  RapidOpenCL-1.1.0.1.tar.gz
mv v1.1.0.tar.gz rapidjson-1.1.0.tar.gz
~/imx-yocto-bsp/imx8qxpmek_wayland/tmp/work/aarch64-mx8-poky-linux/imx-gpu-sdk/5.2.0-r0/git/.Thirdparty/.DownloadCache$ ls
gli-0.8.2.0.tar.gz      RapidOpenCL-1.1.0.1.tar.gz    tclap-1.2.2.tar.gz
rapidjson-1.1.0.tar.gz  RapidVulkan-1.0.68.0a.tar.gz
cd ~/imx-yocto-bsp/imx8qxpmek_wayland/
bitbake fsl-image-validation-imx
```

- 总共时长：

  大约 15 个小时(包括解决问题的时间)。

1.2.1 编译 sdk 及安装

```
bitbake fsl-image-validation-imx -c populate_sdk
```

或者：

```
bitbake fsl-image-qt5-validation-imx -c populate_sdk
```

总共时长大约：2 个小时。

sdk 编译结束后，安装文件地址如下：

```
~/imx-yocto-bsp/imx8qxpmek_wayland/tmp/deploy/sdk
fsl-imx-wayland-glibc-x86_64-fsl-image-validation-imx-aarch64-toolchain-4.14-sumo.sh
```

QT 如下：

```
~/imx-yocto-bsp/imx8qxpmek_xwayland/tmp/deploy/sdk
fsl-imx-xwayland-glibc-x86_64-fsl-image-qt5-validation-imx-aarch64-toolchain-4.14-sumo.sh
```

安装 sdk:

运行安装脚本：

```
pwd
~/imx-yocto-bsp/imx8qxpmek_wayland
mkdir sdk
./fsl-imx-wayland-glibc-x86_64-fsl-image-validation-imx-aarch64-toolchain-4.14-sumo.sh
```

Xwayland+QT 如下：

```
pwd
~/imx-yocto-bsp/imx8qxpmek_wayland
mkdir sdk
./fsl-imx-xwayland-glibc-x86_64-fsl-image-qt5-validation-imx-aarch64-toolchain-4.14-sumo.sh
```

输入安装 sdk 的目标目录(default: /opt/fsl-imx-fb/4.9.51-mx8-ga):

```
~/imx-yocto-bsp/imx8qxpmek_wayland/sdk
You are about to install the SDK to "~/imx-yocto-bsp/imx8qxpmek_wayland/sdk". Proceed[Y/n]? Y
```

Xwayland+QT 如下：

```
~/imx-yocto-bsp/imx8qxpmek_xwayland/sdk
You are about to install the SDK to "~/imx-yocto-bsp/imx8qxpmek_xwayland/sdk". Proceed[Y/n]? Y
```

安装结束后，可以在以下目录找到安装的 sdk 及编译 toolchain：

```
~/imx-yocto-bsp/imx8qxpmek_wayland/sdk/environment-setup-aarch64-poky-linux
Sysroots
```

Xwayland+QT 如下：

```
~/imx-yocto-bsp/imx8qxpmek_xwayland/sdk/environment-setup-aarch64-poky-linux
sysroots
```

1.2.2 独立编译

1. 编译 SCfirmware

打开一个终端：

```
pwd
~/imx-yocto-bsp
mkdir standalone
cd standalone
tar xzvf  imx-scfw-porting-kit-1.2.tar.gz    (untar scfw tools kit)
    |->packages
    |    |-> imx-scfw-porting-kit-1.2.bin
chmod a+x ./ imx-scfw-porting-kit-1.2.bin
sh ./ imx-scfw-porting-kit-1.2.bin   (run the bin file to install scfw tools kit)
cd imx-scfw-porting-kit-1.2/src
tar xzvf scfw_export_mx8qm_b0.tar.gz
tar xzvf scfw_export_mx8qx_b0.tar.gz  (untar source codes)
```

从以下地址 下载编译工具链：
https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads (E.g. Linux 64-bit File: gcc-arm-none-eabi-6-2017-q2-update-linux.tar.bz2 (95.90 MB))
注意最新验证过的工具链版本是 2017-q2，不建议使用最新的工具链。

```
pwd
~/imx-scfw-porting-kit-1.2
mkdir toolchain
mv gcc-arm-none-eabi-6-2017-q2-update-linux.tar.bz2 toolchain/
cd toolchain
tar jxvf gcc-arm-none-eabi-6-2017-q2-update-linux.tar.bz2
pwd
~/imx-scfw-porting-kit-1.2/src/scfw_export_mx8qx_b0
export TOOLS= ../.. /toolchain/
make qx B=mek R=B0 (如果需要看串口调试信息就增加 M=1 参数，U=2 表示使用 SCU 本身串口，注意重新编译之前要 make clean-qx 一下)
```

编译结束打印为：

```
make qx B=mek R=B0 M=1 U=2
Generating platform/board/mx8qx_mek/dcd/imx8qx_dcd_1.2GHz.h
Generating platform/board/mx8qx_mek/dcd/dcd.h from platform/board/mx8qx_mek/dcd/imx8qx_dcd_1.2GHz.h
Generating platform/board/mx8qx_mek/dcd/imx8qx_dcd_1.2GHz_retention.h
Generating platform/board/mx8qx_mek/dcd/dcd_retention.h from
platform/board/mx8qx_mek/dcd/imx8qx_dcd_1.2GHz_retention.h
Compiling platform/drivers/pmic/fsl_pmic.c
Compiling platform/drivers/pmic/pf8100/fsl_pf8100.c
```

```
Compiling platform/drivers/pmic/pf100/fsl_pf100.c
Compiling platform/board/mx8qx_mek/board.c
Compiling platform/board/board_common.c
Compiling platform/board/pmic.c
Linking build_mx8qx_b0/scfw_tcm.elf ....
Objcopy build_mx8qx_b0/scfw_tcm.bin ....
done.
```

2. 编译 uboot：

另打开一个终端

```
pwd
~/imx-yocto-bsp/standalone
git clone  https://source.codeaurora.org/external/imx/uboot-imx
cd uboot-imx
git tag |grep rel_imx_4.14.
 ....
 rel_imx_4.14.98_2.0.0_ga
 ...
git checkout rel_imx_4.14.98_2.0.0_ga
git status
HEAD detached at rel_imx_4.14.98_2.0.0_ga
nothing to commit, working directory clean

ls configs |grep imx8qxp
 …

  imx8qxp_mek_defconfig
…
source ~ /imx-yocto-bsp/imx8qxpmek_xwayland/sdk/environment-setup-aarch64-poky-linux
make  imx8qxp_mek_defconfig
make
```

编译 log 如下：

```
…
LDS     u-boot.lds
 LD      u-boot
 OBJCOPY u-boot.srec
```

```
 OBJCOPY u-boot-nodtb.bin
start=$(aarch64-poky-linux-nm u-boot | grep __rel_dyn_start | cut -f 1 -d ' '); end=$(aarch64-poky-linux-nm u-boot | grep __rel_dyn_end | cut -f 1 -d ' '); tools/relocate-rela u-boot-nodtb.bin 0x80020000 $start $end
 DTC     arch/arm/dts/fsl-imx8dx-17x17-val.dtb
 DTC     arch/arm/dts/fsl-imx8qm-ddr4-arm2.dtb
 DTC     arch/arm/dts/fsl-imx8qm-lpddr4-arm2.dtb
 DTC     arch/arm/dts/fsl-imx8qm-mek.dtb
 DTC     arch/arm/dts/fsl-imx8qm-mek-xen.dtb
 DTC     arch/arm/dts/fsl-imx8qxp-17x17-val.dtb
 DTC     arch/arm/dts/fsl-imx8qxp-lpddr4-arm2.dtb
 DTC     arch/arm/dts/fsl-imx8qxp-mek.dtb
make[2]: 'arch/arm/dts/fsl-imx8qxp-mek.dtb' is up to date.
 SHIPPED dts/dt.dtb
 FDTGREP dts/dt-spl.dtb
 CAT     u-boot-dtb.bin
 COPY    u-boot.bin
 SYM     u-boot.sym
 COPY    u-boot.dtb
 LD      u-boot.elf
 CHK     include/config.h
 CFG     u-boot.cfg
 CFGCHK  u-boot.cfg
```

编译结束后的输出镜像为：

```
u-boot.bin
arch/arm/dts/fsl-imx8qxp-mek.dtb
```

3. 编译 Linux 内核：

```
 pwd
~/imx-yocto-bsp/standalone
git clone  https://source.codeaurora.org/external/imx/linux-imx
cd linux-imx
git tag |grep rel_imx_4.14
 ...
  rel_imx_4.14.98_1.0.0_ga
```

```
...
git checkout rel_imx_4.14.98_2.0.0_ga
git status
 HEAD detached at rel_imx_4.14.98_2.0.0_ga
source ~ /imx-yocto-bsp/imx8qxpmek_wayland/sdk/environment-setup-aarch64-poky-linux
make defconfig
LDFLAGS=""  CC="$CC"  make
LDFLAGS=""  CC="$CC"  make dtbs clean
LDFLAGS=""  CC="$CC"  make dtbs //just make dtb
```

编译结束后的输出镜像为：

```
arch/arm64/boot/dts/freescale/fsl-imx8qxp-mek.dtb
arch/arm64/boot/Image
```

4. 编译 ATF:

```
pwd
~/imx-yocto-bsp/standalone
git clone https://source.codeaurora.org/external/imx/imx-atf
cd imx-atf
git tag
 ...
  rel_imx_4.14.98_2.0.0_ga
 ...

git checkout rel_imx_4.14.98_2.0.0_ga
git status
HEAD detached at rel_imx_4.14.98_2.0.0_ga
source  ~/imx-yocto-bsp/imx8qxpmek_wayland/sdk/environment-setup-aarch64-poky-linux
LDFLAGS=""  make PLAT=imx8qx
```

编译 log 为：

```
…
  LD      build/imx8qxp/release/bl31/bl31.elf
  BIN     build/imx8qxp/release/bl31.bin
Built build/imx8qxp/release/bl31.bin successfully
```

编译结束后的输出镜像为：

```
./build/imx8qxp/release/bl31.bin
```

5. 运行 imx-mkimage 脚本生成 flash.bin 镜像:

另打开一个终端，不要与编译 uboot&kernel 同用一个终端：

```
pwd
~/imx-yocto-bsp/standalone
git clone https://source.codeaurora.org/external/imx/imx-mkimage
cd imx-mkimage
git tag
 ...
 rel_imx_4.14.98_2.0.0_ga
 ...
git checkout rel_imx_4.14.98_2.0.0_ga
git status
HEAD detached at rel_imx_4.14.98_1.0.0_ga
```

imx-mkimage 需要调用 host PC 的 GCC 工具，所以需要退出之前的 terminal。重新进入，从而退出之前 source 的交叉编译变量。

将 mx8qx-ahab-container.img, sc firmware bin, atf 和 uboot 拷贝至对应 iMX8QX 目录：

```
cp ../../imx8qxpmek_xwayland/tmp/deploy/images/imx8qxpmek/imx-boot-tools/mx8qx-ahab-container.img ./iMX8QX/
cp ../packages/imx-scfw-porting-kit-1.2/src/scfw_export_mx8qx_b0/build_mx8qx_b0/scfw_tcm.bin ./iMX8QX/
cp ../imx-atf/build/imx8qx/release/bl31.bin ./iMX8QX/
cp ../uboot-imx/u-boot.bin ./iMX8QX/
ls ./iMX8QX/
bl31.bin                    imx8dx_ddr3_dcd_16bit_933MHz.cfg  imx8qx_dcd_16bit_1.2GHz.cfg
imx8qx_ddr3_dcd_800MHz.cfg     imx8qx_ddr3_dcd_933MHz.cfg     lib          mx8qx-ahab-container.img  scripts
u-boot
imx8dx_ddr3_dcd_16bit_1066MHz.cfg  imx8qx_dcd_1.2GHz.cfg         imx8qx_ddr3_dcd_1066MHz_ecc.cfg
imx8qx_ddr3_dcd_800MHz_ecc.cfg  imx8qx_ddr3_dcd_933MHz_ecc.cfg  mkimage_fit_atf.sh  scfw_tcm.bin
soc.mak
```

运行 imx-mkimage 脚本生成 flash.bin 镜像

```
make SOC=iMX8QX flash_b0
include misc.mak
include m4.mak
include android.mak
```

```
include test.mak
include autobuild.mak
include rev_a.mak
include alias.mak
./../mkimage_imx8 -commit > head.hash
630+1 records in
630+1 records out
645154 bytes (645 kB, 630 KiB) copied, 0.00327583 s, 197 MB/s
./../mkimage_imx8 -soc QX -rev B0 -append mx8qx-ahab-container.img -c -scfw scfw_tcm.bin -ap u-boot-atf.bin a35 0x80000000 -out flash.bin
SOC: QX
REVISION: B0
New Container:  0
SCFW:   scfw_tcm.bin
AP:     u-boot-atf.bin  core: a35 addr: 0x80000000
Output: flash.bin
CONTAINER FUSE VERSION: 0x00
CONTAINER SW VERSION:   0x0000
ivt_offset:     1024
rev:    2
Platform:       i.MX8QXP B0
ivt_offset:     1024
container image offset (aligned):a800
flags: 0x10
Hash of the images = sha384
1+0 records in
1+0 records out
150528 bytes (151 kB, 147 KiB) copied, 0.000348217 s, 432 MB/s
292+1 records in
292+1 records out
149568 bytes (150 kB, 146 KiB) copied, 0.000953368 s, 157 MB/s
SCFW file_offset = 0xa800 size = 0x24c00
Hash of the images = sha384
1+0 records in
1+0 records out
```

```
777216 bytes (777 kB, 759 KiB) copied, 0.00122179 s, 636 MB/s
1516+1 records in
1516+1 records out
776226 bytes (776 kB, 758 KiB) copied, 0.00426954 s, 182 MB/s
AP file_offset = 0x2f400 size = 0xbdc00
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x590
DONE.
Note: Please copy image to offset: IVT_OFFSET + IMAGE_OFFSETHash of the images = sha384
1+0 records in
1+0 records out
760832 bytes (761 kB) copied, 0.00102483 s, 742 MB/s
1485+1 records in
1485+1 records out
760702 bytes (761 kB) copied, 0.00259972 s, 293 MB/s
AP file_offset = 0x28c00 size = 0xb9c00
CST: CONTAINER 0 offset: 0x400
CST: CONTAINER 0: Signature Block: offset is at 0x590
DONE.
Note: Please copy image to offset: IVT_OFFSET + IMAGE_OFFSET
```

结束后生成的 flash.bin 在：

```
./iMX8QX/flash.bin
```

注意：也可以如下使用 wget 命令获得 mx8qx-ahab-container.img

```
wget http://www.freescale.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.1.bin
chmod +x firmware-imx-8.1.bin  && ./ firmware-imx-8.1.bin --auto-accept
```

mx8qx-ahab-container.img 位于 firmware-imx-8.1/firmware/seco/mx8qx-ahab-container.img

6. 烧写镜像到 sdcard

bootloader:

```
vmuser@ubuntu:~$ cat /proc/partitions
major minor  #blocks  name
...
   8       32    7761920 sdc
```

```
  8       33      32768 sdc1
  8       34    6918144 sdc2
sudo dd if=flash.bin of=/dev/sdc bs=1k seek=32
sync
```

kernel and btb:

```
cp Image to Boot imx8qx
cp fsl-imx8qxp.dtb to Boot imx8qx
```

2 Device Tree

2.1 恩智浦的 device Tree 结构

Device Tree 的目录在

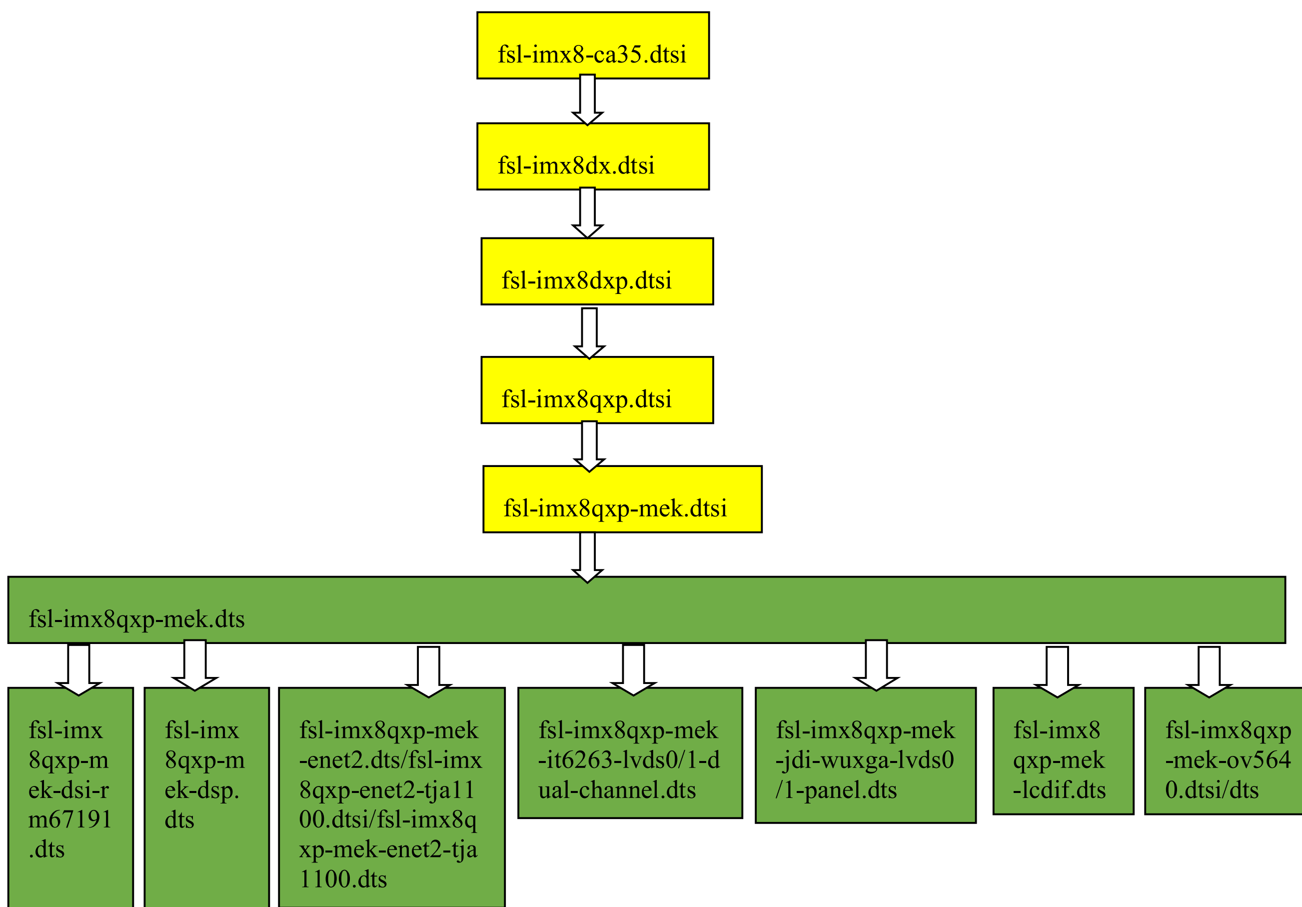
```
pwd
~/imx-yocto-bsp/standalone/linux-imx/arch/arm64/boot/dts/freescale
ls fsl-imx8*
```

```
.
├── fsl-imx8-ca35.dtsi
├── fsl-imx8dx-17x17-val.dts
├── fsl-imx8dx.dtsi
├── fsl-imx8dx-lpddr4-arm2.dts
├── fsl-imx8dxp.dtsi
├── fsl-imx8dxp-lpddr4-arm2.dts
├── fsl-imx8qxp-17x17-val.dts
├── fsl-imx8qxp-ddr3l-val.dts
├── fsl-imx8qxp.dtsi
├── fsl-imx8qxp-enet2-tja1100.dtsi
├── fsl-imx8qxp-lpddr4-arm2-a0.dts
├── fsl-imx8qxp-lpddr4-arm2-dsi-rm67191.dts
├── fsl-imx8qxp-lpddr4-arm2-dsp.dts
├── fsl-imx8qxp-lpddr4-arm2.dts
├── fsl-imx8qxp-lpddr4-arm2-enet2.dts
├── fsl-imx8qxp-lpddr4-arm2-enet2-tja1100.dts
├── fsl-imx8qxp-lpddr4-arm2-gpmi-nand.dts
```

```
├── fsl-imx8qxp-lpddr4-arm2-lpspi.dts
├── fsl-imx8qxp-lpddr4-arm2-lpspi-slave.dts
├── fsl-imx8qxp-lpddr4-arm2-mlb.dts
├──fsl-imx8qxp-lpddr4-arm2-mqs.dts
├── fsl-imx8qxp-lpddr4-arm2-spdif.dts
├──fsl-imx8qxp-lpddr4-arm2-wm8962.dts
├── fsl-imx8qxp-mek-a0.dts
├──fsl-imx8qxp-mek-dom0.dts
├── fsl-imx8qxp-mek-dsi-rm67191.dts
├── fsl-imx8qxp-mek-dsp.dts
├── fsl-imx8qxp-mek.dts
├──fsl-imx8qxp-mek.dtsi
├── fsl-imx8qxp-mek-enet2.dts
├── fsl-imx8qxp-mek-enet2-tja1100.dts
├── fsl-imx8qxp-mek-inmate.dts
├── fsl-imx8qxp-mek-it6263-lvds0-dual-channel.dts
├──fsl-imx8qxp-mek-it6263-lvds1-dual-channel.dts
├──fsl-imx8qxp-mek-jdi-wuxga-lvds0-panel.dts
├──fsl-imx8qxp-mek-jdi-wuxga-lvds1-panel.dts
├──fsl-imx8qxp-mek-lcdif.dts
├──fsl-imx8qxp-mek-lvds0-it6263.dtsi
├──fsl-imx8qxp-mek-lvds1-it6263.dtsi
├──fsl-imx8qxp-mek-ov5640.dts
├──fsl-imx8qxp-mek-ov5640.dtsi
├──fsl-imx8qxp-mek-ov5640-rpmsg.dts
├──fsl-imx8qxp-mek-root.dts
├──fsl-imx8qxp-mek-rpmsg.dts
├──fsl-imx8qxp-mek-rpmsg.dtsi
├──fsl-imx8qxp-xen.dtsi
├──
#include <dt-bindings/clock/imx8qxp-clock.h>
#include <dt-bindings/interrupt-controller/arm-gic.h>
/include/dt-bindings
#include <dt-bindings/interrupt-controller/arm-gic.h>
#include <dt-bindings/soc/imx_rsrc.h>
#include <dt-bindings/soc/imx8_hsio.h>
```

```
#include <dt-bindings/soc/imx8_pd.h>
#include <dt-bindings/clock/imx8qxp-clock.h>
#include <dt-bindings/input/input.h>
#include <dt-bindings/pinctrl/pads-imx8qxp.h>
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/thermal/thermal.h>
├── Makefile
```

其中 imx8qxp-mek 板的 dts 由以下文件组成:



- fsl-imx8-ca35.dtsi: SOC 的 A35 簇部分：包括 cpu,pmu 和 psci(Power State Coordination Interface).
- fsl-imx8dx.dtsi: SOC 中 i.MX8DX/DXP/QXP 共用部分。是 SOC 层的主要文件。
- fsl-imx8dxp.dtsi: SOC 中 i.MX8/DXP/QXP 共用部分：包括 vpu_decoder 与 i.MX8DX 相比有不同。
- fsl-imx8qxp.dtsi: SOC 中 i.MX8QXP 独有部分：比 i.MX8DXP 相比多出两个 CPU 核。

- fsl-imx8qxp-mek.dtsi:MEK 板级主要文件，包括所有外设驱动部分。

1. fsl-imx8qxp-mek.dts: MEK 板的 DTB 文件，直接包含 fsl-imx8qxp-mek.dtsi。
2. fsl-imx8qxp-mek-dsi-rm67191.dts: MEK 板支持两块直接连接的 rm67191 MiPi DSI 接口屏的 DTB 文件。
3. fsl-imx8qxp-mek-dsp.dts: MEK 板支持 HiFi DSP codec 的 DTB 文件。
4. fsl-imx8qxp-mek-enet2.dts/fsl-imx8qxp-enet2-tja1100.dtsi/fsl-imx8qxp-mek-enet2-tja1100.dts

   : MEK 板支持通过 enet2 连接到底板的 NXP tja1100 的 100Mhz 汽车级以太网 PHY。
5. fsl-imx8qxp-mek-it6263-lvds0/1-dual-channel.dts: MEK 板支持通过 LVDS0 或 LVDS1 接口的 dual channel 模式来连接 LVDS to HDMI 桥，来连接 HDMI 显示器(默认的 fsl-imx8qxp-mek.dts 是使用 single channel 模式来连接的。)
6. fsl-imx8qxp-mek-jdi-wuxga-lvds0/1-panel.dts：MEK 板支持通过 LVDS0 或 LVDS1 接口的 dual channel 模式来连接 jdi_tx26d202vm0bwa dual channel LVDS 屏
7. fsl-imx8qxp-mek-lcdif.dts：MEK 板支持通过 LCDIF 接口连接 Seiko 43WVFIG adapter card。
8. fsl-imx8qxp-mek-ov5640.dtsi/dts：MEK 板支持通过 MiPi CSI 接口连接 OV5649 Camera sensor。

编译 imx8qxp-mek 板 BSP 时，会合并相关的所有 dts/dtsi 文件，生成相应的 1~8 的 DTB 文件，默认 uboot 使用的 fdt_file=fsl-imx8qxp-mek.dtb。

2.2 device Tree 的由来(no updates)

“每次正式的 linux kernel release 之后都会有两周的 merge window，在这个窗口期间，kernel 各个部分的维护者都会提交各自的 patch，将自己测试稳定的代码请求并入 kernel main line。每到这个时候，Linus 就会比较繁忙，他需要从各个内核维护者的分支上取得最新代码并 merge 到自己的 kernel source tree 中。Tony Lindgren，内核 OMAP development tree 的维护者，发送了一个邮件给 Linus，请求提交 OMAP 平台代码修改，并给出了一些细节描述：

1、简单介绍本次改动

2、关于如何解决 merge conficts。有些 git mergetool 就可以处理，不能处理的，给出了详细介绍和解决方案

一切都很平常，也给出了足够的信息，然而，正是这个 pull request 引发了一场针对 ARM linux 的内核代码的争论。我相信 Linus 一定是对 ARM 相关的代码早就不爽了，ARM 的 merge 工作量较大倒在其次，主要是他认为 ARM 很多的代码都是垃圾，代码里面有若干愚蠢的 table，而多个人在维护这个 table，从而导致了冲突。因此，在处理完 OMAP 的 pull request 之后（Linus 并非针对 OMAP 平台，只是 Tony Lindgren 撞在枪口上了），他发出了怒吼：

Gaah. Guys, this whole ARM thing is a f*cking pain in the ass.

负责 ARM linux 开发的 Russell King 脸上挂不住，进行了反驳：事情没有那么严重，这次的 merge conficts 就是 OMAP 和 IMX/MXC 之间一点协调的问题，不能抹杀整个 ARM linux 团队的努力。其他的各个 ARM 平台维护者也加入讨论：ARM 平台如何复杂，如何庞大，对于 arm linux code 我们已经有一些思考，正在进行中……一时间，讨论的气氛有些尖锐，但总体是坦诚和友好的。

对于一件事情，不同层次的人有不同层次的思考。这次争论涉及的人包括：

1、内核维护者（CPU 体系结构无关的代码）

2、维护 ARM 系统结构代码的人

3、维护 ARM sub architecture 的人（来自各个 ARM SOC vendor）

维护 ARM sub architecture 的人并没有强烈的使命感，作为公司的一员，他们最大的目标是以最快的速度支持自己公司的 SOC，尽快的占领市场。这些人的软件功力未必强，对 linux kernel 的理解未必深入（有些人可能很强，但是人在江湖身不由己）。在这样的情况下，很多 SOC specific 的代码都是通过 copy and paste，然后稍加修改代码就提交了。此外，各个 ARM vendor 的 SOC family 是一长串的 CPU list，每个 CPU 多多少少有些不同，这时候 # ifdef 就充斥了各个源代码中，让 ARM mach-和 plat-目录下的代码有些不忍直视。

作为维护 ARM 体系结构的人，其能力不容置疑。以 Russell King 为首的 team 很好的维护了 ARM 体系结构的代码。基本上，除了 mach-和 plat-目录，其他的目录中的代码和目录组织是很好的。作为 ARM linux 的维护者，维护一个不断有新的 SOC 加入的 CPU architecture code 的确是一个挑战。在 Intel X86 的架构一统天下的时候，任何想正面攻击 Intel 的对手都败下阵来。想要击倒巨人（或者说想要和巨人并存）必须另辟蹊径。ARM 的策略有两个，一个是 focus 在嵌入式应用上，也就意味着要求低功耗，同时也避免了和 Intel 的正面对抗。另外一个就是博采众家之长，采用 license IP 的方式，让更多的厂商加入 ARM 建立的生态系统。毫无疑问，ARM 公司是成功的，但是这种模式也给 ARM linux 的维护者带来了噩梦。越来越多的芯片厂商加入 ARM 阵营，越来越多的 ARM platform 相关的代码被加入到内核，不同厂商的周边 HW block 设计又各不相同……

内核维护者是真正对操作系统内核软件有深入理解的人，他们往往能站在更高的层次上去观察问题，发现问题。Linus 注意到每次 merge window 中，ARM 的代码变化大约占整个 ARCH 目录的 60%，他认为这是一个很明显的符号，意味着 ARM linux 的代码可能存在问题。其实，60%这个比率的确很夸张，因为 unicore32 是在 2.6.39 merge window 中第一次全新提交，它的代码是全新的，但是其代码变化大约占整个 ARCH 目录的 9.6%（需要提及的是 unicore32 是一个中国芯）。有些维护 ARM linux 的人认为这是 CPU 市场占用率的体现，不是问题，直到内核维护者贴出实际的代码并指出问题所在。内核维护者当然想 linux kernel 支持更多的硬件平台，但是他们更愿意为 linux kernel 制定更长远的规划。例如：对于各种繁杂的 ARM 平台，用一个 kernel image 来支持。

经过争论，确定的问题如下：

1、ARM linux 缺少 platform（各个 ARM sub architecture，或者说各个 SOC）之间的协调，导致 arm linux 的代码有重复。值得一提的是在本次争论之前，ARM 维护者已经进行了不少相关的工作（例如 PM 和 clock tree）来抽象相同的功能模块。

2、ARM linux 中大量的 board specific 的源代码应该踢出 kernel，否则这些垃圾代码和 table 会影响 linux kernel 的长期目标。

3、各个 sub architecture 的维护者直接提交给 Linux 并入主线的机制缺乏层次。

新内核的解决之道：

针对 ARM linux 的现状，最需要解决的是人员问题，也就是如何整合 ARM sub architecture（各个 ARM Vendor）的资源。因此，内核社区成立了一个 ARM sub architecture 的 team，该 team 主要负责协调各个 ARM 厂商的代码（not ARM core part），Russell King 继续负责 ARM core part 的代码。此外，建立一个 ARM platform consolidation tree。ARM sub architecture team 负责 review 各个 sub architecture 维护者提交的代码，并在 ARM platform consolidation tree 上维护。在下一个 merge window 到来的时候，将 patch 发送给 Linus。

针对重复的代码问题，如果不同的 SOC 使用了相同的 IP block（例如 I2C controller），那么这个 driver 的 code 要从各个 arch/arm/mach-xxx 中独立出来，变成一个通用的模块供各个 SOC specific 的模块使用。移动到哪个目录呢？对于 I2C 或者 USB OTG 而言，这些 HW block 的驱动当然应该移动到 kernel/drivers 目录。因为，对于这些外设，可能是 in-chip，也可能是 off-chip 的，但是对于软件而言，它们是没有差别的（或者说好的软件抽象应该掩盖底层硬件的不同）。对于那些 system level 的 code 呢？例如 clock control、interrupt control。其实这些也不是 ARM-specific，应该属于 linux kernel 的核心代码，应该放到 linux/kernel 目录下，属于 core-Linux-kernel frameworks。当然对于 ARM 平台，也需要保存一些和 framework 交互的 code，这些 code 叫做 ARM SoC core architecture code。OK，总结一下：

1、ARM 的核心代码仍然保存在 arch/arm 目录下

2、ARM SoC core architecture code 保存在 arch/arm 目录下

3、ARM SOC 的周边外设模块的驱动保存在 drivers 目录下

4、ARM SOC 的特定代码在 arch/arm/mach-xxx 目录下

5、ARM SOC board specific 的代码被移除，由 Device Tree 机制来负责传递硬件拓扑和硬件资源信息。

OK，终于来到了 Device Tree 了。本质上，Device Tree 改变了原来用 hardcode 方式将 HW 配置信息嵌入到内核代码的方法，改用 bootloader 传递一个 DB 的形式。对于基于 ARM CPU 的嵌入式系统，我们习惯于针对每一个 platform 进行内核的编译。但是随着 ARM 在消费类电子上的广泛应用（甚至桌面系统、服务器系统），我们期望 ARM 能够象 X86 那样用一个 kernel image 来支持多个 platform。在这种情况下，如果我们认为 kernel 是一个 black box，那么其输入参数应该包括：

1、识别 platform 的信息

2、runtime 的配置参数

3、设备的拓扑结构以及特性

对于嵌入式系统，在系统启动阶段，bootloader 会加载内核并将控制权转交给内核，此外，还需要把上述的三个参数信息传递给 kernel，以便 kernel 可以有较大的灵活性。在 linux kernel 中，Device Tree 的设计目标就是如此。“

2.3 device Tree 的基础与语法(no updates)

简单的说，如果要使用 Device Tree，首先用户要了解自己的硬件配置和系统运行参数，并把这些信息组织成 Device Tree source file。通过 DTC（Device Tree Compiler），可以将这些适合人类阅读的 Device Tree source file 变成适合机器处理的 Device Tree binary file（有一个更好听的名字，DTB，device tree blob）。在系统启动的时候，boot program（例如：firmware、bootloader）可以将保存在 flash 中的 DTB copy 到内存（当然也可以通过其他方式，例如可以通过 bootloader 的交互式命令加载 DTB，或者 firmware 可以探测到 device 的信息，组织成 DTB 保存在内存中），并把 DTB 的起始地址传递给 client program（例如 OS kernel，bootloader 或者其他特殊功能的程序）。对于计算机系统（computer system），一般是 firmware->bootloader->OS，对于嵌入式系统，一般是 bootloader->OS。

2.3.1 Device Tree 的基本概念

在描述 Device Tree 的结构之前，我们先问一个基础问题：是否 Device Tree 要描述系统中的所有硬件信息？答案是否定的。基本上，那些可以动态探测到的设备是不需要描述的，例如 USB device。不过对于 SOC 上的 usb host controller，它是无法动态识别的，需要在 device tree 中描述。同样的道理，在 computer system 中，PCI device 可以被动态探测到，不需要在 device tree 中描述，但是 PCI bridge 如果不能被探测，那么就需要描述之。 需要描述的内容一般包括

- CPUs
- Memory
- Buses
- Peripheral connections
- Interrupt Controllers
- GPIO controllers
- Clock controllers

……

DTS 是一个由节点及其属性构成的一个简单的树，其优点主要有

- 内核在解析 device tree 是动态创建平台设备
- 平台设备的配置及数据可能通过 device tree 来描述
- 允许内核和板级专用的配置数据脱勾

Device tree 主要包括以下部分

- DTS: Device Tree Source: 文本描述文件， 在 arch/arm/boot/dts 下
- DTB:Device Treee Blob:由 DTS 通过特定的编译器编译出来的二进制文件， 也在 arch/arm/boot/dts 下,内核在启动时会解析它
- DTC:Device Tree Compiler:位于 script/dtc/dtc,编译命令为

```
make ARCH=arm imx6q-sabresd.dtb
```

为了了解 Device Tree 的结构，我们首先给出一个 Device Tree 的示例：

```
/ {   //root node “/”
 node1 { //child nodes “node1”,” node2”
   a-string-property = "A string"; //字符串型属性，由” ” 定义
   a-string-list-property = "first string", "second string"; //字符串列表型属性，由” ” 定义， “， “分开。
   a-byte-data-property = [0x01 0x23 0x34 0x56]; //BYTE 数据型属性，由[]定义。
   child-node1 { // children nodes of node1: "child-node1" and "child-node2"
     first-child-property;
     second-child-property = <1>;
     a-string-property = "Hello, world";
   };
   child-node2 {
   };
 };
 node2 {
   an-empty-property;
   a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */ //u32 数据型属性，由<>定义
   child-node1 {
   };
 };
};
```

2.3.2 节点 node

从上图中可以看出，device tree 的基本单元是 node。系统中的每个设备用一个 node 来描述，这些 node 被组织成树状结构，除了 root node，每个 node 都只有一个 parent。一个 device tree 文件中只能有一个 root node。每个 node 中包含了若干的 property/value 来描述该 node 的一些特性。每个 node 用节点名字（node name）标识，节点名字的格式是

[label:]node-name[@unit-address]。如果该 node 没有 reg 属性（后面会描述这个 property），那么该节点名字中必须不能包括@和 unit-address。unit-address 的具体格式是和设备挂在那个 bus 上相关。例如对于 cpu，其 unit-address 就是从 0 开始编址，以此加一。而具体的设备，例如以太网控制器，其 unit-address 就是寄存器地址。root node 的 node name 是确定的，必须是“/”。

- node-name 说明了何种设备，必须使用字符开头，

Table 2-1 Characters for node names

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

- unit-address:访问此设备的主地址，必须唯一，必须和此节点的 reg 属性的开始地址一致

2.3.3 根节点 root node

只能有一个 root node,它用来描述从 CPU 端看到的地址空间，至少需要用 cpu 和 memory 节点组成，如下:

```
/ {
	model = "NXP i.MX6 Quad SABRE Smart Device Board";
	compatible = "fsl,imx6q-sabresd", "fsl,imx6q";
cpus {
		#address-cells = <1>;
		#size-cells = <0>;
		cpu0: cpu@0 {
…
		};
		cpu@1 {
…
		};
		cpu@2 {
…
		};
		cpu@3 {
```

```
…
                    };
          };
          soc {
…
          }
          memory {
                    reg = <0x10000000 0x40000000>;
          };
}
```

2.3.4 别名节点

aliases 节点定义了一些别名。为何要定义这个 node 呢？因为 Device tree 是树状结构，当要引用一个 node 的时候要指明相对于 root node 的 full path，例如/node1/child-node1。如果多次引用，每次都要写这么复杂的字符串多少是有些麻烦，因此可以在 aliases 节点定义一些设备节点 full path 的缩写。

```
/ {
    aliases {
        mxcfb0 = &mxcfb1;
        mxcfb1 = &mxcfb2;
        mxcfb2 = &mxcfb3;
        mxcfb3 = &mxcfb4;
    };
```

*Given the alias mxcfb0, it will lookup aliases nodes for the full device path

```
Example
aliases {
serial0 = "/simple-bus@fe000000/serial@llc500";
ethernet0 = "/simple-bus@fe000000/ethernet@31c000";
}
```

2.3.5 CPU 节点

- Describes CPUs or cores in the system
- Standard properties include: reg, clock-frequency, reservation-granule-size, etc
- TLB, L1 cache, as well as multi level and shared caches can be described

```
cpus {
```

```
		#address-cells = <1>;
		#size-cells = <0>;
		cpu0: cpu@0 {
			compatible = "arm,cortex-a9";
			device_type = "cpu";
			reg = <0>;
			next-level-cache = <&L2>;
			operating-points = <
				/* kHz    uV */
				1200000 1275000
				996000  1250000
				852000  1250000
				792000  1175000
				396000  975000
			>;
			fsl,soc-operating-points = <
				/* ARM kHz  SOC-PU uV */
				1200000 1275000
				996000  1250000
				852000  1250000
				792000  1175000
				396000  1175000
			>;
			clock-latency = <61036>; /* two CLK32 periods */
			clocks = <&clks IMX6QDL_CLK_ARM>,
				 <&clks IMX6QDL_CLK_PLL2_PFD2_396M>,
				 <&clks IMX6QDL_CLK_STEP>,
				 <&clks IMX6QDL_CLK_PLL1_SW>,
				 <&clks IMX6QDL_CLK_PLL1_SYS>,
				 <&clks IMX6QDL_PLL1_BYPASS>,
				 <&clks IMX6QDL_CLK_PLL1>,
				 <&clks IMX6QDL_PLL1_BYPASS_SRC> ;
			clock-names = "arm", "pll2_pfd2_396m", "step",
				      "pll1_sw", "pll1_sys", "pll1_bypass", "pll1", "pll1_bypass_src";
			arm-supply = <&reg_arm>;
			pu-supply = <&reg_pu>;
```

```
            soc-supply = <&reg_soc>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <1>;
            next-level-cache = <&L2>;
        };
        cpu@2 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <2>;
            next-level-cache = <&L2>;
        };
        cpu@3 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <3>;
            next-level-cache = <&L2>;
        };
```

2.3.6 Memory 节点

memory device node 是所有设备树文件的必备节点，它定义了系统物理内存的 layout。device_type 属性定义了该 node 的设备类型，例如 cpu、serial 等。对于 memory node，其 device_type 必须等于 memory。reg 属性定义了访问该 device node 的地址信息，该属性的值被解析成任意长度的（address，size）数组，具体用多长的数据来表示 address 和 size 是在其 parent node 中定义（#address-cells 和#size-cells）。对于 device node，reg 描述了 memory-mapped IO register 的 offset 和 length。对于 memory node，定义了该 memory 的起始地址和长度。

Table 3-3 Memory node properties

Property Name	Usage	Value Type	Definition
device_type	R	<string>	Value shall be "memory".
reg	R	<prop-encoded-array>	Consists of an arbitrary number of address and size pairs that specify the physical address and size of the memory ranges.
initial-mapped-area	O	<prop-encoded-array>	Specifies the address and size of the Initial Mapped Area (see section 5.3). Is a prop-encoded-array consisting of a triplet of (*effective address, physical address, size*). The effective and physical address shall each be 64-bit (<u64> value), and the size shall be 32-bits (<u32> value).
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

一个 32bit 的内存分布示例：

RAM: starting address 0x0, length 0x40000000 (1GB)

```
        #address-cells = 1 and a #size-cells = 1:
memory {
            device_type = "memory";
            reg = <0x10000000 0x40000000>;
 };
```

一个 64bit 的内存分布示例：（64bit 由两个 32bit 来描述）

RAM: starting address 0x0, length 0x80000000 (2GB)

RAM: starting address 0x100000000, length 0x100000000 (4GB)

方法 1：

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

方法 2：

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

2.3.7 可选节点

chosen node 主要用来描述由系统 firmware 指定的 runtime parameter。如果存在 chosen 这个 node，其 parent node 必须是名字是“/”的根节点。原来通过 tag list 传递的一些 linux kernel 的运行时参数可以通过 Device Tree 传递。例如 command line 可以通过 bootargs 这个 property 这个属性传递；initrd 的开始地址也可以通过 linux,initrd-start 这个 property 这个属性传递。在实际中，建议增加一个 bootargs 的属性，例如：

```
chosen {        bootargs = "console=ttymxc0,115200";    };
```

我们知道，device tree 用于 HW platform 识别，runtime parameter 传递以及硬件设备描述。chosen 节点并没有描述任何硬件设备节点的信息，它只是传递了 runtime parameter。

Table 3-4 Chosen node properties

Property Name	Usage	Value Type	Definition
bootargs	O	<string>	A string that specifies the boot arguments for the client program. The value could potentially be a null string if no boot arguments are required.
stdout-path	O	<string>	A string that specifies the full path to the node representing the device to be used for boot console output. If the character ":" is present in the value it terminates the path. The value may be an alias. If the stdin-path property is not specified, stdout-path should be assumed to define the input device.
stdin-path	O	<string>	A string that specifies the full path to the node representing the device to be used for boot console input. If the character ":" is present in the value it terminates the path. The value may be an alias.
Usage legend: R=Required, O=Optional, OR=Optional but Recommended, SD=See Definition Note: All other standard properties (section 2.3) are allowed but are optional.			

2.3.8 属性

了解了基本的 device tree 的结构后，我们总要把这些结构体现在 device tree source code 上来。在 linux kernel 中，扩展名是 dts 的文件就是描述硬件信息的 device tree source file，在 dts 文件中，一个 node 被定义成：

```
[label:] node-name[@unit-address] {
  [properties definitions]
  [child nodes]
}
```

“[]”表示 option，因此可以定义一个只有 node name 的空节点。label 方便在 dts 文件中引用，具体后面会描述。child node 的格式和 node 是完全一样的，因此，一个 dts 文件中就是若干嵌套组成的 node，property 以及 child note、child note property 描述。

各家 ARM vendor 也会共用一些硬件定义信息，这个文件就是 skeleton.dtsi。我们自下而上（类似 C＋＋中的从基类到顶层的派生类）逐个进行分析。

1、skeleton.dtsi。位于 linux-3.14\arch\arm\boot\dts 目录下，具体该文件的内容如下：

```
/ {
   #address-cells = <1>;
   #size-cells = <1>;
   chosen { };
   aliases { };
   memory { device_type = "memory"; reg = <0 0>; };
};
```

device tree 顾名思义是一个树状的结构，既然是树，必然有根。“/”是根节点的 node name。“{”和“}”之间的内容是该节点的具体的定义，其内容包括各种属性的定义以及 child node 的定义。chosen、aliases 和 memory 都是 sub node，sub node 的结构和 root node 是完全一样的，因此，sub node 也有自己的属性和它自己的 sub node，最终形成了一个树状的 device tree。属性的定义采用

property ＝ value 的形式。例如#address-cells 和#size-cells 就是 property，而<1>就是 value。value 有以下几种情况：

1）属性值是空<empty> Value is empty—used for conveying true-false information.

2）属性值是 32bit unsigned integers，用尖括号表示。例如#size-cells = <1> , <u32> A 32-bit integer in big-endian format. Example: the 32-bit value 0x11223344 would be represented in memory as:

address	0x11
address+1	0x22
address+2	0x33
address+3	0x44

```
interrupts = <17 0xc>;
```

3）属性值是 64bit unsigned integers，用尖括号表示。<u64> -- A 64-bit integer in big-endian format, would be represented as two <u32> cells

```
clock-frequency = <0x00000001 0x00000000>;
```

4）<phandle> -- specifies a numerical identifier for a node that is unique within the device tree

5）属性值是 binary data，用方括号表示。例如 binary-property = [0x01 0x23 0x45 0x67] <bytestring> -- each byte represented by two hexadecimal digits

```
local-mac-address = [00 00 12 34 56 78];
```

or equivalently:

```
local-mac-address = [000012345678];
```

6）属性值是 text string, 用双引号表示。例如 device_type = "memory" <string> -- Strings are printable and NULL-terminated

7）属性值是 string list，用双引号表示。<stringlist> -- A list of <string> values concatenated together

2.3.9 Compatible 属性和 model 属性

在描述 compatible 属性之前要先描述 model 属性。model 属性指明了该设备属于哪个设备生产商的哪一个 model。一般而言，我们会给 model 赋值"manufacturer,model"。例如 model = "fsl,imx6q-sabresd"。fsl 是生产商，imx6q-sabresd 是 model 类型，指明了具体的是哪一个系列的 SOC。OK，现在我们回到 compatible 属性，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 driver 来驱动该设备。假设定义该属性：compatible = "aaaaaa", "bbbbb"。那么操作操作系统可能首先使用 aaaaaa 来匹配适合的 driver，如果没有匹配到，那么使用字符串 bbbbb 来继续寻找适合的 driver，对于本例，compatible = "fsl,imx6q-sabresd", "fsl,imx6q";，这里只定义了一个 list。对于 root node，compatible

属性是用来匹配 machine type 的（在 device tree 代码分析文章中会给出更细致的描述）。对于普通的 HW block 的节点，例如 interrupt-controller，compatible 属性是用来匹配适合的 driver 的。

2.3.10 Phandle 属性

A <u32> value specifies a numerical identifier for a node that is unique within the device tree.

Used by other nodes that need to refer to the node associated with the property.

Example

```
pic@10000000 {
phandle = <1>;
interrupt-controller;
};
```

*A phandle value of 1 is defined. Another device node could reference the pic node with a phandle value of 1:

```
interrupt-parent = <1>;
```

The DTC tool automatically inserts the phandle properties when the DTS is compiled into the binary DTB format when no explicit phandle properties.

2.3.11 属性标签

Defines a human readable string describing a device

```
[label:] node-name[@unit-address]
[label:] property-name = value;
[label:] property-name;
```

Labels may also appear before or after any component of a property value, or between cells of a cell array, or between bytes of a bytestring.

```
reg = reglabel: <0 sizelabel: 0x1000000>;
prop = [ab cd ef byte4: 00 ff fe];
str = start: "string value" end: ;
```

示例 1

```
intc: interrupt-controller@00a01000 {
    compatible = "arm,cortex-a9-gic";
    #interrupt-cells = <3>;
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-controller;
    reg = <0x00a01000 0x1000>,
          <0x00a00100 0x100>;
};
```

The label 'intc:' label is used to assign a phandle to the interrupt-parent property in the root node.

```
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    interrupt-parent = <&intc>;
    ranges;
```

```
intc: interrupt-controller@00a01000 {
    compatible = "arm,cortex-a9-gic";
    #interrupt-cells = <3>;
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-controller;
    reg = <0x00a01000 0x1000>,
          <0x00a00100 0x100>;
};
```

References & followed by a node's label in a <> to denote phandle.

interrupt-parent = < &intc >;

or they may be & followed by a node's full path in braces.

interrupt-parent = < &{/soc/interrupt-controller@00a01000} >;

示例 2

```
#include "imx6q-sabresd.dts"

&cpu0 {
    arm-supply = <&reg_arm>;
    soc-supply = <&reg_soc>;
    pu-supply = <&reg_pu>; /* u
};
```

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu0: cpu@0 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <0>;
        next-level-cache = <&L2>;
        arm-supply = <&reg_arm>;
        pu-supply = <&reg_pu>;
        soc-supply = <&reg_soc>;
    };
```

Outside a cell array, a reference to another node will be expanded to that node's full path.

ethernet0 = &EMAC0;

&cpu0 expands to "/cpus/cpu@0"

2.3.12 寻址属性

如果一个 device node 中包含了有寻址需求（要定义 reg property）的 sub node（后文也许会用 child node，和 sub node 是一样的意思），那么就必须要定义这两个属性。“#”是 number 的意思，#address-cells 这个属性是用来描述 sub node 中的 reg 属性的地址域特性的，也就是说需要用多少个 u32 的 cell 来描述该地址域。同理可以推断#size-cells 的含义，下面对 reg 的描述中会给出更详细的信息。

- **#address-cells** :

  defines the number of <u32> cells used to encode the address field in a child node’s reg property

- **#size-cells**

  property defines the number of <u32> cells used to encode the size field in a child node’s reg property.

- **reg:**

  A list of tuples in the form

reg = <address1 length1 [address2 length2] [address3 length3] ... >

The reg property is interpreted by its parent node’s #address-cells and #size-cells.

示例 1

- assuming #address-cells and #size-cells are both 1,

  reg = <0x3000 0x20 0xFE00 0x100>;

will be decoded as 2 memory blocks of 0x3000…0x3020 and 0xFE00…0xFF00.

示例 2

- cpus are assigned addresses 0 and 1 and no size field.

```
cpu0: cpu@0 {
    compatible = "arm,cortex-a9";
    device_type = "cpu";
    reg = <0>;
};

cpu@1 {
    compatible = "arm,cortex-a9";
    device_type = "cpu";
    reg = <1>;
    next-level-cache = <&L2>;
};
```

示例 3

```
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    interrupt-parent = <&intc>;
    ranges;

    dma_apbh: dma-apbh@00110000 {
        compatible = "fsl,imx6q-dma-apbh
        reg = <0x00110000 0x2000>;
        interrupts = <0 13 0x04>, <0 13
        interrupt-names = "gpmi0", "gpmi
        #dma-cells = <1>;
        dma-channels = <4>;
        clocks = <&clks 106>;
    };
```

示例 4

```
external-bus {
    #address-cells = <2>
    #size-cells = <1>;

    ethernet@0,0 {
        compatible = "smc,smc91c111";
        reg = <0 0 0x1000>;
    };

    i2c@1,0 {
        compatible = "acme,a1234-i2c-bus";
        reg = <1 0 0x1000>;
        rtc@58 {
            compatible = "maxim,ds1338";
        };
    };

    flash@2,0 {
        compatible = "samsung,k8f1315ebm", "cfi-flash";
        reg = <2 0 0x4000000>;
    };
};
```

the addresses form are <0 0 0>, <1 0 0>, <2 0 0>. It represent <chip select number, offset, length>.

2.3.13 地址翻译

- 地址区域
  - The root-direct children describe the CPU's view of the address space (memory mapped address).
  - Root-indirect children use the parent domain defined by their parent.
  - Parent nodes are free to define whatever addressing scheme makes sense for the bus.
  - Root-indirect children need be translated to a root address domain.
- 地址范围
  - Ranges is a list of address translations.

- Each entry in the ranges table is a tuple in the form of (child-bus-address, parent-bus-address, length).
- The size of each field is determined by taking the child's #address-cells value, the parent's #address-cells value, and the child's #size-cells value.

示例 1:

Take the range entry <0 0 0x10100000 0x10000> for example:

"0 0" is child address,

"0x10100000" is parent address

"0x10000" is the child address length

示例 2:

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    ...
    external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0  0x10100000   0x10000     // Chipselect 1, Ethernet
                  1 0  0x10160000   0x10000     // Chipselect 2, i2c controller
                  2 0  0x30000000   0x1000000>; // Chipselect 3, NOR Flash

        ethernet@0,0 {...};
        i2c@1,0 {...};
        flash@2,0 {...};
    };
};
```

- Ranges is empty

  If it's defined with an <empty> value, it specifies that the parent and child address space is identical, and no address translation is required.

```
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    interrupt-parent = <&intc>;
    ranges;

    dma_apbh: dma-apbh@00110000 {
        compatible = "fsl,imx6q-dma-apbh", "fsl,imx28-dma-apbh";
        reg = <0x00110000 0x2000>;
        interrupts = <0 13 0x04>, <0 13 0x04>, <0 13 0x04>, <0 13 0x04>;
        interrupt-names = "gpmi0", "gpmi1", "gpmi2", "gpmi3";
        #dma-cells = <1>;
        dma-channels = <4>;
        clocks = <&clks 106>;
    };
```

2.3.14 中断

- **interrupt-controller** - An empty property declaring a node as a device that receives interrupt signals
- **#interrupt-cells** - a property of the interrupt controller node. It states how many cells are in an *interrupt specifier* for this interrupt controller (Similar to #address-cells and #size-cells).

- **interrupt-parent** - specify a *phandle* to the interrupt controller that it is attached to.

*Nodes that do not have an interrupt-parent property can also inherit the property from their parent node.

- **interrupts** - A property of a device node containing a list of *interrupt specifiers*, one for each interrupt output signal on the device.

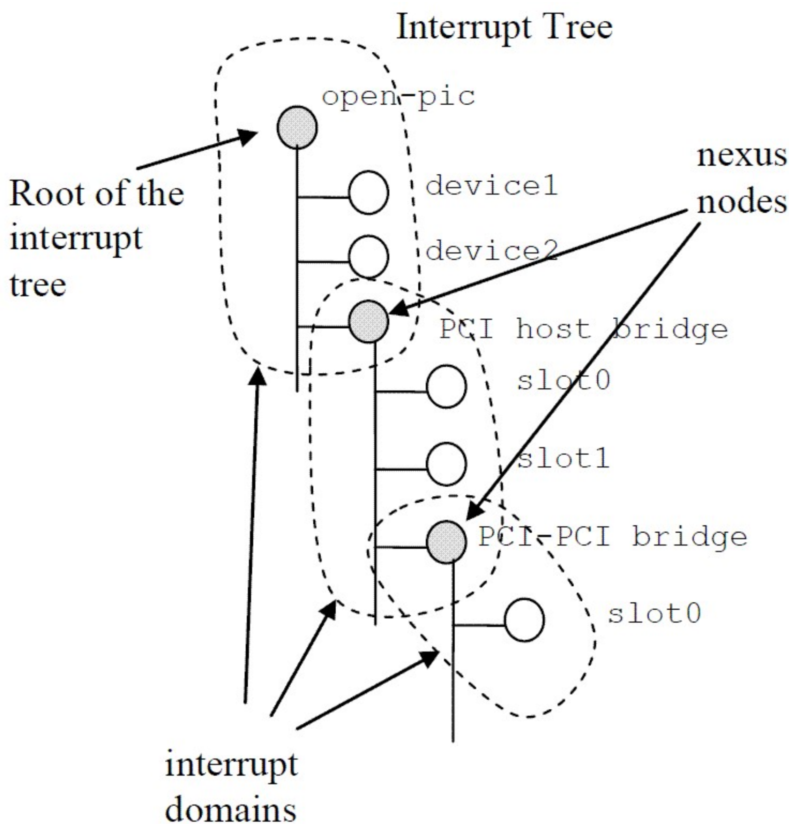
示例 1

```
gpio2: gpio@020a0000 {
    compatible = "fsl,imx6q-gpio", "fsl,imx35-gpio";
    reg = <0x020a0000 0x4000>;
    interrupts = <0 68 0x04 0 69 0x04>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};
egalax_ts@04 {
    compatible = "eeti,egalax_ts";
    reg = <0x04>;
    interrupt-parent = <&gpio2>;
    interrupts = <28 2>;
    wakeup-gpios = <&gpio2 28 0>;
};
```

具体各个 HW block 的 interrupt source 是如何物理的连接到 interruptcontroller 的呢？在 dts 文件中是用 interrupt-parent 这个属性来标识的。且慢，这里定义 interrupt-parent 属性的是 root node，难道 root node 会产生中断到 interrupt controller 吗？当然不会，只不过如果一个能够产生中断的 device node 没有定义 interrupt-parent 的话，其 interrupt-parent 属性就是跟随 parent node。因此，与其在所有的下游设备中定义 interrupt-parent，不如统一在 root node 中定义了。

intc 是一个 lable，标识了一个 device node（在本例中是标识了 intc: interrupt-controller@00a01000 这个 device node）。实际上，interrupt-parent 属性值应该是是一个 u32 的整数值(这个整数值在 Device Tree 的范围内唯一识别了一个 device node，也就是 phandle），不过，在 dts 文件中中，可以使用类似 c 语言的 Labels and References 机制。定义一个 lable，唯一标识一个 node 或者 property，后续可以使用&来引用这个 lable。DTC 会将 lable 转换成 u32 的整数值放入到 DTB 中，用户层面就不再关心具体转换的整数值了。

关于 interrupt，我们值得进一步描述。在 Device Tree 中，有一个概念叫做 interrupt tree，也就是说 interrupt 也是一个树状结构。我们以下图为例(该图来自 Power_ePAPR_APPROVED_v1.1)：



系统中有一个 interrrupt tree 的根节点，device1、device2 以及 PCI host bridge 的 interrupt line 都是连接到 root interrupt controller 的。PCI host bridge 设备中有一些下游的设备，也会产生中断，但是他们的中断都是连接到 PCI host bridge 上的 interrupt controller（术语叫做 interrupt nexus），然后报告到 root interrupt controller 的。每个能产生中断的设备都可以产生一个或者多个 interrupt，每个 interrupt source（另外一个术语叫做 interrupt specifier，描述了 interrupt source 的信息）都是限定在其所属的 interrupt domain 中。

在了解了上述的概念后，我们可以回头再看看 interrupt-parent 这个属性。其实这个属性是建立 interrupt tree 的关键属性。它指明了设备树中的各个 device node 如何路由 interrupt event。另外，需要提醒的是 interrupt controller 也是可以级联的，上图中没有表示出来。那么在这种情况下如何定义 interrupt tree 的 root 呢？那个没有定义 interrupt-parent 的 interrupt controller 就是 root。

```
intc: interrupt-controller@00a01000 {
        compatible = "arm,cortex-a9-gic";
        #interrupt-cells = <3>;
        #address-cells = <1>;
        #size-cells = <1>;
        interrupt-controller;
```

```
        reg = <0x00a01000 0x1000>,
              <0x00a00100 0x100>;
    };
```

2.3.15 节点状态

- **Property**: status
- **Value type**: <string>
- **Description**: -- indicates the operational status of a device

Value	Description
"okay"	Indicates the device is operational
"disabled"	Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). Refer to the device binding for details on what *disabled* means for a given device.
"fail"	Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair.
"fail-sss"	Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The *sss* portion of the value is specific to the device and indicates the error condition detected.

可以使用 okay 和 disabled 来打开和注掉某一个驱动的设备注册。

2.3.16 Bindings

- A binding documents what a particular compatible value means, what properties it should have, what child nodes it might have, and what device it represents
  - Located in Documentation/devicetree/bindings/
  - Each binding page specifies the properties and child nodes that are expected and/or allowed for the binding
  - grouped into categories to make them easy to find an understand
  - Please read the binding doc for each module under Documentation/devicetree/bindings/ before you create the DTS file for the first time

```
$ ls Documentation/devicetree/bindings/pinctrl | grep fsl*
fsl,imx35-pinctrl.txt
fsl,imx51-pinctrl.txt
fsl,imx53-pinctrl.txt
fsl,imx6dl-pinctrl.txt
fsl,imx6q-pinctrl.txt
fsl,imx6sl-pinctrl.txt
fsl,imx-pinctrl.txt
fsl,mxs-pinctrl.txt
fsl,vf610-pinctrl.txt
```

Documentation/devicetree/binding/pinctrl/fsl,imx6q-pinctrl.txt

```
* NXP IMX6Q IOMUX Controller

Please refer to fsl,imx-pinctrl.txt in this directory for common binding part
and usage.

Required properties:
- compatible: "fsl,imx6q-iomuxc"
- fsl,pins: two integers array, represents a group of pins mux and config
  setting. The format is fsl,pins = <PIN_FUNC_ID CONFIG>, PIN_FUNC_ID is a
  pin working on a specific function, CONFIG is the pad setting value like
  pull-up for this pin. Please refer to imx6q datasheet for the valid pad
  config settings.

CONFIG bits definition:
PAD_CTL_HYS                 (1 << 16)
PAD_CTL_PUS_100K_DOWN       (0 << 14)
PAD_CTL_PUS_47K_UP          (1 << 14)
PAD_CTL_PUS_100K_UP         (2 << 14)
PAD_CTL_PUS_22K_UP          (3 << 14)
PAD_CTL_PUE                 (1 << 13)
PAD_CTL_PKE                 (1 << 12)
PAD_CTL_ODE                 (1 << 11)
PAD_CTL_SPEED_LOW           (1 << 6)
PAD_CTL_SPEED_MED           (2 << 6)
PAD_CTL_SPEED_HIGH          (3 << 6)
PAD_CTL_DSE_DISABLE         (0 << 3)
PAD_CTL_DSE_240ohm          (1 << 3)
PAD_CTL_DSE_120ohm          (2 << 3)
PAD_CTL_DSE_80ohm           (3 << 3)
PAD_CTL_DSE_60ohm           (4 << 3)
PAD_CTL_DSE_48ohm           (5 << 3)
PAD_CTL_DSE_40ohm           (6 << 3)
PAD_CTL_DSE_34ohm           (7 << 3)
PAD_CTL_SRE_FAST            (1 << 0)
PAD_CTL_SRE_SLOW            (0 << 0)

Refer to imx6q-pinfunc.h in device tree source folder for all available
imx6q PIN_FUNC_ID.
```

Arch\arm\boot\dts\imx6q-pinfunc.h

```
#define MX6QDL_PAD_SD2_DAT1__SD2_DATA1      0x04c 0x360 0x000 0x0 0x0
#define MX6QDL_PAD_SD2_DAT1__ECSPI5_SS0     0x04c 0x360 0x834 0x1 0x0
#define MX6QDL_PAD_SD2_DAT1__EIM_CS2_B      0x04c 0x360 0x000 0x2 0x0
#define MX6QDL_PAD_SD2_DAT1__AUD4_TXFS      0x04c 0x360 0x7c8 0x3 0x0
#define MX6QDL_PAD_SD2_DAT1__KEY_COL7       0x04c 0x360 0x8f0 0x4 0x0
#define MX6QDL_PAD_SD2_DAT1__GPIO1_IO14     0x04c 0x360 0x000 0x5 0x0
```

2.3.17 DTC&DTB

- dtc tool locates at scripts/dtc/, it is built out by rule hostprogs-y in scripts/dtc/Makefile

```
1 # scripts/dtc makefile
2
3 hostprogs-y := dtc
4 always      := $(hostprogs-y)
5
6 dtc-objs    := dtc.o flattree.o fstree.o data.o livetree.o treesource.o \
7                srcpos.o checks.o util.o
8 dtc-objs    += dtc-lexer.lex.o dtc-parser.tab.o
```

```
dtb-$(CONFIG_ARCH_MXC) += \
    imx25-karo-tx25.dtb \
    imx25-pdk.dtb \
    imx27-apf27.dtb \
    imx27-apf27dev.dtb \
    imx27-pdk.dtb \
    imx27-phytec-phycore.dtb \
    imx31-bug.dtb \
    imx51-apf51.dtb \
    imx51-apf51dev.dtb \
    imx51-babbage.dtb \
    imx53-ard.dtb \
    imx53-evk.dtb \
    imx53-mba53.dtb \
    imx53-qsb.dtb \
    imx53-smd.dtb \
    imx6dl-sabreauto.dtb \
    imx6dl-sabreauto-gpmi-weim.dtb \
    imx6dl-sabresd.dtb \
    imx6dl-sabresd-hdcp.dtb \
    imx6dl-sabresd-ldo.dtb \
    imx6dl-wandboard.dtb \
    imx6q-arm2.dtb \
    imx6q-sabreauto.dtb \
    imx6q-sabreauto-gpmi-weim.dtb \
    imx6q-sabrelite.dtb \
    imx6q-sabresd.dtb \
    imx6q-sabresd-hdcp.dtb \
    imx6q-sabresd-ldo.dtb \
    imx6q-sbc6x.dtb \
    imx6sl-evk.dtb \
    imx6sl-evk-ldo.dtb \
    vf610-twr.dtb
```

- dtb targes:

arch/arm/boot/dts/Makefile defines

which .dtb targets will be built out

- build command:

scripts/dtc/dtc -I dts -O dtb -o */path/to/*my-tree.dtb /path/to/my-tree.dts

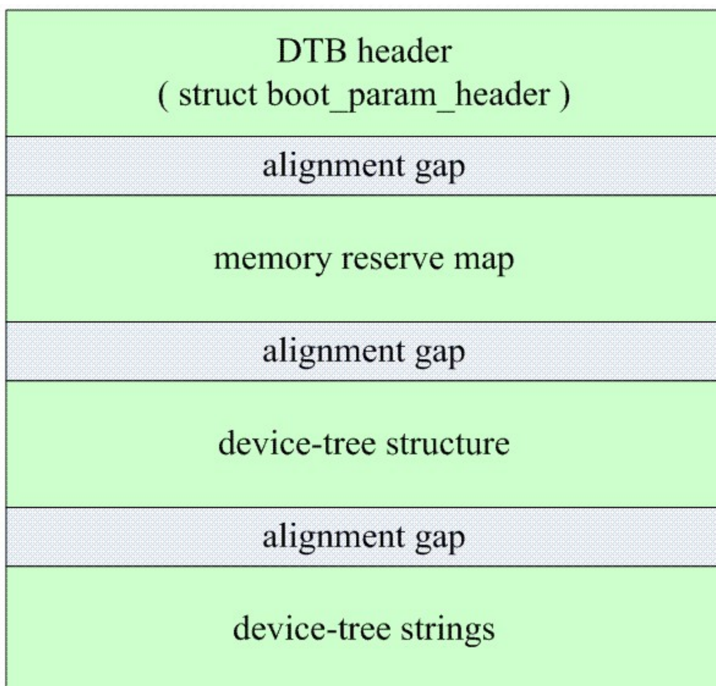
- libfdt

./scripts/dtc/libfdt manipulate binary DT

2.3.18 Device Tree binary 格式

2.3.18.1 DTB 整体结构

经过 Device Tree Compiler 编译，Device Tree source file 变成了 Device Tree Blob（又称作 flattened device tree）的格式。Device Tree Blob 的数据组织如下图所示：



2.3.18.2 DTB header。

对于 DTB header，其各个成员解释如下：

header field name	description
magic	用来识别 DTB 的。通过这个 magic，kernel 可以确定 bootloader 传递的参数 block 是一个 DTB 还是 tag list。
totalsize	DTB 的 total size
off_dt_struct	device tree structure block 的 offset
off_dt_strings	device tree strings block 的 offset

off_mem_rsvmap	offset to memory reserve map。有些系统，我们也许会保留一些 memory 有特殊用途（例如 DTB 或者 initrd image），或者在有些 DSP+ARM 的 SOC platform 上，有写 memory 被保留用于 ARM 和 DSP 进行信息交互。这些保留内存不会进入内存管理系统。
version	该 DTB 的版本。
last_comp_version	兼容版本信息
boot_cpuid_phys	我们在哪一个 CPU（用 ID 标识）上 booting
dt_strings_size	device tree strings block 的 size。和 off_dt_strings 一起确定了 strings block 在内存中的位置
dt_struct_size	device tree structure block 的 size。和和 off_dt_struct 一起确定了 device tree structure block 在内存中的位置

2.3.18.3 memory reserve map 的格式描述

这个区域包括了若干的 reserve memory 描述符。每个 reserve memory 描述符是由 address 和 size 组成。其中 address 和 size 都是用 U64 来描述。

2.3.18.4 device tree structure block 的格式描述

device tree structure block 区域是由若干的分片组成，每个分片开始位置都是保存了 token，以此来描述该分片的属性和内容。共计有 5 种 token：

（1）FDT_BEGIN_NODE (0x00000001)。该 token 描述了一个 node 的开始位置，紧挨着该 token 的就是 node name（包括 unit address）

（2）FDT_END_NODE (0x00000002)。该 token 描述了一个 node 的结束位置。

（3）FDT_PROP (0x00000003)。该 token 描述了一个 property 的开始位置，该 token 之后是两个 u32 的数据，分别是 length 和 name offset。length 表示该 property value data 的 size。name offset 表示该属性字符串在 device tree strings block 的偏移值。length 和 name offset 之后就是长度为 length 具体的属性值数据。

（4）FDT_NOP (0x00000004)。

（5）FDT_END (0x00000009)。该 token 标识了一个 DTB 的结束位置。

一个可能的 DTB 的结构如下：

（1）若干个 FDT_NOP（可选）

（2）FDT_BEGIN_NODE

node name

paddings

（3）若干属性定义。

（4）若干子节点定义。

（5）若干个 FDT_NOP（可选）

（6）FDT_END

2.3.18.5 device tree strings bloc 的格式描述

device tree strings bloc 定义了各个 node 中使用的属性的字符串表。由于很多属性会出现在多个 node 中，因此，所有的属性字符串组成了一个 string block。这样可以压缩 DTB 的 size。

2.4 device Tree 的代码分析(no updates)

本节主要内容是：以 Device Tree 相关的数据流分析为索引，对 ARM linux kernel 的代码进行解析。主要的数据流包括：

1、初始化流程。也就是扫描 dtb 并将其转换成 Device Tree Structure。

2、传递运行时参数传递以及 platform 的识别流程分析

3、如何将 Device Tree Structure 并入 linux kernel 的设备驱动模型。

2.4.1 如何通过 Device Tree 完成运行时参数传递以及 platform 的识别功能？

2.4.1.1 汇编部分的代码分析

linux/arch/arm/kernel/head.S 文件定义了 bootloader 和 kernel 的参数传递要求：

```
MMU = off, D-cache = off, I-cache = dont care, r0 = 0, r1 = machine nr, r2 = atags or dtb pointer.
```

目前的 kernel 支持旧的 tag list 的方式，同时也支持 device tree 的方式。r2 可能是 device tree binary file 的指针（bootloader 要传递给内核之前要 copy 到 memory 中），也可以能是 tag list 的指针。在 ARM 的汇编部分的启动代码中（主要是 head.S 和 head-common.S），machine type ID 和指向 DTB 或者 atags 的指针被保存在变量__machine_arch_type 和__atags_pointer 中，这么做是为了后续 c 代码进行处理。

2.4.1.2 和 device tree 相关的 setup_arch 代码分析

具体的 c 代码都是在 setup_arch 中处理，这个函数是一个总的入口点。具体代码如下（删除了部分无关代码）：

//arch\arm\kernel\setup.c

```
void __init setup_arch(char **cmdline_p)
{
    const struct machine_desc *mdesc;

……

    mdesc = setup_machine_fdt(__atags_pointer);
    if (!mdesc)
        mdesc = setup_machine_tags(__atags_pointer, __machine_arch_type);
    machine_desc = mdesc;
    machine_name = mdesc->name;

……
}
```

对于如何确定 HW platform 这个问题，旧的方法是静态定义若干的 machine 描述符（struct machine_desc ），在启动过程中，通过 machine type ID 作为索引，在这些静态定义的 machine 描述符中扫描，找到那个 ID 匹配的描述符。在新的内核中，首先使用 setup_machine_fdt 来 setup machine 描述符，如果返回 NULL，才使用传统的方法 setup_machine_tags 来 setup machine 描述符。传统的方法需要给出__machine_arch_type（bootloader 通过 r1 寄存器传递给 kernel 的）和 tag list 的地址（用来进行 tag parse）。__machine_arch_type 用来寻找 machine 描述符；tag list 用于运行时参数的传递。随着内核的不断发展，相信有一天 linux kernel 会完全抛弃 tag list 的机制。

2.4.1.3 匹配 platform（machine 描述符）

setup_machine_fdt 函数的功能就是根据 Device Tree 的信息，找到最适合的 machine 描述符。具体代码如下：

//arch\arm\kernel\devtree.c

```
const struct machine_desc * __init setup_machine_fdt(unsigned int dt_phys)
{
    const struct machine_desc *mdesc, *mdesc_best = NULL;

    if (!dt_phys || !early_init_dt_scan(phys_to_virt(dt_phys)))
        return NULL;

    mdesc = of_flat_dt_match_machine(mdesc_best, arch_get_next_mach);

    if (!mdesc) {
        出错处理
    }

    /* Change machine number to match the mdesc we're using */
    __machine_arch_type = mdesc->nr;

    return mdesc;
}
```

early_init_dt_scan 函数有两个功能，一个是为后续的 DTB scan 进行准备工作，另外一个是运行时参数传递。具体请参考下面一个 section 的描述。

of_flat_dt_match_machine 是在 machine 描述符的列表中 scan，找到最合适的那个 machine 描述符。我们首先看如何组成 machine 描述符的列表。和传统的方法类似，也是静态定义的。DT_MACHINE_START 和 MACHINE_END 用来定义一个 machine 描述符。编译的时候，compiler 会把这些 machine descriptor 放到一个特殊的段中（.arch.info.init），形成 machine 描述符的列表。machine 描述符用下面的数据结构来标识（删除了不相关的 member）：

```
#define DT_MACHINE_START(_name, _namestr)                    \
static const struct machine_desc __mach_desc_##_name  \
 __used                                                 \
 __attribute__((__section__(".arch.info.init"))) = {   \
     .nr          = ~0,                                \
     .name        = _namestr,
#define MACHINE_END                                    \
};
```

```
struct machine_desc {
  unsigned int     nr;      /* architecture number    */
  const char *const     *dt_compat;   /* array of device tree 'compatible' strings     */
……
  };
```

nr 成员就是过去使用的 machine type ID。内核 machine 描述符的 table 有若干个 entry，每个都有自己的 ID。bootloader 传递了 machine type ID，指明使用哪一个 machine 描述符。目前匹配 machine 描述符使用 compatible strings，也就是 dt_compat 成员，这是一个 string list，定义了这个 machine 所支持的列表。在扫描 machine 描述符列表的时候需要不断的获取下一个 machine 描述符的 compatible 字符串的信息，具体的代码如下：

```
//arch\arm\kernel\devtree.c
static const void * __init arch_get_next_mach(const char *const **match)
{
  static const struct machine_desc *mdesc = __arch_info_begin;
  const struct machine_desc *m = mdesc;

  if (m >= __arch_info_end)
    return NULL;
```

```
    mdesc++;
    *match = m->dt_compat;
    return m;
}
```

__arch_info_begin 指向 machine 描述符列表第一个 entry。通过 mdesc++不断的移动 machine 描述符指针（Note：mdesc 是 static 的）。match 返回了该 machine 描述符的 compatible string list。具体匹配的算法倒是很简单，就是比较字符串而已，一个是 root node 的 compatible 字符串列表，一个是 machine 描述符的 compatible 字符串列表，得分最低的（最匹配的）就是我们最终选定的 machine type。

```
/**
 * of_flat_dt_match_machine - Iterate match tables to find matching machine.
 *
 * @default_match: A machine specific ptr to return in case of no match.
 * @get_next_compat: callback function to return next compatible match table.
 *
 * Iterate through machine match tables to find the best match for the machine
 * compatible string in the FDT.
 */
const void * __init of_flat_dt_match_machine(const void *default_match,
                const void * (*get_next_compat)(const char * const**))
{
        const void *data = NULL;
        const void *best_data = default_match;
        const char *const *compat;
        unsigned long dt_root;
        unsigned int best_score = ~1, score = 0;
        dt_root = of_get_flat_dt_root();
        while ((data = get_next_compat(&compat))) {
                score = of_flat_dt_match(dt_root, compat);
                if (score > 0 && score < best_score) {
                        best_data = data;
```

```
                        best_score = score;
                }
        }
        if (!best_data) {
                const char *prop;
                long size;
                pr_err("\n unrecognized device tree list:\n[ ");
                prop = of_get_flat_dt_prop(dt_root, "compatible", &size);
                if (prop) {
                        while (size > 0) {
                                printk("'%s' ", prop);
                                size -= strlen(prop) + 1;
                                prop += strlen(prop) + 1;
                        }
                }
                printk("]\n\n");
                return NULL;
        }
        pr_info("Machine model: %s\n", of_flat_dt_get_machine_name());
        return best_data;
}
```

2.4.1.4 运行时参数传递

运行时参数是在扫描 DTB 的 chosen node 时候完成的，具体的动作就是获取 chosen node 的 bootargs、initrd 等属性的 value，并将其保存在全局变量（boot_command_line，initrd_start、initrd_end）中。使用 tag list 方法是类似的，通过分析 tag list，获取相关信息，保存在同样的全局变量中。具体代码位于 early_init_dt_scan 函数中：

//drivers/of/fdt.c

```
bool __init early_init_dt_scan(void *params)
{
```

```
    if (!params)
        return false;

    /* 全局变量 initial_boot_params 指向了 DTB 的 header*/
    initial_boot_params = params;

    /* 检查 DTB 的 magic，确认是一个有效的 DTB */
    if (be32_to_cpu(initial_boot_params->magic) != OF_DT_HEADER) {
        initial_boot_params = NULL;
        return false;
    }

    /* 扫描 /chosen node，保存运行时参数（bootargs）到 boot_command_line，此外，还处理 initrd 相关的 property，并保存在 initrd_start 和 initrd_end 这两个全局变量中 */
    of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);

    /* 扫描根节点，获取 {size,address}-cells 信息，并保存在 dt_root_size_cells 和 dt_root_addr_cells 全局变量中 */
    of_scan_flat_dt(early_init_dt_scan_root, NULL);

    /* 扫描 DTB 中的 memory node，并把相关信息保存在 meminfo 中，全局变量 meminfo 保存了系统内存相关的信息。*/
    of_scan_flat_dt(early_init_dt_scan_memory, NULL);

    return true;
}
```

设定 meminfo（该全局变量确定了物理内存的布局）有若干种途径：

1、通过 tag list（tag 是 ATAG_MEM）传递 memory bank 的信息。

2、通过 command line（可以用 tag list，也可以通过 DTB）传递 memory bank 的信息。

3、通过 DTB 的 memory node 传递 memory bank 的信息。

目前当然是推荐使用 Device Tree 的方式来传递物理内存布局信息。

2.4.2 初始化流程

在系统初始化的过程中，我们需要将 DTB 转换成节点是 device_node 的树状结构，以便后续方便操作。具体的代码位于 setup_arch->unflatten_device_tree 中。

//drivers/of/fdt.c

```
void __init unflatten_device_tree(void)
{
    __unflatten_device_tree(initial_boot_params, &of_allnodes, // create tree of device_nodes from flat blob.  All nodes are saved in "of_allnodes".
                early_init_dt_alloc_memory_arch);
```

```
    /* Get pointer to "/chosen" and "/aliases" nodes for use everywhere */
    of_alias_scan(early_init_dt_alloc_memory_arch);
}
```

我们用 struct device_node 来抽象设备树中的一个节点，具体解释如下：

```
struct device_node {
    const char *name;—————————————————————————device node name
    const char *type;——————————————————————————对应 device_type 的属性
    phandle phandle;——————————————————————————对应该节点的 phandle 属性
    const char *full_name; ——————————————————从“/”开始的，表示该 node 的 full path
    struct    property *properties;——————————————该节点的属性列表
    struct    property *deadprops; ———————————如果需要删除某些属性，kernel 并非真的删除，而是挂入到
deadprops 的列表
    struct    device_node *parent;——————parent、child 以及 sibling 将所有的 device node 连接起来
    struct    device_node *child;
    struct    device_node *sibling;
    struct    device_node *next; —————————通过该指针可以获取相同类型的下一个 node
    struct    device_node *allnext;————————通过该指针可以获取 node global list 下一个 node
    struct    proc_dir_entry *pde;—————————开放到 userspace 的 proc 接口信息
    struct    kref kref;———————————————该 node 的 reference count
    unsigned long _flags;
    void    *data;
};
```

unflatten_device_tree 函数的主要功能就是扫描 DTB，将 device node 被组织成：

1、global list。全局变量 struct device_node *of_allnodes 就是指向设备树的 global list

2、tree。

这些功能主要是在__unflatten_device_tree 函数中实现，具体代码如下（去掉一些无关紧要的代码）：

```
static void __unflatten_device_tree(struct boot_param_header *blob,———需要扫描的 DTB
            struct device_node **mynodes,——————————global list 指针
            void * (*dt_alloc)(u64 size, u64 align))———————内存分配函数
{
    unsigned long size;
    void *start, *mem;
    struct device_node **allnextp = mynodes;

    此处删除了 health check 代码，例如检查 DTB header 的 magic，确认 blob 的确指向一个 DTB。

    /* scan 过程分成两轮，第一轮主要是确定 device-tree structure 的长度，保存在 size 变量中 */
    start = ((void *)blob) + be32_to_cpu(blob->off_dt_struct);
```

```
    size = (unsigned long)unflatten_dt_node(blob, 0, &start, NULL, NULL, 0);
    size = ALIGN(size, 4);

    /* 初始化的时候，并不是扫描到一个 node 或者 property 就分配相应的内存，实际上内核是一次性的分配了一大片内存，这些内存包括了所有的 struct device_node、node name、struct property 所需要的内存。*/
    mem = dt_alloc(size + 4, __alignof__(struct device_node));
    memset(mem, 0, size);

    *(__be32 *)(mem + size) = cpu_to_be32(0xdeadbeef);   //用来检验后面 unflattening 是否溢出

    /* 这是第二轮的 scan，第一次 scan 是为了得到保存所有 node 和 property 所需要的内存 size，第二次就是实打实的要构建 device node tree 了 */
    start = ((void *)blob) + be32_to_cpu(blob->off_dt_struct);
    unflatten_dt_node(blob, mem, &start, NULL, &allnextp, 0);


    此处略去校验溢出和校验 OF_DT_END。
}
```

具体的 scan 是在 unflatten_dt_node 函数中，如果已经清楚地了解 DTB 的结构，其实代码很简单，这里就不再细述了。

```
/**
 * unflatten_dt_node - Alloc and populate a device_node from the flat tree
 * @blob: The parent device tree blob
 * @mem: Memory chunk to use for allocating device nodes and properties
 * @p: pointer to node in flat tree
 * @dad: Parent struct device_node
 * @allnextpp: pointer to ->allnext from last allocated device_node
 * @fpsize: Size of the node path up at the current depth.
 */
static void * unflatten_dt_node(struct boot_param_header *blob,
                                void *mem,
                                void **p,
                                struct device_node *dad,
                                struct device_node ***allnextpp,
                                unsigned long fpsize)
{
        struct device_node *np;
```

```
	struct property *pp, **prev_pp = NULL;
	char *pathp;
	u32 tag;
	unsigned int l, allocl;
	int has_name = 0;
	int new_format = 0;

	tag = be32_to_cpup(*p);
	if (tag != OF_DT_BEGIN_NODE) {
		pr_err("Weird tag at start of node: %x\n", tag);
		return mem;
	}
	*p += 4;
	pathp = *p;
	l = allocl = strlen(pathp) + 1;
	*p = PTR_ALIGN(*p + l, 4);

	/* version 0x10 has a more compact unit name here instead of the full
	 * path. we accumulate the full path size using "fpsize", we'll rebuild
	 * it later. We detect this because the first character of the name is
	 * not '/'.
	 */
	if ((*pathp) != '/') {
		new_format = 1;
		if (fpsize == 0) {
			/* root node: special case. fpsize accounts for path
			 * plus terminating zero. root node only has '/', so
			 * fpsize should be 2, but we want to avoid the first
			 * level nodes to have two '/' so we use fpsize 1 here
			 */
```

```
                                        fpsize = 1;
                                        allocl = 2;
                                        l = 1;
                                        *pathp = '\0';
                        } else {
                                        /* account for '/' and path size minus terminal 0
                                         * already in 'l'
                                         */
                                        fpsize += l;
                                        allocl = fpsize;
                        }
                }

        np = unflatten_dt_alloc(&mem, sizeof(struct device_node) + allocl,
                                        __alignof__(struct device_node));
        if (allnextpp) {
                        char *fn;
                        np->full_name = fn = ((char *)np) + sizeof(*np);
                        if (new_format) {
                                        /* rebuild full path for new format */
                                        if (dad && dad->parent) {
                                                        strcpy(fn, dad->full_name);
#ifdef DEBUG
                                                        if ((strlen(fn) + l + 1) != allocl) {
                                                                        pr_debug("%s: p: %d, l: %d, a: %d\n",
                                                                                        pathp, (int)strlen(fn),
                                                                                        l, allocl);
                                                        }
#endif
                                                        fn += strlen(fn);
```

```
                }
                *(fn++) = '/';
            }
            memcpy(fn, pathp, l);

            prev_pp = &np->properties;
            **allnextpp = np;
            *allnextpp = &np->allnext;
            if (dad != NULL) {
                np->parent = dad;
                /* we temporarily use the next field as `last_child'*/
                if (dad->next == NULL)
                    dad->child = np;
                else
                    dad->next->sibling = np;
                dad->next = np;
            }
            kref_init(&np->kref);
        }
        /* process properties */
        while (1) {
            u32 sz, noff;
            char *pname;

            tag = be32_to_cpup(*p);
            if (tag == OF_DT_NOP) {
                *p += 4;
                continue;
            }
            if (tag != OF_DT_PROP)
```

```
				break;
			*p += 4;
			sz = be32_to_cpup(*p);
			noff = be32_to_cpup(*p + 4);
			*p += 8;
			if (be32_to_cpu(blob->version) < 0x10)
				*p = PTR_ALIGN(*p, sz >= 8 ? 8 : 4);

			pname = of_fdt_get_string(blob, noff);
			if (pname == NULL) {
				pr_info("Can't find property name in list !\n");
				break;
			}
			if (strcmp(pname, "name") == 0)
				has_name = 1;
			l = strlen(pname) + 1;
			pp = unflatten_dt_alloc(&mem, sizeof(struct property),
						__alignof__(struct property));
			if (allnextpp) {
				/* We accept flattened tree phandles either in
				 * ePAPR-style "phandle" properties, or the
				 * legacy "linux,phandle" properties.  If both
				 * appear and have different values, things
				 * will get weird.  Don't do that. */
				if ((strcmp(pname, "phandle") == 0) ||
				    (strcmp(pname, "linux,phandle") == 0)) {
					if (np->phandle == 0)
						np->phandle = be32_to_cpup((__be32*)*p);
				}
				/* And we process the "ibm,phandle" property
```

```
                                * used in pSeries dynamic device tree
                                * stuff */
                               if (strcmp(pname, "ibm,phandle") == 0)
                                        np->phandle = be32_to_cpup((__be32 *)*p);
                               pp->name = pname;
                               pp->length = sz;
                               pp->value = *p;
                               *prev_pp = pp;
                               prev_pp = &pp->next;
                      }
                      *p = PTR_ALIGN((*p) + sz, 4);
             }
             /* with version 0x10 we may not have the name property, recreate
              * it here from the unit name if absent
              */
             if (!has_name) {
                      char *p1 = pathp, *ps = pathp, *pa = NULL;
                      int sz;

                      while (*p1) {
                               if ((*p1) == '@')
                                        pa = p1;
                               if ((*p1) == '/')
                                        ps = p1 + 1;
                               p1++;
                      }
                      if (pa < ps)
                               pa = p1;
                      sz = (pa - ps) + 1;
                      pp = unflatten_dt_alloc(&mem, sizeof(struct property) + sz,
```

```
                        __alignof__(struct property));
        if (allnextpp) {
            pp->name = "name";
            pp->length = sz;
            pp->value = pp + 1;
            *prev_pp = pp;
            prev_pp = &pp->next;
            memcpy(pp->value, ps, sz - 1);
            ((char *)pp->value)[sz - 1] = 0;
            pr_debug("fixed up name for %s -> %s\n", pathp,
                (char *)pp->value);
        }
    }
    if (allnextpp) {
        *prev_pp = NULL;
        np->name = of_get_property(np, "name", NULL);
        np->type = of_get_property(np, "device_type", NULL);

        if (!np->name)
            np->name = "<NULL>";
        if (!np->type)
            np->type = "<NULL>";
    }
    while (tag == OF_DT_BEGIN_NODE || tag == OF_DT_NOP) {
        if (tag == OF_DT_NOP)
            *p += 4;
        else
            mem = unflatten_dt_node(blob, mem, p, np, allnextpp,
                            fpsize);
        tag = be32_to_cpup(*p);
```

```
        }
        if (tag != OF_DT_END_NODE) {
                pr_err("Weird tag at end of node: %x\n", tag);
                return mem;
        }
        *p += 4;
        Returnmem;
}
```

2.4.3 如何并入 linux kernel 的设备驱动模型

在 linux kernel 引入统一设备模型之后，bus、driver 和 device 形成了设备模型中的铁三角。在驱动初始化的时候会将代表该 driver 的一个数据结构（一般是 xxx_driver）挂入 bus 上的 driver 链表。device 挂入链表分成两种情况，一种是即插即用类型的 bus，在插入一个设备后，总线可以检测到这个行为并动态分配一个 device 数据结构（一般是 xxx_device，例如 usb_device），之后，将该数据结构挂入 bus 上的 device 链表。bus 上挂满了 driver 和 device，那么如何让 device 遇到"对"的那个 driver 呢？那么就要靠缘分了，也就是 bus 的 match 函数。

上面是一段导论，我们还是回到 Device Tree。导致 Device Tree 的引入 ARM 体系结构的代码其中一个最重要的原因的太多的静态定义的表格。例如：一般代码中会定义一个 static struct platform_device *xxx_devices 的静态数组，在初始化的时候调用 platform_add_devices。这些静态定义的 platform_device 往往又需要静态定义各种 resource，这导致静态表格进一步增大。如果 ARM linux 中不再定义这些表格，那么一定需要一个转换的过程，也就是说，系统应该会根据 Device tree 来动态的增加系统中的 platform_device。当然，这个过程并非只是发生在 platform bus 上（具体可以参考"Platform Device"的设备），也可能发生在其他的非即插即用的 bus 上，例如 AMBA 总线、PCI 总线。一言以蔽之，如果要并入 linux kernel 的设备驱动模型，那么就需要根据 device_node 的树状结构（root 是 of_allnodes）将一个个的 device node 挂入到相应的总线 device 链表中。只要做到这一点，总线机制就会安排 device 和 driver 的约会。

当然，也不是所有的 device node 都会挂入 bus 上的设备链表，比如 cpus node，memory node，choose node 等。

2.4.3.1 cpus node 的处理

这部分的处理可以参考 setup_arch->arm_dt_init_cpu_maps 中的代码，具体的代码如下：

```
//arch\arm\kernel\devtree.c
void __init arm_dt_init_cpu_maps(void)
{
```

```
  scan device node global list，寻找 full path 是“/cpus”的那个 device node。cpus 这个 device node 只是一个容器，其中包括了各个 cpu node 的定义以及所有 cpu node 共享的 property。
  cpus = of_find_node_by_path("/cpus");
  for_each_child_of_node(cpus, cpu) {        遍历 cpus 的所有的 child node
    u32 hwid;
    if (of_node_cmp(cpu->type, "cpu"))       我们只关心那些 device_type 是 cpu 的 node
      continue;
    if (of_property_read_u32(cpu, "reg", &hwid)) {    读取 reg 属性的值并赋值给 hwid
      return;
    }
    reg 的属性值的 8 MSBs 必须设置为 0，这是 ARM CPU binding 定义的。
    if (hwid & ~MPIDR_HWID_BITMASK)
      return;
    不允许重复的 CPU id，那是一个灾难性的设定
    for (j = 0; j < cpuidx; j++)
      if (WARN(tmp_map[j] == hwid, "Duplicate /cpu reg "
               "properties in the DT\n"))
        return;
数组 tmp_map 保存了系统中所有 CPU 的 MPIDR 值（CPU ID 值），具体的 index 的编码规则是： tmp_map[0]保存了booting CPU 的 id 值，其余的 CPU 的 ID 值保存在 1～NR_CPUS 的位置。
    if (hwid == mpidr) {
      i = 0;
      bootcpu_valid = true;
    } else {
      i = cpuidx++;
    }
    tmp_map[i] = hwid;
  }
根据 DTB 中的信息设定 cpu logical map 数组。
  for (i = 0; i < cpuidx; i++) {
    set_cpu_possible(i, true);
    cpu_logical_map(i) = tmp_map[i];
  }
}
```

要理解这部分的内容，需要理解 ARM CPUs binding 的概念，可以参考 linux/Documentation/devicetree/bindings/arm 目录下的 CPU.txt 文件的描述。

2.4.3.2 memory 的处理

这部分的处理可以参考 setup_arch->setup_machine_fdt->early_init_dt_scan->early_init_dt_scan_memory 中的代码。具体如下：

//drivers/of/fdt.c

```
int __init early_init_dt_scan_memory(unsigned long node, const char *uname,
                int depth, void *data)
{
  char *type = of_get_flat_dt_prop(node, "device_type", NULL); 获取 device_type 属性值
  __be32 *reg, *endp;
  unsigned long l;
  在初始化的时候，我们会对每一个 device node 都要调用该 call back 函数，因此，我们要过滤掉那些和 memory block 定义无关的 node。和 memory block 定义有的节点有两种，一种是 node name 是 memory@形态的，另外一种是 node 中定义了 device_type 属性并且其值是 memory。
  if (type == NULL) {
    if (depth != 1 || strcmp(uname, "memory@0") != 0)
      return 0;
  } else if (strcmp(type, "memory") != 0)
    return 0;
  获取 memory 的起始地址和 length 的信息。有两种属性和该信息有关，一个是 linux,usable-memory，不过最新的方式还是使用 reg 属性。
reg = of_get_flat_dt_prop(node, "linux,usable-memory", &l);
  if (reg == NULL)
    reg = of_get_flat_dt_prop(node, "reg", &l);
  if (reg == NULL)
    return 0;
  endp = reg + (l / sizeof(__be32));
reg 属性的值是 address，size 数组，那么如何来取出一个个的 address/size 呢？由于 memory node 一定是 root node 的 child，因此 dt_root_addr_cells（root node 的#address-cells 属性值）和 dt_root_size_cells（root node 的#size-cells 属性值）之和就是 address，size 数组的 entry size。
  while ((endp - reg) >= (dt_root_addr_cells + dt_root_size_cells)) {
    u64 base, size;
    base = dt_mem_next_cell(dt_root_addr_cells, ®);
    size = dt_mem_next_cell(dt_root_size_cells, ®);
    early_init_dt_add_memory_arch(base, size); 将具体的 memory block 信息加入到内核中。
  }
```

```
    return 0;
}
```

2.4.3.3 interrupt controller 的处理

初始化是通过 start_kernel->init_IRQ->machine_desc->init_irq()实现的。我们用 imx6q-sabresd 为例来描述 interrupt controller 的处理过程。下面是 machine 描述符的定义。

//arch/arm/mach-imx/mach-imx6q.c

```
DT_MACHINE_START(IMX6Q, "NXP i.MX6 Quad/DualLite (Device Tree)")
	/*
	 * i.MX6Q/DL maps system memory at 0x10000000 (offset 256MiB), and
	 * GPU has a limit on physical address that it accesses, which must
	 * be below 2GiB.
	 */
	.dma_zone_size	= (SZ_2G - SZ_256M),
	.smp		= smp_ops(imx_smp_ops),
	.map_io		= imx6q_map_io,
	.init_irq	= imx6q_init_irq,
	.init_machine	= imx6q_init_machine,
	.init_late	= imx6q_init_late,
	.dt_compat	= imx6q_dt_compat,
	.restart	= mxc_restart,
MACHINE_END

static void __init imx6q_init_irq(void)
{
	imx_init_revision_from_anatop();//from anatop register, read the chipset TO version and init the revision
	imx_init_l2cache();//initical pl310 L2 cache
	imx_src_init();//initial system reset controller
	imx_gpc_init();//initial gpc
	irqchip_init();
```

```
}
```

在 driver/irqchip/irq-gic.c 文件中定义了 interrupt controller，如下：

```
IRQCHIP_DECLARE(cortex_a9_gic, "arm,cortex-a9-gic", gic_of_init);
```

当然，系统中可以定义更多的irqchip，不过具体用哪一个是根据DTB中的interrupt controller node 中的 compatible 属性确定的。在 driver/irqchip/irqchip.c 文件中定义了 irqchip_init 函数，如下：

```
void __init irqchip_init(void)
{
    of_irq_init(__irqchip_begin);
}
```

__irqchip_begin 就是所有的 irqchip 的一个列表，of_irq_init 函数是遍历 Device Tree，找到匹配的 irqchip。具体的代码如下：

```
void __init of_irq_init(const struct of_device_id *matches)
{
    struct device_node *np, *parent = NULL;
    struct intc_desc *desc, *temp_desc;
    struct list_head intc_desc_list, intc_parent_list;

    INIT_LIST_HEAD(&intc_desc_list);
    INIT_LIST_HEAD(&intc_parent_list);

    遍历所有的 node，寻找定义了 interrupt-controller 属性的 node，如果定义了 interrupt-controller 属性则说明该 node
就是一个中断控制器。

    for_each_matching_node(np, matches) {
        if (!of_find_property(np, "interrupt-controller", NULL) ||
                !of_device_is_available(np))
            continue;

分配内存并挂入链表，当然还有根据 interrupt-parent 建立 controller 之间的父子关系。对于 interrupt controller，它也可
能是一个树状的结构。
        desc = kzalloc(sizeof(*desc), GFP_KERNEL);
        if (WARN_ON(!desc))
            goto err;

        desc->dev = np;
        desc->interrupt_parent = of_irq_find_parent(np);
        if (desc->interrupt_parent == np)
            desc->interrupt_parent = NULL;
```

```
    list_add_tail(&desc->list, &intc_desc_list);
}
```

正因为 interrupt controller 被组织成树状的结构，因此初始化的顺序就需要控制，应该从根节点开始，依次递进到下一个 level 的 interrupt controller。

while (!list_empty(&intc_desc_list)) { intc_desc_list 链表中的节点会被一个个的处理，每处理完一个节点就会将该节点删除，当所有的节点被删除，整个处理过程也就是结束了。

```
    list_for_each_entry_safe(desc, temp_desc, &intc_desc_list, list) {
        const struct of_device_id *match;
        int ret;
        of_irq_init_cb_t irq_init_cb;
```

最开始的时候 parent 变量是 NULL，确保第一个被处理的是 root interrupt controller。在处理完 root node 之后，parent 变量被设定为 root interrupt controller，因此，第二个循环中处理的是所有 parent 是 root interrupt controller 的 child interrupt controller。也就是 level 1（如果 root 是 level 0 的话）的节点。

```
        if (desc->interrupt_parent != parent)
            continue;

        list_del(&desc->list);      －－－－－从链表中删除
        match = of_match_node(matches, desc->dev);－－－－－匹配并初始化
        if (WARN(!match->data,－－－－－－－－－－－match->data 是初始化函数
            "of_irq_init: no init function for %s\n",
            match->compatible)) {
            kfree(desc);
            continue;
        }

        irq_init_cb = (of_irq_init_cb_t)match->data;
        ret = irq_init_cb(desc->dev, desc->interrupt_parent);－－－－－执行初始化函数
        if (ret) {
            kfree(desc);
            continue;
        }
```

处理完的节点放入 intc_parent_list 链表，后面会用到

```
        list_add_tail(&desc->list, &intc_parent_list);
    }
```

对于 level 0，只有一个 root interrupt controller，对于 level 1，可能有若干个 interrupt controller，因此要遍历这些 parent interrupt controller，以便处理下一个 level 的 child node。

```
    desc = list_first_entry_or_null(&intc_parent_list,
                    typeof(*desc), list);
    if (!desc) {
        pr_err("of_irq_init: children remain, but no parents\n");
```

```
            break;
        }
        list_del(&desc->list);
        parent = desc->dev;
        kfree(desc);
    }

    list_for_each_entry_safe(desc, temp_desc, &intc_parent_list, list) {
        list_del(&desc->list);
        kfree(desc);
    }
err:
    list_for_each_entry_safe(desc, temp_desc, &intc_desc_list, list) {
        list_del(&desc->list);
        kfree(desc);
    }
}
```

只有该 node 中有 interrupt-controller 这个属性定义，那么 linux kernel 就会分配一个 interrupt controller 的描述符（struct intc_desc）并挂入队列。通过 interrupt-parent 属性，可以确定各个 interrupt controller 的层次关系。在 scan 了所有的 Device Tree 中的 interrupt controller 的定义之后，系统开始匹配过程。一旦匹配到了 interrupt chip 列表中的项次后，就会调用相应的初始化函数。如果 CPU 是 imx6q 的话，匹配到的是 irqchip 的初始化函数是 gic_of_init。

OK，我们已经通过 compatible 属性找到了适合的 interrupt controller，那么如何解析 reg 属性呢?我们知道，对于 imx6q 的 interrupt controller 而言，其#interrupt-cells 的属性值是 3，定义可以参考 binding 文件/documentation/devicetree/binding/arm/gic.txt。每个域的解释如下：

```
* ARM Generic Interrupt Controller

ARM SMP cores are often associated with a GIC, providing per processor interrupts (PPI), shared processor interrupts (SPI)
and software generated interrupts (SGI).

Primary GIC is attached directly to the CPU and typically has PPIs and SGIs. Secondary GICs are cascaded into the upward
interrupt controller and do not have PPIs or SGIs.

Main node required properties:

- compatible : should be one of: "arm,gic-400"

        "arm,cortex-a15-gic"

        "arm,cortex-a9-gic"

        "arm,cortex-a7-gic"

        "arm,arm11mp-gic"

- interrupt-controller : Identifies the node as an interrupt controller
```

- #interrupt-cells : Specifies the number of cells needed to encode an interrupt source. The type shall be a <u32> and the value shall be 3.

The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI interrupts.

The 2nd cell contains the interrupt number for the interrupt type.

SPI interrupts are in the range [0-987]. PPI interrupts are in the range [0-15].

The 3rd cell is the flags, encoded as follows: bits[3:0] trigger type and level flags. 1 = low-to-high edge triggered 2 = high-to-low edge triggered 4 = active high level-sensitive 8 = active low level-sensitive bits[15:8] PPI interrupt cpu mask. Each bit corresponds to each of

the 8 possible cpus attached to the GIC. A bit set to '1' indicated the interrupt is wired to that CPU. Only valid for PPI interrupts.

- reg : Specifies base physical address(s) and size of the GIC registers. The first region is the GIC distributor register base and size. The 2nd region is the GIC cpu interface register base and size.

Optional

- interrupts : Interrupt source of the parent interrupt controller on secondary GICs, or VGIC maintenance interrupt on primary GIC (see

below).

- cpu-offset : per-cpu offset within the distributor and cpu interface regions, used when the GIC doesn't have banked registers. The offset is cpu-offset * cpu-nr.

Example:

```
	intc: interrupt-controller@fff11000 { compatible = "arm,cortex-a9-gic";
		#interrupt-cells = <3>;
		#address-cells = <1>;
		interrupt-controller;
		reg = <0xfff11000 0x1000>, <0xfff10100 0x100>; };
```

* GIC virtualization extensions (VGIC)

For ARM cores that support the virtualization extensions, additional properties must be described (they only exist if the GIC is the primary interrupt controller).

Required properties:

- reg : Additional regions specifying the base physical address and size of the VGIC registers. The first additional region is the GIC virtual interface control register base and size. The 2nd additional region is the GIC virtual cpu interface register base and size.

- interrupts : VGIC maintenance interrupt.

```
Example:
        interrupt-controller@2c001000 { compatible = "arm,cortex-a15-gic";
                #interrupt-cells = <3>;
                interrupt-controller;
                reg = <0x2c001000 0x1000>, <0x2c002000 0x1000>,
                        <0x2c004000 0x2000>,
                        <0x2c006000 0x2000>; interrupts = <1 9 0xf04>; };
```

2.4.3.4 machine 初始化

machine 初始化的代码可以沿着 start_kernel->rest_init->kernel_init->kernel_init_freeable->do_basic_setup->do_initcalls 路径寻找。在 do_initcalls 函数中，kernel 会依次执行各个 initcall 函数，在这个过程中，会调用 customize_machine，具体如下：

```
static int __init customize_machine(void)
{

    if (machine_desc->init_machine)
        machine_desc->init_machine();
    else
        of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);

    return 0;
}
arch_initcall(customize_machine);
```

在这个函数中，一般会调用 machine 描述符中的 init_machine callback 函数来把各种 Device Tree 中定义各个设备节点加入到系统。如果 machine 描述符中没有定义 init_machine 函数，那么直接调用 of_platform_populate 把所有的 platform device 加入到 kernel 中。对于 imx6q-sabresd，其 machine 描述符中的 init_machine callback 函数就是 imx6q_init_machine，代码如下：

```
static void __init imx6q_init_machine(void)
{
    struct device *parent;
    imx_print_silicon_rev(cpu_is_imx6dl() ? "i.MX6DL" : "i.MX6Q",
                    imx_get_soc_revision());
```

```
    mxc_arch_reset_init_dt();


    parent = imx_soc_device_init();
    if (parent == NULL)
            pr_warn("failed to initialize soc device\n");
    of_platform_populate(NULL, of_default_bus_match_table, , --------传入 NULL 参数表示从 root node 开始 scan
                            imx6q_auxdata_lookup, parent);
    imx6q_enet_init();
    imx_anatop_init();
    imx6q_csi_mux_init();
    cpu_is_imx6q() ?  imx6q_pm_init() : imx6dl_pm_init();
}
```

由此可见，最终生成platform device的代码来自of_platform_populate函数。该函数的逻辑比较简单，遍历 device node global list 中所有的 node，并调用 of_platform_bus_create 处理，of_platform_bus_create 函数代码如下：

```
//drivers/of/platform.c
static int of_platform_bus_create(struct device_node *bus,-------------要创建的那个 device node
          const struct of_device_id *matches,-------要匹配的 list
          const struct of_dev_auxdata *lookup,------附属数据
          struct device *parent, bool strict)---------------parent 指向父节点。strict 是否要求完全匹配
{
  const struct of_dev_auxdata *auxdata;
  struct device_node *child;
  struct platform_device *dev;
  const char *bus_id = NULL;
  void *platform_data = NULL;
  int rc = 0;
删除确保 device node 有 compatible 属性的代码。
  auxdata = of_dev_lookup(lookup, bus);  在传入的 lookup table 寻找和该 device node 匹配的附加数据
  if (auxdata) {
    bus_id = auxdata->name;-----------------如果找到，那么就用附加数据中的静态定义的内容
    platform_data = auxdata->platform_data;
  }
```

ARM 公司提供了 CPU core，除此之外，它设计了 AMBA 的总线来连接 SOC 内的各个 block。符合这个总线标准的 SOC 上的外设叫做 ARM Primecell Peripherals。如果一个 device node 的 compatible 属性值是 arm,primecell 的话，可以调用 of_amba_device_create 来向 amba 总线上增加一个 amba device。

```
    if (of_device_is_compatible(bus, "arm,primecell")) {
        of_amba_device_create(bus, bus_id, platform_data, parent);
        return 0;
    }
```

如果不是 ARM Primecell Peripherals，那么我们就需要向 platform bus 上增加一个 platform device 了

```
    dev = of_platform_device_create_pdata(bus, bus_id, platform_data, parent);
    if (!dev || !of_match_node(matches, bus))
        return 0;
```

一个 device node 可能是一个桥设备，因此要重复调用 of_platform_bus_create 来把所有的 device node 处理掉。

```
    for_each_child_of_node(bus, child) {
        pr_debug("    create child: %s\n", child->full_name);
        rc = of_platform_bus_create(child, matches, lookup, &dev->dev, strict);
        if (rc) {
            of_node_put(child);
            break;
        }
    }
    return rc;
}
```

具体增加 platform device 的代码在 of_platform_device_create_pdata 中，代码如下：

```
static struct platform_device *of_platform_device_create_pdata(
                struct device_node *np,
                const char *bus_id,
                void *platform_data,
                struct device *parent)
{
    struct platform_device *dev;

    if (!of_device_is_available(np))---------check status 属性，确保是 enable 或者 OK 的。
        return NULL;
```

of_device_alloc 除了分配 struct platform_device 的内存，还分配了该 platform device 需要的 resource 的内存（参考 struct platform_device 中的 resource 成员）。当然，这就需要解析该 device node 的 interrupt 资源以及 memory address 资源。

```
dev = of_device_alloc(np, bus_id, parent);
if (!dev)
    return NULL;
```

设定 platform_device 中的其他成员

```
dev->dev.coherent_dma_mask = DMA_BIT_MASK(32);
if (!dev->dev.dma_mask)
    dev->dev.dma_mask = &dev->dev.coherent_dma_mask;
dev->dev.bus = &platform_bus_type;
dev->dev.platform_data = platform_data;

if (of_device_add(dev) != 0) {------------------把这个 platform device 加入统一设备模型系统中
    platform_device_put(dev);
    return NULL;
}

return dev;
}
```

2.4.4 如何基于 device tree 来开发驱动

2.4.4.1 of_match_table

设备驱动需要与.dts 中下定义设备一致，从而使用相应的 probe 函数被调用，所以平台驱动中需要定义一下.of_match_table,如 drivers\pinctrol\pinctrol-imx6q.c

```
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,
        },
        .probe = imx6q_pinctrl_probe,
        .remove = imx_pinctrl_remove,
};
```

相应的.dts 定义为:

```
                iomuxc: iomuxc@020e0000 {
                        compatible = "fsl,imx6dl-iomuxc", "fsl,imx6q-iomuxc";
                        reg = <0x020e0000 0x4000>;
                };
```

2.4.4.2 memory and IRQ resources

Memory and IRQ resources work in the same way with non-DT probe. ,IORESOURCE_DMA does not work with DT.

see sound/soc/fsl/imx-ssi.c   vs. sound/soc/fsl/fsl_ssi.c

```
ssi->irq = platform_get_irq(pdev, 0);

ssi->clk = devm_clk_get(&pdev->dev, NULL);
if (IS_ERR(ssi->clk)) {
    ret = PTR_ERR(ssi->clk);
    dev_err(&pdev->dev, "Cannot get the clock: %d\n",
        ret);
    goto failed_clk;
}
clk_prepare_enable(ssi->clk);

res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
ssi->base = devm_ioremap_resource(&pdev->dev, res);
```

```
ssi->irq = platform_get_irq(pdev, 0);

ssi->clk = devm_clk_get(&pdev->dev, NULL);
if (IS_ERR(ssi->clk)) {
    ret = PTR_ERR(ssi->clk);
    dev_err(&pdev->dev, "Cannot get the clock: %d\n",
        ret);
    goto failed_clk;
}
clk_prepare_enable(ssi->clk);

res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
ssi->base = devm_ioremap_resource(&pdev->dev, res);
```

2.4.4.3 platform_data is retrieved from DT

- For non-DT, platform_data is used pass hardware configuration between board file and device driver.

```
struct esdhc_platform_data {
    unsigned int wp_gpio;
    unsigned int cd_gpio;
    enum wp_types wp_type;
    enum cd_types cd_type;
};
```

- For DT probe, all these data should be retrieved from DT
    - Bindings
        - Documentation/devicetree/bindings/mmc/fsl-imx-esdhc.txt

- Documentation/devicetree/bindings/gpio/gpio.txt

- Helper functions
  - of_get_property(), of_get_named_gpio(), of_property_read_u32() etc.
  - include/linux/of.h

```
function
  of_node_get
  of_node_put
  of_have_populated_dt
  of_node_is_root
  of_node_check_flag
  of_node_set_flag
  of_read_number
  of_read_ulong
  of_node_full_name
  of_find_matching_node
  of_get_child_count
  of_node_full_name
  of_find_node_by_name
  of_get_parent
  of_have_populated_dt
  of_get_child_by_name
  of_get_child_count
  of_device_is_compatible
  of_device_is_available
  of_find_property
  of_find_compatible_node
  of_property_read_u32_index
  of_property_read_u8_array
  of_property_read_u16_array
  of_property_read_u32_array
  of_property_read_string
  of_property_read_string_inde
  of_property_count_strings
  of_get_property
  of_property_read_u64
  of_property_match_string
  of_parse_phandle
  of_parse_phandle_with_args
  of_count_phandle_with_args
  of_alias_get_id
  of_machine_is_compatible
```

2.4.4.4 platform_data is retrieved from DT 示例 i2c

- Imx6qdl.dtsi

```
i2c1: i2c@021a0000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
        reg = <0x021a0000 0x4000>;
        interrupts = <0 36 0x04>;
        clocks = <&clks 125>;
        status = "disabled";
};

i2c2: i2c@021a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <0 37 0x04>;
        clocks = <&clks 126>;
        status = "disabled";
};

i2c3: i2c@021a8000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
        reg = <0x021a8000 0x4000>;
        interrupts = <0 38 0x04>;
        clocks = <&clks 127>;
        status = "disabled";
};
```

- Run "cat cat /proc/iomem | grep i2c"

```
root@imx6qsabresd:~# cat /proc/iomem | grep i2c
021a0000-021a3fff : /soc/aips-bus@02100000/i2c@021a0000
021a4000-021a7fff : /soc/aips-bus@02100000/i2c@021a4000
021a8000-021abfff : /soc/aips-bus@02100000/i2c@021a8000
```

- of_match_table defined for i2c_imx_driver

```
static const struct of_device_id i2c_imx_dt_ids[] = {
    { .compatible = "fsl,imx1-i2c", .data = &imx_i2c_c
    { .compatible = "fsl,imx21-i2c", .data = &imx_i2c_
    { /* sentinel */ }
};
```

- platform_driver_probe

```
static struct platform_driver i2c_imx_driver = {
    .remove     = __exit_p(i2c_imx_remove),
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = i2c_imx_dt_ids,
    },
    .id_table   = imx_i2c_devtype,
};

static int __init i2c_adap_imx_init(void)
{
    return platform_driver_probe(&i2c_imx_driver, i2c_imx_probe);
}
subsys_initcall(i2c_adap_imx_init);
```

- i2c_imx_probe: apply resources, irq, add i2c adapt

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if (!res) {
    dev_err(&pdev->dev, "can't get device resources\n");
    return -ENOENT;
}
irq = platform_get_irq(pdev, 0);
if (irq < 0) {
    dev_err(&pdev->dev, "can't get irq number\n");
    return -ENOENT;
}

base = devm_ioremap_resource(&pdev->dev, res);
if (IS_ERR(base))
    return PTR_ERR(base);
```

```
/* Request IRQ */
ret = devm_request_irq(&pdev->dev, irq, i2c_imx_isr, 0,
            pdev->name, i2c_imx);
```

```
/* Set up clock divider */
bitrate = IMX_I2C_BIT_RATE;
ret = of_property_read_u32(pdev->dev.of_node,
                "clock-frequency", &bitrate);
if (ret < 0 && pdata && pdata->bitrate)
    bitrate = pdata->bitrate;
i2c_imx_set_clk(i2c_imx, bitrate);
```

imx6qdl-sabresd.dtsi

```
&i2c1 {
    clock-frequency = <100000>;
```

- of_i2c_register_devices() parse i2c child nodes, get addr, irq etc.

for_each_available_child_of_node{} loop

-> request_module: load device driver

-> i2c_new_device: generate device.

-> driver probe

From now on, it’s same as non-DT

- Child node wm8962 of i2c1

imx6qdl-sabresd.dtsi

```
&i2c1 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1_2>;
    status = "okay";

    codec: wm8962@1a {
        compatible = "wlf,wm8962";
        reg = <0x1a>;
        clocks = <&clks 201>;
```

```
static const struct of_device_id wm8962_of_match[] = {
	{ .compatible = "wlf,wm8962", },
	{ }
};
MODULE_DEVICE_TABLE(of, wm8962_of_match);

static struct i2c_driver wm8962_i2c_driver = {
	.driver = {
		.name = "wm8962",
		.owner = THIS_MODULE,
		.of_match_table = wm8962_of_match,
		.pm = &wm8962_pm,
	},
	.probe =	wm8962_i2c_probe,
	.remove =	wm8962_i2c_remove,
	.id_table = wm8962_i2c_id,
};
```

wm8962_set_pdata_from_of parse and get conf.

```
/* If platform data was supplied, update the default data in pr
if (pdata) {
	memcpy(&wm8962->pdata, pdata, sizeof(struct wm8962_pdata));
} else if (i2c->dev.of_node) {
	ret = wm8962_set_pdata_from_of(i2c, &wm8962->pdata);
	if (ret != 0)
		return ret;
}
```

- i2c adapt log

```
root@imx6qsabresd:~# dmesg | grep i2c
i2c-core: driver [max17135] using legacy suspend method
i2c-core: driver [max17135] using legacy resume method
i2c i2c-0: IMX I2C adapter registered
i2c i2c-1: IMX I2C adapter registered
i2c i2c-2: IMX I2C adapter registered
imx6q-pinctrl 20e0000.iomuxc: pin MX6Q_PAD_KEY_COL3 already requested by 21a4000.i2c; cannot claim for 20e0000.hdmi_video
input: eGalax Touch Screen as /devices/soc0/soc.1/2100000.aips-bus/21a8000.i2c/i2c-2/2-0004/input/input0
input: max11801_ts as /devices/soc0/soc.1/2100000.aips-bus/21a4000.i2c/i2c-1/1-0048/input/input1
i2c-core: driver [isl29023] using legacy suspend method
i2c-core: driver [isl29023] using legacy resume method
i2c /dev entries driver
i2c-core: driver [mag3110] using legacy suspend method
i2c-core: driver [mag3110] using legacy resume method
i2c-core: driver [cs42888] using legacy suspend method
i2c-core: driver [cs42888] using legacy resume method
input: WM8962 Beep Generator as /devices/soc0/soc.1/2100000.aips-bus/21a0000.i2c/i2c-0/0-001a/input/input5
```

- i2c device - wm8962 log

```
root@imx6qsabresd:~# dmesg | grep 8962
wm8962-supply: no parameters
wm8962 0-001a: customer id 0 revision D
input: WM8962 Beep Generator as /devices/soc0/soc.1/2100000.aips-bus/21a0000.i2c/i2c-0/0-001a/input/input5
imx-wm8962 sound.23:  wm8962 <-> 202c000.ssi mapping ok
input: wm8962-audio AMIC as /devices/soc0/sound.23/sound/card0/input6
input: wm8962-audio Headphone Jack as /devices/soc0/sound.23/sound/card0/input7
  #0: wm8962-audio
evbug: Connected device: input5 (WM8962 Beep Generator at 0-001a)
evbug: Connected device: input6 (wm8962-audio AMIC at ALSA)
evbug: Connected device: input7 (wm8962-audio Headphone Jack at ALSA)
```

2.4.4.5 DTS 示例 gpio

- gpio-cells: how many cells to specify a gpio
- gpio-controller: specify its identity.
- gpio-phandle : phandle to gpio controller node
- single-gpio ::= <gpio-phandle> <gpio-specifier>
- gpio-list ::= <single-gpio> [gpio-list]
- gpio-specifier : Array of #gpio-cells specifying specific gpio, it’s “25 0” in this example.

```
gpio1: gpio@0209c000 {
    compatible = "fsl,imx6q-gpio", "fsl
    reg = <0x0209c000 0x4000>;
    interrupts = <0 66 0x04 0 67 0x04>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};
```

```
leds {
    compatible = "gpio-leds";

    debug-led {
        label = "Heartbeat";
        gpios = <&gpio3 25 0>;
        linux,default-trigger = "heartbeat";
    };
};
```

- Node using GPIOs: gpio-keys/volume-up uses the gpio pin “4 0” of controller &gpio1 (imx6qdl-sabresd.dtsi)

```
gpio-keys {
    compatible = "gpio-keys";
    power {
        label = "Power Button";
        gpios = <&gpio3 29 0>;
        linux,code = <116>; /* KEY_POWER */
        gpio-key,wakeup;
    };

    volume-up {
        label = "Volume Up";
        gpios = <&gpio1 4 0>;
        linux,code = <115>; /* KEY_VOLUMEUP */
    };
```

```
aliases {
    gpio0 = &gpio1;
    gpio1 = &gpio2;
    gpio2 = &gpio3;
    gpio3 = &gpio4;
    gpio4 = &gpio5;
    gpio5 = &gpio6;
    gpio6 = &gpio7;
```

- Pin allocation is managed by pinctrl system
- Gpio is maintained by gpio drivers
    - different pin ranges managed by different gpio drivers.
        - let gpio drivers announce their pin ranges to the pinctrl subsystem and call 'pinctrl_request_gpio' in order to request the corresponding pin before any gpio usage.
        - gpio controller can use a pinctrl phandle and pins to announce the pinrange to the pin ctrl subsystem

```
qe_pio_e: gpio-controller@1460 {
    #gpio-cells = <2>;
    compatible = "fsl,qe-pario-bank-e", "fsl,qe-pario-bank";
    reg = <0x1460 0x18>;
    gpio-controller;
    gpio-ranges = <&pinctrl1 0 20 10>, <&pinctrl2 10 50 20>;
}
```

*<&pinctrl1 0 20 10>: from base pin 20 to pin 29 under pinctrl1 with gpio offset 0 is handled by this gpio controller

3 恩智浦 i.MX8XBSP 包文件目录结构

恩智浦 linux-4.14.98 BSP 开发包文件目录结构如下:

```
linux-imx
|->arch
| |->arm64
| | |->boot
| | | |->dts
| | | | |->freescale
| | |->configs
| | | |->defconfig
| | |->kernel
| | | |->head.s
| | | |->setup.c
| | |->mm
| | | |->proc.S
|->block
|->crypto
|->documention
| |->devicetree
| |  |->bindings  //dts bindings doc
| | |  |->arm
| | | |  |->imx
| | | | |  |->busfreq-imx6.txt
| | | |  |->gic.txt//gic 中断控制器
| | | |  |->gpio
| | | | |   |->gpio-mxs.txt
```

```
| | | | |->pinctrl
| | | | |   |->fsl,imx6q-pinctrl.txt
|->drivers
| |->of //dts 相关驱动代码
| |->pinctrl //pinctrl 相关驱动代码
| |->block
| |->char
| |->mtd
| |->serial
| | |->mxc_uart.c
|->fs:
| |->ext3
| |->jffs2
| |->ubifs:
| |->yaffs2:
|->include
| |->asm-generic
| |->linux
| |->mtd
| ->init
| |->main.c
|->ipc
|->kernel
| |->printk.c
|->lib
|->mm
|->net
|->scriptes
|->security
|->sound
|->usr
```

|->virt

4 恩智浦 i.MX8XBSP 的编译(no updates)

Makefile 主要用于控制:

- 要编译那一些文件
- .如何编译这些文件
- 如何链接为目标文件及链接顺序

Top folder makefile	所有 makefile 文件的核心，从总体上控制着内核的编译，连接
.config	配置文件，在 make menuconfig 时生成，所有的 makefile 文件，（包括顶层目录及各级子目录）都是根据.config 来决定使用那些文件。
Arch/$(ARCH)/Makefile	对应体系结构的 makefile,它用来决定那些体系结构相关的文件参与内核的生成，并提供一些规则赞成特定格式的内核映象
Scripts/Makefile.*	Makefile 共用的通用规则，脚本等
Kbuild Makefiles	各级子目录下的 makefile,相对简单，被上层 makefile 调用来编译当前目录下的文件

Document /Documentation/kbuild/makefiles.txt 对内核 makefile 的作用，用法

4.1 需要编译哪些文件

- Top make file 决定内核根目录下那些子目录将被编进内核
- Arch/$(ARCH)/Makefile 决定 arch/$(ARCH)目录下那些文件，目录将被编译进内核
- 各级子目录下的 makefile 决定此目录下那些文件将被编译进内核，那些将被编译成模块，进入那些子目录继续调用 它们的 makefile。

1. 顶层 makefile “\Makefile”

顶层 makefile 将 13 个子目录分为 5 类：

a) init-y := init/
b) drivers-y := drivers/ sound/ firmware/
c) net-y := net/
d) libs-y := lib/
e) core-y := usr/

core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/

include $(srctree)/arch/$(SRCARCH)/Makefile[arch 目录被直接包含,扩展以上 5 类]

ARCH/CROSS_COMPILE 变量设置

ARCH ?= $(SUBARCH)

CROSS_COMPILE ?=

2. arch/arm/Makefile

head-y := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o[除前面 5 类，还有一类 head-y,直接以文件名出现，对于有 MMU 的 CPU，使用文件 head.s]

core-y += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/[进一步扩展了 core-y 的内容]

machine-$(CONFIG_ARCH_MX6) := mx6[CONFIG_ARCH_MX6 在配置内核中定义，可以是 y,编译到内核，m,编译为模块，空，不编译]

machdirs := $(patsubst %,arch/arm/mach-%/,$(machine-y))

platdirs := $(patsubst %,arch/arm/plat-%/,$(plat-y))

core-y += $(machdirs) $(platdirs)

libs-y := arch/arm/lib/ $(libs-y) /[进一步扩展了 lib-y 的内容]

编译内核时，将依次进入 init-y,core-y,libs-y,drivers-y,net-y,所列出的目录中执行他们的 makefile,每个子目录都会生成一个 built-in.o(libs-y 所列目录会生成 lib.a),最后，head-y 所表示的文件将和这些 built-in.o,lib.a 一起连接为 vmlinux.’

3. 各子目录 makefile

Make menuconfig->.config, TOP makefile include it

include/config/auto.conf[auto.conf 只是将.config 中的注释去掉，并根据 top makefile 中定义的变量增加一些变量]

auto.conf 中定义的变量值只有两类,y,m. 各级子目录的 makefile 使用这些变量来决定那些文件编译到内核中，那些编译成模块，通过四种办法来确定。

a) Obj-y,编译进内核：obj-y 中定义的.o 文件由当前目录下的.c,.s 文件编译生成，它们连同下一层子目录中的 built-in.o 文件一起组合成(使用”$(LD) -r”命令)当然 built-in.o 文件。此文件又被上一层 makefile 使用。
b) Obj-m,编译成可加载模块.
c) Lib-y,编译成库文件，要把 lib.a 编译时内核，在 top makefile 中的 libs-y 变量列出的当前目录，并且内核代码一般放在 lib/, arch/$(ARCH)/lib/下。
d) Obj-y,obj-m 还可以用来指定要进入的下一层子目录。

4.2 如何编译这些文件

即编译选项，连接选项是什么，这些选项分三类，

- 全局的，适用于整个内核代码树。定义在 top makefile, arch/$(ARCH)/makefile 中定义，包括：CFLAGS,AFLAGS,LDFLAGS,ARFLGAS
- 局部的，适用于某个 makefile 的所有文件。EXTRA_CFLAGS, EXTRA_AFLAGS, EXTRA_LDFLAGS, EXTRA_ARFLGAS
- 个体的，只适用于某个文件。包括 CFLAG_XX, AFLAGS_XX

4.3 如何链接为目标文件及链接顺序

1. Top makefile, arch/$(ARCH)/makefile 定义了 6 类目录，head-y, init-y, drivers-y, net-y, libs-y, core-y.除了 head-y,这样目录的后面直接加上 built-in.o 或 lib.a，表示要连接进内核

- ***init-y := $(patsubst %/, %/built-in.o, $(init-y))***
- ***core-y := $(patsubst %/, %/built-in.o, $(core-y))***
- ***drivers-y := $(patsubst %/, %/built-in.o, $(drivers-y))***
- ***net-y := $(patsubst %/, %/built-in.o, $(net-y))***

libs-y1 := $(patsubst %/, %/lib.a, $(libs-y))

libs-y2 := $(patsubst %/, %/built-in.o, $(libs-y))

- ***libs-y := $(libs-y1) $(libs-y2)***

[patsubst 将 init-y 转换为 init/built-in.o]

2.

vmlinux-init := $(head-y) $(init-y)

vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)

vmlinux-all := $(vmlinux-init) $(vmlinux-main)

vmlinux-lds := arch/$(SRCARCH)/kernel/vmlinux.lds

- vmlinux-all,表示所有构成 vmlinux 的目标文件，按顺序为 head-y,init-y,core-y,libs-y,drivers-y,net-y 即：***arch/arm/kernel/head.o,arch/arm/kernel/init_task.o,init/built-in.o,user/built-in.o*** 等等
- vmlinux-lds 指定连接脚本为 arch/$(ARCH)/kernel/vmlinux.lds,它是由 arch/arm/kernel/vmlinux.lds.s 文件生成的，规则在 script/makefile.build 中。

```
SECTIONS
{
 . = 0xC0000000 + 0x00008000; //代码段的起始地址，是一个虚拟地址
 .text.head : {
  _stext = .;
  _sinittext = .;
  *(.text.head)
 }
```

```
.init : { /* Init code and data                */内核初始化的代码与数据
……
}
.text : { /* Real text segment                 */真正的代码段
 _text = .; /* Text and read-only data        */代码段和只读数据的开始地址
……
}
. = ALIGN((4096)); .rodata : AT(ADDR(.rodata) - 0) { … } . = ALIGN((4096));\\只读数据
_etext = .; /* End of text and rodata section */代码段和只读数据的结束地址
. = ALIGN(8192);
__data_loc = .;
.data : AT(__data_loc) {//数据段
 __data_start = .; /* address in memory *//数据段开始地址
…
 _edata = .; /数据段结束地址
}
_edata_loc = __data_loc + SIZEOF(.data);
.bss : {//bss 段，没有初始化或初值=0 的全局，静态变量
 _end = .;
}
   /* Stabs debugging sections.   */调试信息段
.stab 0 : { *(.stab) }
}
```

总结：

- 配置文件.config 中定义了一系列的变量，makefile 将结合它们来决定那些文件被编译进内核，那些被编译模块，涉及那些子目录。
- 顶层 makefile 和 arch/$(ARCH)/Makefile 决定根目录下那些子目录，arch/$(ARCH)下那些文件和目录将被编译进内核。
- 各级子目录下的 Makefile 决定所在目录下那些文件编译进内核，那些编译进模块，进入那些子目录继续调用它们的 makefile.
- Top makefile 和 arch/$(ARCH)/Makefile 设置了可以影响所有文件编译，连接选项。
- 各级子目录下的 makefile 中可以设置所有文件的编译，连接选项。

- Top makefile 按照一定的顺序组织文件，根据连接脚本 arch/$(ARCH)/kernel/vmlinux.lds 生成内核映象文件 vmlinux

4.4 kernel Kconfig

Make menuconfig 读取 arch/$(ARCH)/Kconfig 来生成配置界面，这个文件是所有 Kconfig 的入口，包括了其它目录下的 Kconfig 文件。Kconfig 用于配置内核，它是各种配置界面的源文件，最后生成配置文件.config,其语法参考\documentation\kbuild\kconfig-language.txt.

5 恩智浦 BSP 的内核初始化过程(no updates)

标准 linux 内核初始化过程如下:

内核引导第一阶段：汇编语言

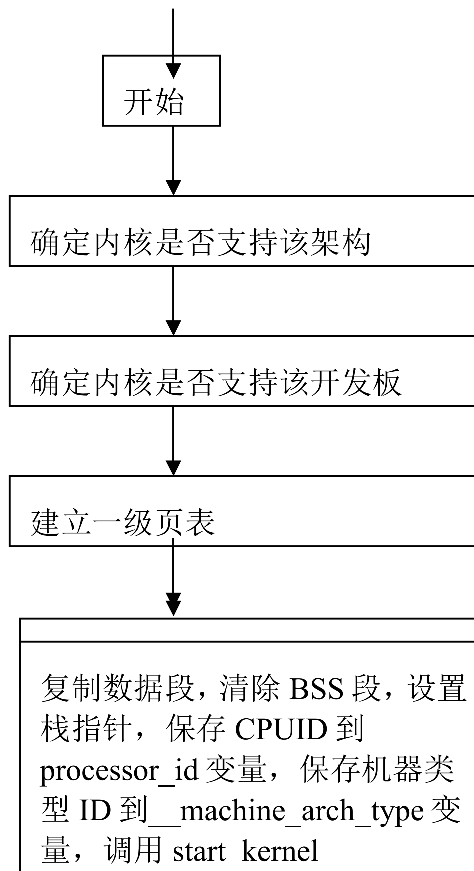
Arch/arm/kernel/head.s
arch/arm/kernel/head-common.s
arch/arm/mm/proc-v7.s

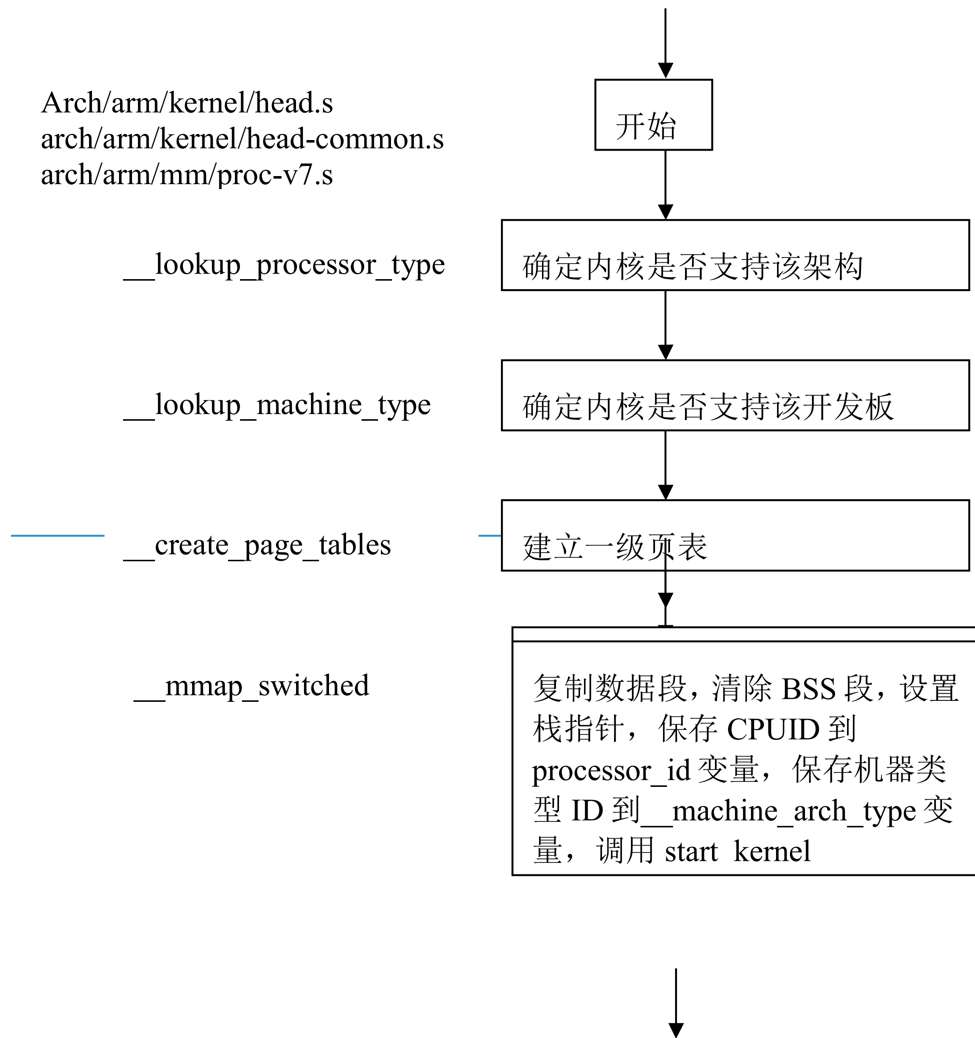
__lookup_processor_type

__lookup_machine_type

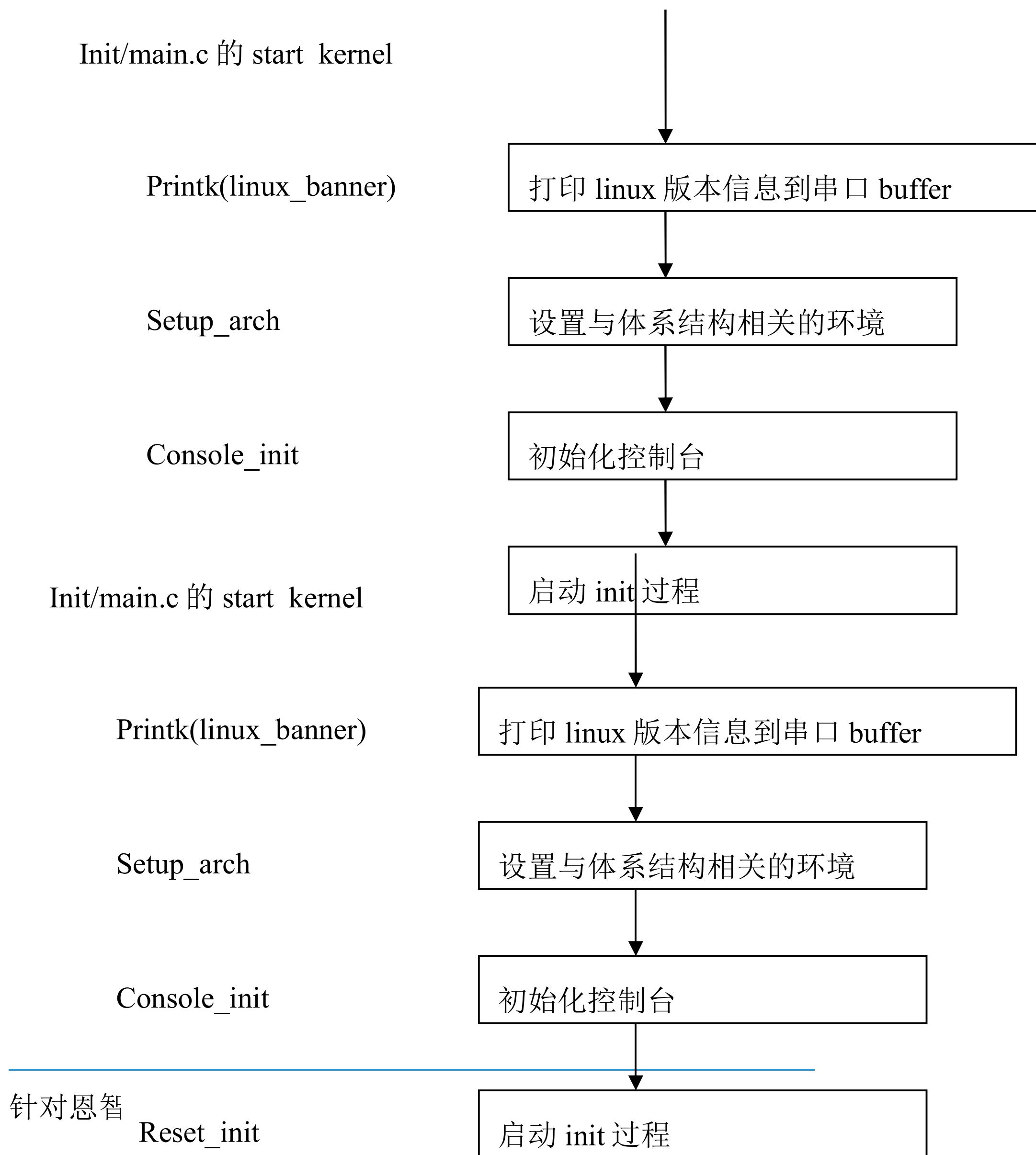
__create_page_tables

__mmap_switched





内核启动的第二阶段：C 语言



5.1 初始化的汇编代码

Head.s(\arch\arm\kernel)

```
/*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
```

```
* r1 = machine nr, r2 = atags or dtb pointer.
*
```

目前的 kernel 支持旧的 tag list 的方式，同时也支持 device tree 的方式。r2 可能是 device tree binary file 的指针（bootloader 要传递给内核之前要 copy 到 memory 中），也可以能是 tag list 的指针。在 ARM 的汇编部分的启动代码中（主要是 head.S 和 head-common.S），machine type ID 和指向 DTB 或者 atags 的指针被保存在变量 __machine_arch_type 和__atags_pointer 中，这么做是为了后续 c 代码进行处理。

```
* This code is mostly position independent, so if you link the kernel at
* 0xc0008000, you call this at __pa(0xc0008000).
*
* See linux/arch/arm/tools/mach-types for the complete list of machine
* numbers for r1.
*
* We're trying to keep crap to a minimum; DO NOT add any machine specific
* crap here - that's what the boot loader (or in extreme, well justified
* circumstances, zImage) is for.
*/

    @ ensure svc mode and all interrupts masked
        safe_svcmode_maskall r9
```

进入 svc 模式，关中断

```
        mrc     p15, 0, r9, c0, c0          @ get processor id
```

从协处理器 CP15 的寄存器 C0 中获得 CPUID，

[31:24]	厂商编号：0x41=A: ARM 公司
[23:20]	产品子编号
[19:16]	ARM 体系版本号：
[15:4]	产品主编号
[3:0]	处理器版本号

```
        bl      __lookup_processor_type         @ r5=procinfo r9=cupid
```

确定内核是否支持当前 cpu, 如果支持，r5 返回一个用于描述处理器的结构体的地址，proc_info_list 结构体原形定义在 include/asm-arm/procinfo.h,

```
    struct proc_info_list {
        unsigned int            cpu_val;
        unsigned int            cpu_mask;
```

```
	unsigned long		__cpu_mm_mmu_flags;	/* used by head.S */
	unsigned long		__cpu_io_mmu_flags;	/* used by head.S */
	unsigned long		__cpu_flush;		/* used by head.S */
	const char		*arch_name;
	const char		*elf_name;
	unsigned int		elf_hwcap;
	const char		*cpu_name;
	struct processor	*proc;
	struct cpu_tlb_fns	*tlb;
	struct cpu_user_fns	*user;
	struct cpu_cache_fns	*cache;
};
```

表示它支持的 cpu, arm 这个结构体定义在 arch/arm/mm/proc-v7.S

```
section ".proc.info.init", #alloc, #execinstr
```

不同的 proc_info_list 结构支持不同的 cpu,他们定义在.proc.info.init 段中,连接时这些结构体组织在一起，在\arch\arm\kernel\vmlinux.lds 中，

```
	__proc_info_begin = .; //proc_info_list 结构开始地址
				*(.proc.info.init)
			__proc_info_end = .; //proc_info_list 结构结束地址
```

以上标志说明此 section 的数据链接到了 proc_info_list 中，其数据结构如下：

```
	/*
	 * Match any ARMv7 processor core.
	 */
	.type	__v7_proc_info, #object
__v7_proc_info:
	.long	0x000f0000		@ Required ID value
	.long	0x000f0000		@ Mask for ID
```

函数__lookup_processor_type 在\arch\arm\kernel\head-common.S 中,根据前面读出的 cpuid,存在寄存器 r9 中，然后从__proc_info_begin 向后搜索，找到匹配的.

```
/*
 * Read processor ID register (CP#15, CR0), and look up in the linker-built
 * supported processor list.  Note that we can't use the absolute addresses
 * for the __proc_info lists since we aren't running with the MMU on
 * (and therefore, we are not in the correct address space).  We have to
 * calculate the offset.
```

```
 *
 *      r9 = cpuid
 * Returns:
 *      r3, r4, r6 corrupted
 *      r5 = proc_info pointer in physical address space
 *      r9 = cpuid (preserved)
 */
        movs    r10, r5                                 @ invalid processor?
        moveq   r0, #'p'                        @ yes, error 'p'
THUMB( it   eq )                    @ force fixup-able long branch encoding
        beq     __error_p
bl      __vet_atags
bl      __create_page_tables
```

//创建一级页表以建立虚拟地址到物理地址的映射关系，它用到了__lookup_processor_type 函数返回的 proc_info_list 结构体

```
        /*
         * The following calls CPU specific code in a position independent
         * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
         * xxx_proc_info structure selected by __lookup_processor_type
         * above.  On return, the CPU will be ready for the MMU to be
         * turned on, and r0 will hold the CPU control register value.
         */
        ldr     r13, =__mmap_switched                   @ address to jump to after
                                                        @ mmu has been enabled
        adr     lr, BSYM(1f)                            @ return (PIC) address
        mov     r8, r4                                  @ set TTBR1 to swapper_pg_dir
          b     __enable_mmu
* Enable the MMU.  This completely changes the structure of the visible
 * memory space.  You will not be able to trace execution through this.
 * If you have an enquiry about this, *please* check the linux-arm-kernel
 * mailing list archives BEFORE sending another post to the list.
 *
 *  r0  = cp#15 control register
 *  r1  = machine ID
 *  r2  = atags or dtb pointer
```

```
 *  r9  = processor ID
 *  r13 = *virtual* address to jump to upon completion
 *
 * other registers depend on the function called upon completion
 */
#include "head-common.S"
```

Head-common.S(\arch\arm\kernel)

```
b       start_kernel
```

//跳转到 C 代码运行

5.2 初始化的 C 代码

start_kernel (\init\Main.c) 从此，内核启动进入第二阶段

```
        * Need to run as early as possible, to initialize the
        * lockdep hash:
        */
|->         lockdep_init();
|->         smp_setup_processor_id();
|->         debug_objects_early_init();
            /*
             * Set up the the initial canary ASAP:
             */
|->         boot_init_stack_canary();
|->         cgroup_init_early();
|->         local_irq_disable();
            early_boot_irqs_disabled = true;
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
|->         boot_cpu_init();
|->         page_address_init();
```

|-> pr_notice("%s", linux_banner);只是打印到 buffer,只能在 console_init 函数注册，初始化后才能真正输出

|-> ***setup_arch***(&command_line)[\init\Main.c]:initial the cpu device, and parser the command line and store it into command_line

在新的内核中，首先使用 setup_machine_fdt 来 setup machine 描述符，如果返回 NULL，才使用传统的方法 setup_machine_tags 来 setup machine 描述符

```
void __init setup_arch(char **cmdline_p)
{
  const struct machine_desc *mdesc;
……
  mdesc = setup_machine_fdt(__atags_pointer);
  if (!mdesc)
      mdesc = setup_machine_tags(__atags_pointer, __machine_arch_type);
  machine_desc = mdesc;
  machine_name = mdesc->name;
……
}
```

| |->setup_machine_fdt[arch\arm\kernel\devtree.c] setup_machine_fdt 函数的功能就是根据 Device Tree 的信息，找到最适合的 machine 描述符。具体代码如下：

```
/**
 * setup_machine_fdt - Machine setup when an dtb was passed to the kernel
 * @dt_phys: physical address of dt blob
 *
 * If a dtb was passed to the kernel in r2, then use it to choose the
 * correct machine_desc and to setup the system.
 */
const struct machine_desc * __init setup_machine_fdt(unsigned int dt_phys)
{
  const struct machine_desc *mdesc, *mdesc_best = NULL;
  if (!dt_phys || !early_init_dt_scan(phys_to_virt(dt_phys)))
      return NULL;
  mdesc = of_flat_dt_match_machine(mdesc_best, arch_get_next_mach);
  if (!mdesc) {
      出错处理
  }
  /* Change machine number to match the mdesc we're using */
  __machine_arch_type = mdesc->nr;
  return mdesc;
}
```

early_init_dt_scan 函数有两个功能，一个是为后续的 DTB scan 进行准备工作，另外一个是运行时参数传递

运行时参数是在扫描DTB的chosen node时候完成的，具体的动作就是获取chosen node的bootargs、initrd等属性的value，并将其保存在全局变量（boot_command_line，initrd_start、initrd_end）中。使用tag list方法是类似的，通过分析tag list，获取相关信息，保存在同样的全局变量中。具体代码位于early_init_dt_scan函数中：

```
bool __init early_init_dt_scan(void *params)
{
    if (!params)
        return false;
    /* 全局变量 initial_boot_params 指向了 DTB 的 header*/
    initial_boot_params = params;
    /* 检查 DTB 的 magic，确认是一个有效的 DTB */
    if (be32_to_cpu(initial_boot_params->magic) != OF_DT_HEADER) {
        initial_boot_params = NULL;
        return false;
    }
    /* 扫描 /chosen node，保存运行时参数（bootargs）到 boot_command_line，此外，还处理 initrd 相关的 property，并保存在 initrd_start 和 initrd_end 这两个全局变量中 */
    of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);
    /* 扫描根节点，获取 {size,address}-cells 信息，并保存在 dt_root_size_cells 和 dt_root_addr_cells 全局变量中 */
    of_scan_flat_dt(early_init_dt_scan_root, NULL);
    /* 扫描 DTB 中的 memory node，并把相关信息保存在 meminfo 中，全局变量 meminfo 保存了系统内存相关的信息。*/
    of_scan_flat_dt(early_init_dt_scan_memory, NULL);
    return true;
}
```

设定meminfo（该全局变量确定了物理内存的布局）有若干种途径：

1、通过tag list（tag是ATAG_MEM）传递memory bank的信息。

2、通过command line（可以用tag list，也可以通过DTB）传递memory bank的信息。

3、通过DTB的memory node传递memory bank的信息。

目前当然是推荐使用Device Tree的方式来传递物理内存布局信息。

of_flat_dt_match_machine是在machine描述符的列表中scan，找到最合适的那个machine描述符。我们首先看如何组成machine描述符的列表。和传统的方法类似，也是静态定义的。DT_MACHINE_START和MACHINE_END用来定义一个machine描述符。编译的时候，compiler会把这些machine descriptor放到一个特殊的段中（.arch.info.init），形成machine描述符的列表。machine描述符用下面的数据结构来标识（删除了不相关的member）：

```
struct machine_desc {
    unsigned int        nr;        /* architecture number    */
    const char *const     *dt_compat;    /* array of device tree 'compatible' strings    */
……
  };
```

nr 成员就是过去使用的 machine type ID。内核 machine 描述符的 table 有若干个 entry，每个都有自己的 ID。bootloader 传递了 machine type ID，指明使用哪一个 machine 描述符。目前匹配 machine 描述符使用 compatible strings，也就是 dt_compat 成员，这是一个 string list，定义了这个 machine 所支持的列表。在扫描 machine 描述符列表的时候需要不断的获取下一个 machine 描述符的 compatible 字符串的信息，具体的代码如下：

```
static const void * __init arch_get_next_mach(const char *const **match)
{
  static const struct machine_desc *mdesc = __arch_info_begin;
  const struct machine_desc *m = mdesc;
  if (m >= __arch_info_end)
      return NULL;
  mdesc++;
  *match = m->dt_compat;
  return m;
}
```

__arch_info_begin 指向 machine 描述符列表第一个 entry。通过 mdesc++不断的移动 machine 描述符指针（Note：mdesc 是 static 的）。match 返回了该 machine 描述符的 compatible string list。具体匹配的算法倒是很简单，就是比较字符串而已，一个是 root node 的 compatible 字符串列表，一个是 machine 描述符的 compatible 字符串列表，得分最低的（最匹配的）就是我们最终选定的 machine type

在系统初始化的过程中，我们需要将 DTB 转换成节点是 device_node 的树状结构，以便后续方便操作。具体的代码位于 setup_arch->unflatten_device_tree 中。

```
void __init unflatten_device_tree(void)
{
  __unflatten_device_tree(initial_boot_params, &of_allnodes,
              early_init_dt_alloc_memory_arch);
  /* Get pointer to "/chosen" and "/aliases" nodes for use everywhere */
  of_alias_scan(early_init_dt_alloc_memory_arch);
}
```

我们用 struct device_node 来抽象设备树中的一个节点，具体解释如下：

```
struct device_node {
  const char *name;－－－－－－－－－－－－－－－－－－－－－－－－device node name
  const char *type;－－－－－－－－－－－－－－－－－－－－－－－－－对应 device_type 的属性
  phandle phandle;－－－－－－－－－－－－－－－－－－－－－－－－－对应该节点的 phandle 属性
  const char *full_name; －－－－－－－－－－－－－－－－－从“/”开始的，表示该 node 的 full path
  struct    property *properties;－－－－－－－－－－－－－－该节点的属性列表
  struct    property *deadprops; －－－－－－－－－－－如果需要删除某些属性，kernel 并非真的删除，而是挂入到
deadprops 的列表
  struct    device_node *parent;－－－－－－parent、child 以及 sibling 将所有的 device node 连接起来
  struct    device_node *child;
  struct    device_node *sibling;
  struct    device_node *next; －－－－－－－－－通过该指针可以获取相同类型的下一个 node
```

```
  struct   device_node *allnext;————————通过该指针可以获取 node global list 下一个 node
  struct   proc_dir_entry *pde;————————开放到 userspace 的 proc 接口信息
  struct   kref kref;———————————————该 node 的 reference count
  unsigned long _flags;
  void    *data;
};
```

unflatten_device_tree 函数的主要功能就是扫描 DTB，将 device node 被组织成：

1、global list。全局变量 struct device_node *of_allnodes 就是指向设备树的 global list

2、tree。

这些功能主要是在__unflatten_device_tree 函数中实现，具体代码如下（去掉一些无关紧要的代码）：

```
static void __unflatten_device_tree(struct boot_param_header *blob,———需要扫描的 DTB
             struct device_node **mynodes,——————————global list 指针
             void * (*dt_alloc)(u64 size, u64 align))———————内存分配函数
{
  unsigned long size;
  void *start, *mem;
  struct device_node **allnextp = mynodes;
  此处删除了 health check 代码，例如检查 DTB header 的 magic，确认 blob 的确指向一个 DTB。
  /* scan 过程分成两轮，第一轮主要是确定 device-tree structure 的长度，保存在 size 变量中 */
  start = ((void *)blob) + be32_to_cpu(blob->off_dt_struct);
  size = (unsigned long)unflatten_dt_node(blob, 0, &start, NULL, NULL, 0);
  size = ALIGN(size, 4);
  /* 初始化的时候，并不是扫描到一个 node 或者 property 就分配相应的内存，实际上内核是一次性的分配了一大片
内存，这些内存包括了所有的 struct device_node、node name、struct property 所需要的内存。*/
  mem = dt_alloc(size + 4, __alignof__(struct device_node));
  memset(mem, 0, size);
  *(__be32 *)(mem + size) = cpu_to_be32(0xdeadbeef);   //用来检验后面 unflattening 是否溢出
  /* 这是第二轮的 scan，第一次 scan 是为了得到保存所有 node 和 property 所需要的内存 size，第二次就是实打实的
要构建 device node tree 了 */
  start = ((void *)blob) + be32_to_cpu(blob->off_dt_struct);
  unflatten_dt_node(blob, mem, &start, NULL, &allnextp, 0);

  此处略去校验溢出和校验 OF_DT_END。
}
```

| |->setup_processor：处理器相关设置，它会调用引导阶段的 lookup_process_type 函数，来获得该处理器的 proc_info_list 结构

| |->setup_machine_tags (machine_arch_type):得到开发板的 machine_desc 结构，以后根据开发板的这个结构来进行开发板的一些操作,在\arch\arm\mach-mx6\board-mx6q_sabersd.c 中会定义板级的信息：

```
/*
 * initialize __mach_desc_MX6Q_SABRESD data structure.
 */
MACHINE_START(MX6Q_SABRESD, "NXP i.MX 6Quad/DualLite/Solo Sabre-SD Board")
	/* Maintainer: NXP Semiconductor, Inc. */
	.boot_params = MX6_PHYS_OFFSET + 0x100,
	.fixup = fixup_mxc_board,
	.map_io = mx6_map_io,
	.init_irq = mx6_init_irq,
	.init_machine = mx6_sabresd_board_init,
	.timer = &mx6_sabresd_timer,
	.reserve = mx6q_sabresd_reserve,
MACHINE_END
```

其中 MACHINE_START 和 MACHINE_END 定义在\arch\arm\include\asm\mach\arch.h

```
/*
 * Set of macros to define architecture features.  This is built into
 * a table by the linker.
 */
#define MACHINE_START(_type,_name)			\
static const struct machine_desc __mach_desc_##_type	\
 __used							\
 __attribute__((__section__(".arch.info.init"))) = {	\
	.nr		= MACH_TYPE_##_type,		\
	.name		= _name,

#define MACHINE_END				\
};
```

machine_desc 结构,从\arch\arm\kernel\vmlinux.lds,连接在

```
 __arch_info_begin = .;
 *(.arch.info.init)
 __arch_info_end = .;
```

machine_arch_type=MACH_TYPE_MX6Q_SABRESD, 定义在 [\include\asm\Mach-typ.h], 从结构体 __mach_desc_ MX6Q_SABRESD 中获得机器相关全局变量.

```
if (mdesc->soft_reboot)
		reboot_setup("s");
```

如果是软件重启，将重启模式改为 s.

```
|->mm_init_owner(&init_mm, &init_task);
|->mm_init_cpumask(&init_mm);
 |->setup_command_line
|->setup_nr_cpu_ids();
|->setup_per_cpu_areas();
|->smp_prepare_boot_cpu();          /* arch-specific boot-cpu hooks */
|->build_all_zonelists(NULL, NULL);
|->page_alloc_init();
|->pr_notice("Kernel command line: %s\n", boot_command_line);
|->parse_early_param();
|->parse_args("Booting kernel", static_command_line, __start___param,
    __stop___param - __start___param, &unknown_bootoption);
```

比如对于命令"console=ttymxc",它的处理过程就是 parse_args,unknown_bootoption,最后调用代码处理函数：***__setup("console=", console_setup);***命令行"console=.."经过 console_setup 处理后在全局变量 console_cmdline 中保存这些信息，在后面 console_init 函数初始化控制台后，根据这些信息选择控制台

```
|->jump_label_init();
            /*
             * These use large bootmem allocations and must precede
             * kmem_cache_init()
             */
|->setup_log_buf(0);
|->pidhash_init();
|->vfs_caches_init_early();
|->sort_main_extable();
|->trap_init();
|->mm_init();
            /*
             * Set up the scheduler prior starting any interrupts (such as the
             * timer interrupt). Full topology setup happens at smp_init()
             * time - but meanwhile we still have a functioning scheduler.
             */
|->sched_init();
            /*
```

```
            * Disable preemption - early bootup scheduling is extremely
            * fragile until we cpu_idle() for the first time.
            */
|->preempt_disable();
           if (WARN(!irqs_disabled(), "Interrupts were enabled *very* early, fixing it\n"))
|->local_irq_disable();
|->idr_init_cache();
|->rcu_init();
|->tick_nohz_init();
|->context_tracking_init();
|->radix_tree_init();
           /* init some links before init_ISA_irqs() */
|->early_irq_init();
|->init_IRQ.
|   |->machine_desc->init_irq()=mx6_init_irq 初始化中断
|->tick_init();
|->init_timers();
|->hrtimers_init();
|->softirq_init();
|->timekeeping_init();
|->time_init();
|->sched_clock_postinit();
|->perf_event_init();
|->profile_init();
|->call_function_init();
           WARN(!irqs_disabled(), "Interrupts were enabled early\n");
           early_boot_irqs_disabled = false;
|->local_irq_enable();
|->kmem_cache_init_late();
```

|->console_init: Initialize the console device. This is called *early*, so we can't necessarily depend on lots of kernel help here. Just do some early initializations, and do the complex setup later.

他调用__con_initcall_start 到__con_initcall_end 之间定义的每个函数，这些函数使用宏 console_initcall 来指定[\drivers\serial\mxc_uart.c]

```
|     |->console_initcall(mxcuart_console_init);注册 mxc_console,
    * This structure contains the information such as the name of the UART driver
```

```
 * that appears in the /dev folder, major and minor numbers etc. This structure
 * is passed to the serial_core.c file.
 */
static struct uart_driver mxc_reg = {
	.owner = THIS_MODULE,
	.driver_name = "ttymxc",
	.dev_name = "ttymxc",
	.major = SERIAL_MXC_MAJOR,
	.minor = SERIAL_MXC_MINOR,
	.nr = MXC_UART_NR,
	.cons = MXC_CONSOLE,
};

/*!
 * This structure contains the pointers to the UART console functions. It is
 * passed as an argument when registering the console.
 */
static struct console mxc_console = {
	.name = "ttymxc",
	.write = mxcuart_console_write,
	.device = uart_console_device,
	.setup = mxcuart_console_setup,//串口初始化函数，对于 index=-1,将从串口 0 打出,系统通过 index 查表 mxc_ports 得到相关串口资源，定义在[/arch/arm/mach-mx51/serial.c 中]
	.flags = CON_PRINTBUFFER,
	.index = -1,
	.data = &mxc_reg,
};
  |->lockdep_info();
	/*
	 * Need to run this when irqs are enabled, because it wants
	 * to self-test [hard/soft]-irqs on/off lock inversion bugs
	 * too:
	 */
  |->locking_selftest();
  |->page_cgroup_init();
```

```
 |-> debug_objects_mem_init();
 |-> kmemleak_init();
 |-> setup_per_cpu_pageset();
 |-> numa_policy_init();
     if (late_time_init)
 |->         late_time_init();
 |-> sched_clock_init();
 |-> calibrate_delay(); //maybe delay the boot time, can by pass the uboot command to it.
 |-> pidmap_init();
 |-> anon_vma_init();
 |-> acpi_early_init();
thread_info_cache_init();
|->  cred_init();
|->  fork_init(totalram_pages);
|->  proc_caches_init();
|->  buffer_init();
|->  key_init();
|->  security_init();
|->  dbg_late_init();
|->  vfs_caches_init(totalram_pages);
|->  signals_init();
     /* rootfs populating might need page-writeback */
|->  page_writeback_init()
|->cgroup_init();
|->  cpuset_init();
|->  taskstats_init_early();
|->  delayacct_init();
|->  check_bugs();
|->  sfi_init_late();
     if (efi_enabled(EFI_RUNTIME_SERVICES)) {
             efi_late_init();
             efi_free_boot_services();
```

```
        }
|->     ftrace_init();
  |->rest_init[\init\Main.c]:
/*
 * We need to finalize in a non-__init function or else race conditions
 * between the root thread and the init thread may cause start_kernel to
 * be reaped by free_initmem before the root thread has proceeded to
 * cpu_idle.
 *
 * gcc-3.4 accidentally inlines this function, so use noinline.
 */
```

| |-> create kernel thread: kernel_init: it is the init thread, which will run /etc/init,/bin/init or /sbin/init

Kthreadd

Kernel schedule

Cpu_idle

rest_init()函数:

Start_kernel()函数负责初始化内核各子系统，最后调用 reset_init()，启动一个叫做 init 的内核线程，继续初始化。在 init 内核线程中，将执行下列 init()函数的程序。Init()函数负责完成根文件系统的挂接、初始化设备驱动程序和启动用户空间的 init 进程等重要工作。

```
 |->kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
 static int __ref kernel_init(void *unused)
    |  |->  kernel_init_freeable();
  |  |   |-> do_basic_setup //初始化设备驱动程序
  |  |  |   |-> do_initcalls
for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
                do_initcall_level(level);
static void __init do_initcall_level(int level)
{
        extern const struct kernel_param __start___param[], __stop___param[];
        initcall_t *fn;

        strcpy(initcall_command_line, saved_command_line);
        parse_args(initcall_level_names[level],
                   initcall_command_line, __start___param,
                   __stop___param - __start___param,
```

```
                        level, level,
                        &repair_env_string);

        for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
                    do_one_initcall(*fn);
    }
| | |->try_to_run_init_process ("/sbin/init");
    | | |->try_to_run_init_process ("/etc/init");
    | | |->try_to_run_init_processbin/init");
    | | |->try_to_run_init_process ("/bin/sh");
```

分析上面一段代码可以看出，设备的初始化是通过 do_basic_setup()函数调用 do_initcalls()函数，实现__early_initcall_end, __initcall_end 段之间的指针函数执行的。而到底是那些驱动函数怎么会被集中到这个段内的呢？我们知道系统内存空间的分配是由链接器 ld 读取链接脚本文件决定。链接器将同样属性的文件组织到相同的段里面去，如所有的.text 段都被放在一起。在链接脚本里面可以获得某块内存空间的具体地址。我们来看下\arch\arm\kernel\vmlinux.lds.S 文件。由于文件过长，只贴出和__initcall_start, __initcall_end 相关的部分。

```
__initcall_start = .;
          *(.initcall1.init)
          *(.initcall2.init)
          *(.initcall3.init)
          *(.initcall4.init)
          *(.initcall5.init)
          *(.initcall6.init)
          *(.initcall7.init)
      __initcall_end = .;
```

从脚本文件中我们可以看出， 在__initcall_start, __initcall_end 之间放置的是属行为(.initcall*.init)的函数数据 。在 linux/include/linux/init.h 文件中可以知道，(.initcall*.init)属性是由__define_initcall(level, fn)宏设定的。

```
#define __define_initcall(level,fn) \
   static initcall_t __initcall_##fn __attribute_used__ \
   __attribute__((__section__(".initcall" level ".init"))) = fn

/*
    * A "pure" initcall has no dependencies on anything else, and purely
    * initializes variables that couldn't be statically initialized.
    *
    * This only exists for built-in code, not for modules.
    */
    #define pure_initcall(fn)                   __define_initcall("0",fn,0)
    #define core_initcall(fn)                   __define_initcall("1",fn,1)
```

```
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)             __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn) __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                 __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)             __define_initcall("6",fn,6)
#define device_initcall_sync(fn) __define_initcall("6s",fn,6s)
#define late_initcall(fn)         __define_initcall("7",fn,7)
#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
```

由此可以判断，所有的设备驱动函数都必然通过*_initcall(fn)宏的处理。以此为入口，可以查询所有的设备驱动。

```
core_initcall(fn)
……

postcore_initcall(fn)
…
```

arch_initcall(fn)

arch_initcall(customize_machine);

static int __init customize_machine(void) linux/arch/arm/kernel/setup.c 在此处调用 函数 machine_desc->init_machine=mx6_board_init

static int __init mxc_init_uart(void)   linux/arch/arm/mach-mx51/serial.c

```
subsys_initcall(fn)
…

fs_initcall(fn)
…

device_initcall(fn)
…

late_initcall(fn)
…
```

5.3 init_machine

init_machine is call by customize_machine[\arch\arm\kernel\Setup.c], which is kernel module initial function: arch_initcall(customize_machine).

```
DT_MACHINE_START(IMX6Q, "NXP i.MX6 Quad/DualLite (Device Tree)")
	/*
	 * i.MX6Q/DL maps system memory at 0x10000000 (offset 256MiB), and
	 * GPU has a limit on physical address that it accesses, which must
	 * be below 2GiB.
	 */
	.dma_zone_size	= (SZ_2G - SZ_256M),
	.smp		= smp_ops(imx_smp_ops),
	.map_io		= imx6q_map_io,
	.init_irq	= imx6q_init_irq,
	.init_machine	= imx6q_init_machine,
	.init_late	= imx6q_init_late,
	.dt_compat	= imx6q_dt_compat,
	.restart	= mxc_restart,
MACHINE_END
```

Init_machine point to imx6q_init_machine [\arch\arm\mach-imx\mach-imx6q.c]

```
|->imx_print_silicon_rev(cpu_is_imx6dl() ? "i.MX6DL" : "i.MX6Q",
				imx_get_soc_revision());//print i.mx silicon
|->mxc_arch_reset_init_dt();//initial the dtb
|	|-> of_find_compatible_node(NULL, NULL, "fsl,imx6q-gpc");
//set the wdog from the dtb
|	|-> imx_soc_device_init
	root = of_find_node_by_path("/");
	ret = of_property_read_string(root, "model", &soc_dev_attr->machine);
	of_node_put(root);
…
	case MXC_CPU_IMX6Q:
		soc_id = "i.MX6Q";
…
|	|->soc_device_register
|	|-> soc_device_to_device
|-> int of_platform_populate(struct device_node *root,
```

```
                                const struct of_device_id *matches,
                                const struct of_dev_auxdata *lookup,
                                struct device *parent)
/**
 * of_platform_populate() - Populate platform_devices from device tree data
 * @root: parent of the first level to probe or NULL for the root of the tree
 * @matches: match table, NULL to use the default
 * @lookup: auxdata table for matching id and platform_data with device nodes
 * @parent: parent to hook devices from, NULL for toplevel
 *
 * Similar to of_platform_bus_probe(), this function walks the device tree
 * and creates devices from nodes.  It differs in that it follows the modern
 * convention of requiring all device nodes to have a 'compatible' property,
 * and it is suitable for creating devices which are children of the root
 * node (of_platform_bus_probe will only create children of the root which
 * are selected by the @matches argument).
 *
 * New board support should be using this function instead of
 * of_platform_bus_probe().
 *
 * Returns 0 on success, < 0 on failure.
 */
//found the root
root = root ? of_node_get(root) : of_find_node_by_path("/");
        if (!root)
                return -EINVAL;
//from the root to find each child
for_each_child_of_node(root, child) {
                rc = of_platform_bus_create(child, matches, lookup, parent, true);
                if (rc)
                        break;
        }
|  |-> static int of_platform_bus_create(struct device_node *bus,
                                const struct of_device_id *matches,
                                const struct of_dev_auxdata *lookup,
                                struct device *parent, bool strict)
```

```
/**
 * of_platform_bus_create() - Create a device for a node and its children.
 * @bus: device node of the bus to instantiate
 * @matches: match table for bus nodes
 * @lookup: auxdata table for matching id and platform_data with device nodes
 * @parent: parent for new device, or NULL for top level.
 * @strict: require compatible property
 *
 * Creates a platform_device for the provided device_node, and optionally
 * recursively create devices for all the child nodes.
 */
| | |-> static struct platform_device *of_platform_device_create_pdata(
                                        struct device_node *np,
                                        const char *bus_id,
                                        void *platform_data,
                                        struct device *parent)
/**
 * of_platform_device_create_pdata - Alloc, initialize and register an of_device
 * @np: pointer to node to create device for
 * @bus_id: name to assign device
 * @platform_data: pointer to populate platform_data pointer with
 * @parent: Linux device model parent device.
 *
 * Returns pointer to created platform device, or NULL if a device was not
 * registered.  Unavailable devices will not get registered.
 */
```

总结：

通过分析恩智浦 **i.MX6X** 内核驱动代码的目录结构与启动流程，我们了解到：

1. 恩智浦 **i.MX6X(**包括 **i.MX6Q/D/DL/S/SL)**都可以使用同一下内核镜像来支持，使用 **DTB** 来传递板级相关信息。

2. 恩智浦 **i.MX6X** 开发板的板级信息通过 **machine_desc** 数据结构链接在固定的内存地址，然后内核通过 **uboot** 传递过来的板级信息来查找相对应的开发板的 **machine_desc** 数据结构，然后使用这个数据结构中的函数来初始化相应的开发板的资源，包括注册相关的驱动。

3. 因为以上原因，将恩智浦 **i.MX6X** 的 **BSP** 移植到自己板上所要做的工作，将不用修改 **make menuconfig** 来增减驱动，而是创建或修改 **machine_desc** 数据结构，其中最重要的工作是修改其 **dts** 文件，来完成对此板所需要驱动的注册工作。

4. i.MX6X 的 **io** 管脚配置，集中在 **imx6qdl-sabresd.dts** 中配置。

6 恩智浦 BSP 的内核定制

关于 i.MX8X BSP 的情况，请参考 BSP 用户手册" imx-yocto-L4.14.98_2.0.0_ga\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。

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。

6.1 DDR 修改

根据文档《MX8X_4.14.98_ga_BootLoader_V5-20190903_chn.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\fsl-imx8dx.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\fsl-imx8dx.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_4.14.98_ga_BootLoader_V5-20190903_chn.pdf文档如何去掉。

/*
        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。

        dsp_reserved:… // audio DSP使用，没有DSP可以注掉，注意一下调用处和DSP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
…
```

6.2 IO 管脚配置与 Pinctrl 驱动

6.2.1 i.MX8QXP MEK 板的管脚配置

i.MX8QXP MEK 板的 IO 管脚配置大部分定义在:

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

```
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;
        imx8qxp-mek {
                pinctrl_hog: hoggrp {
                        fsl,pins = <
                                SC_P_MCLK_OUT0_ADMA_ACM_MCLK_OUT0 0x0600004c
                                SC_P_COMP_CTL_GPIO_1V8_3V3_GPIORHB_PAD      0x000514a0
                        >;
                };
                pinctrl_csi0_lpi2c0: csi0lpi2c0grp {
                        fsl,pins = <
                                SC_P_MIPI_CSI0_I2C0_SCL_MIPI_CSI0_I2C0_SCL      0xc2000020
                                SC_P_MIPI_CSI0_I2C0_SDA_MIPI_CSI0_I2C0_SDA      0xc2000020
                        >;
                };
                pinctrl_esai0: esai0grp {
                        fsl,pins = <
…
                        >;
                };
                pinctrl_lpuart0: lpuart0grp {
                        fsl,pins = <
                                SC_P_UART0_RX_ADMA_UART0_RX    0x06000020
                                SC_P_UART0_TX_ADMA_UART0_TX    0x06000020
                        >;
                };
                pinctrl_lpuart1: lpuart1grp {
                        fsl,pins = <
```

```
…
                                >;
                        };
                        pinctrl_lpuart2: lpuart2grp {
                                fsl,pins = <
                                        SC_P_UART2_TX_ADMA_UART2_TX     0x06000020
                                        SC_P_UART2_RX_ADMA_UART2_RX     0x06000020
                                >;
                        };
                        pinctrl_lpuart3: lpuart3grp {
                                fsl,pins = <
…
                                >;
                        };
                        pinctrl_fec1: fec1grp {
                                fsl,pins = <
…
                                >;
                        };
                        pinctrl_fec2: fec2grp {
                                fsl,pins = <
…
                                >;
                        };
                        pinctrl_flexcan1: flexcan0grp {
                                fsl,pins = <
                                        SC_P_FLEXCAN0_TX_ADMA_FLEXCAN0_TX               0x21
                                        SC_P_FLEXCAN0_RX_ADMA_FLEXCAN0_RX               0x21
                                >;
                        };
                        pinctrl_flexcan2: flexcan1grp {
                                fsl,pins = <
                                        SC_P_FLEXCAN1_TX_ADMA_FLEXCAN1_TX               0x21
                                        SC_P_FLEXCAN1_RX_ADMA_FLEXCAN1_RX               0x21
                                >;
                        };
```

```
		pinctrl_lcdif: lcdif_grp {
			fsl,pins = <
…
			>;
		};
		pinctrl_lcdif_pwm: lcdif_pwm_grp {
			fsl,pins = <
				SC_P_SPI0_CS1_ADMA_LCD_PWM0_OUT			0x00000060
			>;
		};
		pinctrl_flexspi0: flexspi0grp {
			fsl,pins = <
…
			>;
		};
		pinctrl_cm40_i2c: cm40i2cgrp {
			fsl,pins = <
				SC_P_ADC_IN1_M40_I2C0_SDA	0x0600004c
				SC_P_ADC_IN0_M40_I2C0_SCL	0x0600004c
			>;
		};
		pinctrl_ioexp_rst: ioexp_rst_grp {
			fsl,pins = <
				SC_P_SPI2_SDO_LSIO_GPIO1_IO01		0x06000021
			>;
		};
		pinctrl_ioexp_rst_sleep: ioexp_rst_sleep_grp {
			fsl,pins = <
				SC_P_SPI2_SDO_LSIO_GPIO1_IO01		0x07800021
			>;
		};
		pinctrl_pwm_mipi_lvds0: mipi_lvds0_pwm_grp {
			fsl,pins = <
				SC_P_MIPI_DSI0_GPIO0_00_MIPI_DSI0_PWM0_OUT	0x00000020
			>;
```

```
		};

		pinctrl_i2c0_mipi_lvds0: mipi_lvds0_i2c0_grp {
			fsl,pins = <
				SC_P_MIPI_DSI0_I2C0_SCL_MIPI_DSI0_I2C0_SCL	0xc6000020
				SC_P_MIPI_DSI0_I2C0_SDA_MIPI_DSI0_I2C0_SDA	0xc6000020
				SC_P_MIPI_DSI0_GPIO0_01_LSIO_GPIO1_IO28		0x00000020
			>;
		};
		pinctrl_pwm_mipi_lvds1: mipi_lvds1_pwm_grp {
			fsl,pins = <
				SC_P_MIPI_DSI1_GPIO0_00_MIPI_DSI1_PWM0_OUT	0x00000020
			>;
		};
		pinctrl_i2c0_mipi_lvds1: mipi_lvds1_i2c0_grp {
			fsl,pins = <
				SC_P_MIPI_DSI1_I2C0_SCL_MIPI_DSI1_I2C0_SCL	0xc6000020
				SC_P_MIPI_DSI1_I2C0_SDA_MIPI_DSI1_I2C0_SDA	0xc6000020
				SC_P_MIPI_DSI1_GPIO0_01_LSIO_GPIO2_IO00		0x00000020
			>;
		};
		pinctrl_isl29023: isl29023grp {
			fsl,pins = <
				SC_P_SPI2_SDI_LSIO_GPIO1_IO02	0x00000021
			>;
		};
		pinctrl_lpi2c1: lpi1cgrp {
			fsl,pins = <
				SC_P_USB_SS3_TC1_ADMA_I2C1_SCL	0x06000021
				SC_P_USB_SS3_TC3_ADMA_I2C1_SDA	0x06000021
			>;
		};
		pinctrl_sai1: sai1grp {
			fsl,pins = <
…
			>;
```

```
		};

		pinctrl_usdhc1: usdhc1grp {
			fsl,pins = <
…
			>;
		};
		pinctrl_typec: typecgrp {
			fsl,pins = <
				SC_P_ENET0_REFCLK_125M_25M_LSIO_GPIO5_IO09	0x60
				SC_P_SPI2_SCK_LSIO_GPIO1_IO03			0x06000021
			>;
		};
		pinctrl_usdhc1_100mhz:
		pinctrl_usdhc1_200mhz:
		pinctrl_usdhc2_gpio: usdhc2gpiogrp {
			fsl,pins = <
				SC_P_USDHC1_RESET_B_LSIO_GPIO4_IO19	0x00000021
				SC_P_USDHC1_WP_LSIO_GPIO4_IO21		0x00000021
				SC_P_USDHC1_CD_B_LSIO_GPIO4_IO22	0x00000021
			>;
		};
		pinctrl_usdhc2: usdhc2grp {
			fsl,pins = <
…
			>;
		};
		pinctrl_usdhc2_100mhz:
		pinctrl_usdhc2_200mhz:
		pinctrl_pcieb: pcieagrp{
			fsl,pins = <
				SC_P_PCIE_CTRL0_PERST_B_LSIO_GPIO4_IO00			0x06000021
				SC_P_PCIE_CTRL0_CLKREQ_B_LSIO_GPIO4_IO01	0x06000021
				SC_P_PCIE_CTRL0_WAKE_B_LSIO_GPIO4_IO02			0x04000021
			>;
		};
```

```
		pinctrl_mipi_csi0_gpio: mipicsi0gpiogrp{
			fsl,pins = <
				SC_P_MIPI_CSI0_GPIO0_00_MIPI_CSI0_GPIO0_IO00	0x00000021
				SC_P_MIPI_CSI0_GPIO0_01_MIPI_CSI0_GPIO0_IO01	0x00000021
			>;
		};
		pinctrl_gpio3: gpio3grp{
			fsl,pins = <
				SC_P_MIPI_CSI0_GPIO0_01_LSIO_GPIO3_IO07		0xC0000041
				SC_P_MIPI_CSI0_GPIO0_00_LSIO_GPIO3_IO08		0xC0000041
			>;
		};
		pinctrl_wifi: wifigrp{
			fsl,pins = <
				SC_P_SCU_BOOT_MODE3_SCU_DSC_RTC_CLOCK_OUTPUT_32K	0x20
			>;
		};
		pinctrl_wifi_init: wifi_initgrp{
			fsl,pins = <
				SC_P_SCU_BOOT_MODE3_SCU_DSC_BOOT_MODE3		0x20
			>;
		};
		pinctrl_parallel_csi: parallelcsigrp {
			fsl,pins = <
…
			>;
		};
	};
};
```

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:

```
SC_P_MCLK_OUT0_ADMA_ACM_MCLK_OUT0		0x0600004c
#define SC_P_MCLK_OUT0_ADMA_ACM_MCLK_OUT0		SC_P_MCLK_OUT0		0
```

```
#define SC_P_MCLK_OUT0                    76   /* ADMA.ACM.MCLK_OUT0, ADMA.ESAI0.TX_HF_CLK,
ADMA.LCDIF.CLK, ADMA.SPI2.SDO, LSIO.GPIO0.IO20 */
```

其中 SC_P_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 mux_mode	mux_mode mux_mode 000b - ADMA_ACM_MCLK_OUT0 001b - ADMA_ESAI0_TX_HF_CLK 010b - ADMA_LCDIF_CLK 011b - ADMA_SPI2_SDO 100b - LSIO_GPIO0_IO20

IOpad 对应寄存器定义为：

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

=0b11: INOUT.

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

=0b10: pull down

0 PDRV	Drive Drive 0b - Output is configured in High Drive mode both in 1.8 V and 3.3 V applications 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：

```
SC_P_USDHC1_RESET_B_LSIO_GPIO4_IO19 0x00000021
```

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

=0b01: pull up

0 PDRV	Drive Drive 0b - Output is configured in High Drive mode both in 1.8 V and 3.3 V applications 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	i.mx6q/dl pin controller 的 low level driver。
pinctrl-imx.c	

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： a、设置引脚复用功能

	b、配置引脚的电气特性 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

6.2.2 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

6.2.2.1 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.

6.2.2.2 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;
}
```

6.3 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\fsl-imx8qxp-mek.dtsi

```
	brcmfmac: brcmfmac { //disable wifi/bt/modem
status = "disabled";
	};
	modem_reset: modem-reset { status = "disabled";};

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: sound {//remove all sound device
```

```
status = "disabled"
}
sound-amix-sai { status = "disabled"}
sound-cs42888 { status = "disabled"}
…
&acm {status = "disabled"};
&amix {status = "disabled"};
&asrc0{status = "disabled"};
&esai0 {status = "disabled"};
&sai4{status = "disabled"};;
&sai5 {status = "disabled"};
..
&sai1 { status = "disabled"};

lvds_backlight0/1: lvds_backlight@0/1 {//remove all the pwm backligh
status = "disabled"
}
lcdif_backlight: lcdif_backlight { status = "disabled"}
…
&pwm_mipi_lvds1  { 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"}
//fec 去掉
&fec1/2 { status = "disabled"}
//can 去掉
&flexcan1/2 { status = "disabled"}
//spi 去掉
&flexspi0 { status = "disabled"}
//i2c 去掉
&i2c0_cm40 { status = "disabled"}
//pcie 去掉
&pcieb{ status = "disabled"}
//camera 去掉
```

```
&mipi_csi_0 { status = "disabled"}
&cameradev { status = "disabled"}
&parallel_csi { status = "disabled"}
&isi_0/1/2/3 { status = "disabled"}
&i2c0_csi0 { status = "disabled"
  max9286_mipi@6A          { status = "disabled"}}
//显示接口去掉
&i2c0_mipi_lvds0/1 { status = "disabled"}
&ldb1/2_phy { status = "disabled"}
&ldb1/2 {status = "disabled"}
&mipi_dsi_phy1/2  {status = "disabled"};
&mipi_dsi1/2  {status = "disabled"}
&mipi_dsi_bridge1/2  {status = "disabled"}
```

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:

https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-8-processors/i.mx-software-and-development-tool:IMX-SW

Linux Binary Demo Files - i.MX 8QXPlus MEK

Unzip it。

2. download uuu 1.2.91 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\4.14.98_ga\L4.14.98_2.0.0_ga_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\4.14.98_ga\L4.14.98_2.0.0_ga_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
```

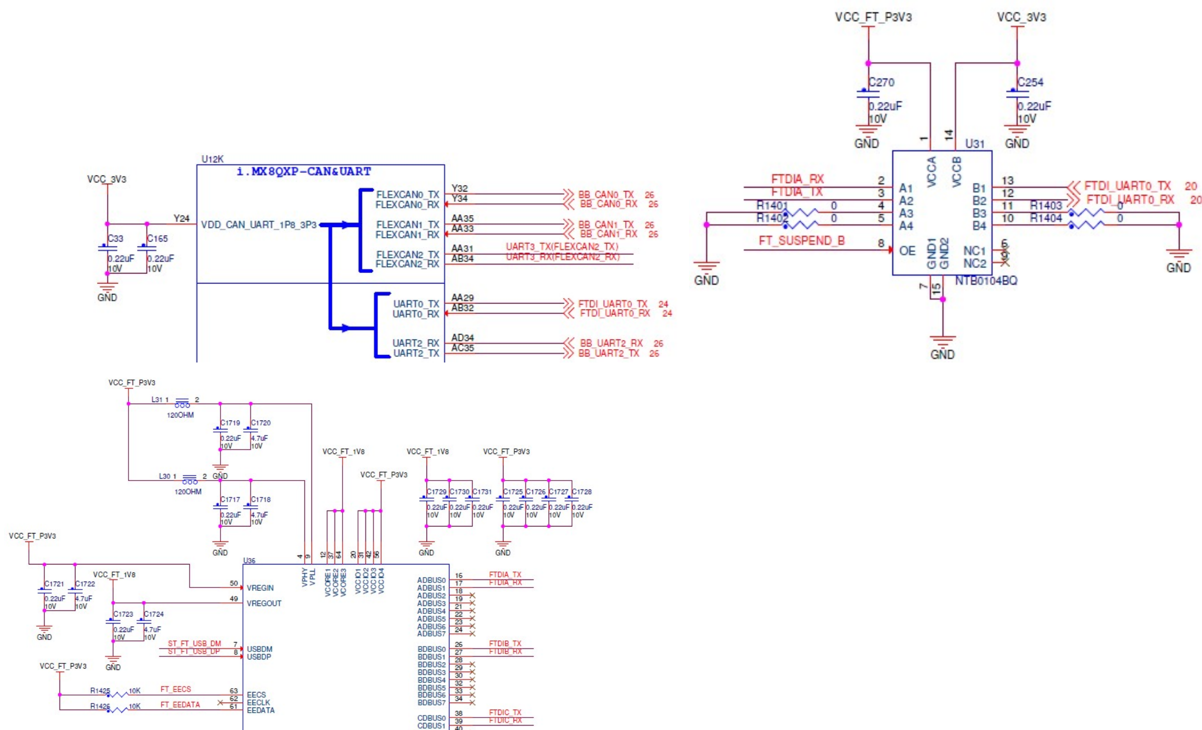
```
Initializing 'bootloader'
switch to partitions #1, OK
mmc0(part 1) is current device
Writing 'bootloader'

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
```

6.4 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。



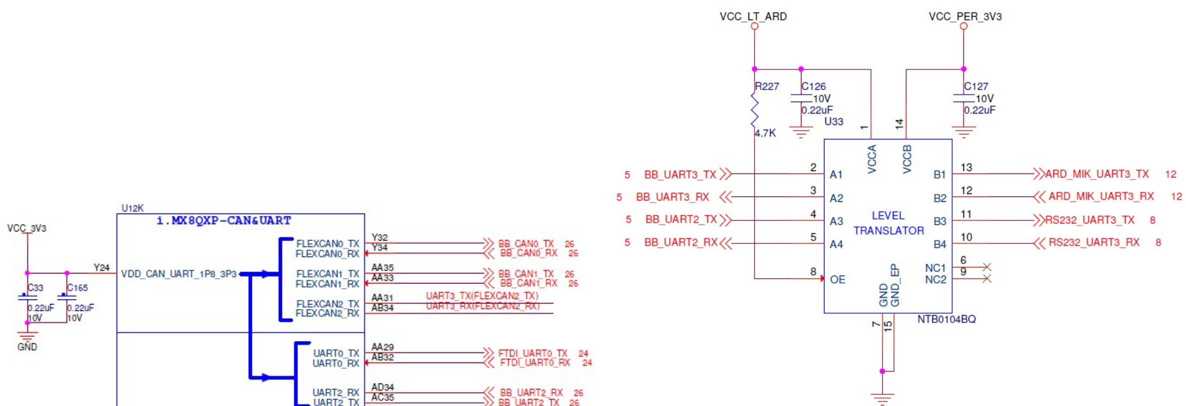
相关软件 DTB 设置为：

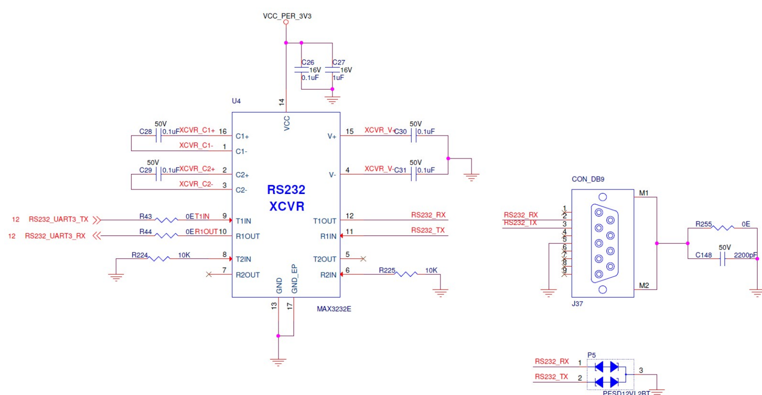
//arch\arm64\boot\dts\freescale\fsl-imx8qxp-mek.dtsi

```
chosen {
		bootargs = "console=ttyLP0,115200 earlycon=lpuart32,0x5a060000,115200";
		stdout-path = &lpuart0;
	};
…
		pinctrl_lpuart0: lpuart0grp {
			fsl,pins = <
				SC_P_UART0_RX_ADMA_UART0_RX	0x06000020
				SC_P_UART0_TX_ADMA_UART0_TX	0x06000020
			>;
		};
&pd_dma_lpuart0 {
	debug_console;
};
&lpuart0 {
	pinctrl-names = "default";
	pinctrl-0 = <&pinctrl_lpuart0>;
```

```
		status = "okay";
};
	//arch\arm64\boot\dts\freescale\fsl-imx8dx.dtsi
	lpuart0: serial@5a060000 {
		compatible = "fsl,imx8qm-lpuart";
		reg = <0x0 0x5a060000 0x0 0x1000>;
		interrupts = <GIC_SPI 345 IRQ_TYPE_LEVEL_HIGH>;
		interrupt-parent = <&wu>;
		clocks = <&clk IMX8QXP_UART0_CLK>,
			 <&clk IMX8QXP_UART0_IPG_CLK>;
		clock-names = "per", "ipg";
		assigned-clocks = <&clk IMX8QXP_UART0_CLK>;
		assigned-clock-rates = <80000000>;
		power-domains = <&pd_dma_lpuart0>;
		status = "disabled";
	};
…
			pd_dma_lpuart0: PD_DMA_UART0 {
				reg = <SC_R_UART_0>;
				#power-domain-cells = <0>;
				power-domains = <&pd_dma>;
				wakeup-irq = <345>;
			};
```

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\fsl-imx8qxp-mek.dtsi
chosen {
		bootargs = "console=ttyLP2,115200 earlycon=lpuart32,0x5a080000,115200";
		stdout-path = &lpuart2;
	};
…
		pinctrl_lpuart2: lpuart2grp {
			fsl,pins = <
				SC_P_UART2_TX_ADMA_UART2_TX	0x06000020
				SC_P_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\fsl-imx8dx.dtsi
	lpuart2: serial@5a080000 {
		compatible = "fsl,imx8qm-lpuart";
		reg = <0x0 0x5a080000 0x0 0x1000>;
		interrupts = <GIC_SPI 347 IRQ_TYPE_LEVEL_HIGH>;
```

```
                interrupt-parent = <&wu>;
                clocks = <&clk IMX8QXP_UART2_CLK>,
                        <&clk IMX8QXP_UART2_IPG_CLK>;
                clock-names = "per", "ipg";
                assigned-clocks = <&clk IMX8QXP_UART2_CLK>;
                assigned-clock-rates = <80000000>;
                power-domains =  <&pd_dma_lpuart2>; /* 调试串口去掉 DMA PM  <&pd_dma2_chan13>;*/
                /* dma-names = "tx","rx"; */ /*调试串口去掉 DMA 功能*/
                /* dmas = <&edma2 13 0 0>,
                        <&edma2 12 0 1>; */
                status = "disabled";
        };…
/*调试串口去掉 DMA PM*/
                        pd_dma_lpuart2: PD_DMA_UART2 {
                                reg = <SC_R_UART_2>;
                                #power-domain-cells = <0>;
                                power-domains = <&pd_dma>;
                        /*      #address-cells = <1>;
                                #size-cells = <0>; */
                                wakeup-irq = <347>;
/*
                                pd_dma2_chan12: PD_UART2_RX {
                                        reg = <SC_R_DMA_2_CH12>;
                                        power-domains =<&pd_dma_lpuart2>;
                                        #power-domain-cells = <0>;
                                        #address-cells = <1>;
                                        #size-cells = <0>;

                                        pd_dma2_chan13: PD_UART2_TX {
                                                reg = <SC_R_DMA_2_CH13>;
                                                power-domains =<&pd_dma2_chan12>;
                                                #power-domain-cells = <0>;
                                                #address-cells = <1>;
                                                #size-cells = <0>;
                                        };
```

```
                                    };
*/
                            };
```

如下将 UART0 改为普通串口：增加 DMA 功能：

```
        //arch\arm64\boot\dts\freescale\fsl-imx8qxp-mek.dtsi
/* 去掉 UART0 的 debug 串口功能，这个必须要修改
&pd_dma_lpuart0 {
        debug_console;
};
*/
        //arch\arm64\boot\dts\freescale\fsl-imx8dx.dtsi
        lpuart0: serial@5a060000 {
                compatible = "fsl,imx8qm-lpuart";
                reg = <0x0 0x5a060000 0x0 0x1000>;
                interrupts = <GIC_SPI 345 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-parent = <&wu>;
                clocks = <&clk IMX8QXP_UART0_CLK>,
                         <&clk IMX8QXP_UART0_IPG_CLK>;
                clock-names = "per", "ipg";
                assigned-clocks = <&clk IMX8QXP_UART0_CLK>;
                assigned-clock-rates = <80000000>;
                power-domains = <&pd_dma2_chan9>; /* <&pd_dma_lpuart0>; */
/*add dma support*/
dmas = <&edma2 9 0 0>,
              <&edma2 8 0 1>;
                status = "disabled";
        };
…
                        pd_dma_lpuart0: PD_DMA_UART0 {
                                reg = <SC_R_UART_0>;
                                #power-domain-cells = <0>;
                                power-domains = <&pd_dma>;
                                #address-cells = <1>;//增加 DMA 通道的 PM
                                #size-cells = <0>;
                                wakeup-irq = <345>;
```

//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

```
                    pd_dma2_chan8: PD_UART0_RX {
                    reg = <SC_R_DMA_2_CH8>;
                    power-domains =<&pd_dma_lpuart0>;
                    #power-domain-cells = <0>;
                    #address-cells = <1>;
                    #size-cells = <0>;
                    pd_dma2_chan9: PD_UART0_TX {
                        reg = <SC_R_DMA_2_CH9>;
                        power-domains =<&pd_dma2_chan8>;
                        #power-domain-cells = <0>;
                        #address-cells = <1>;
                        #size-cells = <0>;
                    };
                };
            };
```

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

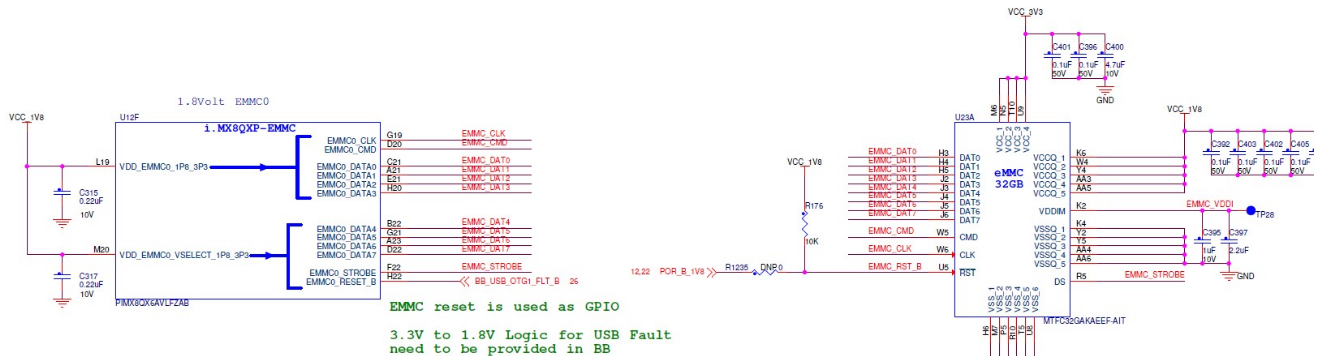
console=ttyLP2,115200 earlycon=lpuart32,0x5a080000,115200 root=/dev/mmcblk1p2 rootwait rw

可以操作 ttyLP0 如下：

echo aaaaa > /dev/ttyLP0

6.5 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：

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

static struct esdhc_soc_data usdhc_imx8qm_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 4.14.98 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/fsl-imx8qxp-mek.dtsi
        pinctrl_usdhc1_100mhz/_200mhz: usdhc1grp100mhz /200mhz {
            fsl,pins = <
                …
            >;
        };

        pinctrl_usdhc2_gpio: usdhc2gpiogrp {
            fsl,pins = <
                SC_P_USDHC1_RESET_B_LSIO_GPIO4_IO19     0x00000021
                SC_P_USDHC1_WP_LSIO_GPIO4_IO21          0x00000021
                SC_P_USDHC1_CD_B_LSIO_GPIO4_IO22        0x00000021
            >;
        };

        pinctrl_usdhc2/_100mhz/_200mhz: usdhc2grp/100mhz/200mhz {
            fsl,pins = <
                …
            >;
        };
```

//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_100mhz>, <&pinctrl_usdhc2_gpio>;
    pinctrl-2 = <&pinctrl_usdhc2_200mhz>, <&pinctrl_usdhc2_gpio>;
    bus-width = <4>;//sdslot support 4bit
    cd-gpios = <&gpio4 22 GPIO_ACTIVE_LOW>; //cd gpio 需要确认 iomux 已经设置为 gpio
    wp-gpios = <&gpio4 21 GPIO_ACTIVE_HIGH>;//wp gpio 需要确认 iomux 已经设置为 gpio
    vmmc-supply = <&reg_usdhc2_vmmc>;//电源需要确认设置好了电压
    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_100mhz>;
    pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
    bus-width = <8>;
    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：

6.5.1 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.

6.5.2 对应源代码

Linux-4.14.98/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

6.5.3 MMC 驱动分层图

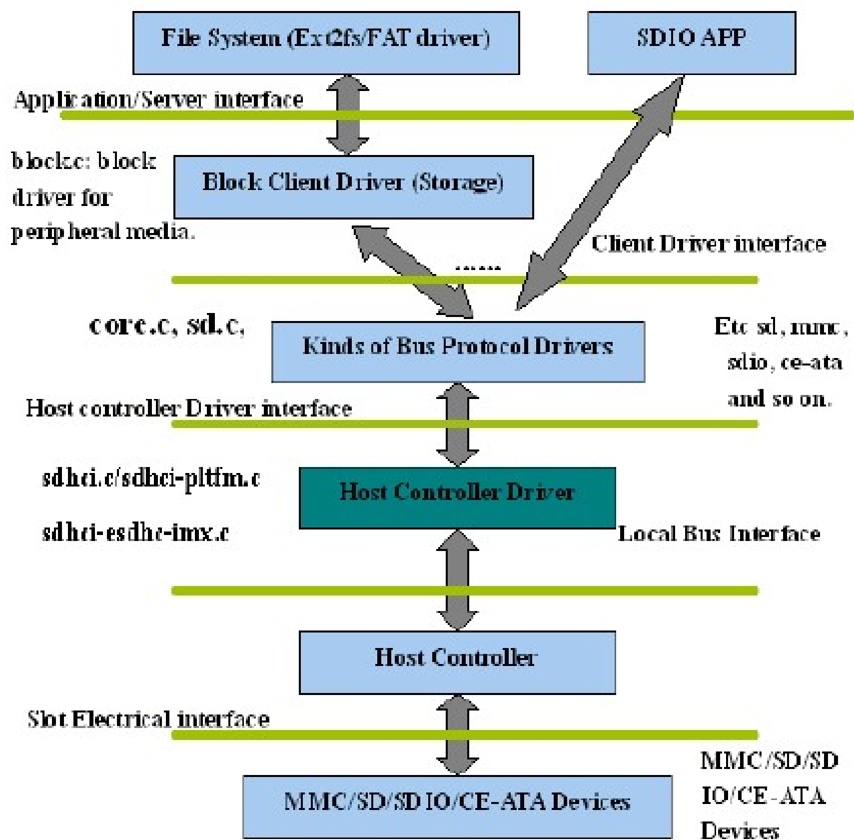


Figure 34-1. MMC Drivers Layering

6.5.4 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: usdhc@5b010000 {
        compatible = "fsl,imx8qm-usdhc", "fsl,imx6sl-usdhc";
        interrupt-parent = <&gic>;
        interrupts = <GIC_SPI 232 IRQ_TYPE_LEVEL_HIGH>;
        reg = <0x0 0x5b010000 0x0 0x10000>;
        clocks = <&clk IMX8QXP_SDHC0_IPG_CLK>,
                <&clk IMX8QXP_SDHC0_CLK>,
                <&clk IMX8QXP_CLK_DUMMY>;
```

```
                clock-names = "ipg", "per", "ahb";
                assigned-clocks = <&clk IMX8QXP_SDHC0_SEL>, <&clk IMX8QXP_SDHC0_DIV>;
                assigned-clock-parents = <&clk IMX8QXP_CONN_PLL1_CLK>;
                assigned-clock-rates = <0>, <400000000>;
                power-domains = <&pd_conn_sdch0>;
                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_100mhz>;
        pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
        bus-width = <8>;
        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-4.14.98\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
```

6.6 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/fsl-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 'fsl-imx8qxp-mek-jdi-wuxga-lvds0-panel.dtb'
=> sav
=> pri
fdt_file=fsl-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：

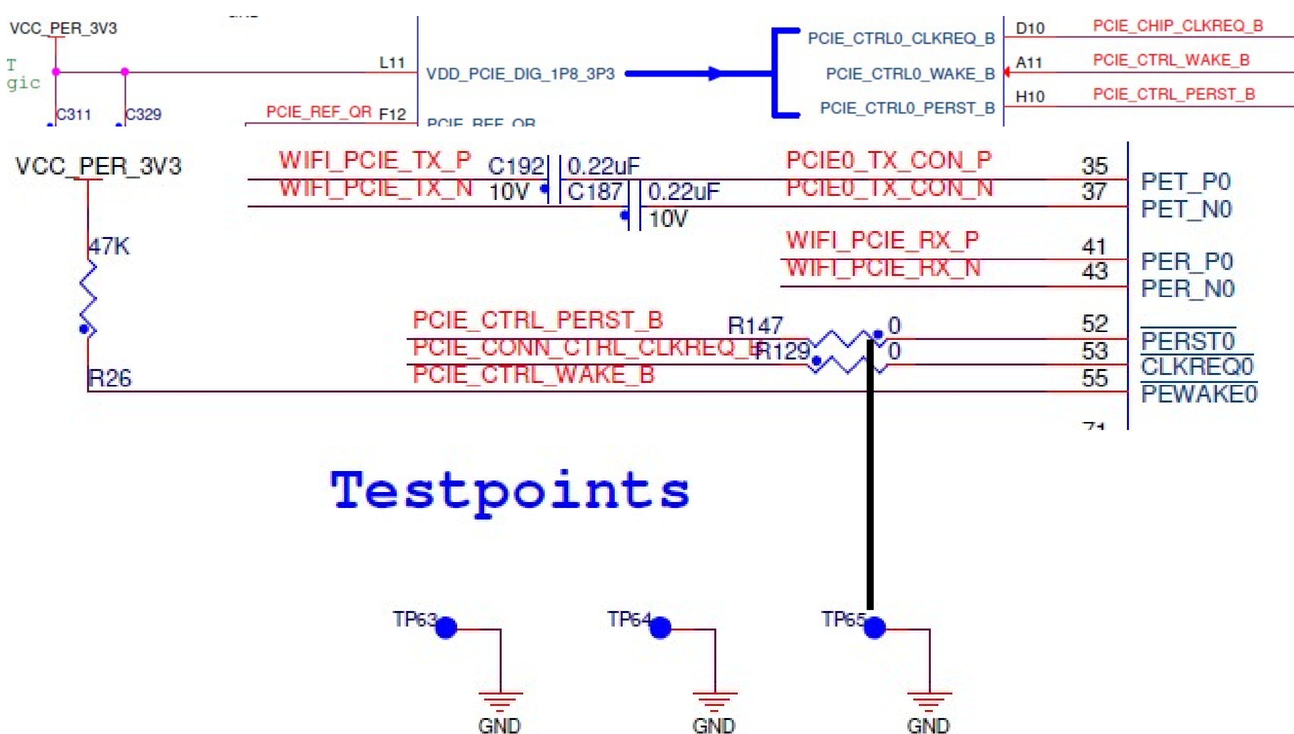
```
drivers/gpu/drm/panel/panel-simple.c
static const struct display_timing jdi_tx26d202vm0bwa_timing = {
	.pixelclock = { 104604000, 104604000, 104604000 },//pixel clock=52.3MhzX2=(1920+200+180)X(720+30+8)X60=104604000
	.hactive = { 1920, 1920, 1920 }, //horizontal effective time =960X2=1920
	.hfront_porch = { 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：

6.7 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 = <
                            SC_P_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。

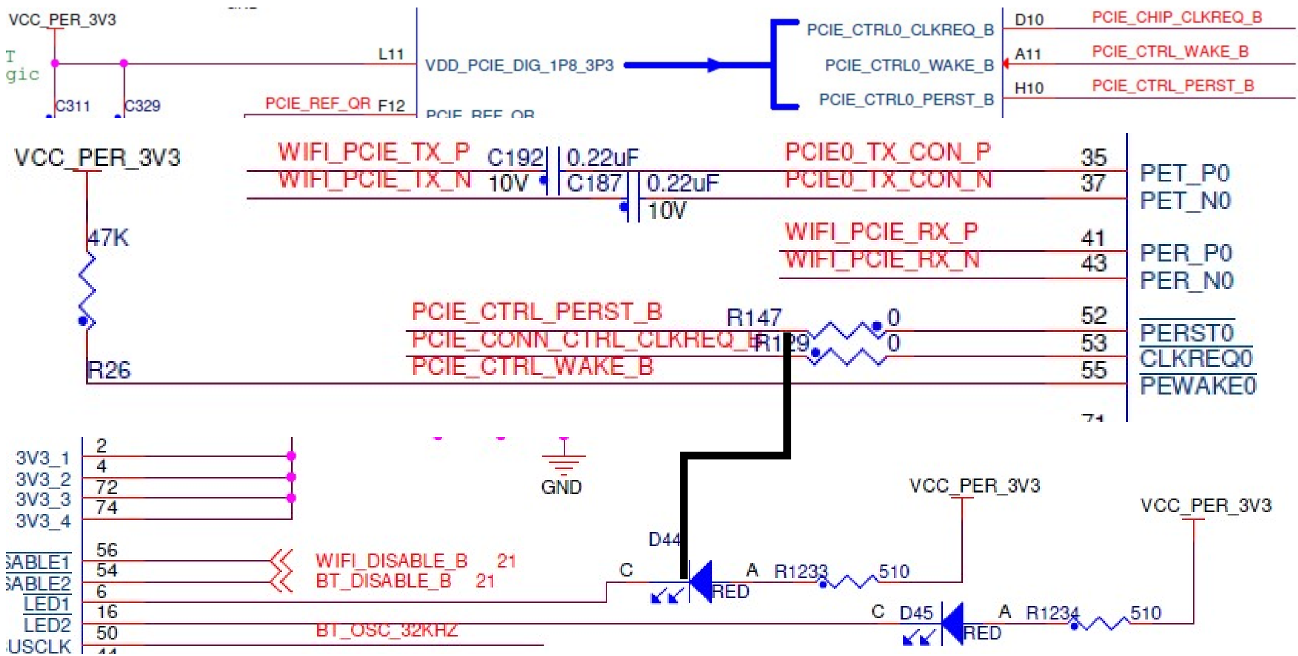
- 测试命令

```
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
    …
```

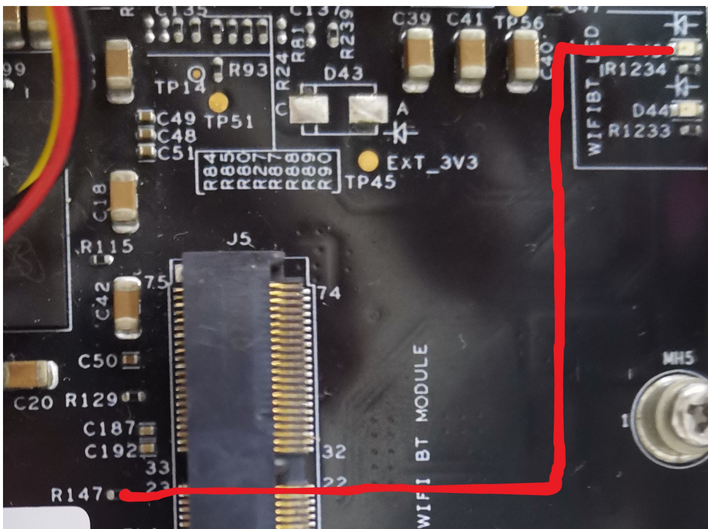
6.8 GPIO_LED 驱动定制

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 = <
				SC_P_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-ctrlllock 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-ctrlllock 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。

6.9 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])/ 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.

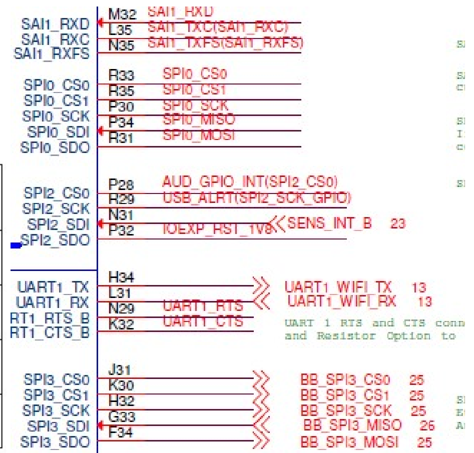
6.10 SPI 与 SPI Slave 驱动

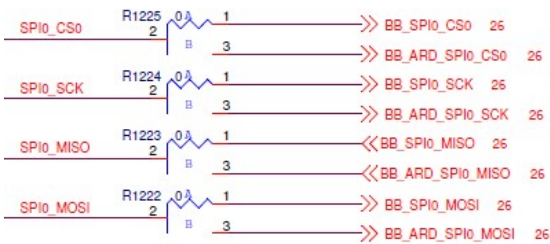
i.MX8QXP MEK 板 BSP 默认没有打开 SPI 接口，事实上在汽车的应用中，是有可能使用 SPI 接口来连接外设的，甚至有可能是 MCU 做主通过 SPI 接口来访问 i.MX8X,所以 i.MX8X 是做从的。所以如下说明在 i.MX8QXP MEK 板上如何集成 SPI Master 和 SPI Slave 的。

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)。

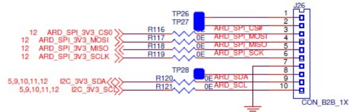
SPI TABLE

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



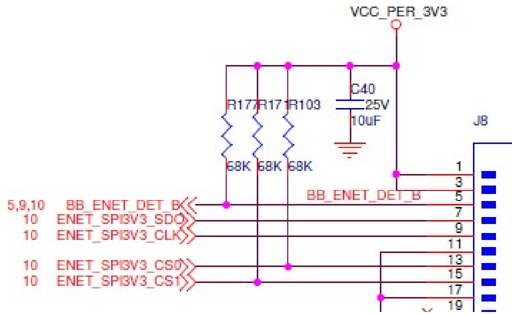


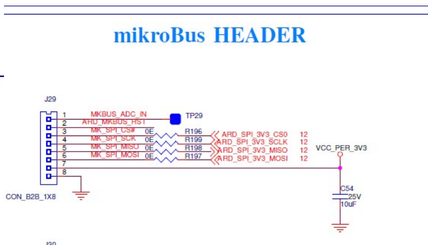
SPI0 is muxed with Audio In and Arduino Bus. The resistor options will be swapped only if the Arduino Bus is needed.



The connectors used are 2.54mm sockets

mikroBus HEADER





软件上可以参考 fsl-imx8qxp-lpddr4-arm2-lpspi.dts/fsl-imx8qxp-lpddr4-arm2-lpspi-slave.dts，如何集成 SPI:

```
arch/arm64/boot/dts/freescale/fsl-imx8qxp-mek.dtsi
pinctrl_lpspi0: lpspi0grp {
        fsl,pins = <
                SC_P_SPI0_SCK_ADMA_SPI0_SCK        0x0600004c
                SC_P_SPI0_SDO_ADMA_SPI0_SDO        0x0600004c
                SC_P_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 = <
				SC_P_SPI0_CS0_LSIO_GPIO1_IO08		0x21
			>;
		};

		pinctrl_lpspi3: lpspi3grp {
			fsl,pins = <
				SC_P_SPI3_SCK_ADMA_SPI3_SCK		0x0600004c
				SC_P_SPI3_SDO_ADMA_SPI3_SDO		0x0600004c
				SC_P_SPI3_SDI_ADMA_SPI3_SDI		0x0600004c
				SC_P_SPI3_CS0_ADMA_SPI3_CS0		0x0600004c
			>;
		};
…
&lpspi0 {
	#address-cells = <1>;
	#size-cells = <0>;
	fsl,spi-num-chipselects = <1>;
	pinctrl-names = "default";
	pinctrl-0 = <&pinctrl_lpspi0 &pinctrl_lpspi0_cs>;
	cs-gpios = <&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>;
	};
};

&lpspi3 {
	#address-cells = <1>;
```

```
		#size-cells = <0>;
		fsl,spi-num-chipselects = <1>;
		pinctrl-names = "default";
		pinctrl-0 = <&pinctrl_lpspi3>;
		spi-slave;
		status = "okay";
};
```

//注意目前 4.14.98 BSP 的 SPI SOC 层 DTS 并不完整，需要参考已经有的 SPI 接口的设置和芯片手册，自行添加：

arch/arm64/boot/dts/freescale/fsl-imx8dx.dtsi

5A03_0000	5A03_FFFF	64KB	LPSPI3
5A02_0000	5A02_FFFF	64KB	LPSPI2
5A01_0000	5A01_FFFF	64KB	LPSPI1
5A00_0000	5A00_FFFF	64KB	LPSPI0

```
lpspi3: lpspi@5a030000 { //寄存器值
		compatible = "fsl,imx7ulp-spi";
		reg = <0x0 0x5a030000 0x0 0x10000>;
```

			transmit
251	219	SPI3_INT	Following interrupts are grouped on this line • SPI#3 interrupt • DMA#2 interrupt#6 – SPI receive • DMA#2 interrupt#7 – SPI transmit
252	220	I2C0_INT	Following interrupts are

```
		interrupts = <GIC_SPI 219 IRQ_TYPE_LEVEL_HIGH>;  //中断号
		interrupt-parent = <&gic>;
		clocks = <&clk IMX8QXP_SPI3_CLK>,
			<&clk IMX8QXP_SPI3_IPG_CLK>;
		clock-names = "per", "ipg";
		assigned-clocks = <&clk IMX8QXP_SPI3_CLK>;
		assigned-clock-rates = <20000000>;
		power-domains = <&pd_dma2_chan7>; //johnli details as follows DMA channel
		dma-names = "tx","rx";
```

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

```
        dmas = <&edma2 7 0 0>, <&edma2 6 0 1>;
        status = "disabled";
    };

            pd_dma_lpspi3: PD_DMA_SPI_3 { //PD
                reg = <SC_R_SPI_3>;
                #power-domain-cells = <0>;
                power-domains = <&pd_dma>;
                wakeup-irq = <339>;
//johnli add
                pd_dma2_chan6: PD_LPSPI3_RX {
                    reg = <SC_R_DMA_2_CH6>;
                    power-domains =<&pd_dma_lpspi3>;
                    #power-domain-cells = <0>;
                    #address-cells = <1>;
                    #size-cells = <0>;

                    pd_dma2_chan7: PD_LPSPI3_TX {
                        reg = <SC_R_DMA_2_CH7>;
                        power-domains =<&pd_dma2_chan6>;
                        #power-domain-cells = <0>;
                        #address-cells = <1>;
                        #size-cells = <0>;
                    };
                };
//end
```

```
        };
```

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.
-		 */
-		temp |= TCR_CONT;
-		if (is_first_xfer)
-			temp &= ~TCR_CONTC;
-		else
-			temp |= TCR_CONTC;
 	} else {
 		temp |= fsl_lpspi->config.bpw - 1;
 		temp |= (fsl_lpspi->config.mode & 0x3) << 30;
```

- 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。

6.11 USB 3.0 TypeC 改成 USB 3.0 TypeA(未验证)

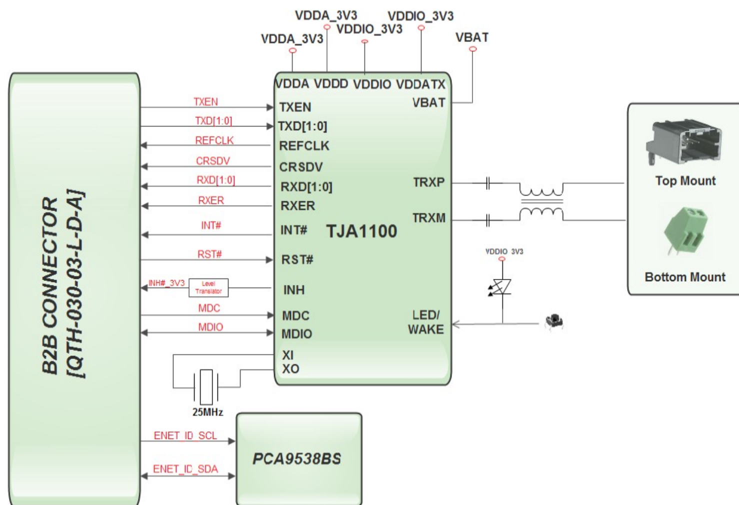
在我们 i.MX8QXP MEK 开发板设计中。USB_OTG1 是一个 USB2.0,连接到底板上的 USB OTG 口上，而 USB_OTG2/SS3 是一个 USB3.0 口，是连接到一个 Type C 接口上的，而在大多数汽车产品设计中，可能不会使用 Type C 接口中，而还是连接为 Type A 接口的。所以如下软件修改：

```
Arch/arm64/boot/dts/freescale/fsl-imx8qxp-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。

6.12 汽车级以太网驱动定制

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 fsl-imx8qxp-mek.dtsi/ fsl-imx8qxp-enet2-tja1100.dtsi/fsl-imx8qxp-mek-enet2-tja1100.dts/
fsl-imx8qxp-mek-enet2.dts 中得到：
//iomux 冲突，去掉
&fec1 {
    status = "disabled";
};
&esai0 {
    status = "disabled";
};
&fec2 {
    pinctrl-0 = <&pinctrl_fec2_rmii>;
    clocks = <&clk IMX8QXP_ENET1_IPG_CLK>, \\ 100Mhz 网的 clock tree
             …<&clk IMX8QXP_ENET1_TX_CLK>;
    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
  fsl,magic-packet; //以下两项使用 rmii 应该不起作用
    fsl,rgmii_rxc_dly;
  status = "okay";
    mdio {
            #address-cells = <1>;
            #size-cells = <0>;
            ethphy2: ethernet-phy@5 { //phy_addr=5
```

```
                compatible = "ethernet-phy-ieee802.3-c22";
                reg = <5>;
                tja110x,refclk_in;
            };
    };
};

&iomuxc {
    imx8qxp-mek {
            pinctrl_fec2_rmii: fec2rmiigrp {
                fsl,pins = <
                        SC_P_ENET0_MDC_CONN_ENET1_MDC                   0x06000020
…//使用 RMII 接口 PIN，收发各两根数据线
                        SC_P_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：

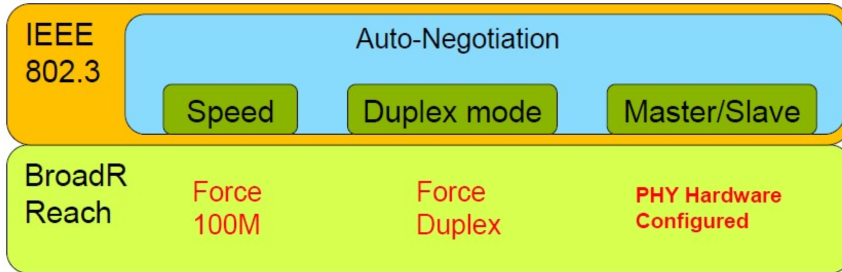
1. drivers/net/phy/tja110x.h 中定义了：#define NXP_PHY_ID_TJA1102P1 (0x00000000U)，这个 TJA1102 是一个双路 PHY 芯片，但是他的第二个 PHY 的 ID 默认设置成了 0x00。这样会导致一个问题，就是如果因为某些硬件原因导致了 mdio 访问 PHY 失败，get_phy_id 函数会返回 phy_id=0x00。这个时候理论上应该是找不到 PHY，从而不能加载相应 PHY 驱动，但是因为 TJA1102P1 的 PHY ID 刚好设置成了 0x00。所以错误的加载了此驱动：

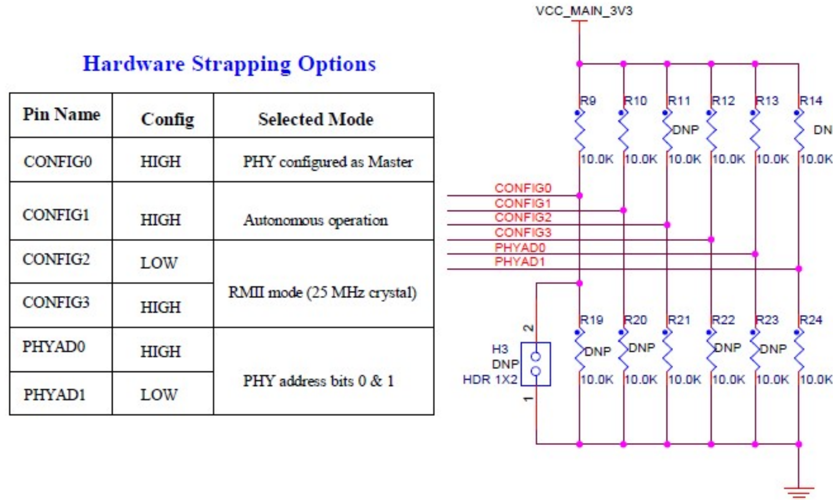
```
24.045570] TJA1102_p1 5b050000.ethernet-1:02: power mode transition failed
[   24.052818] TJA1102_p1 5b050000.ethernet-1:02: attached PHY driver [TJA1102_p1] (mii_bus:phy_addr=5b050000.ethernet-1:02, irq=POLL)
```

   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.

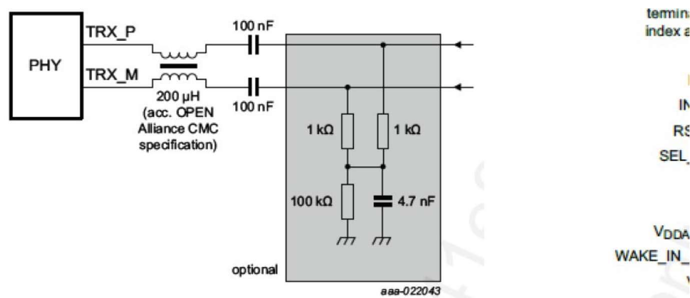




Hardware Strapping Options

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 = <
..
                //SC_P_ENET0_MDC_CONN_ENET1_MDC                 0x06000020
                //SC_P_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。

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 = <
            SC_P_MIPI_CSI0_GPIO0_00_MIPI_CSI0_GPIO0_IO00          0x21 //reset 使用此GPIO
来连接PHY的Reset管脚。
            SC_P_MIPI_CSI0_I2C0_SDA_LSIO_GPIO3_IO06               0x21 //power 使用此GPIO
来给PHY芯片的3.3V供电
                SC_P_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB0_PAD 0x000014a0
                SC_P_COMP_CTL_GPIO_1V8_3V3_ENET_ENETB1_PAD 0x000014a0
                SC_P_ENET0_MDC_CONN_ENET0_MDC                  0x06000020
….//此处为标准ENET0的IOMUX, 1G收发各四根数据线
                SC_P_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 32 Devices per Port
32 Registers per Port	64K Registers per Device
Operation of Clause 22 ● To Read a Clause 22 Register Perform:Read Register RRRRR from PHY AAAAA ● To Write a Clause 22 Register Perform:Write Register RRRRR to PHY AAAAA ● Each Operation Takes 1 Step	**Operation of Clause 45** ● To Read a Clause 45 Register Perform:Write Address AAAAAAAAAAAAAAAA to Device EEEEE on Port PPPPP; Read Register From Device EEEEE on Port PPPPP ● To Write a Clause 45 Register Perform:Write Address AAAAAAAAAAAAAAAA to Device EEEEE on Port PPPPP; Write Register To Device EEEEE on Port PPPPP ● 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

Only 13 & 14 are Left

Same

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:

Clause 22 Register 13

15 14	13 … 5	4 … 0
FN	Reserved	EEEEE

00 = Address Reg
01 = Data Reg (no post increment)
10 = Data Reg w/Post Increment on reads & writes
11 = Data Reg w/Post Increment on writes only

	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

Clause 22 Register 14

15 … 0
Clause 45 Address or Data

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 AAAAAAAAAAAAAAAA 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 10 = Data, post increment on reads and writes 01 = Data, no post increment 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
```

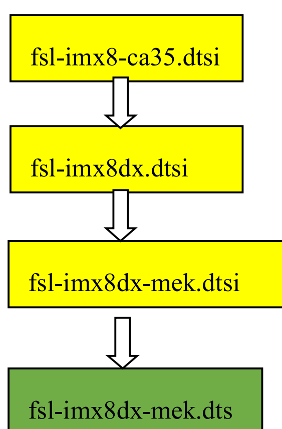
然后在 UUU 的 script 中增加以下命令就可以用 uboot 命令来烧写 MAC 地址了：

```
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
```

6.13 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 4.14.98 中没有 i.MX8DX MEK 板的 DTS 配置，需要自己开发，主要就是只使用两个 C35 ARM 核：

i.MX8DX MEK 的 DTS 可以设计为如下：



其中：fsl-imx8dx-mek.dtsi 可以从 fsl-imx8qxp-mek.dtsi 拷贝一份，然后头部修改为：

```
// #include "fsl-imx8qxp.dtsi"
#include "fsl-imx8dx.dtsi"
…
```

然后多余的驱动可以去掉。

fsl-imx8dx-mek.dts 比较简单，直接包含 fsl-imx8dx-mek.dtsi：

```
 #include "fsl-imx8dx-mek.dtsi"
```

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_FSL_IMX8QXP) += fsl-imx8qxp-lpddr4-arm2.dtb \
				fsl-imx8qxp-mek.dtb \
			fsl-imx8dx-mek.dtb \...
```

这样就不会初始化定义在\linux-imx\arch\arm64\boot\dts\freescale\fsl-imx8qxp.dtsi 中的两个 C35 核：

```
	cpus {
```

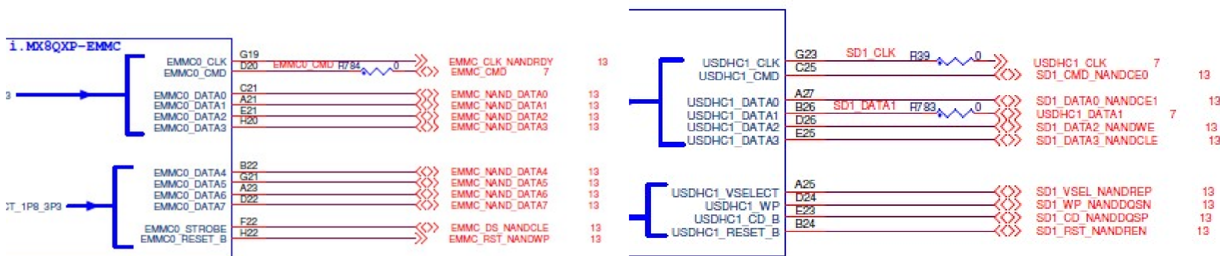
```
            A35_2: cpu@2 {
                ...
            };
            A35_3: cpu@3 {
…
            };
        };

        pmu {
            interrupt-affinity = <&A35_0>, <&A35_1>, <&A35_2>, <&A35_3>;
        };
```

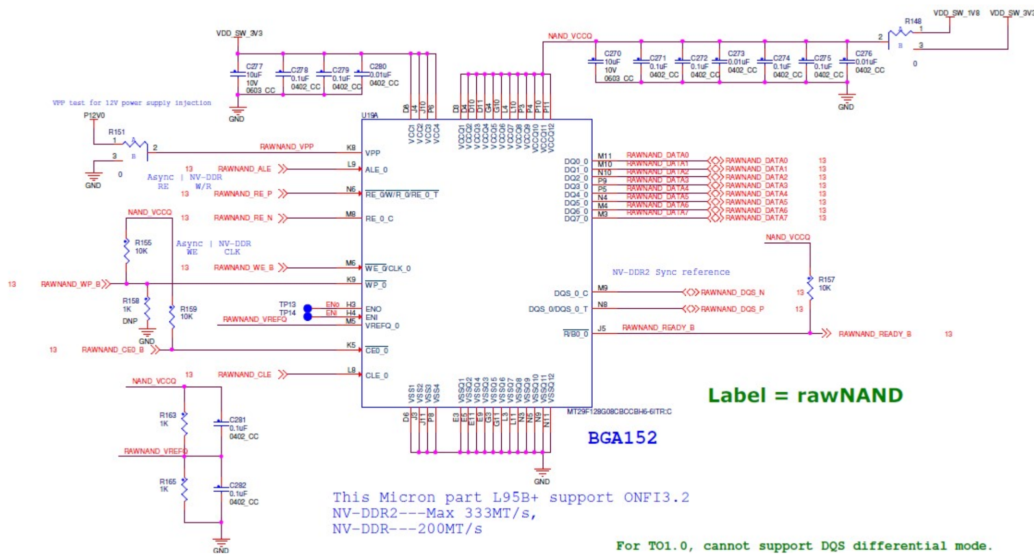
而只会使用定义在\linux-imx\arch\arm64\boot\dts\freescale\fsl-imx8dx.dtsi fsl-imx8-ca35.dtsi 中的两个 C35 核：

```
 /delete-node/ &A35_2;
 /delete-node/ &A35_3;
```

6.14 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：





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 = <
                        SC_P_EMMC0_CLK_CONN_NAND_READY_B    0x0e00004c
                        SC_P_EMMC0_DATA0_CONN_NAND_DATA00  0x0e00004c
                        …
                        SC_P_EMMC0_DATA7_CONN_NAND_DATA07  0x0e00004c
                        SC_P_EMMC0_STROBE_CONN_NAND_CLE         0x0e00004c
                        SC_P_EMMC0_RESET_B_CONN_NAND_WP_B  0x0e00004c

                        SC_P_USDHC1_DATA0_CONN_NAND_CE1_B   0x0e00004c
                        SC_P_USDHC1_DATA2_CONN_NAND_WE_B   0x0e00004c
                        SC_P_USDHC1_DATA3_CONN_NAND_ALE         0x0e00004c
                        SC_P_USDHC1_CMD_CONN_NAND_CE0_B         0x0e00004c

                        /* i.MX8QXP NAND use nand_re_dqs_pins */
                        SC_P_USDHC1_CD_B_CONN_NAND_DQS      0x0e00004c
                        SC_P_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 SC_P_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 PULL	Pull Down Pull Up Pull Down Pull Up 00b - Prohibited 01b - pull up 10b - pull down 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
```

freescale semiconductor

Document Number: ANnnnn
09/2013