NXP Semiconductor
Application Note

Document Number: ANnnnn
Rev. 6, 2/2020

i.MX8X 板级开发板 4.19.35 的 Bootloader 定制

by John Li (nxa08200)
GSM CAS
NXP Semiconductor, Inc.
Shanghai,

本文说明i.MX8X 板级开发包版本4.19.35的bootloader细节，以帮助客户了解i.MX8X的bootloader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-L4.19.35_1.1.0\
i.MX_Yocto_Project_User's_Guide.pdf
i.MX_Linux_User's_Guide.pdf。预先熟悉一下i.MX8X的编译环境，本文部分内容与之重复。

历史		
V1	● 创建本文	John.Li
V2	● 更新到 4.14.78_ga	John.Li
V3	● 主要更新第二章	John.Li
V4	● 更新到 4.14.98_ga ● 更新 SCU 与串口定制	John.Li
V5	● 添加 usb/nandflash 定制及其它	John.Li
V6	● 更新至 4.19.35	John.Li

目录

1 i.MX8X 板级开发包镜像结构

Linux	
flash.bin	Compound bootloader
Image	Linux kernel
fsl-imx8qxp-mek*.dtb	Device tree files
rootfs.tar.bz2	Rootfs

Compound bootloader	
scfw_tcm.bin	SCFW
bl31.bin	ATF
u-boot.bin	UBOOT
imx8qx_dcd.cfg	[Optional] DCD for DDR
scd.bin	[Optional] Image for SCD
csf*.bin	[Optional] Command Sequence File, for HAB
m4_image	[Optional] Image for M4 core

2 创建 i.MX8QXP Linux 4.19.35 板级开发包编译环境

2.1 下载板级开发包

i.MX8QXP_QM Linux 4.19.35 板级开发包下载地址 如下:

www.nxp.com/imx-> i.MX 8 Processors->*i.MX 8X->Tools&Software -> Embedded Software -> BSP,Drivers and Middelware ->i.MX Software and Development Tools

i.MX BSP Updates and Releases
Linux

Linux 4.19.35_1.1.0
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 L4.19.35_1.1.0 Documentation

可以首先下载：

1. Linux L4.19.35_1.1.0 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-1.2.7.1.tar.gz

i.MX8QXP MEK Linux Demo 镜像也可以下载。

2.2 创建 yocto 编译环境:

Ubuntu 18.04 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-L4.19.35_1.1.0_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-warrior

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp
repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-linux-warrior -
m imx-4.19.35-1.1.0.xml  --repo-url=https://mirrors.tuna.tsinghua.edu.cn/git/git-repo
repo sync
```

3. Yocto 编译，和烧写 SDcard 镜像：

编译配置命令如下：

```
DISTRO=<distro name> MACHINE= imx8qxpmek source fsl-setup-release.sh -b imx8qxpmek_"distro name"
```

Distro name 列表如下：

- fsl-imx-wayland - Wayland weston graphics.
- fsl-imx-xwayland - Wayland graphics and X11. X11 applications using EGL are not supported.

Bitbake image 与编译配置，以及建议使用功能，编译结果输出，烧写命令，SDK 编译与安装如下列表：

Image name	fsl-image-machine-test	imx-image-multimedia	imx-image-full
Suggested distro	fsl-imx-wayland		fsl-imx-xwayland
Target	An FSL Community i.MX core image with console environment - no GUI interface.	Builds an i.MX image with a GUI without any Qt content	Builds an opensource Qt 5 image with Machine Learningfeatures. These images are only supported for i.MX SoCwith hardware graphics.
Provided by layer	meta-freescale-distro	meta-fsl-bsp-release/imx/meta-sdk	
Application feature	Tbox 之类的无屏应用	Cluster&surround view 之类的不需要复杂 GUI 支持的	Infortainment 之类的需要复杂 GUI 如 QT 支持的
Build configs	DISTRO=fsl-imx-wayland MACHINE=imx8qxpmek source fsl-setup-release.sh -b imx8qxpmek_wayland		DISTRO=fsl-imx-xwayland MACHINE=imx8qxpmek

sample			source fsl-setup-release.sh -b imx8qxpmek xwayland
Bitbake sample	bitbake fsl-image-machine-test	bitbake imx-image-multimedia	bitbake imx-image-full
Output folder and image	/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/images/ imx8qxpmek/ fsl-image-machine-test -imx8qxpmek.sdcard.bz2 -> fsl-image- machine-test -imx8qxpmek -20200219011834.rootfs.wic.bz2 (sdcard 镜像，解压后可以直接烧写) fsl-image- machine-test -imx8qxpmek.tar.bz2 -> fsl-image-machine-test -imx8qxpmek -20200219011834.rootfs.tar.bz2 (rootfs 压缩包) u-boot.bin-> u-boot-sd-2019.04-r0.bin (uboot) imx-boot-tools/scfw-tcm.bin -> 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)	/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/images/ imx8qxpmek/ imx-image-multimedia -imx8qxpmek.sdcard.bz2 -> imx-image-multimedia-imx8qxpmek -20200218072756.rootfs.wic.bz2 (sdcard 镜像，解压后可以直接烧写) imx-image-multimedia -imx8qxpmek.tar.bz2 -> imx-image-multimedia-imx8qxpmek -20200218072756.rootfs.tar.bz2 (rootfs 压缩包) u-boot.bin-> u-boot-sd-2019.04-r0.bin (uboot) imx-boot-tools/scfw-tcm.bin -> 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)	/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/images/ imx8qxpmek/ imx-image-full -imx8qxpmek.sdcard.bz2 -> imx-image- full -imx8qxpmek -20200220071128.rootfs.wic.bz2 (sdcard 镜像，解压后可以直接烧写) imx-image- full -imx8qxpmek.tar.bz2 -> imx-image- full -imx8qxpmek -20200220071128.rootfs.tar.bz2 (rootfs 压缩包) u-boot.bin-> u-boot-sd-2019.04-r0.bin (uboot) imx-boot-tools/scfw-tcm.bin -> 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)
Burning SDcard image	bunzip2 -dk -f fsl-image- machine-test -imx8qxpmek.sdcard.bz2 sudo dd if= fsl-image- machine-test -imx8qxpmek.sdcard of=/dev/sd<partition> bs=1M	bunzip2 -dk -f imx-image-multimedia -imx8qxpmek.sdcard.bz2 sudo dd if= imx-image-multimedia -imx8qxpmek.sdcard of=/dev/sd<partition> bs=1M	bunzip2 -dk -f imx-image- full -imx8qxpmek.sdcard.bz2 sudo dd if= imx-image- full -imx8qxpmek.sdcard of=/dev/sd<partition> bs=1M conv=fsync

	conv=fsync	conv=fsync	
Compiling SDK	bitbake fsl-image-machine-test -c populate_sdk	bitbake imx-image-multimedia -c populate_sdk	bitbake imx-image-full -c populate_sdk
Compiling SDK output	~/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/sdk fsl-imx-wayland-glibc -x86_64-fsl-image-machine-test -aarch64-toolchain-4.19-warrior.sh	~/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/sdk fsl-imx-wayland-glibc -x86_64-fsl-image-multimedia -aarch64-toolchain-4.19-warrior.sh	~/imx-yocto-bsp/ imx8qxpmek_wayland/ tmp/deploy/sdk fsl-imx-wayland-glibc -x86_64-fsl-image-full -aarch64-toolchain-4.19-warrior.sh
Install SDK	创建 SDK 安装目录 pwd ~/imx-yocto-bsp/ imx8qxpmek_wayland mkdir sdk 到 SDK 输出目录下运行安装脚本 ./fsl-imx-wayland-glibc -x86_64-fsl-image-machine-test -aarch64-toolchain-4.19-warrior.sh 输入安装 sdk 的目标目录 (default: /opt/fsl-imx-wayland/4.19-warrior): ~/imx-yocto-bsp/ imx8qxpmek_wayland/sdk You are about to install the SDK to "~/imx-yocto-bsp/ imx8qxpmek_wayland/sdk". Proceed[Y/n]? Y	创建 SDK 安装目录 pwd ~/imx-yocto-bsp/ imx8qxpmek_wayland mkdir sdk 到 SDK 输出目录下运行安装脚本 ./ fsl-imx-wayland-glibc -x86_64-fsl-image-multimedia -aarch64-toolchain-4.19-warrior.sh 输入安装 sdk 的目标目录 (default: /opt/fsl-imx-wayland/4.19-warrior): ~/imx-yocto-bsp/ imx8qxpmek_wayland/sdk You are about to install the SDK to "~/imx-yocto-bsp/ imx8qxpmek_wayland/sdk". Proceed[Y/n]? Y	创建 SDK 安装目录 pwd ~/imx-yocto-bsp/ imx8qxpmek_xwayland mkdir sdk 到 SDK 输出目录下运行安装脚本 ./ fsl-imx-wayland-glibc -x86_64-fsl-image-full -aarch64-toolchain-4.19-warrior.sh 输入安装 sdk 的目标目录 (default: /opt/fsl-imx-wayland/ 4.19-warrior): ~/imx-yocto-bsp/ imx8qxpmek_xwayland/sdk You are about to install the SDK to "~/imx-yocto-bsp/ imx8qxpmek_wayland/sdk". Proceed[Y/n]? Y
Install SDK output	~/imx-yocto-bsp/imx8qxpmek_wayland/sdk/ ● environment-setup-aarch64-poky-linux ● sysroots		~/imx-yocto-bsp/ imx8qxpmek_xwayland/sdk/ ● environment-setup-aarch64-poky-linux ● sysroots

注意：

1. 中断后重新编译 命令：

   source setup-environment <build-dir>

2. 以上 yocto 编译测试环境为：

   - 编译主机与主机操作系统：

     处理器 Inte(R) Core(TM) i7-8650U CPU@1.90GHZ 2.11GHZ

     RAM 8.00GB

     System: 64 位操作系统

     Windows: Windows 10 企业版

     建议编译主机使用 SSD 硬盘，实现中发现使用 SSD 硬盘的主机编译 yocto 要快很多。

   - 虚拟机工具，版本，设置与 ubuntu 镜像版本：

     VMware® Workstation 15 Player 15.5.1 build-15018445

     虚拟机设置：

     内存： 4GB

     处理器 4

     硬盘 200GB

     网络适配器 NAT

     4.19.35 BSP 需要 python3 支持，所以需要用 ubuntu18.04(16.04 和 14.04 不支持 python3)，如下虚拟机系统：

     uname -a

     Linux ubuntu 5.3.0-40-generic #32~18.04.1-ubuntu SMP Mon Feb 3 14:05:58 UTC 2020 x86_64 x86_64 GNU/Linux

   - 测试网络环境：

     家用上海电信 1G 光纤宽带。

3. 编译问题解决：

   - 不支持 python3，如上所说，更新到 ubuntu1804，可以由低版本的升级，也可以直接安装 18.04 的 iso。
   - 编译主机如果有代理服务器设置，有可能出现编译 imx-image-full 时 tensorflow-lite 不通过的情况，解决方式如下：

     A:如果只需要 QT，并不需要 ML(machine learning)可以如下方法去掉 ML 支持：

     pwd

     ~/imx-yocto-bsp/imx8qxpmek_xwayland

```
vi conf/bblayers.conf
# BBLAYERS += " ${BSPDIR}/sources/meta-fsl-bsp-release/imx/meta-ml "
vi ../sources/meta-fsl-bsp-release/imx/meta-sdk/recipes-fsl/images/imx-image-full.bb
#ML_PKGS_mx8 = "packagegroup-imx-ml"
```

B:可以加上如下 PATCH 通过 tensorflow-lite 编译：

```
--- sources/meta-fsl-bsp-release/imx/meta-ml/recipes-libraries/tensorflow-lite/tensorflow-lite_1.13.2.bb
        2020-02-20 16:49:41.320540594 +0800
+++ sources/meta-fsl-bsp-release/imx/meta-ml/recipes-libraries/tensorflow-lite/tensorflow-lite_1.13.2.bb.modified
        2020-02-20 17:08:23.831918603 +0800
@@ -16,6 +16,13 @@
 EXTRA_OEMAKE = "'TARGET_ARCH=${TARGET_ARCH}' 'CXX=${CXX}' 'CC=${CC}' 'AR=${AR}'
'CXXFLAGS=${CXXFLAGS}'"

 do_configure(){
+
+       export HTTP_PROXY=${http_proxy}
+       export HTTPS_PROXY=${https_proxy}
+       export http_proxy=${http_proxy}
+       export https_proxy=${https_proxy}
+
+

         ${S}/tensorflow/lite/tools/make/download_dependencies.sh
```

C: 没有代理服务器设置的编译主机预计测试可以通过(待测试)。如果编译过程中出现 do_fetch 失败，请多试几次 bitbake，最终可以成功(imx-gpu-viv-6.4.0.p1.0-aarch64.bin 本次测试中经常会出现 do_fetch 失败)。

也可以根据文档《i.MX_Linux_Release_Notes.pdf》：

wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/<package file name>：

imx-gpu-viv-6.4.0.p1.0-aarch64.bin：i.MX Graphics libraries for i.MX 8.

说明，执行命令：

~/imx-yocto-bsp/imx8qxpmek_xwayland/tmp/work/aarch64-mx8-poky-linux/imx-gpu-viv/1_6.4.0.p1.0-aarch64-r0/

wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/ imx-gpu-viv-6.4.0.p1.0-aarch64.bin .

就可以快速下载下 gpu 安装包。

4. 总共时长：

编译时长要根据编译主机配置，网络情况和 git 服务器的情况而定，一般是从几个小时到几天都有可能，建议尽量使用代理服务器。

2.3 独立编译

1. 编译 SCfirmware

打开一个终端：

```
tar xzvf  imx-scfw-porting-kit-1.2.7.1.tar.gz    (untar scfw tools kit)
 |->packages
 |     |-> imx-scfw-porting-kit-1.2.7.1.bin
chmod a+x ./ imx-scfw-porting-kit-1.2.7.1.bin
./ imx-scfw-porting-kit-1.2.7.1.bin   (run the bin file to install scfw tools kit)
cd imx-scfw-porting-kit-1.2.7.1/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-1.2.7.1
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-1.2.7.1/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
Assembling platform/board/board.S
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_4.19.
 ....
 rel_imx_4.19.35_1.1.0
 ...
git checkout rel_imx_4.19.35_1.1.0
git status
HEAD detached at rel_imx_4.19.35_1.1.0
nothing to commit, working directory clean

ls configs |grep imx8qxp
 …

 imx8qxp_mek_defconfig
…
source ~ /imx-yocto-bsp/imx8qxpmek_wayland/sdk/environment-setup-aarch64-poky-linux (sdk 安装地址)
make  imx8qxp_mek_defconfig
make
```

编译 log 如下：

```
…
LDS     spl/u-boot-spl.lds
 LD      spl/u-boot-spl
 OBJCOPY spl/u-boot-spl-nodtb.bin
 COPY    spl/u-boot-spl.dtb
 CAT     spl/u-boot-spl-dtb.bin
 COPY    spl/u-boot-spl.bin
WARNING './ahab-container.img' not found, resulting binary is not-functional
```

```
WARNING './ahab-container.img' not found, resulting binary is not-functional
make[1]: Nothing to be done for 'SPL'.
  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
  CAT     u-boot-dtb.bin
  COPY    u-boot.bin
  SYM     u-boot.sym
  MKIMAGE u-boot.img
  COPY    u-boot.dtb
  MKIMAGE u-boot-dtb.img
  LD      u-boot.elf
```

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_4.19
  ...
   rel_imx_4.19.35_1.1.0
  ...
git checkout rel_imx_4.19.35_1.1.0
git status
  HEAD detached at rel_imx_4.19.35_1.1.0
source ~ /imx-yocto-bsp/imx8qxpmek_wayland/sdk/environment-setup-aarch64-poky-linux
make defconfig
make
make dtbs clean
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_4.19.35_1.1.0

...

git checkout rel_imx_4.19.35_1.1.0

git status

HEAD detached at rel_imx_4.19.35_1.1.0

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_4.19.35_1.1.0
  ...
git checkout rel_imx_4.19.35_1.1.0
git status
HEAD detached at rel_imx_4.19.35_1.1.0
```

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：

```
pwd
~/standalone/imx-mkimage
cp ../../imx-yocto-bsp/imx8qxpmek_wayland/tmp/deploy/images/imx8qxpmek/imx-boot-tools/mx8qx-ahab-container.img ./iMX8QX/
cp ../scfw/packages/imx-scfw-porting-kit-1.2.7.1/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 -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

SCFW file_offset = 0xa800 size = 0x25000

Hash of the images = sha384

AP file_offset = 0x2f800 size = 0xdec00

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/imx-seco-2.3.1.bin

chmod +x imx-seco-2.3.1.bin && ./ imx-seco-2.3.1.bin --auto-accept

mx8qx-ahab-container.img 位于 imx-seco-2.3.1.bin/firmware/seco/mx8qx-ahab-container.img

6. 更新镜像到已经烧写了*.sdcard 的 sdcard 里

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-mek.dtb to Boot imx8qx

3 i.MX8X SC firmware

3.1 SC firmware 目录结构

imx-scfw-porting-kit-1.2.7.1

|->doc

| |->pdf/ sc_fw_port.pdf

| |->pdf/sc_fw_api.pdf, sc_fw_api_qx_b0.pdf, sc_fw_api_qm_b0.pdf

| |->pdf/ sc_fw_rn.pdf

|->src

| |->scfw_export_mx8qm

| |->scfw_export_mx8qx_b0

| | |->bin

| | |->Makefile

| | |->build_mx8qx_b0

| | |->makefiles

| | |->platform

| | | |->board

| | | | |->pmic.c/pmic.h pmic 上层访问 APIs

| | | | |->mx8qx_mek

| | | | | |->board.c (customer's board customization file)

| | | | | | |->dcd

| | | | | | | | |->imx8qx_dcd_1.2GHz.cfg (DDR parameters setting file)

| | | |->drivers

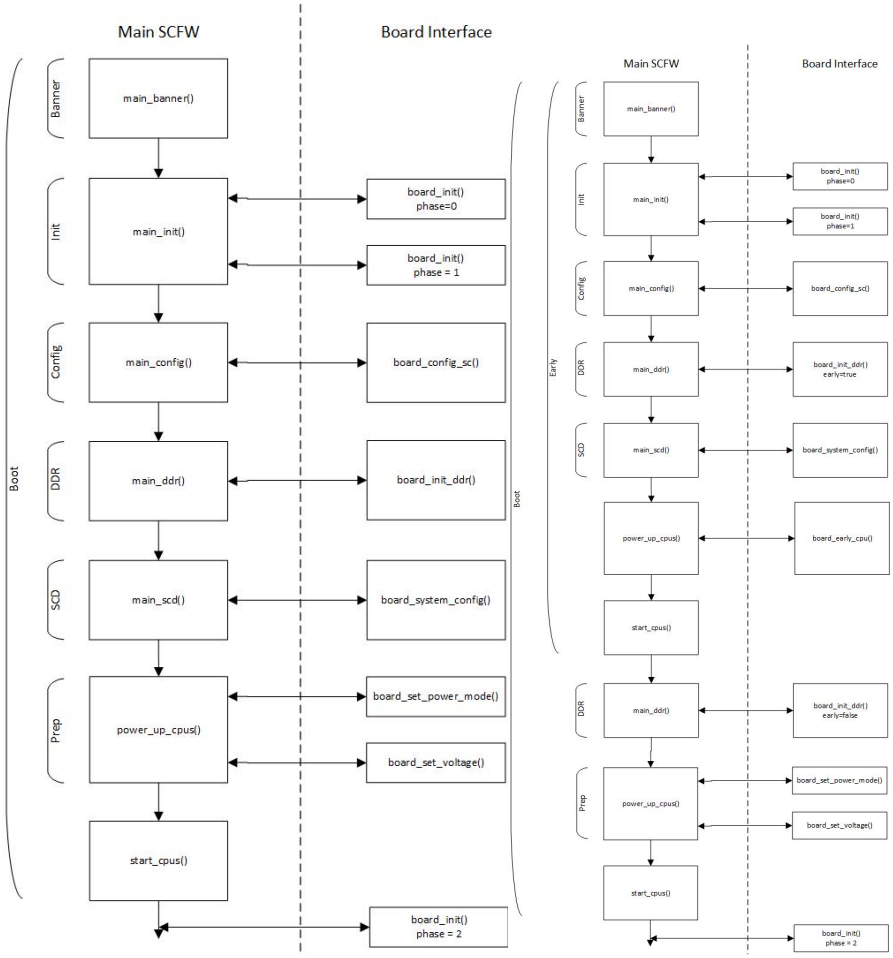
| | | | |->pmic

| | | | |->fsl_pmic.c\fsl_pmic.h

| | | | |->pf8100

| | | | | |->fsl_pf8100.c\fsl_pf8100.h

3.2 SC firmware 启动流程



3.3 SC firmware 定制

SC firmware 是以二进制镜像加部分源代码的方式发布的，请参考文档 sc_fw_port.pdf 了解如何编译及修改到一块新板上，通常情况下如果客户设计的新板的 DDR 设计与 NXP 的开发板不同，则需要修改相应的静态参数（动态校准参数会在每次开机是自动生成），另外与启动相关的硬件区别可能也需要修改，比如说如果修改了电源设计，则需要修改 PMIC 相关代码。

3.3.1 DDR 配置：

DDR 参数直接编译到 SC firmware 镜像中的方式，这种情况下 DCD (.cfg) 文件首先转换成“.h”头文件，然后编译到 SC firmware 镜像中。

```
\mx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\Makefile
#Set Default DDR config file
ifeq ($(Z),1)
        DDR_CON ?= imx8qx_dcd_emul
else
        DDR_CON ?= imx8qx_dcd_1.2GHz //所以默认是使用以下DDR配置cfg文件
endif
```

```
\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\dcd\imx8qx_dcd_1.2GHz.cfg
```

可以使用“Programming Aids：MX8QXP_C0_B0_LPDDR4_RPA_1.2GHz_v13.xlsx” excel 表自动生成 DCD 段值，(下载地址https://community.nxp.com/docs/DOC-335253 如果无法下载，请与你对应的 NXP FAE 联系)。

以下以一个特殊例子，i.MX8DX 16bit DDR3L 256MB 内存为例说明如何使用配置工具 (iMX8DX DDR3L 的配置工具为：MX8DualX_B0_DDR3L_RPA_v17.xlsx)：

i.MX8X 的 DDR 功能是：

External Memory

• DDR3-1866, i.e. 933MHz (1066 MHz is supported, but JEDEC standard is unavailable), with optional ECC protection

• LPDDR4-2400, i.e. 1200MHz, without ECC protection

使用配置表：“MX8DualX_B0_DDR3L_RPA_v17.xlsx”如下配置：

Device Information	
SoC:	**MX8 DX**
Memory type:	**DDR3L**
Manufacturer:	Micron
Memory part number:	MT41K1G8SN-107 IT:A
Density per Device (Gb)[1]:	2
Number of Devices per chip select	1
Density per chip select (Gb)	2
Number of Chip Selects used[2]	1
Total DRAM density (Gb)	2
Number of ROW Addresses[2]	14
Number of COLUMN Addresses[2]	10
Number of BANK addresses[2]	3
Number of BANKS[2]	8
Device Width	16
Total Data Bus Width	16

Clock Cycle Freq (MHz)[3]	933
Clock Cycle Time (ns)	1.071

然后从 DCD CFG file 一页拷贝出 DDR configuration 内容。

不再需要进行离线的 DDR 校准（i.MX8X 每次启动时自动校准。

所以 DDR 压力测试工具只需要用于压力测试。

DDR 配置初始化源代码：

```
\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\dcd\dcd.h (此文件为编译自动生成)
#ifndef DCD_H
#define DCD_H
#include "imx8qx_dcd_1.2GHz.h"
#endif

\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\board.c
/*----------------------------------------------------------------------*/
/* Take action on DDR                                                   */
/*----------------------------------------------------------------------*/
void board_ddr_config (void)
{
  …
    #include "dcd/dcd.h"
  …
}
```

所以如果客户修改了 DDR 设计，软件修改方式如下：

1. 根据 DDR 数据手册和 DDR 设计，填写 Programming Aids，得到 DDR DCD(.cfg)配置文件。
2. 在 SC firmware 以下目录中修改 DCD 文件，然后重新编译 scfw。

   \imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\dcd\imx8qx_dcd_1.2GHz.cfg

3. 然后修改 uboot 中的内存尺寸参数(以 i.MX8DX+256MB DDR3L 为例)：

```
#define CONFIG_SYS_SDRAM_BASE          0x80000000
#define PHYS_SDRAM_1                   0x80000000
#define PHYS_SDRAM_2                   0x880000000
#define PHYS_SDRAM_1_SIZE          0x10000000      /* 256MB */
```

```
#define PHYS_SDRAM_2_SIZE              0x0       /* 0 GB */
```

4. 重新编译 SC firmware 和 flash.bin。

3.3.2 PMIC 定制

强烈建议客户参考 NXP 开发板的设计，使用对应的 PMIC 芯片及设计，因为现在板级开发包中的电源管理代码是由 SC firmware 最终调用 的，所以 PMIC 的访问代码是在 SC firmware 中，如果客户修改了电源设计，比如说换了其它的 PMIC，则需要相应修改 SC firmware 中 PMIC 相应代码：

SC firmware PMIC 代码如下：

PMIC 初始化代码：

Power_up_cpus -> prep_cpu-> pm_force_resource_power_mode-> pm_update_ridx-> ss_trans_power_mode-> sc_ss_ep[ss].ss_trans_power_mode -> ss_trans_power_mode_base->ss_trans_pd

soc_trans_pd->board_set_power_mode->board_get_pmic_info-> pmic_init(board.c)

PMIC 调用 APIs

board_set_voltage-> board_get_pmic_info

pmic_init-> pmic_ignore_current_limit

|-> pmic_update_timing

所以如果修改了电源设计，如换了 PMIC，需要重写以下代码：
\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\board.c

的以下函数：

pmic_init/ board_set_power_mode/board_set_voltage/ board_get_pmic_info/ pmic_ignore_current_limit/ pmic_update_timing

然后在

\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\pmic.c pmic.h

的 pmic 上层函数要增加新的 PMIC 访问代码。

最后还需要添加相应 PMIC 驱动在如下目录
\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\drivers\pmic, 在此处添加新的 PMIC 驱动，可以参考 fsl_pmic.h 的 I2C 访问 APIs i2c_write/read 来访问新的 PMIC(仅指可以通过 I2C 访问的 PMIC,比如说通过 SPI 口访问的则 SPI 访问驱动接口没有开源)。这个驱动需要实现定义在

\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\pmic.h 中的 pmic 访问 APIs:

```
#ifdef PMIC /* Replace function prefix with defined PMIC */
  #define FUNC_PRE(name, function)          FUNC_PREPRE(name,function)
  #define FUNC_PREPRE(name, function)       name ## function
  #define PMIC_SET_MODE          FUNC_PRE(PMIC, _pmic_set_mode)
  #define PMIC_GET_MODE          FUNC_PRE(PMIC, _pmic_get_mode)
```

```
#define PMIC_SET_VOLTAGE        FUNC_PRE(PMIC,  pmic_set_voltage)
#define PMIC_GET_VOLTAGE        FUNC_PRE(PMIC, _pmic_get_voltage)
#define PMIC_IRQ_SERVICE        FUNC_PRE(PMIC,  pmic_irq_service)
#define PMIC_REGISTER_ACCESS    FUNC_PRE(PMIC,  pmic_register_access)
#define GET_PMIC_VERSION        FUNC_PRE(PMIC, _get_pmic_version)
#define GET_PMIC_TEMP           FUNC_PRE(PMIC, _get_pmic_temp)
#define SET_PMIC_TEMP_ALARM     FUNC_PRE(PMIC, _set_pmic_temp_alarm)
#else /*
```

所以还是强烈客户参考 NXP 开发板的设计，不用修改相应电源设计。

另外注意一下 scfw 中有为 A35 核和 GPU 开电的代码，这段代码不开源，所以 A35 与 GPU 电源在上电时是没有电的，由 SCU 开电。

如果是使用 SPF-29683-D1.pdf 的设计， SCU 用自己的调试串口，可以如下使用 SCU 命令操作 PMIC: 以下为读出 SCU_LDO 的电压为 1.6V:

pmic.r 8 0x88

PMIC 0x8 read register 0x88: 0x1

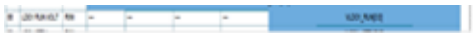
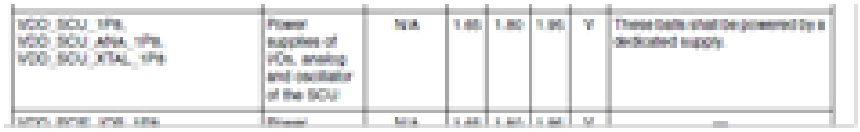


Table 62. LDO output voltage configuration

Set point	VLDOx_RSM[3:0] VLDOx_STBY[3:0]	VLDOx output (V)
0	0000	1.5
1	0001	1.6
2	0010	1.8
3	0011	1.85
4	0100	2.15
5	0101	2.5
6	0110	2.8
7	0111	3.0
8	1000	3.1
9	1001	3.15
10	1010	3.2
11	1011	3.3
12	1100	3.35
13	1101	4.0
14	1110	4.9
15	1111	5.0



也可以修改源代码操作 PMIC:如下设置 SCU LDO1=1.6V.

src\scfw_export_mx8qm_b0\platform\board\mx8qm_mek\board.c

```
//johnlii test
                    {
```

```
                uint32_t vol_mv;
                sc_err_t rtn = SC_ERR_NONE;
                rtn=PMIC_SET_VOLTAGE(PMIC_0_ADDR, PF8100_LDO1,
                  1600, REG_RUN_MODE);
                PMIC_GET_VOLTAGE(PMIC_0_ADDR, PF8100_LDO1, &vol_mv, REG_RUN_MODE)
                board_print(1, "SCFW: johnli debug,set pf8100_1_ldo1=scu=%d,error=%x\n",vol_mv,rtn);
                    }
                //end
        /* Enable PMIC IRQ at NVIC level */
Src\scfw_export_mx8qm_b0\platform\drivers\pmic\pf8100\fsl_pf8100.c
static const uint32_t ldo_table_b0[16] =
{
  1500U, /* lookup for LDO values  B0*/
  1600U,
  1800U,
…
};
SCFW: johnli debug,set pf8100_1_ldo1=scu=1600,error=0
Print out :
```

3.3.3 调试串口接口定制 UART interface

目前的 NXP MEK 开发板(8qx)，其调试串口是默认关闭的，可以如下方式打开：

Build with debug information enable:

Make qx B=mek R=B0 M=1

关于 debug 信息“sc_fw_port.pdf”解释如下：

Option	Action
V=0	quite output (default)
V=1	verbose output
D=0	configure for no debug
D=1	configure for debug (default)
DL=<level>	configure debug level (0-5)
M=0	no debug monitor (default)
M=1	include debug monitor
B=<board>	configure board (default=val)
U=<uart>	configure debug UART (default=0)
DDR_CON=<file>	specify DDR config file w/o extension
R=<srev>	silicon revision
T=<test>	run tests rather than boot next core

3.5 Production

When the SCFW is compiled for release into production devices, it is critical that this is done without debug (default is debug enabled, D=1) and without the debug monitor (default is no monitor, M=0). For example:

```
make qm R=B0 D=0 M=0
```

Turning off debug will eliminate the linking of the standard C library.

Debug Monitor

A debug monitor can be compiled into the SCFW using the M=1 make option. This can be used to R/W memory or registers, R/W power state, and dump some resource manager state. See Debug Monitor for more info.

Production SCFW should never have the monitor enabled (M=0, the default)!

NXP MEK开发板(8qx)SC firmware的调试串口默认是使用如下M4核的串口和IOMUX:

imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\board\mx8qx_mek\board.c

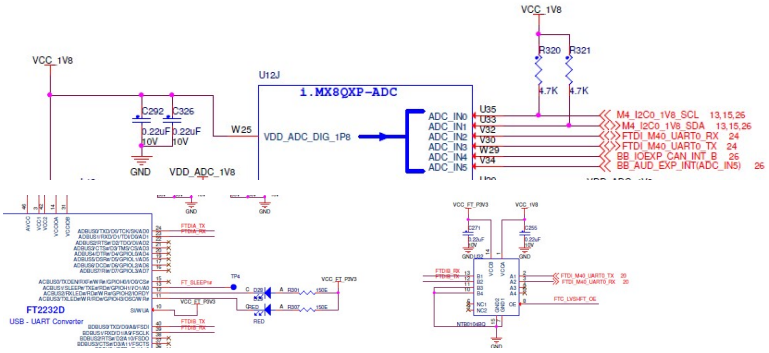
```
#if DEBUG_UART == 2
   /*! Use alternate debug UART */
   #define ALT_DEBUG_SCU_UART
#endif
#ifdef ALT_DEBUG_SCU_UART
```

```
  #define LPUART_DEBUG       LPUART_SC
#else
  #define LPUART_DEBUG       LPUART_M4_0
#endif
/*! Configure debug UART instance */
#ifdef ALT_DEBUG_SCU_UART
  #define LPUART_DEBUG_INST   0U
#else
  #define LPUART_DEBUG_INST   2U
#endif
```

board_config_debug_uart(board.c)

```
/* Configure pads */
        (void) pad_set_mux(SC_PT, SC_P_ADC_IN2, 1, SC_PAD_CONFIG_NORMAL,
          SC_PAD_ISO_OFF);
        (void) pad_set_mux(SC_PT, SC_P_ADC_IN3, 1, SC_PAD_CONFIG_NORMAL,
          SC_PAD_ISO_OFF);
```

原理图如下：



所以建议保留此 IOPAD 给 SC firmware/m4 核调试使用：.

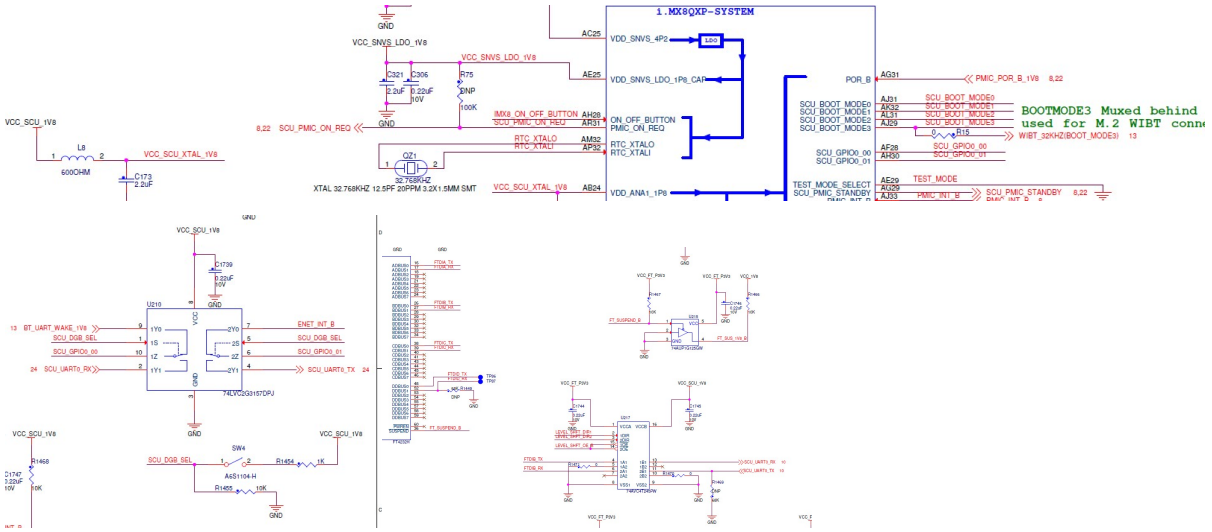
SC firmware 串口调试的 API 文件：

\imx-scfw-porting-kit-1.2.7.1\src\scfw_export_mx8qx_b0\platform\main\ debug.h

中将 debug_print 重定义到每一个调试级别函数。

以上设计是源自 i.MX8QXP MEK 板 SPF-29683-C1.pdf 的设计，SC firmware/m4 内核复用同一个调试串口，所以这个串口的电源域是 VCC_1V8 的电源域。不是 SCU 的 VCC_SCU_1V8 这个电源域。

从 SPF-29683-D1.pdf 的设计中：



使能 SW4 键，可以使能 SCU 自己的调试串口：(iomux from SCU_GPIO0_00/01)，使用编译命令 make qx B=mek R=B0 M=1 U=2。源代码分析如下:

```
    pwd
~/imx-yocto-bsp/standalone/packages/imx-scfw-porting-kit-1.2.7.1/src/scfw_export_mx8qx_b0
vi makefile
…
# Configure UART
ifdef u
  U := $(u)
endif
ifndef U
  U = 0  //default= 0
endif
FLAGS += -DDEBUG_UART=$(U)
…
vi platform/board/mx8qx_mek/board.c

#if DEBUG_UART == 3
  /*! Use debugger terminal emulation */
  #define DEBUG_TERM_EMUL
#endif
#if DEBUG_UART == 2
  /*! Use alternate debug UART */
```

```
    #define ALT_DEBUG_SCU_UART
#endif
#if (defined(MONITOR) || defined(EXPORT_MONITOR) || defined(HAS_TEST) \
        || (DEBUG_UART == 1)) && !defined(DEBUG_TERM_EMUL) \
        && !defined(ALT_DEBUG_SCU_UART)
    #define ALT_DEBUG_UART
#endif

/*! Configure debug UART */
#ifdef ALT_DEBUG_SCU_UART
    #define LPUART_DEBUG        LPUART_SC
#else
    #define LPUART_DEBUG        LPUART_M4_0
#endif

/*! Configure debug UART instance */
#ifdef ALT_DEBUG_SCU_UART
    #define LPUART_DEBUG_INST   0U
#else
    #define LPUART_DEBUG_INST   2U
#endif
…
void board_config_debug_uart(sc_bool_t early_phase)
{
    #if defined(ALT_DEBUG_SCU_UART) && !defined(DEBUG_TERM_EMUL) \
            && defined(DEBUG) && !defined(SIMU)
        /* Power up UART */
        pm_force_resource_power_mode_v(SC_R_SC_UART,
            SC_PM_PW_MODE_ON);
…

void board_config_sc(sc_rm_pt_t pt_sc)
{
    …
```

```
#ifdef ALT_DEBUG_SCU_UART
    (void) rm_set_pad_movable(pt_sc, SC_P_SCU_GPIO0_00,
        SC_P_SCU_GPIO0_01, SC_FALSE);
#endif
}
```

即可以调换成此串口输出，使用 SCU 自己的串口可以保证在只有 SCU 电源域供电的情况下也可以正常输出。

3.3.4 访问 GPIO

有时，需要在 SC firmware 中操作 GPIO，比如说开 GPIO 电源，或是设置调试标志灯之类的， i.MX8QXP MEK SC firmware 没有相关代码，可以参考：
\imx-scfw-porting-kit-1.1\src\scfw_export_mx8qm_b0\platform\board\mx8qm_mek\board.c

```
board_init->
…
    (void) pad_set_mux(SC_PT, SC_P_SCU_GPIO0_01, 0, SC_PAD_CONFIG_NORMAL,
        SC_PAD_ISO_OFF);
    (void) pad_set_mux(SC_PT, SC_P_SCU_GPIO0_02, 0, SC_PAD_CONFIG_NORMAL,
        SC_PAD_ISO_OFF);

    /* Assert base board reset SC_GPIO_01 */
    config.outputLogic  = 0U;
    FGPIO_PinInit(FGPIOA, 1U, &config);

    /* SCU_LED on SC_GPIO_02 */
    config.outputLogic  = 1U;
    FGPIO_PinInit(FGPIOA, 2U, &config);

    SystemTimeDelay(2U);
```

3.3.5 xRCD

Two methods for xRDC configuration: new SCD data header (similar to DCD) in “flash.bin” and SCU built-in (recommended).

For details on how to write/build SCD, please refer to the document in “SCFW porting kit” package.

4 i.MX8X ATF

ATF (ARM Trusted Firmware)是默认使能的，正常的启动如下： BOOTROM -> SCUFW -> ATF -> UBOOT -> OS

什么是 ATF，如下解释：

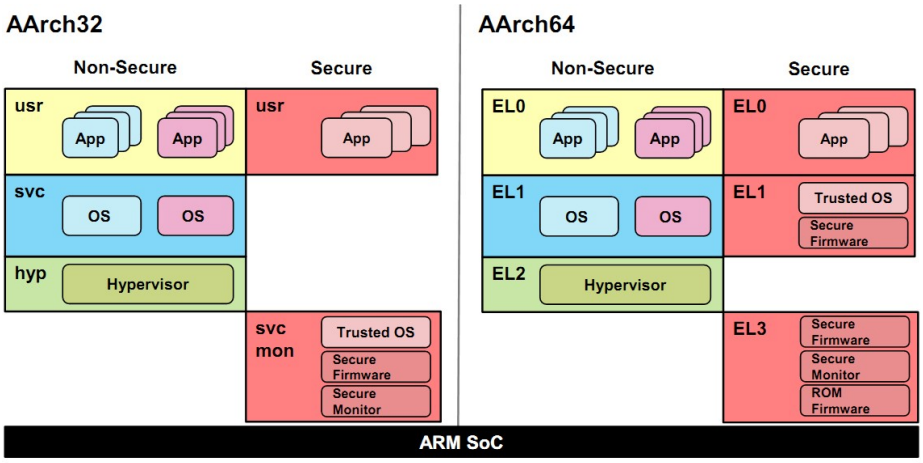
“*ARM Trusted Firmware provides a reference implementation of secure world software for ARMv8-A, including a Secure Monitor executing at Exception Level 3 (EL3). It implements various ARM interface standards, such as:*

The Power State Coordination Interface (PSCI)

Trusted Board Boot Requirements (TBBR, ARM DEN0006C-1)

SMC Calling Convention

System Control and Management Interface”



为什么使用 ATF:

- Secure Monitor. Support for TEE.
- PSCI, coordinate power management for secure/non-secure world..

EL1/EL2 (OS/VMM) are supposed to use PSCI call (SMC) for any cluster level power management, such as DVFS, Idle, Suspend/Resume and etc.

正常情况下，如果不使用 TEE，不需要修改 ATF，以下仅说明如何打开 ATF 的调试串口消息，默认是关闭的：

\imx-atf\plat\imx\imx8qxp\include\platform_def.h

```
#define DEBUG_CONSOLE                   0  /*change to 1 to  open console*/
#ifdef TEE_IMX8
#define DEBUG_CONSOLE_A35               1
```

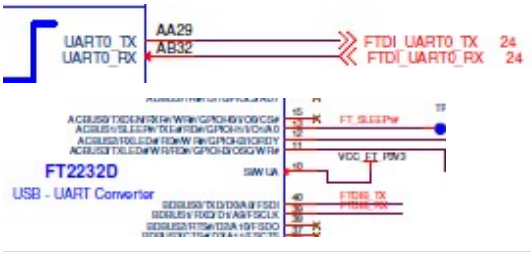
```
#else
#define DEBUG_CONSOLE_A35                    0 /*change to 1 to  open console on a35*/
#endif
   #define IMX_BOOT_UART_BASE                0x5a060000
```

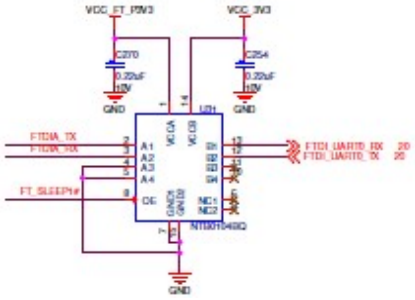
\imx-atf\plat\imx\imx8qxp\imx8qxp_bl31_setup.c

```
   bl31_early_platform_setup2
#if DEBUG_CONSOLE_A35
        /* Power up UART0 */
        sc_pm_set_resource_power_mode(ipc_handle, SC_R_UART_0, SC_PM_PW_MODE_ON);

        /* Set UART0 clock root to 80 MHz */
        sc_pm_clock_rate_t rate = 80000000;
        sc_pm_set_clock_rate(ipc_handle, SC_R_UART_0, 2, &rate);

        /* Enable UART0 clock root */
        sc_pm_clock_enable(ipc_handle, SC_R_UART_0, 2, true, false);

#define UART_PAD_CTRL       (PADRING_IFMUX_EN_MASK | PADRING_GP_EN_MASK | \
                             (SC_PAD_CONFIG_OUT_IN << PADRING_CONFIG_SHIFT) | \
                             (SC_PAD_ISO_OFF << PADRING_LPCONFIG_SHIFT) | \
                             (SC_PAD_28FDSOI_DSE_DV_LOW << PADRING_DSE_SHIFT) | \
                             (SC_PAD_28FDSOI_PS_PD << PADRING_PULL_SHIFT))
        /* Configure UART pads */
        sc_pad_set(ipc_handle, SC_P_UART0_RX, UART_PAD_CTRL);

        sc_pad_set(ipc_handle, SC_P_UART0_TX, UART_PAD_CTRL);

        lpuart32_serial_init(IMX_BOOT_UART_BASE);
#endif


#if DEBUG_CONSOLE
        console_lpuart_register(IMX_BOOT_UART_BASE, IMX_BOOT_UART_CLK_IN_HZ,
                IMX_CONSOLE_BAUDRATE, &console);
#endif


   #if DEBUG_CONSOLE_A35
   /*
    * Note:
    * This piece code is from uboot, take care of license issue.
    */
   static void lpuart32_serial_setbrg(unsigned int base, int baudrate)
   {
        …
```

```
}

static int lpuart32_serial_init(unsigned int base)
{
        …
}
#endif
```

所以 ATF 使用 uboot/kernel 所使用的调试串口，如果要查看 ATF 消息，不建议修改 A35 核的调试串口。





5 FSL Uboot 定制

相对与 4.1.15 使用的 uboot 2016_03 相比，目前 4.19.35 的 uboot 2019_04 增加的两个重要功能 是：

- 如同内核一样，支持 FDT(Flatted device tree)
- 与内核类似，支持 DM(driver model)

以下详细说明这两个功能：

5.1 FDT 支持

5.1.1 说明

FDT，flatted device tree，扁平设备树。简单理解为将部分设备信息结构存放到 device tree 文件中。 uboot 最终将其 device tree 编译成 dtb 文件，使用过程中通过解析该 dtb 来获取板级设备信息。 uboot 的 dtb 和 kernel 中的 dtb 是一致的。

DTB 头结构体如下：

```
Scripts\dtc\libfdt\fdt.h
struct fdt_header {
        fdt32_t magic;                    /* magic word FDT_MAGIC */
        fdt32_t totalsize;           /* total size of DT block */
        fdt32_t off_dt_struct;            /* offset to structure */
        fdt32_t off_dt_strings;           /* offset to strings */
        fdt32_t off_mem_rsvmap;                   /* offset to memory reserve map */
        fdt32_t version;             /* format version */
        fdt32_t last_comp_version;        /* last compatible version */

        /* version 2 fields below */
        fdt32_t boot_cpuid_phys;          /* Which physical CPU id we're
                                             booting on */
        /* version 3 fields below */
        fdt32_t size_dt_strings;    /* size of the strings block */

        /* version 17 fields below */
        fdt32_t size_dt_struct;           /* size of the structure block */
};
```

其中，magic 是一个固定的值，0xd00dfeed（大端）

```
#define FDT_MAGIC 0xd00dfeed        /* 4: version, 4: total size */
```

只要提取待验证 dtb 的地址上的数据的前四个字节，与 0xd00dfeed（大端）或者 0xedfe0dd0（小端）进行比较，如果匹配的话，就说明对应待验证 dtb 就是一个合法的 dtb。

以下编译宏：

```
\uboot-imx\configs\ imx8qxp_mek_defconfig
```

CONFIG_OF_CONTROL=y 用于表示是否使用了 dtb 的方式

5.1.2 dtb 在 uboot 中的位置

dtb可以以两种形式编译到uboot的镜像中。

- dtb 和 uboot 的 bin 文件分离
    - 如何使能
      需要打开 CONFIG_OF_SEPARATE 宏来使能。

\uboot-imx\dts\ Kconfig

```
choice
    prompt "Provider of DTB for DT control"
    depends on OF_CONTROL
config OF_SEPARATE
    …
config OF_EMBED
```

默认选择： CONFIG_OF_SEPARATE=y // 是否将 dtb 和 uboot 分离表

    - 编译说明
      在这种方式下，uboot 的编译和 dtb 的编译是分开的，先生成 uboot 的 bin 文件，然后再另外生成 dtb 文件。
    - 最终位置
      dtb 最终会追加到 uboot 的 bin 文件的最后面。也就是 uboot.img 的最后一部分。
      因此，可以通过 uboot 的结束地址符号，也就是_end 符号来获取 dtb 的地址。
- dtb 集成到 uboot 的 bin 文件内部
    - 如何使能
      需要打开 CONFIG_OF_EMBED 宏来使能。
    - 编译说明
      在这种方式下，在编译 uboot 的过程中，也会编译 dtb。
    - 最终位置
      注意：最终 dtb 是包含到了 uboot 的 bin 文件内部的。
      dtb 会位于 uboot 的.dtb.init.rodata 段中，并且在代码中可以通过__dtb_dt_begin 符号获取其符号。

因为这种方式不够灵活，文档上也不推荐，所以后续也不具体研究，简单了解一下即可。

- 另外，也可以通过 fdtcontroladdr 环境变量来指定 dtb 的地址
  可以通过直接把 dtb 加载到内存的某个位置，并在环境变量中设置 fdtcontroladdr 为这个地址，达到动态指定 dtb 的目的。
  在调试中使用。

  i.MX8X MEK 板的 DTS 源文件为：

```
\uboot-imx\arch\arm\dts\fsl-imx8qxp-mek.dts
                    |->fsl-imx8qxp.dtsi
                    |    |->fsl-imx8-ca35.dtsi
                    |    |    |->dt-bindings/clock/imx8qxp-clock.h, dt-bindings/interrupt-controller/arm-gic.h
                    |    |->dt-bindings/soc/imx_rsrc.h, imx8_hsio.h, imx8_pd.h,
dt-bindings/clock/imx8qxp-clock.h, dt-bindings/pinctrl/pads-imx8qxp.h, dt-bindings/gpio/gpio.h
```

5.1.3 uboot 中如何获取 dtb

在uboot初始化过程中，需要对dtb做两个操作：

- 获取 dtb 的地址，并且验证 dtb 的合法性
- 因为我们使用的 dtb 并没有集成到 uboot 的 bin 文件中，也就是使用的 CONFIG_OF_SEPARATE 方式。因此，在 relocate uboot 的过程中并不会去 relocate dtb。因此，这里我们还需要自行为 dtb 预留内存空间并进行 reloca
- relocate 之后，还需要重新获取一次 dtb 的地址

  这部分过程是在 init_board_f 中实现, 对应代码 common/board_f.c

```
static const init_fnc_t init_sequence_f
        |->fdtdec_setup, /* 获取 dtb 的地址，并且验证 dtb 的合法性*/
int fdtdec_setup(void)
{
…
        /* FDT is at end of image */
        gd->fdt_blob = (ulong *)&_end; /* 当使用 CONFIG_OF_SEPARATE 的方式时，也就是 dtb 追加到 uboot 的 bin 文件后面时，通过_end 符号来获取 dtb 地址 */
…
        /* Allow the early environment to override the fdt address */
    /* 可以通过环境变量 fdtcontroladdr 来指定 gd->fdt_blob，也就是指定 fdt 的地址 */
```

```
        gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
                                           (uintptr_t)gd->fdt_blob);
…
/* 最终都把 dtb 的地址存储在 gd->fdt_blob 中 */
/* 在 fdtdec_prepare_fdt 中检查 fdt 的合法性  */
// 判断 dtb 是否存在，以及是否有四个字节对齐
// 然后再调用 fdt_check_header 看看头部是否正常。fdt_check_header 主要是检查 dtb 的 magic 是否正确
        return fdtdec_prepare_fdt();
}
      |->reserve_fdt, /* 为 dtb 分配新的内存地址空间 */
static int reserve_fdt(void)
{
#ifndef CONFIG_OF_EMBED
        /*
         * If the device tree is sitting immediately above our image then we
         * must relocate it. If it is embedded in the data section, then it
         * will be relocated with other data.
         */
         /*
          当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
   当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate
    */
        if (gd->fdt_blob) {
                /* 获取 dtb 的 size */
                gd->fdt_size = ALIGN(fdt_totalsize(gd->fdt_blob) + 0x1000, 32);
        /* 为 dtb 分配新的内存空间 */
                gd->start_addr_sp -= gd->fdt_size;
                gd->new_fdt = map_sysmem(gd->start_addr_sp, gd->fdt_size);
                debug("Reserving %lu Bytes for FDT at: %08lx\n",
                      gd->fdt_size, gd->start_addr_sp);
        }
#endif
```

```
        return 0;
}
        |->reloc_fdt, /* relocate dtb  */
static int reloc_fdt(void)
{
#ifndef CONFIG_OF_EMBED
/*
当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate

*/
 /* 检查 GD_FLG_SKIP_RELOC 标识 */
        if (gd->flags & GD_FLG_SKIP_RELOC)
                return 0;
        if (gd->new_fdt) {
                /* relocate dtb 空间 */
                memcpy(gd->new_fdt, gd->fdt_blob, gd->fdt_size);
                /* 切换 gd->fdt_blob 到 dtb 的新的地址空间上 */
                gd->fdt_blob = gd->new_fdt;
        }
#endif

        return 0;
}
```

5.1.4 uboot 中 dtb 解析的常用接口

gd->fdt_blob 已经设置成了 dtb 的地址了。注意，fdt 提供的接口都是以 gd->fdt_blob（dtb 的地址）为参数的。以下只简单说明几个接口的功能，另外，用节点在 dtb 中的偏移地址来表示一个节点。也就是节点变量 node 中，存放的是节点的偏移地址。

- lib/fdtdec.c 中
    - **fdt_path_offset**
      int fdt_path_offset(const void *fdt, const char *path)
      eg：node = fdt_path_offset(gd->fdt_blob, “/aliases”);
      功能：获得 dtb 下某个节点的路径 path 的偏移。这个偏移就代表了这个节点。

- **fdt_getprop**
  const void *fdt_getprop(const void *fdt, int nodeoffset, const char *name, int *lenp)
  eg： mac = fdt_getprop(gd->fdt_blob, node, “mac-address”, &len);
  功能：获得节点 node 的某个字符串属性值。

- **fdtdec_get_int_array、fdtdec_get_byte_array**
  int fdtdec_get_int_array(const void *blob, int node, const char *prop_name, u32 *array, int count)
  eg： ret = fdtdec_get_int_array(blob, node, “interrupts”, cell, ARRAY_SIZE(cell));
  功能：获得节点 node 的某个整形数组属性值。

- **fdtdec_get_addr**
  fdt_addr_t fdtdec_get_addr(const void *blob, int node, const char *prop_name)
  eg：fdtdec_get_addr(blob, node, “reg”);
  功能：获得节点 node 的地址属性值。

- **fdtdec_get_config_int、fdtdec_get_config_bool、fdtdec_get_config_string**
  功能：获得 config 节点下的整形属性、bool 属性、字符串等等。

- **fdtdec_get_chosen_node**
  int fdtdec_get_chosen_node(const void *blob, const char *name)
  功能：获得 chosen 下的 name 节点的偏移

- **fdtdec_get_chosen_prop**
  const char *fdtdec_get_chosen_prop(const void *blob, const char *name)
  功能：获得 chosen 下 name 属性的值

- lib/fdtdec_common.c 中

  - **fdtdec_get_int**
    int fdtdec_get_int(const void *blob, int node, const char *prop_name, int default_val)
    eg： bus->udelay = fdtdec_get_int(blob, node, “i2c-gpio,delay-us”, DEFAULT_UDELAY);
    功能：获得节点 node 的某个整形属性值。

  - **fdtdec_get_uint**
    功能：获得节点 node 的某个无符号整形属性值。

5.2 DM(driver model)支持

5.2.1 说明

uboot 引入了驱动模型（driver model），这种驱动模型为驱动的定义和访问接口提供了统一的方法。提高了驱动之间的兼容性以及访问的标准型。 uboot 驱动模型和 kernel 中的设备驱动模型类似，但是又有所区别。

具体细节建议参考./doc/driver-model/README.txt

在 configs/ imx8qxp_mek_defconfig 中定义了如下：

CONFIG_DM=y

DM 和 uclass 是息息相关的，如果我们希望在某个模块引入 DM，那么就需要使用相应模块的 uclass driver 来代替旧版的通用 driver

CONFIG_DM_SERIAL/ I2C/ USB/ GPIO/ PCA953X/ MMC/ SPI/ SPI_FLASH/ETH/

REGULATOR/ REGULATOR_FIXED/ REGULATOR_GPIO/ THERMAL=y

看 driver/serial/Makefile

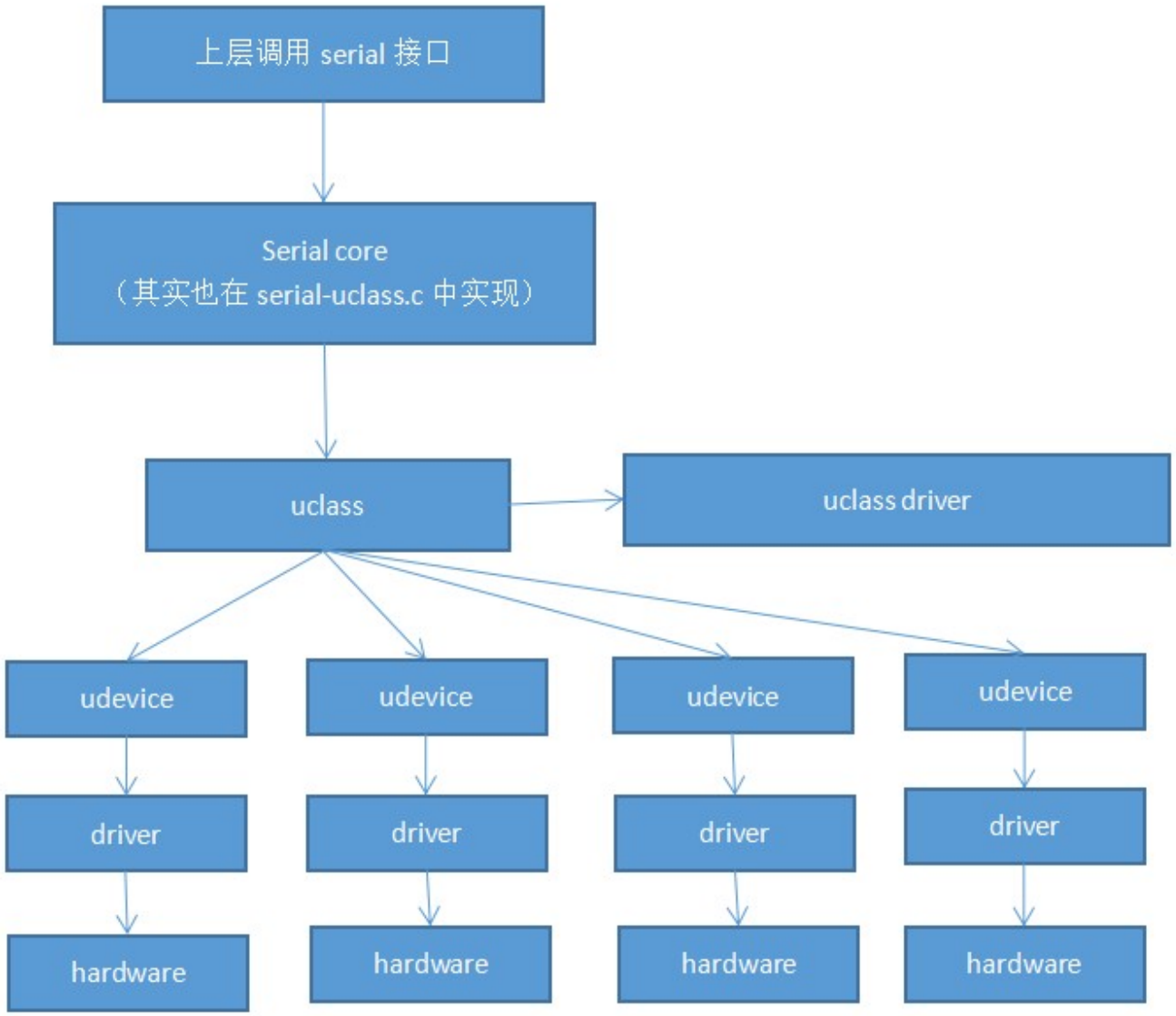
```
ifdef CONFIG_DM_SERIAL
        obj-y += serial-uclass.o ## 引入 dm 的 serial core 驱动
else
        obj-y += serial.o ## 通用的 serial core 驱动
endif
```

5.2.2 uboot DM 整体架构

DM 的四个组成部分

- **udevice**：简单就是指设备对象，可以理解为 kernel 中的 device
- **driver**：udevice 的驱动，可以理解为 kernel 中的 device_driver。和底层硬件设备通信，并且为设备提供面向上层的接口
- **uclass**：uclass，使用相同方式的操作集的 device 的组。相当于是一种抽象。uclass 为那些使用相同接口的设备提供了统一的接口。
  例如，GPIO uclass 提供了 get/set 接口。再例如，一个 I2C uclass 下可能有 10 个 I2C 端口，4 个使用一个驱动，另外 6 个使用另外一个驱动
- **uclass_driver**：对应 uclass 的驱动程序。主要提供 uclass 操作时，如绑定 udevice 时的一些操作

调用关系框架图



相互之间的关系

结合上图来看：

- 上层接口都是和 uclass 的接口直接通讯。
- uclass 可以理解为一些具有相同属性的 udevice 对外操作的接口，uclass 的驱动是 uclass_driver，主要为上层提供接口。
- udevice 的是指具体设备的抽象，对应驱动是 driver，driver 主要负责和硬件通信，为 uclass 提供实际的操作集。

- udevice 找到对应的 uclass 的方式主要是通过：udevice 对应的 driver 的 id 和 uclass 对应的 uclass_driver 的 id 是否匹配。
- udevice 会和 uclass 绑定。driver 会和 udevice 绑定。uclass_driver 会和 uclass 绑定。

uclass和udevice都是动态生成的。在解析fdt中的设备的时候，会动态生成udevice。 然后找到udevice对应的driver，通过driver中的uclass id得到uclass_driver id。从uclass链表中查找对应的uclass是否已经生成，没有生成的话则动态生成uclass

5.2.3 uboot DM 的初始化

Uboot DM 初始化的主要工作包括：

- DM 的初始化
    - 创建根设备 root 的 udevice，存放在 gd->dm_root 中。
      根设备其实是一个虚拟设备，主要是为 uboot 的其他设备提供一个挂载点。
    - 初始化 uclass 链表 gd->uclass_root
- DM 中 udevice 和 uclass 的解析
    - udevice 的创建和 uclass 的创建
    - udevice 和 uclass 的绑定
    - uclass_driver 和 uclass 的绑定
    - driver 和 udevice 的绑定
    - 部分 driver 函数的调用

DM 初始化的接口在 dm_init_and_scan 中。 可以发现在 uboot relocate 之前的 initf_dm 和之后的 initr_dm 都调用了这个函数,主要区别在于参数。 首先说明一下 dts 节点中的“u-boot,dm-pre-reloc”属性，当设置了这个属性时，则表示这个设备在 relocate 之前就需要使用。当 dm_init_and_scan 的参数为 true 时，只会对带有“u-boot,dm-pre-reloc”属性的节点进行解析。而当参数为 false 的时候，则会对所有节点都进行解析。 由于“u-boot,dm-pre-reloc”的情况比较少，

(目前只有：\uboot-imx\arch\arm\dts\fsl-imx8qxp.dtsi

```
tsens: thermal-sensor {
        compatible = "nxp,imx8qxp-sc-tsens";
        u-boot,dm-pre-reloc;
        /* number of the temp sensor on the chip */
        tsens-num = <1>;
```

```
                #thermal-sensor-cells = <1>;
        };使用)
```

所以这里只学习参数为 false 的情况。也就是 initr_dm 里面的 dm_init_and_scan(false)

```
|->dm_init(); // DM 的初始化
int dm_init(void)
{
        if (gd->dm_root) {
                // 根设备已经存在，说明 DM 已经初始化过了
        }
        // 初始化 uclass 链表
        INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);

        ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);
// DM_ROOT_NON_CONST 是指根设备 udevice，root_info 是表示根设备的设备信息
// device_bind_by_name 会查找和设备信息匹配的 driver，然后创建对应的 udevice 和 uclass 并进行绑定，最后放在 DM_ROOT_NON_CONST 中。
// device_bind_by_name 后续我们会进行说明，这里我们暂时只需要了解 root 根设备的 udevice 以及对应的 uclass 都已经创建完成。

        ret = device_probe(DM_ROOT_NON_CONST);
// 对根设备执行 probe 操作，
    // device_probe 后续再进行说明
/*
这里就完成的 DM 的初始化了
（1）创建根设备 root 的 udevice，存放在 gd->dm_root 中。
（2）初始化 uclass 链表 gd->uclass_root

*/

|->dm_scan_platdata(pre_reloc_only); //  从平台设备中解析 udevice 和 uclass,
|->dm_scan_fdt(gd->fdt_blob, pre_reloc_only); // 从 dtb 中解析 udevice 和 uclass
int dm_scan_fdt(const void *blob, bool pre_reloc_only)
{
```

```
// 此时传进来的参数
// parent=gd->dm_root，表示以 root 设备作为父设备开始解析
// blob=gd->fdt_blob，指定了对应的 dtb
// offset=0，从偏移 0 的节点开始扫描
// pre_reloc_only=0，不只是解析 relotion 之前的设备

        return dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);
}
```

| |->dm_scan_fdt_node

```
int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
                     bool pre_reloc_only)
{
        int ret = 0, err;
/*  以下步骤相当于是遍历每一个 dts 节点并且调用 lists_bind_fdt 对其进行解析 */
        for (offset = fdt_first_subnode(blob, offset);
        // 获得 blob 设备树的 offset 偏移下的节点的第一个子节点
             offset > 0;

             offset = fdt_next_subnode(blob, offset)) {
                        // 循环查找下一个子节点
                if (pre_reloc_only &&
                    !fdt_getprop(blob, offset, "u-boot,dm-pre-reloc", NULL))
                        continue;
                if (!fdtdec_get_is_enabled(blob, offset)) {
                        // 判断节点状态是否是 disable，如果是的话直接忽略
                        dm_dbg("   - ignoring disabled device\n");
                        continue;
                }
                // 解析绑定这个节点，dm_scan_fdt 的核心，下面具体分析
                err = lists_bind_fdt(parent, blob, offset, NULL);

                }
        }
```

```
}
```

lists_bind_fdt 是从 dtb 中解析 udevice 和 uclass 的核心。 其具体实现如下：

// parent 指定了父设备，通过 blob 和 offset 可以获得对应的设备的 dts 节点，对应 udevice 结构通过 devp 返回

```
int lists_bind_fdt(struct udevice *parent, const void *blob, int offset,
                   struct udevice **devp)
struct driver *driver = ll_entry_start(struct driver, driver); // 获取 driver table 地址
const int n_ents = ll_entry_count(struct driver, driver); // 获取 driver table 长度
name = fdt_get_name(blob, offset, NULL);
    dm_dbg("bind node %s\n", name);
    // 打印当前解析的节点的名称
for (entry = driver; entry != driver + n_ents; entry++) {
                        // 遍历 driver table 中的所有 driver
                        ret = driver_check_compatible(entry->of_match, &id,
                                                      compat);
                        if (!ret)
                                break;
                }
```

// 找到对应的 driver，调用 device_bind 进行绑定，会在这个函数中创建对应 udevice 和 uclass 并切进行绑定，后面继续说明

```
                ret = device_bind_with_driver_data(parent, entry, name,
                                                   id->data, offset, &dev);
```

// 将 udevice 设置到 devp 指向的地方中，进行返回

```
                        if (devp)
                                *devp = dev;
```

在 device_bind_common 中实现了 udevice 和 uclass 的创建和绑定以及一些初始化操作，这里专门学习一下 device_bind_common。 device_bind_common 的实现如下(去除部分代码)
driver/core/device.c

```
static int device_bind_common(struct udevice *parent, const struct driver *drv,
                              const char *name, void *platdata,
                              ulong driver_data, int of_offset,
                              uint of_platdata_size, struct udevice **devp)


// parent:父设备
```

```
// drv：设备对应的 driver
// name：设备名称
// platdata：设备的平台数据指针
// of_offset：在 dtb 中的偏移，即代表了其 dts 节点
// devp：所创建的 udevice 的指针，用于返回
{
// 获取 driver id 对应的 uclass，如果 uclass 原先并不存在，那么会在这里创建 uclass 并其 uclass_driver 进行绑定
    ret = uclass_get(drv->id, &uc);
// 分配一个 udevice
    dev = calloc(1, sizeof(struct udevice));
dev->platdata = platdata; // 设置 udevice 的平台数据指针
    dev->driver_data = driver_data;
    dev->name = name; // 设置 udevice 的 name
    dev->of_offset = of_offset; // 设置 udevice 的 dts 节点偏移
    dev->parent = parent; // 设置 udevice 的父设备
    dev->driver = drv; // 设置 udevice 的对应的 driver，相当于 driver 和 udevice 的绑定
    dev->uclass = uc; // 设置 udevice 的所属 uclass
dev->platdata = calloc(1,
                                        drv->platdata_auto_alloc_size);
                    // 为 udevice 分配平台数据的空间，由 driver 中的 platdata_auto_alloc_size 决定
// 添加到父设备的子设备链表中
    /* put dev into parent's successor list */
    if (parent)
            list_add_tail(&dev->sibling_node, &parent->child_head);
 // uclass 和 udevice 进行绑定，主要是实现了将 udevice 链接到 uclass 的设备链表中
    ret = uclass_bind_device(dev);
    if (ret)
            goto fail_uclass_bind;

    /* if we fail to bind we remove device from successors and free it */
    if (drv->bind) {
            // 执行 udevice 对应 driver 的 bind 函数
```

```
                ret = drv->bind(dev);
                if (ret)
                        goto fail_bind;
        }
        if (parent && parent->driver->child_post_bind) {
                // 执行父设备的 driver 的 child_post_bind 函数
                ret = parent->driver->child_post_bind(dev);
                if (ret)
                        goto fail_child_post_bind;
        }
        if (uc->uc_drv->post_bind) {
                // 执行所属 uclass 的 post_bind 函数
                ret = uc->uc_drv->post_bind(dev);
                if (ret)
                        goto fail_uclass_post_bind;
        }

        if (parent)
                dm_dbg("Bound device %s to %s\n", dev->name, parent->name);
        // 将 udevice 进行返回
        if (devp)
                *devp = dev;
    // 设置已经绑定的标志
        // 后续可以通过 dev->flags & DM_FLAG_ACTIVATED 或者 device_active 宏来判断设备是否已经被激活
    //上述就完成了 dtb 的解析，udevice 和 uclass 的创建，以及各个组成部分的绑定关系
    //注意，这里只是绑定，即调用了 driver 的 bind 函数，但是设备还没有真正激活，也就是还没有执行设备的 probe 函数。
        dev->flags |= DM_FLAG_BOUND;
```

上述就完成了 dtb 的解析，udevice 和 uclass 的创建，以及各个组成部分的绑定关系。 注意，这里只是绑定，即调用了 driver 的 bind 函数，但是设备还没有真正激活，也就是还没有执行设备的 probe 函数。

5.2.4 DM 工作流程

经过前面的 DM 初始化以及设备解析之后，我们只是建立了 udevice 和 uclass 之间的绑定关系。但是此时 udevice 还没有被 probe，其对应设备还没有被激活。 激活一个设备主要是通过 device_probe 函数，所以在介绍 DM 的工作流程前，先说明 device_probe 函数。

主要工作归纳如下：

- 分配设备的私有数据
- 对父设备进行 probe
- 执行 probe device 之前 uclass 需要调用的一些函数
- 调用 driver 的 ofdata_to_platdata，将 dts 信息转化为设备的平台数据
- 调用 driver 的 probe 函数
- 执行 probe device 之后 uclass 需要调用的一些函数

driver/core/device.c

```
int device_probe(struct udevice *dev)
{
drv = dev->driver; // 获取这个设备对应的 driver
// 为设备分配私有数据
dev->priv = alloc_priv(drv->priv_auto_alloc_size, drv->flags);
// 为设备所属 uclass 分配私有数据
size = dev->uclass->uc_drv->per_device_auto_alloc_size;
dev->flags |= DM_FLAG_ACTIVATED; // 设置 udevice 的激活标志
ret = uclass_pre_probe_device(dev); // uclass 在 probe device 之前的一些函数的调用
// 调用 driver 中的 ofdata_to_platdata 将 dts 信息转化为设备的平台数据
                ret = drv->ofdata_to_platdata(dev);
// 调用 driver 的 probe 函数，到这里设备才真正激活了
                ret = drv->probe(dev);
}
```

通过 uclass 来获取一个 udevice 并且进行 probe 有如下接口：

driver/core/uclass.c

```
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)  //通过索引从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_name(enum uclass_id id, const char *name,
                struct udevice **devp) //通过设备名从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp) //通过序号从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
                struct udevice **devp) //通过 dts 节点的偏移从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
                const char *name, struct udevice **devp) //通过设备的“phandle”属性从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_first_device(enum uclass_id id, struct udevice **devp) //从 uclass 的设备链表中获取第一个 udevice，并且进行 probe
int uclass_next_device(struct udevice **devp) //从 uclass 的设备链表中获取下一个 udevice，并且进行 probe
```

这些接口主要是获取设备的方法上有所区别，但是 probe 设备的方法都是一样的，都是通过调用 uclass_get_device_tail->device_probe 来 probe 设备的。以 uclass_get_device 为例:

```
/*  通过索引从 uclass 中获取 udevice*/

int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)

{

        ret = uclass_find_device(id, index, &dev); //通过索引从 uclass 的设备链表中获取对应的 udevice

        return uclass_get_device_tail(dev, ret, devp); // 调用 uclass_get_device_tail 进行设备的 get，最终会调用 device_probe 来对设备进行 probe

}

int uclass_get_device_tail(struct udevice *dev, int ret,

                                        struct udevice **devp)

{

        ret = device_probe(dev); // 调用 device_probe 对设备进行 probe，这个函数在前面说明过了

        *devp = dev;

}
```

5.2.5 DM 工作流程实例 DM-GPIO

Drivers\gpio\gpio-uclass.c 定义了一个 uclass_driver

```
UCLASS_DRIVER(gpio) = {
        .id                     = UCLASS_GPIO,
```

```
        .name               = "gpio",
        .flags              = DM_UC_FLAG_SEQ_ALIAS,
        .post_probe         = gpio_post_probe,
        .pre_remove         = gpio_pre_remove,
        .per_device_auto_alloc_size = sizeof(struct gpio_dev_priv),
};
```

GPIO DTS support: \uboot-imx\arch\arm\dts\fsl-imx8qxp.dtsi, fsl-imx8qxp-mek.dts

```
gpio0: gpio@5d080000 {
                        compatible = "fsl,imx8qm-gpio", "fsl,imx35-gpio";
….
                        power-domains = <&pd_lsio_gpio0>;
            };
&gpio0 {
            status = "okay";
};
gpio1/2/3/4
```

i.MX GPIO 设备驱动：

```
static const struct udevice_id mxc_gpio_ids[] = {
            { .compatible = "fsl,imx35-gpio" },};


U_BOOT_DRIVER(gpio_mxc) = {
            .name       = "gpio_mxc",
            .id         = UCLASS_GPIO,
            .ops        = &gpio_mxc_ops,
            .probe      = mxc_gpio_probe,
            .priv_auto_alloc_size = sizeof(struct mxc_bank_info),
            .of_match = mxc_gpio_ids,
            .bind       = mxc_gpio_bind,
};
```

在 DM 初始化的过程中 uboot 自己创建对应的 udevice 和 uclass，并自己实现将 udevice 绑定到对应的 uclass 中。

udevice 的 probe 则通过 gpio core API 调用实现：

gpio core API 接口说明如下：其中既提供了兼容老版本的接口，也提供了 DM 框架下的接口。

- DM 框架下的接口
  注意，外部通过 gpio_desc 来描述一个 GPIO，所以这些接口都是以 gpio_desc 作为参数
  - gpio_request_by_name
    int gpio_request_by_name(struct udevice *dev, const char *list_name, int index, struct gpio_desc *desc, int flags)
    通过对应的 udevice 找到其 dtsi 节点中属性名为 list_name 的 GPIO 属性并转化为 gpio_desc，并且 request。
  - gpio_request_by_name_nodev
    int gpio_request_by_name_nodev(const void *blob, int node, const char *list_name, int index, struct gpio_desc *desc, int flags)
    通过对应的 dtsi 节点中属性名为 list_name 的 GPIO 属性并转化为 gpio_desc，并且 request。
  - dm_gpio_request
    int dm_gpio_request(struct gpio_desc *desc, const char *label)
    申请 gpio_desc 描述的 GPIO
  - dm_gpio_get_value
    int dm_gpio_get_value(const struct gpio_desc *desc)
    获取 gpio_desc 描述的 GPIO 的值
  - dm_gpio_set_value
    int dm_gpio_set_value(const struct gpio_desc *desc, int value)
    设置 gpio_desc 描述的 GPIO 的值
  - dm_gpio_set_dir_flags
    int dm_gpio_set_dir_flags(struct gpio_desc *desc, ulong flags)
    设置 gpio_desc 描述的 GPIO 的输入输出方向，带标志
  - dm_gpio_set_dir
    int dm_gpio_set_dir(struct gpio_desc *desc)
    设置 gpio_desc 描述的 GPIO 的输入输出方向
  - dm_gpio_is_valid
    static inline bool dm_gpio_is_valid(const struct gpio_desc *desc)
    判断 gpio_desc 是否可用
- 老接口：
  这些接口是为了兼容老版本的接口，注意，但是最终还是会调用 DM 框架下的接口

- gpio_request
  int gpio_request(unsigned gpio, const char *label)
  申请一个 GPIO
- gpio_direction_input
  int gpio_direction_input(unsigned gpio)
  设置某个 GPIO 为输入
- gpio_direction_output
  int gpio_direction_output(unsigned gpio, int value)
  设置某个 GPIO 为输出
- gpio_get_value
  int gpio_get_value(unsigned gpio)
  获取某个 GPIO 上的值
- gpio_set_value
  int gpio_set_value(unsigned gpio, int value)
  设置 GPIO 的值

实例说明如下：

```
Common/board_r.c
init_sequence_r
|->board_init
|   |->board_gpio_init(board\freescale\imx8qxp_mek\imx8qxp_mek.c)
dm_gpio_lookup_name("gpio@1a_3", &desc);\\调用 i2c-gpio,dts 设置如下
pca9557_a: gpio@1a {
                                        compatible = "nxp,pca9557";
                                        reg = <0x1a>;
                                        gpio-controller;
                                        #gpio-cells = <2>;
                                    };
dm_gpio_request(&desc, "bb_per_rst_b");
dm_gpio_set_dir_flags(&desc, GPIOD_IS_OUT);
dm_gpio_set_value(&desc, 0);
        udelay(50);
dm_gpio_set_value(&desc, 1);
```

这个是使用 DM 接口调用 GPIO 的方式，其中

```
dm_gpio_lookup_name
| -> uclass_first_device
|  | -> uclass_find_first_device
|  |  | -> uclass_get_device_tail
|  |  |   |-> device_probe  //实现了设备的激活
```

使用传统方式调用如下：

```
mx8_iomux_setup_multiple_pads(board_gpios, ARRAY_SIZE(board_gpios));
        /* enable i2c port expander assert reset line */
gpio_request(IOEXP_RESET, "ioexp_rst");
gpio_direction_output(IOEXP_RESET, 1);
```

其中 gpio_request

```
|->gpio_to_device(->uclass_first_device->uclass_get_device_tail->device_probe)
        | |->dm_gpio_request
```

5.3 Uboot 目录 结构

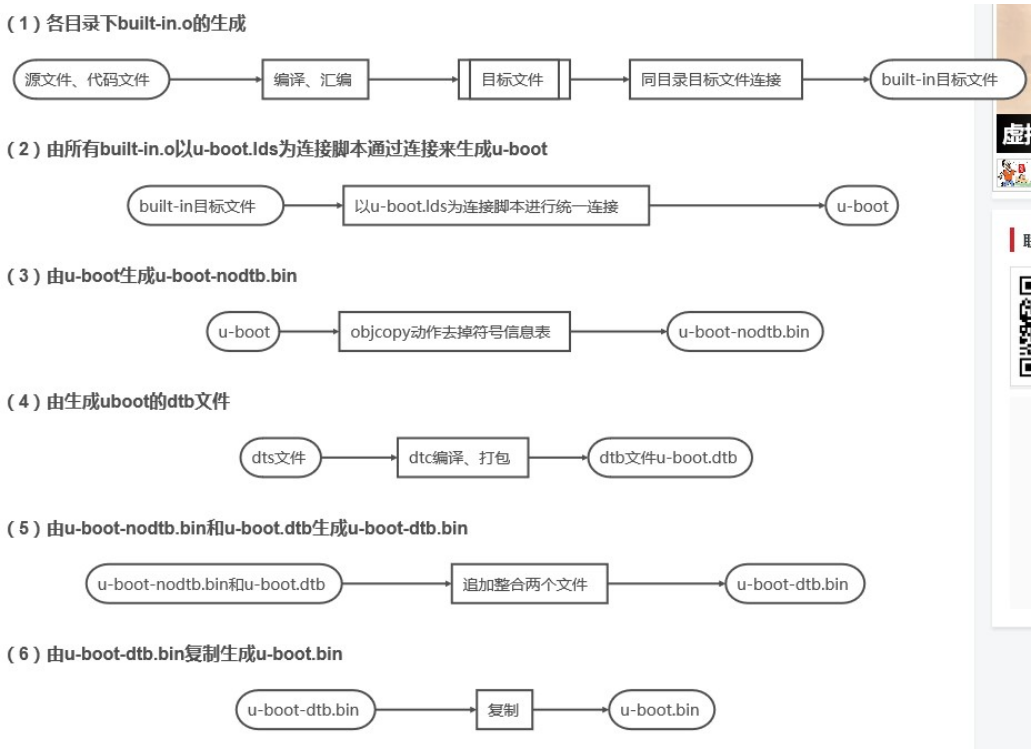
一般定制需要使用的目录有源代码如下：

```
\uboot-imx
|-> arch
|    |->arm
|    |   |->cpu
|    |   |   |->armv8
|    |   |   |->u-boot.lds
|    |   |->dts
|    |   |   |->fsl-imx8-ca35.dtsi, fsl-imx8dx.dtsi , fsl-imx8dxp.dtsi ,fsl-imx8qxp.dtsi, fsl-imx8qxp-mek.dts
|    |   |-> mach-imx
|    |   |   |->imx8
|->board
|    |->freescale
```

```
|    |    |->imx8qxp_mek
|    |    |    |->imx8qxp_mek.c
|->configs
|    |->imx8qxp_mek_defconfig, imx8qxp_mek_fspi_defconfig
|->common
|   |->board_f.c, board_r.c
|->drivers
|    |->gpio
|    |    |->gpio-uclass.c, mxc_gpio.c,pca953x_gpio.c
|    |->i2c
|    |    |->i2c-uclass.c, imx_lpi2c
|    |    |    |->muxes
|    |    |    |    |->pca954x
|    |->net
|    |->power
|    |    |->domain
|    |    |    |->imx8-power-domain.c, power-domain-uclass.c
```

5.4 Uboot 编译

5.4.1 编译流程



5.4.2 生成文件：

文件	说明
u-boot	初步链接后得到的 uboot 文件
u-boot-nodtb.bin	在 u-boot 的基础上，经过 objcopy 去除符号表信息之后的可执行程序
u-boot.dtb	dtb 文件
u-boot-dtb.bin	将 u-boot-nodtb.bin 和 u-boot.dtb 打包在一起的文件
u-boot.bin	在需要 dtb 的情况下，直接由 u-boot-dtb.bin 复制而来，也就是编译 u-boot 的最终目标
u-boot.lds	uboot 的连接脚本
System.map	连接之后的符号表文件
u-boot.cfg	由 uboot 配置生成的文件

5.5 Uboot 初始化流程

板级初始化的流程

_main

|->board_init_f_alloc_reserve :堆栈、GD、early malloc空间的分配
|->board_init_f_init_reserve:堆栈、GD、early malloc空间的初始化
|->board_init_f:uboot relocate前的板级初始化以及relocate的区域规划
|->relocate_code、relocate_vectors:进行uboot和异常中断向量表的重定向 ,旧堆栈的清空
|->board_init_r:uboot relocate后的板级初始化
|->run_main_loop:进入命令行状态，等待终端输入命令以及对命令进行处理

如下详细代码说明:

(\arch\arm\cpu\lib\vector.S): _start

_start:

…

|-> b reset

arch\arm\cpu\armv8\start.S

reset:

….

/* Processor specific initialization */

| |-> bl lowlevel_init

| |->bl _main

Arch\arm\lib\crt0_64.S

ENTRY(_main)

板级初始化的流程

| | |->board_init_f_alloc_reserve

| | |->board_init_f_init_reserve

| | |->board_init_f

Common\board_f.c

if (initcall_run_list(init_sequence_f))

fdtdec_setup, /* 获取 dtb 的地址，并且验证 dtb 的合法性，之前已经说明*/

arch_cpu_init, /* basic arch cpu dependent setup */

/* include Open IPC channel , power on SMMU*/

initf_dm /*此处仅初始化 relocal 之前的驱动*/

```
        board_early_init_f/*早期串口相关 PD 和 IOMUX 设置， 事实上因为以下 serial_init 时并没有 dts 支持，所以这儿没有用处*/
        timer_init,                     /* initialize timer */
        get_clocks, /*get sdhc clock to gd*/
        env_init,                       /* initialize environment */
       init_baud_rate,                  /* initialze baudrate settings */
        serial_init,                    /* serial communications setup *//*此处的问题在于由于 dts 还没有加载，实际上使用 DM 初始化会失败？ */
        console_init_f,                 /* stage 1 init of console */
       display_options,  /* say that we are here */
        display_text_info,              /* show debugging info if required */
       show_board_info
        init_func_i2c
        dram_init /*initial DDR size to gd*/
    /*
         * Now that we have DRAM mapped and working, we can
         * relocate the code and continue running from DRAM.
         *
         * Reserve memory at end of RAM for (top down in that order):
         *  - area that won't get touched by U-Boot and Linux (optional)
         *  - kernel log buffer
         *  - protected RAM
         *  - LCD framebuffer
         *  - monitor code
         *  - board info struct
         */
        setup_dest_addr,
        reserve_round_4k /* Round memory pointer down to next 4 kB limit */
        reserve memory for serveral driver
        setup_machine, /*set board machine id*/
        reserve_global_data, /*reserve memory for gd*/
        reserve_fdt, /* 为 dtb 分配新的内存地址空间 */
       setup_dram_config,
        show_dram_config,
       reloc_fdt, /* relocate dtb  */
```

```
        setup_reloc,
| | |->relocate_code
| | |->board_init_r
Common\board_r.c
if (initcall_run_list(init_sequence_r))
                        hang();
static init_fnc_t init_sequence_r[] = {
…
        initr_caches,   /*caches initial*/
        initr_reloc, /* tell others: relocation done */
        initr_caches，  /*caches enabled*/
         initr_reloc_global_data
        initr_noncached, /*initial noncached memory*/
        initr_dm, /*之前已经分析*/
…
        board_init,         /* Setup chipselects */
board/freescale/imx8qxp_mek/imx8qxp_mek.c
board_gpio_init();/*之前已经分析*/
    //this function explain two way to operate the GPIO, one is use DM interface to access the i2c to gpio chipset to pull a gpio, one way is hard coding to pull a i.mx8 itself gpio
    setup_fec(CONFIG_FEC_ENET_DEV);/*事实上仅 reset eth phy*/
    /* ENET0 connects AR8031 on CPU board, ENET1 connects to base board and MUX with ESAI, default is ESAI */
    #define CONFIG_FEC_ENET_DEV 0
      |-> enet_device_phy_reset
    //reset the enet phy by i2c to gpio chipset
    ret = dm_gpio_lookup_name("gpio@1a_4", &desc);
                    if (ret)
                                return;


                    ret = dm_gpio_request(&desc, "enet0_reset");
                    if (ret)
                                return;


                    dm_gpio_set_dir_flags(&desc, GPIOD_IS_OUT);
```

```
                dm_gpio_set_value(&desc, 0);
                udelay(50);
                dm_gpio_set_value(&desc, 1);
setup_typec();
        set_cpu_clk_info, /* Setup clock information , get the arm clock to gd*/
        initr_serial, /*DM 串口设备 probe*/
         |-> serial_initialize-> serial_init-> serial_find_console_or_panic
if (CONFIG_IS_ENABLED(OF_CONTROL) && blob) {
                /* Check for a chosen console */
                node = fdtdec_get_chosen_node(blob, "stdout-path");
                if (node < 0) {
                        const char *str, *p, *name;

                        /*
                         * Deal with things like
                         *      stdout-path = "serial0:115200n8";
                         *
                         * We need to look up the alias and then follow it to
                         * the correct node.
                         */
                        str = fdtdec_get_chosen_prop(blob, "stdout-path");
                        if (str) {
                                p = strchr(str, ':');
                                name = fdt_get_alias_namelen(blob, str,
                                                p ? p - str : strlen(str));
                                if (name)
                                        node = fdt_path_offset(blob, name);
                        }
                }
                if (node < 0)
                        node = fdt_path_offset(blob, "console");
                if (!uclass_get_device_by_of_offset(UCLASS_SERIAL, node,
                                                &dev)) {
                        gd->cur_serial_dev = dev;
```

```
\uboot-imx\arch\arm\dts\ fsl-imx8qxp-mek.dts
chosen {
                bootargs = "console=ttyLP0,115200 earlycon=lpuart32,0x5a060000,115200";
                stdout-path = &lpuart0;
        };
&lpuart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpuart0>;
        status = "okay";
};
pinctrl_lpuart0: lpuart0grp {
                        fsl,pins = <
                                SC_P_UART0_RX_ADMA_UART0_RX 0x06000020
                                SC_P_UART0_TX_ADMA_UART0_TX 0x06000020
                        >;
                };
initr_announce, /*print ("Now running in RAM - U-Boot at: %08lx\n", gd->relocaddr);*/
…
power_init_board, /*由于 i.MX8X/QM 的 pmic initial move to SC firmware, so its empty*/
…
#ifdef CONFIG_CMD_NAND
        initr_nand,
#endif
…
#ifdef CONFIG_GENERIC_MMC
        initr_mmc,
        |-> mmc_initialize-> mmc_probe->
static int mmc_probe(bd_t *bis)
{
        int ret, i;
        struct uclass *uc;
        struct udevice *dev;

        ret = uclass_get(UCLASS_MMC, &uc);
```

```
        if (ret)
                return ret;

        /*
         * Try to add them in sequence order. Really with driver model we
         * should allow holes, but the current MMC list does not allow that.
         * So if we request 0, 1, 3 we will get 0, 1, 2.
         */
        for (i = 0; ; i++) {
                ret = uclass_get_device_by_seq(UCLASS_MMC, i, &dev);
                if (ret == -ENODEV)
                        break;
        }
        uclass_foreach_dev(dev, uc) {
                ret = device_probe(dev);
                if (ret)
                        printf("%s - probe failed: %d\n", dev->name, ret);
        }

        return 0;
}
```

/*所以默认 emmc 初始化将使用 DM 做设备 probe，所以定义在 board\freescale\imx8qxp_mek\imx8qxp_mek.c 中的函数 board_mmc_init，board_mmc_getcd，iomux structure emmc0, usdhc1_sd 最终不会调用 到，而是使用 dts 设置：arch/arm/dts/fsl-imx8qxp-mek.dts

```
pinctrl_usdhc1/__100mhz/_200mhz
pinctrl_usdhc2_gpio/usdhc2/_100mhz/_200mhz
&usdhc1 {
        pinctrl-names = "default", "state_100mhz", "state_200mhz";
        pinctrl-0 = <&pinctrl_usdhc1>;
…
        bus-width = <8>;
        non-removable;
        status = "okay";
};
```

```
&usdhc2 {
        pinctrl-names = "default", "state_100mhz", "state_200mhz";
        pinctrl-0 = <&pinctrl_usdhc2>, <&pinctrl_usdhc2_gpio>;
…
        bus-width = <4>;
        cd-gpios = <&gpio4 22 GPIO_ACTIVE_LOW>;
        wp-gpios = <&gpio4 21 GPIO_ACTIVE_HIGH>;
        vmmc-supply = <&reg_usdhc2_vmmc>;
        status = "okay";
};
  所以修改 emmc/sd 相关的 iomux 或 gpio ，可以在 DTS 中直接修改*/
    initr_dataflash,
#endif
    initr_env, /* initialize environment  ->set_default_env(NULL);*/
    …
      stdio_add_devices, /*此处可以预先初始化设备 probe，但是目前 i.mx8qxp 平台没有用到*/
    console_init_r,             /* fully init console as a device */
#ifdef CONFIG_DISPLAY_BOARDINFO_LATE
    show_board_info,
/*
 * If the root node of the DTB has a "model" property, show it.
 * Then call checkboard().
 */
#endif
    …
    interrupt_init,
#if defined(CONFIG_ARM) || defined(CONFIG_AVR32)
    initr_enable_interrupts,
#endif
    …

#ifdef CONFIG_CMD_NET
    initr_ethaddr, /*get ethaddr from gd*/
#endif
```

```
#ifdef CONFIG_BOARD_LATE_INIT
        board_late_init,
        setenv("board_name", "MEK");
        setenv("board_rev", "iMX8QXP");
/*customer can change to theirselves name*/
          |-> board_late_mmc_env_init
          |   |-> mmc_get_env_dev /*call sc_misc_get_boot_dev to get the boot dev*/
            |   |->check_mmc_autodetect /*check does uboot environment have “mmcautodetect”*/
            setenv_ulong("mmcdev", dev_no);

                    /* Set mmcblk env */
                    sprintf(mmcblk, "/dev/mmcblk%dp2 rootwait rw",
                                mmc_map_to_kernel_blk(dev_no));
                    setenv("mmcroot", mmcblk);

                    sprintf(cmd, "mmc dev %d", dev_no);
                    run_command(cmd, 0);
#endif
#ifdef CONFIG_FSL_FASTBOOT
        initr_fastboot_setup,
#endif
…
#ifdef CONFIG_CMD_NET
        INIT_FUNC_WATCHDOG_RESET
        initr_net,
        |-> eth_initialize
        |   |->eth_common_init /*此处最终配置 phy 芯片 api*/
        |   |   |->phy_init
        |   |   |   |-> phy_atheros_init();
        |   |   |   |    |-> phy_register(&AR8031_driver);
static struct phy_driver AR8031_driver =  {
        .name = "AR8031/AR8033",
        .uid = 0x4dd074,
        .mask = 0xffffffef,
```

```
        .features = PHY_GBIT_FEATURES,
        .config = ar8031_config,
        .startup = genphy_startup,
        .shutdown = genphy_shutdown,
};
    |   |->uclass_first_device(UCLASS_ETH, &dev); /*此处 probe eth 设备*/
        |   |->eth_set_dev /*此处 probe eth 设备*/
        /*相应 dts 设置
        &fec1 {
         pinctrl-names = "default";
         pinctrl-0 = <&pinctrl_fec1>;
         phy-mode = "rgmii";
         phy-handle = <&ethphy0>;
         fsl,ar8031-phy-fixup;
         fsl,magic-packet;
         status = "okay";

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

                ethphy0: ethernet-phy@0 {
                        compatible = "ethernet-phy-ieee802.3-c22";
                        reg = <0>;
                };

                ethphy1: ethernet-phy@1 {
                        compatible = "ethernet-phy-ieee802.3-c22";
                        reg = <1>;
                };
         };
        };

        &fec2 {
```

```
 pinctrl-names = "default";
 pinctrl-0 = <&pinctrl_fec2>;
 phy-mode = "rgmii";
 phy-handle = <&ethphy1>;
 fsl,ar8031-phy-fixup;
 fsl,magic-packet;
 status = "okay";
};
            pinctrl_fec1: fec1grp {
                        fsl,pins = <
                                    SC_P_ENET0_MDC_CONN_ENET0_MDC                   0x06000048
…
                                    SC_P_ENET0_RGMII_RXD3_CONN_ENET0_RGMII_RXD3     0x06000048
                        >;
            };

            pinctrl_fec2: fec2grp {
                        fsl,pins = <
                                    SC_P_ESAI0_SCKR_CONN_ENET1_RGMII_TX_CTL         0x06000048
…
                                    SC_P_ESAI0_TX1_CONN_ENET1_RGMII_RXD3            0x06000048
                        >;
            };
*/
/*
 * Typically this will just store a device pointer.
 * In case it was not probed, we will attempt to do so.
 * dev may be NULL to unset the active device.
 */
void eth_set_dev(struct udevice *dev)
{
 if (dev && !device_active(dev)) {
            eth_errno = device_probe(dev);
            if (eth_errno)
```

```
                dev = NULL;
            }

            eth_get_uclass_priv()->current = dev;
        }
    #endif
    …
        run_main_loop,
    };
```

5.6 uboot 定制

5.6.1 修改 DDR 配置

由于 DDR 的参数配置已经移到 SC firmware 中，所以 uboot 中唯一要注意的地方是如何修改了 DDR 的尺寸太小，相应需要修改：

\common\board_f.c\init_sequence_f

|->dram_init(arch\arm\mach-imx\imx8\cpu.c)

| |->get_owned_memreg(mr, &start, &end);

Which will call macro which define in include\configs\imx8qxp_mek.h

```
#define CONFIG_SYS_SDRAM_BASE                   0x80000000
```

8000_0000	FFFF_FFFF (SCU - DFFF_FFFF)	DDR Address	2GB (SCU - 1.5GB)	DDR Main memory Note: SCU data structures are limited to first 1.5GB

```
#define PHYS_SDRAM_1                    0x80000000 //片选 1 起始地址
```

Start Address	End Address	Region	Size	Allocation
8_8000_0000	B_FFFF_FFFF	DDR	14GB	DDR Main memory

```
#define PHYS_SDRAM_2                    0x880000000//片选 2 起始地址
#define PHYS_SDRAM_1_SIZE               0x80000000      /* 2 GB */ i.MX8QXP MEK 使用 3GB LPDDR4,片选 1 大小 2GB
/* LPDDR4 board total DDR is 3GB */
#define PHYS_SDRAM_2_SIZE               0x40000000      /* 1 GB */ i.MX8QXP MEK 使用 3GB LPDDR4,片选 2 大小 1GB
```

至于 DTS 中的 memory size，理论上不需要修改，uboot 传给内核的参量优先级更高：

\uboot-imx\arch\arm\dts\fsl-imx8qxp.dtsi

```
memory@80000000 {
                device_type = "memory";
                reg = <0x00000000 0x80000000 0 0x40000000>; //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。
                    /* DRAM space - 1, size : 1 GB DRAM */
        };
//以下为CMA的大小配置，事实上内核DTS会重写掉。
        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;
                /* global autoconfigured region for contiguous allocations */
                linux,cma {
                        compatible = "shared-dma-pool";
                        reusable;
                        size = <0 0x28000000>;
                        alloc-ranges = <0 0x80000000 0 0x80000000>;
                        linux,cma-default;
                };
        };
```

但是需要注意一下在：

\uboot-imx\configs\imx8qxp_mek_defconfig 中定义 了：

```
CONFIG_IMX_BOOTAUX=y
CONFIG_BOOTAUX_RESERVED_MEM_BASE=0x88000000
CONFIG_BOOTAUX_RESERVED_MEM_SIZE=0x08000000
```

这个在\uboot-imx\arch\arm\mach-imx\imx8\cpu.c 中：

```
int ft_system_setup(void *blob, bd_t *bd)
{
#if (CONFIG_BOOTAUX_RESERVED_MEM_SIZE != 0x00)
   int off;
   off = fdt_add_mem_rsv(blob, CONFIG_BOOTAUX_RESERVED_MEM_BASE,
                                CONFIG_BOOTAUX_RESERVED_MEM_SIZE);
            if (off < 0)
                    printf("Failed  to reserve memory for bootaux: %s\n",
                            fdt_strerror(off));
#endif
```

会在内核 DTS 中增加这部分的 reserved memory: 内核 DTS 有说明：
\kernel\arch\arm64\boot\dts\freescale\fsl-imx8dx.dtsi

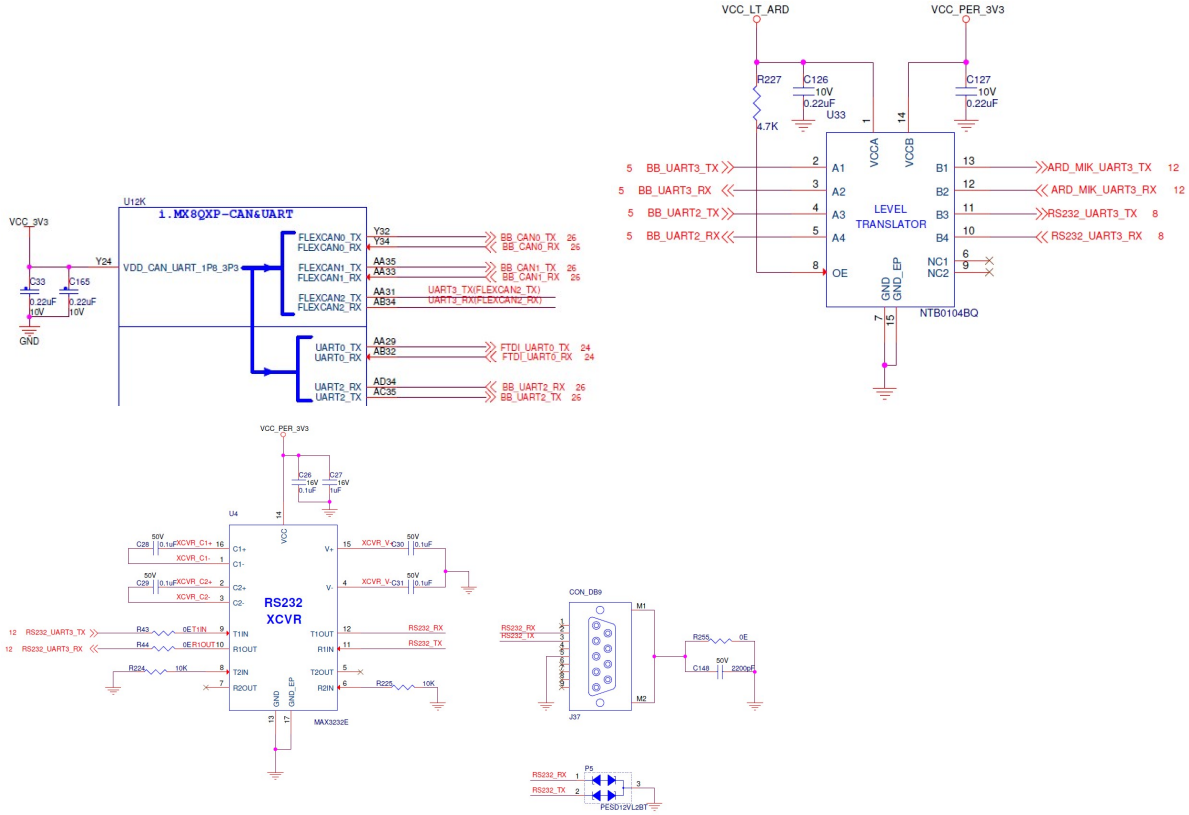
```
reserved-memory {…
/*
             * reserved-memory layout
```

```
        * 0x8800_0000 ~ 0x8FFF_FFFF is reserved for M4
        * Shouldn't be used at A core and Linux side.
        */
```

这个 128MB 是为 M4 预留的，如果不需要，可以在 configure 中注掉。

5.6.2 修改调试串口

一般情况下，不建议修改 A35 核 的调试串口（ATF/Uboot/Kernel 均使用同一下调试串口，与 SC firmware 使用的串口不同），以下示例如何修改为 UART2,是 i.MX8QXP MEK 底板上的 J37 DB9 接口：



Common\board_f.c

```
init_sequence_f
|->board_early_init_f
/board/freescale/imx8qxp_mek/imx8qxp_mek.c
  int board_early_init_f(void)
{
…
```

```
        /* Power up UART0 */
…
        /* Set UART0 clock root to 80 MHz */
…
        /* Enable UART0 clock root */
…
#if 1 //add UART2 initial support
/* Power up UART2 */
        sciErr = sc_pm_set_resource_power_mode(ipcHndl, SC_R_UART_2, SC_PM_PW_MODE_ON);
        if (sciErr != SC_ERR_NONE)
                return 0;
        /* Set UART2 clock root to 80 MHz */
        sc_pm_clock_rate_t rate2 = 80000000;
        sciErr = sc_pm_set_clock_rate(ipcHndl, SC_R_UART_2, 2, &rate2);
        if (sciErr != SC_ERR_NONE)
                return 0;
        /* Enable UART1 clock root */
        sciErr = sc_pm_clock_enable(ipcHndl, SC_R_UART_2, 2, true, false);
        if (sciErr != SC_ERR_NONE)
                return 0;
    LPCG_AllClockOn(LPUART_2_LPCG);
#endif
        setup_iomux_uart();

        return 0;
}
static iomux_cfg_t uart0_pads[] = {
…
};
//add uart2 support
static iomux_cfg_t uart2_pads[] = {
        SC_P_UART2_RX | MUX_PAD_CTRL(UART_PAD_CTRL),
        SC_P_UART2_TX | MUX_PAD_CTRL(UART_PAD_CTRL),
```

```
};
static void setup_iomux_uart(void)
{
        imx8_iomux_setup_multiple_pads(uart0_pads, ARRAY_SIZE(uart0_pads));
        imx8_iomux_setup_multiple_pads(uar2_pads, ARRAY_SIZE(uart2_pads));

}
…
void board_quiesce_devices()
{
        const char *power_on_devices[] = {
                "dma_lpuart0",
                "dma_lpuart2",  /*johnli add 此处关闭 uart2 PM，要不然在内核 PM 初始化时会挂死*/
…
        };

        power_off_pd_devices(power_on_devices, ARRAY_SIZE(power_on_devices));
}
…
Arch\arm\mach-imx\imx8\clock.c
unsigned int mxc_get_clock(enum mxc_clock clk)
{
…
        switch (clk) {
        case MXC_UART_CLK:
                #if 0 /*johnli add*/
                err = sc_pm_get_clock_rate((sc_ipc_t)gd->arch.ipc_channel_handle,
                                        SC_R_UART_0, 2, &clkrate);
                #else
                err = sc_pm_get_clock_rate((sc_ipc_t)gd->arch.ipc_channel_handle,
                                        SC_R_UART_2, 2, &clkrate); //此外是 hard coding
                #endif
```

在 initf_dm 之后,dm 设备驱动绑定成功，可以访问 dtb 来获得设备平台数据，绑定设备，串口是使用这种方式来初始化的：

```
init_sequence_f
|-> serial_init
| |-> serial_find_console_or_panic(); // 调用 serial_find_console_or_panic 进行作为 console 的 serial 的初始化
/* Check for a chosen console */
                node = fdtdec_get_chosen_node(blob, "stdout-path");
node = fdt_path_offset(blob, "console");
                // 这里调用 uclass_get_device_by_of_offset，通过 dts 节点的偏移从 uclass 的设备链表中获取 udevice，并且进行 probe。
// 注意，是在这里完成设备的 probe 的！！！
if (!uclass_get_device_by_of_offset(UCLASS_SERIAL, node,
                                                &dev)) {
// 将 udevice 存储在 gd->cur_serial_dev，后续 uclass 中可以直接通过 gd->cur_serial_dev 获取到对应的设备并且进行操作
// 但是注意，这种并不是通用做法！！！
// o      可以通过先从 root_uclass 链表中提取对应的 uclass，然后通过 uclass->uclass_driver->ops 来进行接口调用，这种方法比较具有通用性。
// o      可以通过调用 uclass 直接 expert 的接口，不推荐，但是 serial-uclass 使用的是这种方式。
```

所以 DTS 的修改方法是：

```
/arch/arm/dts/ fsl-imx8qxp-mek.dts
chosen {
                bootargs = "console=ttyLP2,115200 earlycon=lpuart32,0x5a080000,115200";
                stdout-path = &lpuart2;
        };
        /*johnli add*/

                pinctrl_lpuart2: lpuart2grp {
                        fsl,pins = <
                                SC_P_UART2_RX_ADMA_UART2_RX 0x06000020
                                SC_P_UART2_TX_ADMA_UART2_TX 0x06000020
                        >;
                };
&lpuart2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_lpuart2>;
        status = "okay";
};
```

5.6.3 DM I2C 操作 config

由于 I2C to PMIC 的初始化已经移到 SC firmware 中，所以 uboot 中使用到 I2C 的机会比较少，在 i.MX8QXP MEK 板中，在 uboot 中基本有两个地方使用到了，一个是使用 pca9557 的 I2C to GPIO 的 GPIO 操作，这个的调用方式是通过 GPIO 的 DM 接口调用到 pca9557 的驱动，然后 pca9557 的驱动再调用 I2C 的 DM 接口。

所以一般说来，如果需要在 uboot 中实现一个 I2C 设备驱动，应该如同 pca9557 一样：

1. 在\uboot-imx\configs\imx8qxp_mek_defconfig 中，打开编译宏，也可以用 make menuconfig 增加

```
CONFIG_DM_PCA953X=y
```

2. 在 DTS 中增加此驱动的平台数据：arch\arm\dts\fsl-imx8qxp-mek.dts

```
&i2c1 {
…
pca9557_a: gpio@1a {
                compatible = "nxp,pca9557";
                reg = <0x1a>;
                gpio-controller;
                #gpio-cells = <2>;
            };
…
}
```

3. 其驱动源代码在：

```
Driver\gpio\makefile
  obj-$(CONFIG_DM_PCA953X)    += pca953x_gpio.o
drivers\gpio\pca953x_gpio.c
  static const struct udevice_id pca953x_ids[] = {
{ .compatible = "nxp,pca9557", .data = OF_953X(8, 0), },
}
```

此驱动调用 dm_i2c_read/write

？另一个就是 type C 接口的 i2c 的接口，这个是直接调用 i2c 的 DM 接口，并且 实现在 \board\freescale\imx8qxp_mek\imx8qxp_mek.c 中的，如果客户有其它的 i2c 设备需要在 uboot 中访问，可以参考这个代码：

```
common\board_r.c\init_sequence_r->board\freescale\imx8qxp_mek\imx8qxp_mek.c\board_init->
setup_typec->tcpc_init
ret = uclass_get_device_by_seq(UCLASS_I2C, 1, &bus); \*得到 i2c1,事实上 imx8qxp mek 板上*\
        |-> uclass_find_device_by_seq
        |   |-> uclass_get_device_tail
        |   |   |->device_probe/*此处 probe i2c 设备*/

        ret = dm_i2c_probe(bus, chip, 0, &tcpc_i2c_dev);
        if (ret) {
                printf("%s: Can't find device id=0x%x\n",
                        __func__, chip);
                return;
        }
…dm_i2c_read…
…dm_i2c_write…
```

5.6.4 Ethernet 定制

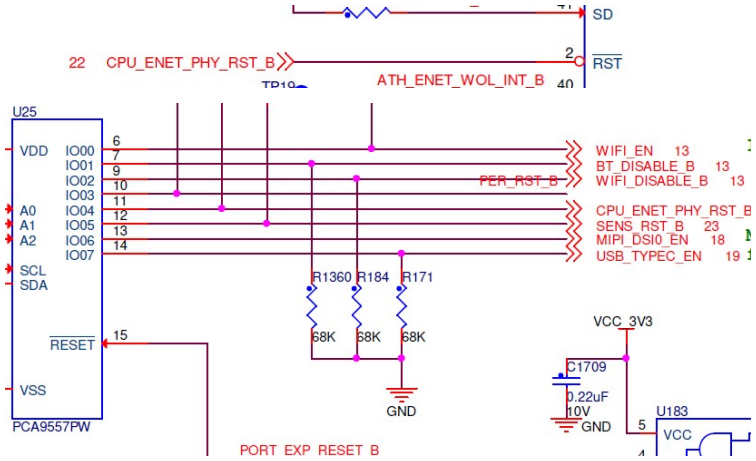
根据 uboot 初始化调用分析：如果修改了一个 ethenet phy，可能需要修改的地方有两个：

1： common\board_r.c\init_sequence_r->board\freescale\imx8qxp_mek\imx8qxp_mek.c\board_init-> setup_fec(CONFIG_FEC_ENET_DEV=0);-> enet_device_phy_reset

使用 gpio@1a_4 I2C to GPIO 的第 4 个脚去 reset phy

```
pca9557_a: gpio@1a {
                        compatible = "nxp,pca9557";
                        reg = <0x1a>;
                        gpio-controller;
                        #gpio-cells = <2>;
                    };
dm_gpio_lookup_name("gpio@1a_4", &desc);
dm_gpio_request(&desc, "enet0_reset");
dm_gpio_set_dir_flags(&desc, GPIOD_IS_OUT);
        dm_gpio_set_value(&desc, 0);
        udelay(50);
        dm_gpio_set_value(&desc, 1)
```

原理图如下：



2：目前 phy 是使用的 ENET0 connects AR8031

所以在 include\configs\imx8qxp_mek.h\中定义 了：

#define CONFIG_PHY_ATHEROS

然后 driver\net\phy\phy.c

```
int phy_init(void)
{
#ifdef CONFIG_PHY_ATHEROS
        phy_atheros_init();
#endif
}
Drivers/net/phy/atheros.c
int phy_atheros_init(void)
{
        phy_register(&AR8031_driver);
        return 0;
}
static struct phy_driver AR8031_driver =  {
        .name = "AR8031/AR8033",
        .uid = 0x4dd074,
        .mask = 0xffffffef,
        .features = PHY_GBIT_FEATURES,
        .config = ar8031_config,
        .startup = genphy_startup,
        .shutdown = genphy_shutdown,
```

```
};
static int ar8031_config(struct phy_device *phydev)
{
        if (phydev->interface == PHY_INTERFACE_MODE_RGMII_TXID ||
            phydev->interface == PHY_INTERFACE_MODE_RGMII_ID) {
                phy_write(phydev, MDIO_DEVAD_NONE, AR803x_PHY_DEBUG_ADDR_REG,
                                AR803x_DEBUG_REG_5);
                phy_write(phydev, MDIO_DEVAD_NONE, AR803x_PHY_DEBUG_DATA_REG,
                                AR803x_RGMII_TX_CLK_DLY);
        }

        if (phydev->interface == PHY_INTERFACE_MODE_RGMII_RXID ||
            phydev->interface == PHY_INTERFACE_MODE_RGMII_ID) {
                phy_write(phydev, MDIO_DEVAD_NONE, AR803x_PHY_DEBUG_ADDR_REG,
                                AR803x_DEBUG_REG_0);
                phy_write(phydev, MDIO_DEVAD_NONE, AR803x_PHY_DEBUG_DATA_REG,
                                AR803x_RGMII_RX_CLK_DLY);
        }

        phydev->supported = phydev->drv->features;

        genphy_config_aneg(phydev);
        genphy_restart_aneg(phydev);

        return 0;
}
```

所以如果使用其它的 PHY，需要检查 driver\net\phy\phy.c

int phy_init(void)中定义的宏，并将宏加入到 include\configs\imx8qxp_mek.h\

然后再检查其驱动 Drivers/net/phy/atheros.c 中定义 的驱动操作函数数组中的.config 函数调用是否能正确配置 PHY 的初始化。

5.6.5 下载用 USB 口定制

iMX8QXP MEK CPU 板上是使用 typeC 的 USBOTG3 的接口，可以用于 uuu 下载，而事实上在大多数的汽车应该设计中，不会使用 typeC 接口，所以有可能是使用此 USB PHY 的非 typeC 接口 USBOTG2 来接 typeA 接口，用于下载，这种情况下如下把 typeC 禁止掉：

\uboot-imx\configs\imx8qxp_mek_defconfig

```
-CONFIG_USB_TCPC=y
+#CONFIG_USB_TCPC=y
```

如果是使用 USBOTG1 来下载(在 i.MX8QXP MEK 上是连接到了底板的 USBOTG 上)，则需要打开其支持：

\uboot-imx\configs\imx8qxp_mek_defconfig

```
-# CONFIG_CI_UDC=y
+CONFIG_CI_UDC=y
```

同时， 把 typeC 加 USBOTG2/3 禁止掉：

```
-CONFIG_USB_TCPC=y
+#CONFIG_USB_TCPC=y
-CONFIG_USB_CDNS3=y
-CONFIG_USB_CDNS3_GADGET=y
+#CONFIG_USB_CDNS3=y
+#CONFIG_USB_CDNS3_GADGET=y
```

5.6.6 I.MX8DX 支持

Bootloader 都是单核启动的，所以 i.MX8DX 可以使用 i.MX8QXP 的配置。但是要修改 arch/arm/dts/fsl-imx8qxp-mek.dts

//#include "fsl-imx8qxp.dtsi"

#include "fsl-imx8dx.dtsi"

直接包含 fsl-imx8dx.dtsi,不在包括 fsl-imx8qxp.dtsi 和 fsl-imx8dxp.dtsi.

在"fsl-imx8dx.dtsi"中已经禁掉了两个核：

\uboot-imx\arch\arm\dts\fsl-imx8dx.dtsi

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

5.6.7 NANDflash 支持

i.MX8QXP MEK 板是不支持 nandflash 的，不过 ARM2 的 BSP 版本支持 Nandflash,可以直接使用 ARM2 板的 Uboot.

```
source ~ /imx-yocto-bsp/imx8qxpmek_x
```

```
wayland/sdk/environment-setup-aarch64-poky-linux
make clean
make distclean
make imx8qxp_lpddr4_arm2_nand_defconfig
make
```

注意在 Make image 时，要指定编译成 nandflash 镜像，如下：

```
 pwd
~/imx-yocto-bsp/standalone/imx-mkimage
vi iMX8QX/soc.mak
flash_nand: $(MKIMG) mx8qx-ahab-container.img scfw_tcm.bin u-boot-atf.bin
        ./$(MKIMG) -soc QX -rev B0 -dev nand 16K -append mx8qx-ahab-container.img -c -scfw scfw_tcm.bin -ap u-boot-atf.bin a35 0x80000000 -out flash.bin
        ./$(MKIMG) -soc QX -rev B0 -append mx8qx-ahab-container.img -c -scfw scfw_tcm.bin -ap u-boot-atf.bin a35 0x80000000 -out flash_fw.bin
```

所以 Make image 的命令为：

```
cd imx-mkimage
….
```

运行 imx-mkimage 脚本生成 nandflash.bin 镜像

```
make SOC=iMX8QX nandflash
```

这个命令会导出两个文件，其中 flash.bin 是要被烧写到 nandflash 中的镜像，flash_fw.bin 是 uuu 用来烧写 nandflash 的镜像，具体的 uuu 烧写 nandflash 方法请见内核文档。

以下为 ARM2 的 uboot nandflash 配置：

\uboot-imx\configs\imx8qxp_lpddr4_arm2_nand_defconfig 中 nandflash 相关：

```
#CONFIG_CMD_MMC=y
#CONFIG_DM_MMC=y
#CONFIG_MMC_IO_VOLTAGE=y
#CONFIG_MMC_UHS_SUPPORT=y
#CONFIG_MMC_HS400_SUPPORT=y
CONFIG_NAND_BOOT=y
CONFIG_CMD_NAND=y
#CONFIG_ENV_IS_IN_MMC=y
CONFIG_ENV_IS_IN_NAND=y
```

\uboot-imx\include\configs\imx8qxp_arm2.h nandflash 相关：

```
#ifdef CONFIG_NAND_BOOT
#ifndef CONFIG_PARSE_CONTAINER
```

```
#define CONFIG_SPL_NAND_RAW_ONLY
#endif
#define CONFIG_SPL_NAND_SUPPORT
#define CONFIG_SPL_DMA_SUPPORT
#define CONFIG_SPL_NAND_MXS
#define CONFIG_SYS_NAND_U_BOOT_OFFS     (0x8000000)  /*Put the FIT out of first 128MB boot area */
#define CONFIG_SPL_NAND_BOOT
#define CONFIG_SYS_NAND_U_BOOT_DST              0x80000000
#define CONFIG_SYS_NAND_U_BOOT_SIZE     (1024 * 1024 )

#define CONFIG_SYS_NAND_U_BOOT_START    0x80000000
#endif
…
 #ifdef CONFIG_NAND_BOOT
#define MFG_NAND_PARTITION
"mtdparts=gpmi-nand:128m(nandboot),16m(nandfit),32m(nandkernel),16m(nanddtb),8m(nandtee),-(nandrootfs) " //此处为nandflash mtdparts 表，其中 bootloader+kernel 用了 128+16+32+16+8=200MB,所以有可能要根据客户选择的 Nandflash 大小，实际 bootloader/kernel image 的大小，和 rootfs 的大小来实际调节。
#endif
….
/* Initial environment variables */
#ifdef CONFIG_NAND_BOOT
#define CONFIG_EXTRA_ENV_SETTINGS               \
     CONFIG_MFG_ENV_SETTINGS \
     "bootargs=console=ttyLP0,115200 ubi.mtd=6 "  \//注意一下此处指定了 nandflash rootfs 所在的 ubi.mtd 的序号，如果按照 nandflash 自己的序号的话，nandrootfs 的序号应该是 5(从 0 开始排),但是因为 ARM2 设计有 QSPINOR,它会占掉 ubi.mtd0, nandflash 会依次后排，所以如果客户的板子只有 nandflash,这儿需要修改为 5，不修改会报 rootfs mount errorr 错误。
          "root=ubi0:nandrootfs rootfstype=ubifs "                  \
          MFG_NAND_PARTITION \
          "\0"\
     "console=ttyLP0,115200 earlycon=lpuart32,0x5a060000,115200\0" \
     "mtdparts=" MFG_NAND_PARTITION "\0" \
     "fdt_addr=0x83000000\0"
#else
…
#ifdef CONFIG_NAND_BOOT
```

```
#define CONFIG_BOOTCOMMAND \
        "nand read ${loadaddr} 0x9000000 0x2000000;"\
        "nand read ${fdt_addr} 0xB000000 0x100000;"\
        "booti ${loadaddr} - ${fdt_addr}"
#else
…
#ifdef CONFIG_NAND_BOOT
#define CONFIG_ENV_OFFSET       (120 << 20)
#elif defined(CONFIG_QSPI_BOOT)
…
#ifdef CONFIG_CMD_NAND
#define CONFIG_NAND_MXS
#define CONFIG_CMD_NAND_TRIMFFS

/* NAND stuff */
#define CONFIG_SYS_MAX_NAND_DEVICE     1
#define CONFIG_SYS_NAND_BASE           0x40000000
#define CONFIG_SYS_NAND_5_ADDR_CYCLE
#define CONFIG_SYS_NAND_ONFI_DETECTION

/* DMA stuff, needed for GPMI/MXS NAND support */
#define CONFIG_APBH_DMA
#define CONFIG_APBH_DMA_BURST
#define CONFIG_APBH_DMA_BURST8
#endif
```

以下为 ARM2 的 uboot nandflash 初始化流程

\uboot-imx\board\freescale\imx8qxp_arm2\ imx8qxp_arm2.c

```
int board_init(void)
{
….

#ifdef CONFIG_NAND_MXS
        imx8qxp_gpmi_nand_initialize();
#endif
…
```

```
}
int board_early_init_f(void)
{…
#ifdef CONFIG_SPL_BUILD
#ifdef CONFIG_NAND_MXS
        imx8qxp_gpmi_nand_initialize();
#endif
#endif
…}
static void imx8qxp_gpmi_nand_initialize(void)
  |->  setup_iomux_gpmi_nand();
  |    |-> imx8_iomux_setup_multiple_pads(gpmi_nand_pads, ARRAY_SIZE(gpmi_nand_pads));
static iomux_cfg_t gpmi_nand_pads[] = {
        SC_P_EMMC0_CLK | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_EMMC0_DATA0 | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
…
        SC_P_EMMC0_DATA7 | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_EMMC0_STROBE | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_EMMC0_RESET_B | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_USDHC1_CMD | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_USDHC1_DATA2 | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_USDHC1_DATA3 | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_USDHC1_DATA0 | MUX_MODE_ALT(1) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),

        /* i.MX8QXP NAND use nand_re_dqs_pins */
        SC_P_USDHC1_CD_B | MUX_MODE_ALT(3) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),
        SC_P_USDHC1_VSELECT | MUX_MODE_ALT(3) | MUX_PAD_CTRL(GPMI_NAND_PAD_CTRL),

};
#define GPMI_NAND_PAD_CTRL        ((SC_PAD_CONFIG_OUT_IN << PADRING_CONFIG_SHIFT) |
(SC_PAD_28FDSOI_DSE_DV_HIGH << PADRING_DSE_SHIFT) \
                                  | (SC_PAD_28FDSOI_PS_PU << PADRING_PULL_SHIFT))
#define SC_PAD_28FDSOI_PS_PU     1U    /*!< Pull-up */ //配置为上拉
…
int board_mmc_init(bd_t *bis)
{
```

…

#ifdef CONFIG_NAND_MXS

return 0; //emmc 初始化直接返回，因为 emmc 与 nandflash 有管脚冲突。

#endif

以下 dts 中可以禁掉 emmc,但是因为代码中已经禁掉了 emmc 初始化，不改也可以：

\uboot-imx\arch\arm\dts\fsl-imx8qxp-lpddr4-arm2.dts

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

&usdhc2 {…
status = "disabled";
}
```

5.7 uboot debug 信息

5.7.1 dm - Driver model low level access

```
=> dm tree
 Class     Probed  Driver     Name
-----------------------------------------
 root      [ + ]   root_drive  root_driver
 pinctrl   [ + ]   imx8_pinct  |-- iomuxc
 pinconfig  [ + ]   pinconfig   |   `-- imx8qxp-mek
 pinconfig  [ + ]   pinconfig   |      |-- hoggrp
 pinconfig  [ + ]   pinconfig   |      |-- lpuart0grp
 pinconfig  [ + ]   pinconfig   |      |-- fec1grp
 pinconfig  [ + ]   pinconfig   |      |-- fec2grp
 pinconfig  [ + ]   pinconfig   |      |-- lpi1cgrp
 pinconfig  [ + ]   pinconfig   |      |-- usdhc1grp
 pinconfig  [   ]   pinconfig   |      |-- usdhc1grp100mhz
 pinconfig  [   ]   pinconfig   |      |-- usdhc1grp200mhz
 pinconfig  [ + ]   pinconfig   |      |-- usdhc2gpiogrp
 pinconfig  [ + ]   pinconfig   |      |-- usdhc2grp
 pinconfig  [   ]   pinconfig   |      |-- usdhc2grp100mhz
 pinconfig  [   ]   pinconfig   |      |-- usdhc2grp200mhz
```

```
pinconfig [   ]  pinconfig  |      |-- flexspi0grp
pinconfig [   ]  pinconfig  |      |-- mipi_lvds0_i2c0_grp
pinconfig [   ]  pinconfig  |      `-- mipi_lvds1_i2c0_grp
simple_bus [ + ]  generic_si  |-- imx8qx-pm
power_doma [ + ]  imx8_power  |  |-- lsio_power_domain
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm0
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm1
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm2
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm3
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm4
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm5
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm6
power_doma [   ]  imx8_power  |  |  |-- lsio_pwm7
power_doma [   ]  imx8_power  |  |  |-- lsio_kpp
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio0
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio1
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio2
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio3
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio4
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio5
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio6
power_doma [ + ]  imx8_power  |  |  |-- lsio_gpio7
power_doma [   ]  imx8_power  |  |  |-- lsio_gpt0
power_doma [   ]  imx8_power  |  |  |-- lsio_gpt1
power_doma [   ]  imx8_power  |  |  |-- lsio_gpt2
power_doma [   ]  imx8_power  |  |  |-- lsio_gpt3
power_doma [   ]  imx8_power  |  |  |-- lsio_gpt4
power_doma [   ]  imx8_power  |  |  |-- lsio_fspi0
power_doma [   ]  imx8_power  |  |  `-- lsio_fspi1
power_doma [ + ]  imx8_power  |  |-- connectivity_power_domain
power_doma [   ]  imx8_power  |  |  |-- conn_usb0
power_doma [   ]  imx8_power  |  |  |-- conn_usb0_phy
power_doma [   ]  imx8_power  |  |  |-- conn_usb1
power_doma [   ]  imx8_power  |  |  |-- conn_usb2
```

```
power_doma [ ]  imx8_power |  |  |-- conn_usb2_phy
power_doma [+]  imx8_power |  |  |-- conn_sdhc0
power_doma [+]  imx8_power |  |  |-- conn_sdhc1
power_doma [ ]  imx8_power |  |  |-- conn_sdhc2
power_doma [+]  imx8_power |  |  |-- conn_enet0
power_doma [+]  imx8_power |  |  |-- conn_enet1
power_doma [ ]  imx8_power |  |  |-- conn_nand
power_doma [ ]  imx8_power |  |  |-- conn_mlb0
power_doma [ ]  imx8_power |  |  |-- conn_dma4_ch0
power_doma [ ]  imx8_power |  |  |-- conn_dma4_ch1
power_doma [ ]  imx8_power |  |  |-- conn_dma4_ch2
power_doma [ ]  imx8_power |  |  |-- conn_dma4_ch3
power_doma [ ]  imx8_power |  |  `-- conn_dma4_ch4
power_doma [ ]  imx8_power |  |-- audio_power_domain
power_doma [ ]  imx8_power |  |  |-- audio_asrc0
power_doma [ ]  imx8_power |  |  |-- audio_asrc1
power_doma [ ]  imx8_power |  |  |-- audio_esai0
power_doma [ ]  imx8_power |  |  |-- audio_spdif0
power_doma [ ]  imx8_power |  |  |-- audio_sai0
power_doma [ ]  imx8_power |  |  |-- audio_sai1
power_doma [ ]  imx8_power |  |  |-- audio_sai2
power_doma [ ]  imx8_power |  |  |-- audio_sai3
power_doma [ ]  imx8_power |  |  |-- audio_sai4
power_doma [ ]  imx8_power |  |  |-- audio_sai5
power_doma [ ]  imx8_power |  |  |-- audio_gpt5
power_doma [ ]  imx8_power |  |  |-- audio_gpt6
power_doma [ ]  imx8_power |  |  |-- audio_gpt7
power_doma [ ]  imx8_power |  |  |-- audio_gpt8
power_doma [ ]  imx8_power |  |  |-- audio_gpt9
power_doma [ ]  imx8_power |  |  |-- audio_gpt10
power_doma [ ]  imx8_power |  |  |-- audio_amix
power_doma [ ]  imx8_power |  |  |-- audio_mqs0
power_doma [ ]  imx8_power |  |  |-- audio_hifi
power_doma [ ]  imx8_power |  |  |-- audio_ocram
```

```
power_doma [   ]   imx8_power  |   |   |-- audio_mclkout0
power_doma [   ]   imx8_power  |   |   |-- audio_mclkout1
power_doma [   ]   imx8_power  |   |   |-- audio_audiopll0
power_doma [   ]   imx8_power  |   |   |-- audio_audiopll1
power_doma [   ]   imx8_power  |   |   |-- audio_audioclk0
power_doma [   ]   imx8_power  |   |   `-- audio_audioclk1
power_doma [ + ]   imx8_power  |   |-- dma_power_domain
power_doma [   ]   imx8_power  |   |   |-- dma_flexcan0
power_doma [   ]   imx8_power  |   |   |-- dma_flexcan1
power_doma [   ]   imx8_power  |   |   |-- dma_flexcan2
power_doma [   ]   imx8_power  |   |   |-- dma_ftm0
power_doma [   ]   imx8_power  |   |   |-- dma_ftm1
power_doma [   ]   imx8_power  |   |   |-- dma_adc0
power_doma [   ]   imx8_power  |   |   |-- dma_lpi2c0
power_doma [ + ]   imx8_power  |   |   |-- dma_lpi2c1
power_doma [   ]   imx8_power  |   |   |-- dma_lpi2c2
power_doma [   ]   imx8_power  |   |   |-- dma_lpi2c3
power_doma [ + ]   imx8_power  |   |   |-- dma_lpuart0
power_doma [   ]   imx8_power  |   |   |-- dma_lpuart1
power_doma [   ]   imx8_power  |   |   |-- dma_lpuart2
power_doma [   ]   imx8_power  |   |   |-- dma_lpuart3
power_doma [   ]   imx8_power  |   |   |-- dma_spi0
power_doma [   ]   imx8_power  |   |   |-- dma_spi1
power_doma [   ]   imx8_power  |   |   |-- dma_spi2
power_doma [   ]   imx8_power  |   |   |-- dma_spi3
power_doma [   ]   imx8_power  |   |   |-- dma_pwm0
power_doma [   ]   imx8_power  |   |   `-- dma_lcd0
power_doma [   ]   imx8_power  |   |-- gpu-power-domain
power_doma [   ]   imx8_power  |   |   `-- gpu0
power_doma [   ]   imx8_power  |   |-- vpu-power-domain
power_doma [   ]   imx8_power  |   |   `-- vpu_core
power_doma [ + ]   imx8_power  |   |-- hsio-power-domain
power_doma [ + ]   imx8_power  |   |   |-- PD_HSIO_SERDES_1
power_doma [ + ]   imx8_power  |   |   |   `-- hsio_pcie1
```

```
power_doma [ + ]   imx8_power  |   |   `-- hsio_gpio
power_doma [   ]   imx8_power  |   |-- cm40_power_domain
power_doma [   ]   imx8_power  |   |   |-- cm40_i2c
power_doma [   ]   imx8_power  |   |   `-- cm40_intmux
power_doma [   ]   imx8_power  |   |-- dc0_power_domain
power_doma [   ]   imx8_power  |   |   |-- dc0_pll0
power_doma [   ]   imx8_power  |   |   |   `-- dc0_pll1
power_doma [   ]   imx8_power  |   |   |-- mipi0_dsi_power_domain
power_doma [   ]   imx8_power  |   |   |   |-- lvds0_power_domain
power_doma [   ]   imx8_power  |   |   |   |-- mipi0_dsi_i2c0
power_doma [   ]   imx8_power  |   |   |   |-- mipi0_dsi_i2c1
power_doma [   ]   imx8_power  |   |   |   `-- mipi0_dsi_pwm0
power_doma [   ]   imx8_power  |   |   `-- mipi1_dsi_power_domain
power_doma [   ]   imx8_power  |   |       |-- lvds1_power_domain
power_doma [   ]   imx8_power  |   |       |-- mipi1_dsi_i2c0
power_doma [   ]   imx8_power  |   |       |-- mipi1_dsi_i2c1
power_doma [   ]   imx8_power  |   |       `-- mipi1_dsi_pwm0
power_doma [   ]   imx8_power  |   `-- imaging_power_domain
power_doma [   ]   imx8_power  |       |-- mipi_csi0_power_domain
power_doma [   ]   imx8_power  |       |   |-- mipi_csi0_i2c
power_doma [   ]   imx8_power  |       |   `-- mipi_csi0_pwm
power_doma [   ]   imx8_power  |       |-- imaging_pdma1
power_doma [   ]   imx8_power  |       |-- imaging_pdma2
power_doma [   ]   imx8_power  |       |-- imaging_pdma3
power_doma [   ]   imx8_power  |       |-- imaging_pdma4
power_doma [   ]   imx8_power  |       |-- imaging_pdma5
power_doma [   ]   imx8_power  |       |-- imaging_pdma6
power_doma [   ]   imx8_power  |       `-- imaging_pdma7
thermal    [   ]   imx_sc_the  |-- thermal-sensor
thermal    [   ]   imx_sc_the  |   `-- cpu-thermal0
gpio       [ + ]   gpio_mxc    |-- gpio@5d080000
gpio       [ + ]   gpio_mxc    |-- gpio@5d090000
gpio       [ + ]   gpio_mxc    |-- gpio@5d0a0000
gpio       [ + ]   gpio_mxc    |-- gpio@5d0b0000
```

```
gpio       [ + ]   gpio_mxc    |-- gpio@5d0c0000
gpio       [ + ]   gpio_mxc    |-- gpio@5d0d0000
gpio       [ + ]   gpio_mxc    |-- gpio@5d0e0000
gpio       [ + ]   gpio_mxc    |-- gpio@5d0f0000
i2c        [   ]   imx_lpi2c   |-- i2c@56226000
simple_bus [   ]   generic_si  |-- camera
i2c        [   ]   imx_lpi2c   |-- i2c@56246000
i2c        [ + ]   imx_lpi2c   |-- i2c@5a810000
i2c_mux    [ + ]   pca954x     |   |-- mux@71
i2c        [   ]   i2c_mux_bu  |   |   |-- i2c@0
i2c        [   ]   i2c_mux_bu  |   |   |-- i2c@1
i2c        [   ]   i2c_mux_bu  |   |   |-- i2c@2
i2c        [ + ]   i2c_mux_bu  |   |   `-- i2c@3
gpio       [ + ]   pca953x     |   |       |-- gpio@1a
gpio       [   ]   pca953x     |   |       `-- gpio@1d
i2c_generi [ + ]   i2c_generi  |   `-- generic_50
usb        [   ]   ehci_mx6    |-- usb@5b0d0000
usb        [   ]   xhci_imx8   |-- usb@0x5b110000
serial     [ + ]   serial_lpu  |-- serial@5a060000
mmc        [ + ]   fsl-esdhc-  |-- usdhc@5b010000
blk        [   ]   mmc_blk     |   `-- usdhc@5b010000.blk
mmc        [ + ]   fsl-esdhc-  |-- usdhc@5b020000
blk        [ + ]   mmc_blk     |   `-- usdhc@5b020000.blk
eth        [ + ]   fecmxc      |-- ethernet@5b040000
eth        [ + ]   fecmxc      |-- ethernet@5b050000
spi        [   ]   fsl_fspi    |-- flexspi@05d120000
spi_flash  [   ]   spi_flash_  |   `-- mt35xu512aba@0
simple_bus [ + ]   generic_si  `-- regulators
regulator  [ + ]   fixed regu      |-- usdhc2_vmmc
regulator  [   ]   fixed regu      `-- regulator@0
```

5.7.2 => fdt

```
=> pri
…
fdt_addr=0x83000000
```

```
…
=>fdt addr ${fdt_addr}
=>fdt print
```

会打印出 DTB 文件内容。