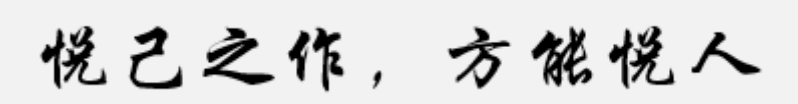

u-boot 移植基础详解

本文系广泛撷取、借鉴和整理（相关的内容在网络上有很多，但很多相互抄，或者是版本太老，或者就是不通用的非常有平台针对性的步骤，碎片化泛滥，甚至就是有待分拣的垃圾厂，当然也有一些好的文章，都会看），经过了细心、耐心、小心、佛心乃至贤者模式、莫生气、开始怀疑、愁然、不畏怯、再到渐入佳境、逐渐明朗的学习过程，再经过了广泛翻阅和对比，边做边写，对主要内容进行了精髓提取。

过于基础的内容不会在此提及，比如 SoC 芯片的启动流程、什么是 bootloader 等等内容，此类背景知识在几乎每一本嵌入式 Linux 书籍都会提及，在此不会赘述；本文专注具体 u-boot 的启动流程、移植和编译等（以百问网的 imx6ull 和 米尔、NXP 的 imx8mm 这两种平台为例），为了区分于网络上碎片碎末、残缺不全、缺斤少两、半斤八两、一个巴掌拍不响的各种文章，本文尽量做到通用化的、全面化的学习记录，并且会说明我是看的什么资料，目的就是在面对新平台时候，可以自主移植、编译和跑起来，融会贯通，而不是把人当作定制化的没有主观能动性的机器。

参考大量文章和教程以及触(连)类(猜)旁(带)通(蒙)的反复测试，记录成功的步骤，而成此文，如有错误恭谢指出！侵删。作者见 "署名" 一节。

---

u-boot 篇

成熟的 bootloader

先说思想，bootloader 是很有用、很方便的一种发明，作为一个 bootloader，本质是一个在 MCU 或者 MPU 首先运行的裸机程序，通过通讯接口（UART、EtherNet、CAN、USB、SD 卡 等等）把编译好的 APP 程序的二进制文件或者镜像首先下载到内存中，或者是从 FLASH 中读取 APP 程序数据放到内存中，并跳转到 APP 程序处开始执行，在内存中跑的程序对于取指等 CPU 操作等待的时间比在 FLASH 中跑的程序快得多；或者通过把从一个接口接收的 APP 程序数据存储到 自己的 FLASH 中，实现自己给自己烧录（自举， IAP），比如把 SD 卡里面的 程序数据读出然后烧入 自己的 FLASH 里面，或者从 USB 接收程序，这些方法方便板子的批量烧录；还可以预先在自己的 FLASH 中存入多个 APP 程序，根据外部指令选择运行哪一个，不用反复烧写便可实现更换程序；说到这里，给 MCU/MPU 加一个 bootloader 后，其灵活性、可操作性就提升了维度。bootloader 要实现这些功能，就当然需要先初始化这些通讯接口和储存设备（DDR / FLASH），还要提供一个人机交互界面（一般为命令行）并实现一些命令，搬运应用程序到内存，准备要跳入操作系统内核的环境并跳转启动该内核，这些即是一个 bootloader 的最小实现。

关于 MCU 的 IAP 更多内容可见我的另一个开源项目 Staok/u-iap: 一个志在实现适用于 MCU 的通用 IAP 程序框架，可以从串口、外部 SPI FLASH、外部 SDIO SD 卡、USB Device MSC 或者 USB HOST MSC等等途径更新 MCU 固件。 (github.com)。

以上说的都是最最基本的，更多 bootloader 的更专业（更官话）的介绍文章如下：

- 搞嵌入式的，为啥要有uboot？ (qq.com)；
- 嵌入式系统 Boot Loader 技术内幕 业精于勤,荒于嬉;行成于思,毁于随-CSDN博客；

u-boot 启动流程

不但知其然，还要知其所以然。如果熟悉《【主线剧情01】ARM IMX6ULL 基础学习记录》里面的内容，那么作为 bootloader 一种的 u-boot 的启动过程，就相当于扩充了、丰富了和复杂了 的那篇文章里所介绍的启动文件 start.s 的过程，主要步骤可以一一对应上。u-boot 启动流程仅作知道一下有这回事，不必深究。

不同版本的 u-boot 以及不同的板子的文件位置和初始化流程会有偏差。以下启动流程为笔者主要用 grep 和 find 命令一点一点看源码梳理出来。

以 i.mx6 为例

以下基于 u-boot 2017.03 的 100ask i.mx6ull 平台：

1. 上电运行 `arch/arm/cpu/armv7/start.S`，在这里面执行以下：
    1. CPU 设为 SVC 模式，关中断（IRQ 和 FIR），关 MMU，关看门狗等；
    2. 执行 `bl cpu_init_crit`，跳到同目录下的`lowlevel_init.S`，再跳到 `同目录/mx6/soc.c`的`s_init()`进行时钟等检查和设置；
    3. 执行 `bl _main`，跳到 `arch/arm/lib/crt0.S`的`_main`。
2. 在`arch/arm/lib/crt0.S`，`_main`里面执行以下：
    1. 设置栈 stack_setup；为 C 环境做准备；清除 BSS 段；
    2. 执行 `board_init_f`，（第一次初始化）基本硬件初始化，如时钟、内存和串口等，这个函数可能位于 `board/<公司名>/<板子名>/<板子名>.c` 文件里或者位于`common/board_f.c`文件里，其里面的宏定义在其对应的头文件里`include/configs/<板子名>.h`；
    3. 调用重定位 `relocate_code`，源码位于`arch/arm/cpu/armv7/start.S`，（第一次搬移）u-boot 自搬到内存；
    4. 执行 `board_init_r`，（第二次初始化）对外设的初始化，位于`common/board_r.c`，初始化 `init_sequence_r` 列表中的初始化函数，最后调用`run_main_loop()`，其再调用 `common/main.c`里的`main_loop()`。
3. 进入 `common/main.c`，`main_loop()`，延迟`bootdelay_process()`，若无输入则进入自启动 `autoboot_command()`，若有输入则进入交互模式 `cli_secure_boot_cmd()`；
4. 根据命令或自启动流程，若要 boot 进入系统，跳到 `cmd/bootm.c` 的 `do_bootm()`，再调用同文件的 `do_bootm_subcommand()`，调用 `common/bootm.c` 里的 `do_bootm_states()`，在这里面检测是否有 OS，是什么 OS，如果有 Linux 系统，则下一步；
5. （第二次搬移）搬运 Linux 内核到内存；
6. 然后调用 `common/bootm_os.c` 里的 `boot_selected_os()`，其调用`arch/arm/lib/bootm.c` 里的 `do_bootm_linux()`，再调用同文件里的`boot_jump_linux()`，跳到内核的起始地址，进入 & 运行内核。

更详细的启动流程介绍可以参看《【正点原子】I.MX6U嵌入式Linux驱动开发指南》的第三十二章节，目前发现的最全面的。

以 i.mx8 为例

以下基于 u-boot 2019.04 的 uboot-imx 中的 imx8mm 平台，这里主要描述各种初始化函数的跳转关系和逻辑：

1. 上电执行`arch/arm/cpu/armv8/start.S`后跳转到`arch/arm/lib/crt0_64.S`的`_main`里：

    1. 首先调用 `common/board_f.c` 里面的 `board_init_f()`（第一次初始化），其调用 `init_sequence_f[]` 数组里的各个初始化函数，数组中有 `board/<公司名>/<板子名>/<板子名>.c` 中的 `board_early_init_f()`；
    2. 然后调用 `relocate_code`（第一次搬运）uboot自搬运到内存；
    3. 调用 `common/board_r.c` 里面的 `board_init_r()`（第二次初始化），其调用 `init_sequence_r[]` 数组里的各个初始化函数，数组中有：
        1. `board/<公司名>/<板子名>/<板子名>.c` 中的 `board_init()`；
        2. `board_early_init_r()`，在 imx8 的文件中没有用到；
        3. `board/<公司名>/<板子名>/<板子名>.c` 中的 `board_late_init()`；
        4. `run_main_loop()`；
2. 跳入 uboot 的主循环 `run_main_loop()` 之后，等待外部输入命令或超时等待自启动；
3. 进行启动，找到 Linux kernel 镜像，搬运其到内存（第二次搬运），跳转启动。

更多 u-boot 启动过程介绍文章：

- 更详细的引导内核过程。
- u-boot分析 - 蜗窝科技 (wowotech.net)。

u-boot 命令

若设置了 u-boot 启动时等待任意键输入几秒，若有输入，则退出自启动模式而进入命令行模式。

- `help`：显示所有命令及其说明；
- `help 命令`：显示`命令`详细的使用说明；
- `pri`：查看所有环境变量，包括开机等待任意输入的时延（秒）、串口波特率（baudrate）、本地 IP 地址（ipaddr）、tftp 服务器端的 IP 地址（serverip）、自启动命令字符串（bootcmd，一般不用动）等等；
- `setenv <环境变量> <要设置的值>`：修改某一个环境变量为要设置的值，`要设置的值` 若是字符串则要加双引号；
- `saveenv`：将当前所有设置过的环境变量保存，掉电不丢失；
- `reset`：复位；
- `dhcp`：执行 DHCP 服务，获取 IP 地址，验证网络功能；
- `setenv my`：恢复系统的所有环境变量为默认，即使之前用户重设的环境变量都恢复默认；
- `protext on/off 0~10000`：对 Nor Flash 区域 [0x0 ~ 0x10000] 设置为写保护或取消写保护；
- `movi`：对 EMMC 进行操作；
- `run bootcmd`：，执行 bootcmd，其是一个环境变量，为一段字符串形式的命令；上电后 u-boot 若处于自启动模式最后执行则执行的多条命令，默认为 `下载内核、设备树和运行内核的多个命令`，可以根据需要增加命令，以分号分隔。

u-boot 移植要点

要点：

- 一般厂家直接提供 u-boot 源码，做查看、修改(增加新功能) 或 u-boot 版本升级这三大块的用处；后两种都需要对新板子做适配/移植。
- 如果没有提供 u-boot 源码，那么就从 u-boot 官方版本中找到一个最相近的板子配置进行移植，这个需要水平较高。
- 一般把 u-boot 做成对应平台通用的和最小化的，即只保留必要的板级外设初始化代码（如串口、网口和 FLASH 等需要主要做适配，都尽量找能现成使用的），其他更多板级外设初始化在 Linux 移植部分中完成。

- 如果要深入学习，有以下要点可以参考：
    - 如果芯片公司或者单位提供了移植好的 u-boot，可以用 beyong 软件把移植好的 u-boot 文件夹与 官方原版（版本要一致）进行对比，看一看改动了哪些文件夹和哪些文件，帮助学习。
    - uboot移植新手入门实践_哔哩哔哩 (゜-゜)つロ 干杯~-bilibili。版本比较新。
    - 正点原子【第三期】手把手教你学Linux之系统移植和跟文件系统构建篇_哔哩哔哩 (゜-゜)つロ 干杯~-bilibili；
    - 韦东山_嵌入式Linux_第1期与2期间的衔接课程_u-boot编译体验和源码深度分析_哔哩哔哩 (゜-゜)つロ 干杯~-bilibili 老版本。
    - 【韦东山】移植U-boot 2012 04 01 到JZ2440_哔哩哔哩 (゜-゜)つロ 干杯~-bilibili 老版本。
    - linux-----uboot和kernel移植 - 灰信网（软件开发博客聚合） (freesion.com)。

芯片公司、开发板厂家和用户三者之间的联系：

- 芯片公司移植的 u-boot 从一开始是基于官方的 u-boot 拿来修改，添加/修改自家的 EVK 评估版的板子型号、相关外设初始化文件，并修改 u-boot 的 Makefile 配置，然后把自家芯片的 EVK 评估版的硬件原理图、u-boot、Linux 和 根文件系统以及使用说明文档等等全部开源，以供下游做应用的公司/厂家和做开发板的公司拿来做修改或直接应用；
- 做开发板的厂家在拿到了芯片公司提供的芯片评估版 EVK 板子的原理图后，与 SoC 直接相关的比如 PMIC、DDR、FLASH、以太网 PHY 芯片等等不会做大改，一般直接照搬过来画自己的开发板。因为在移植 u-boot 的时候就不用再为新选型的芯片做代码适配，一般没必要做这种费力但效果不大的事情，能直接用的就尽量直接用，能不用改的就尽量不改。然后再拿到芯片公司提供的芯片评估版 EVK 板子对应的 u-boot 源码之后，同样的再添加/修改为自家开发板的型号、添加一点点自己板子的外设初始化代码（这个要求比较高）并修改 Makefile，便得到自家开发板适配的又一个 u-boot；
- 当用户（比如现在的我）拿到了开发板厂家 或者 芯片公司提供的 u-boot 源码，即所有相关文件和初始化代码都写好了，便可以直接编译进而使用，或者自己再进一步定制化，那么此时此刻间，具体的移植步骤介绍在以下：

u-boot 配置/移植并编译

注意，网上很多 u-boot 移植教程都是三星的 2440 或者 4412 在 古董级别的 u-boot 上面的移植，不是说不好，而是现在的 u-boot（2019 以后的版本）的 Makefile 结构等等都已变化太大了。

平台相关的目录：arch、board、include 等，平台无关的目录：common、net、fs、drivers 等等。

环境构建

不同的 SoC 需要准备不同的交叉编译器，对于 i.mx 系列，参考其官网手册通过一系列 NXP 提供的自动化脚本生成，以及配置好环境变量，即可使用此专用的交叉编译器编译 u-boot、Linux kernel 和 Linux 驱动/应用。

现以 i.mx8mm 为例，参考从其官网网址下载的嵌入式 Linux 手册包 imx-yocto-L5.4.70_2.3.0.zip 里面的手册来准备编译环境，提前说明，这个步骤是对于 NXP i.mx 系列的独家方法，其中一些编译步骤要花费巨长的时间，读到此的游客了解即可，除非要用，否则可以不用实践一遍。对于买 i.mx 系列开发板的，如果不愿从头做一遍，可以直接使用开发板公司提供的资料里面的环境配置脚本和交叉编译器，就直接跳过这一小节。

以下步骤都在 Linux 主机上进行。

1. 参考 "i.MX_Yocto_Project_User's_Guide.pdf" 手册：
    1. Linux Ubuntu 18.04 主机安装必要包：

```
sudo apt-get install gawk wget git-core diffstat unzip
texinfo gcc-multilib build-essential chrpath socat cpio
python python3 python3-pip python3-pexpect xz-utils
debianutils iputils-ping python3-git python3-jinja2 libegl1-
mesa libsdl1.2-dev pylint3 xterm
```

2. 新建存放资源的文件夹，这里取名 "nxp_imx"，在里面再建立一个放 bsp 的文件夹取名 "imx-yocto-bsp"；
3. 配置 git 信息：

```
git config --global user.name "<Your Name>"
git config --global user.email "<Your Email>"
git config --list
```

4. 先下载 repo 程序：`sudo apt-get install repo` ；然后在 imx-yocto-bsp 文件夹下面执行：

```
repo init -u https://source.codeaurora.org/external/imx/imx-
manifest -b imx-linux-zeus -m imx-5.4.70-2.3.0.xml
repo sync
```

如果上一条命令执行不成功，先删除目录中的 .repo 文件夹，再试；若还不成功，参考[关于使用repo时repo init和repo sync失败的一个解决方案 - 代码的搬运工 - 博客园 (cnblogs.com)](), 参考这个文章的方法，若遇到 repo 超时错误，尝试用方法2：

```
# 先手动下载 repo git 仓库
git clone https://gerrit-googlesource.lug.ustc.edu.cn/git-
repo
# 然后用以下命令替代上面第一条不成功的 repo 命令试试，这一句大概率会成功
./git-repo/repo init -u
https://source.codeaurora.org/external/imx/imx-manifest -b
imx-linux-zeus -m imx-5.4.70-2.3.0.xml
```

5. 上一个步骤执行成功结束后，会得到 imx-yocto-bsp/sources 文件夹和其他文件，You can perform repo synchronization, with the command `repo sync`, periodically to update to the latest code。源文说明：The "sources" directory which contains the recipes used to build one or more
build directories, and a set of scripts used to set up the environment.The recipes used to build the project come from both the community and i.MX. The Yocto Project layers are downloaded to thesources directory. This sets up the recipes that are used to build the project.
6. 构建针对 i.mx8mm 的配置，i.MX provides a script, `imx-setup-release.sh`, that simplifies the setup for i.MX machines. The script sets up a directory andthe configuration files for the specified machine and backend。在`imx-yocto-bsp`目录需要执行的命令模板：

```
DISTRO=<distro name> MACHINE=<machine name> source imx-setup-
release.sh -b <build dir>
```

以上命令需要填的空：

- DISTRO= is the distro, which configures the build environment and it is stored in meta-imx/meta-sdk/conf/distro. 可选：`fsl-imx-wayland` - Wayland weston graphics 和 `fsl-imx-xwayland` - Wayland graphics and X11. X11 applications using EGL are not supported，一般就都选这个吧；
- MACHINE= is the machine name which points to the configuration file in conf/machine in meta-freescale and meta-imx. 要选 `meta-imx/meta-bsp/conf/machine/` 目录下的文件名，具体的板级配置文件，这里选 `imx8mmddr4evk`；
- -b specifies the name of the build directory created by the imx-setup-release.sh script.这里选`-b imx8mmddr4evk_sh`；

最终实际执行的命令：

```
DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk source imx-setup-release.sh -b imx8mmddr4evk_sh
```

7. 上一句完成后会自动进入 `./imx8mmddr4evk_sh` 目录，并且执行信息最后面部分有提示，翻译过来就是`可以执行 bitbake 命令了`。

2. 参考 "i.MX_Porting_Guide.pdf" 手册：
    1. 先说，这一步是构建交叉编译工具链，可以参考 "工具链构建的更多方法" 小节的方法构建，基于 yocto 的方法在个人 PC 上所需的时间太漫长了，除非有高性能的服务器。
    2. Generate a development SDK, which includes the tools, toolchain, and small rootfs to compile against to put on the host machine. The same SDK can be used to build a standalone kernel. 在`./imx8mmddr4evk_sh` 目录要执行的命令模板：

    ```
    DISTRO=Target-Distro MACHINE=Target-Machine bitbake core-image-minimal -c populate_sdk
    ```

    还是需要填空，实际执行的命令为：

    ```
    DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk bitbake core-image-minimal -c populate_sdk
    ```

    `-c populate_sdk`选项表示构建应用层工具链；

    The populate_sdk generates a script file that sets up a standalone environment without Yocto Project.这一步需要 Linux 主机有至少 150G 的空间，这一步要花很长很长时间，第一次执行 5 ~ 10 个小时是有的...也可能是从外网下载包慢，总之做好准备。

    如果这一步漫长的过程中间被中断或者有下载包失败，那么可以在上一级目录中重新执行`DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk source imx-setup-release.sh -b imx8mmddr4evk_sh`一次，然后再执行本步骤的 bitbake 命令，看见会在上一次的进度继续执行。

    运行结束后，下载包的错误信息和编译包的错误信息分别保存在`log.do_fetch`和`log.do_compile`里面，路径`tmp/work/<architecture>/<component>/temp`。

3. From the build directory where the bitbake was run in, copy the `.sh` file from `tmp/deploy/sdk` to the host machine to build on and execute the script to install the SDK. The default location is /opt but can be placed anywhere on the host machine. 这里讲从上一步执行过后产生的 `tmp/deploy/sdk` 目录拿出 `.sh` 脚本文件，这个脚本用于产生上面所说的 toolchain 等等，可以单独使用，在哪里需要用就在哪里执行一下，默认会释放到 `/opt` 目录，不过可以指定释放的目录（写路径全名），释放的交叉编译器在 `/<工具链指定目录>/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/` 目录下面。在下一步进行环境配置。
4. 取得交叉编译器后，配置 Linux 主机环境变量；

```
# 临时生效的命令，下次开机会无效
export ARCH=arm64
source /<工具链指定目录>/environment-setup-aarch64-poky-linux
```

上面这个 source 命令后面这个文件内容很多；在需要编译的时候敲上面两句临时生效一下，也不麻烦。

3. 下载 u-boot-imx 源码，尝试编译 u-boot 的一个板子配置，如果成功便证明可用；

```
# 第一步 找一个地方，下载 u-boot-imx 源码，并进入 u-boot-imx 目录
git clone https://source.codeaurora.org/external/imx/uboot-imx
cd uboot-imx

# 第二步 查看所有 git 分支，切换到与 imx-5.4.70-2.3.0 版本对应的分支上
# git branch -v #列出本地所有的分支, + hash 信息
# git branch -vv #列出本地所有的分支, + hash 信息 + 与远程的关联信息
git branch -a #列出所有的分支(远程和本地)

# 第2.33步 切换分支
git checkout imx_v2020.04_5.4.70_2.3.0

# 第2.66步 同步当前仓库
git pull origin

# 第三步 尝试编译（举例）
make distclean
make imx8mm_ddr4_evk_defconfig
make -j16
```

成功后会产生所需的文件：

- u-boot-nodtb.bin；
- spl / u-boot-spl.bin；
- arch / arm / dts/ imx8mm-ddr4-evk.dtb。

4. 下载 imx-mkimage、imx-atf 和 firmware-imx；

```
# 下载 imx-mkimage，并进入目录
git clone https://source.codeaurora.org/external/imx/imx-mkimage/ &&
cd imx-mkimage

# 查看远程分支
git branch -v

# 切换到与 imx-5.4.70-2.3.0 版本对应的分支上
git checkout imx_5.4.70_2.3.0
```

```
git pull origin

# 退回上一级目录，下载 imx-atf，并同样切换分支
cd ..
git clone https://source.codeaurora.org/external/imx/imx-atf/ && cd imx-atf
git checkout imx_5.4.70_2.3.0
git pull origin

# 返回上一级目录，取得 firmware-imx，在上文提到的 ./imx8mmddr4evk_sh 目录中取得
cd .. && mkdir firmware-imx-8.10
cp -ri /.../imx8mmddr4evk_sh/tmp/work/aarch64-mx8mm-poky-linux/firmware-imx-8m/8.10-r0/firmware-imx-8.10/* ./firmware-imx-8.10/

# 取得 firmware-imx 的另一个途径 8.1 版本
wget http://www.freescale.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.1.bin
    # 解压
./firmware-imx-8.1.bin --auto-accept
```

5. 编译 imx-atf；不换平台的话一般只编译一次即可，后面可以一直用；

```
cd imx-atf
make clean PLAT=imx8mm
LDFLAGS="" make PLAT=imx8mm
```

编译成功后产生 `./build/imx8mm/release/bl31.bin` 文件。

6. 使用 imx-mkimage 链接合成所有文件生成最后二进制文件；

```
# 复制必要的文件
    # uboot-imx
cp uboot-imx/tools/mkimage ./imx-mkimage/iMX8M/mkimage_uboot
cp uboot-imx/arch/arm/dts/imx8mm-ddr4-evk.dtb ./imx-mkimage/iMX8M/imx8mm-ddr4-evk.dtb
cp uboot-imx/spl/u-boot-spl.bin ./imx-mkimage/iMX8M/
cp uboot-imx/u-boot-nodtb.bin ./imx-mkimage/iMX8M/

    # firmware-imx  在这里区分使用 LPDDR4 还是 DDR4，详见 "分析 imx-mkimage" 小节
cp firmware-imx-8.10/firmware/ddr/synopsys/ddr4_dmem_1d.bin ./imx-mkimage/iMX8M/
cp firmware-imx-8.10/firmware/ddr/synopsys/ddr4_dmem_2d.bin ./imx-mkimage/iMX8M/
cp firmware-imx-8.10/firmware/ddr/synopsys/ddr4_imem_1d.bin ./imx-mkimage/iMX8M/
cp firmware-imx-8.10/firmware/ddr/synopsys/ddr4_imem_2d.bin ./imx-mkimage/iMX8M/

    # imx-atf
cp imx-atf/build/imx8mm/release/bl31.bin ./imx-mkimage/iMX8M/
```

```
16
17  # imx-mkimage  在这里区分使用 LPDDR4 还是 DDR4，详见 "分析 imx-mkimage" 小节
18  cd imx-mkimage
19  make SOC=iMX8MM clean
20  make SOC=iMX8MM flash_ddr4_evk
```

最后产生 `./imx-mkimage/iMX8M/flash.bin` 文件为成功；不同的 imx 系列最后产生的文件名不同，具体看手册。

可以把上面的命令们复制到一个 .sh 脚本然后运行。

7. 通过 dd 命令烧写到 SD 卡的 uboot 分区，或者使用 USB 工具 uuu 烧写到 emmc（具体 uuu 工具的使用看其官方手册（github 上）或者 米尔的手册）。

工具链构建的更多方法

以下两种方法选其一即可。

基于 yocto 的 bitbake

我算看明白了，这个方法需要的时间实在太漫长了。除非有服务器，个人 PC 上不推荐跑。

参考《i.MX_Yocto_Project_User's_Guide.pdf》的 5.2 Choosing an i.MX Yocto project image 章节和米尔的手册，第 2 步中，构建其他选项的工具链详细说明：

有以下两种构建方法，对于uboot 的编译来说无明显差别；推荐直接在第二种里面使用 bitbake 命令进行构建。

- 利用上面的步骤，在第 2 步的在 `imx-yocto-bsp` 里面 `DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk source imx-setup-release.sh -b imx8mmddr4evk_sh` 执行完之后，然后再执行本步骤的 bitbake 命令，开始构建。
- 或者 使用 nxp imx 提供的 Yocto 的 10GB 的包（米尔、百问网都有提供），用这个构建（镜像或者产生交叉编译链和环境变量的 SDK .h 脚本）；更多 bitbake 命令见《i.MX_Yocto_Project_User's_Guide.pdf》的 5.4 Bitbake options 章节。

使用 bitbake 命令构建的步骤：

构建好后都会在在 `tmp/deploy/sdk` 目录下：产生 .sh 可执行环境构建文件，还有两个 manifest 文件， `host.manifest` 是工具链中包含主机端的软件包的列表， `target.manifest` 是包含目标设备端的软件包列表。

- 构建底层工具链（适合底层开发的工具链， 用于编译 u-boot 代码；使用范围有限，不推荐使用）：

```
1  # bitbake <参数>：meta-toolchain、meta-toolchain-sdk
2  # 实际执行：（在 imx8mmddr4evk_sh 目录里面，即 imx-setup-release.sh -b <build dir> 的这个目录）
3  DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk bitbake meta-toolchain （加不加 -c populate_sdk ？）
```

- 构建应用层工具链（编译 u-boot、linux 内核、驱动和应用等，附带目标系统的头文件和库文件， 方便应用开发者移植应用在目标设备上）：

```
# bitbake 的选项如下（这里只列出四个）：
# - core-image-minimal：A small image that only allows a device to boot。Provided by layer：Poky；
# - core-image-base：A console-only image that fully supports the target device hardware。Provided by layer：Poky；
# - imx-image-multimedia：This image contains all the packages except QT5/OpenCV/Machine Learning packages。Provided by layer：meta-imx/meta-sdk；
# - imx-image-full：This is the big image which includes imx-image-multimedia + OpenCV + QT5 + Machine Learning packages。Provided by layer：meta-imx/meta-sdk；

# 实际执行（在 imx-yocto-bsp 目录里面）
DISTRO=fsl-imx-xwayland MACHINE=imx8mmddr4evk bitbake imx-image-full -c populate_sdk
```

米尔提供了两个编译好的工具链构建脚本：

工具链文件名	描述
fsl-imx-xwayland-glibc-x86_64-myir-image-full-aarch64-myd imx8mm-toolchain-5.4-zeus.sh	包含一个独立的交叉开发工具链 还提供 qmake, 目标平台的 sysroot, Qt 应用开发所依赖的库 和头文件等。用户可以直接使用 这个 SDK 来建立一个独立的开 发环境
fsl-imx-xwayland-glibc-x86_64-meta-toolchain-aarch64-myd imx8mm-toolchain-5.4-zeus.sh	基础工具链，单独编译 Bootloader，Kernel 或者编译自己的应用程序

使用 GUN-A 编译器

单独编译过 nxp 官方的系统镜像 并生成了配套的工具链， 由于其安装复杂且包含很多我们很多用不到的库，而我们只需要单独编译 uboot kernel rootfs 等，所以使用 ARM 官方的交叉编译工具链。

TODO：目前暂时不知道 使用 ARM 官方编译器会不会导致 imx8mm 的 qt、ml 等特性用不上？！？

1. ARM GUN-A 官方编译器下载页面： GNU Toolchain | GNU-A Downloads – Arm Developer；
    - arm-linux-gnueabihf、aarch64-linux-gnu等ARM交叉编译GCC的区别 _Namcodream521的博客-CSDN博客；
    - 转：ARM交叉编译工具链分类说明 arm-linux-gnueabi和arm-linux-gnueabihf 的区别 Beyoungbehappy的博客-CSDN博客；
    - arm交叉编译器gnueabi、none-eabi、arm-eabi、gnueabihf等的区别 - 涛少& - 博客园 (cnblogs.com)；
2. 在 `x86_64 Linux hosted cross compilers` 下面找到 `AArch64 GNU/Linux target (aarch64-none-linux-gnu)`，并下载；（`AArch64 Linux hosted cross compilers` 下的编译器可以运行在 嵌入式板子 SoC 的 Linux 上）；
3. 使用 `tar xvf` 命令解压；并添加环境变量：

```
export ARCH=arm64
export CROSS_COMPILE=aarch64-none-linux-gnu-
export PATH=$PATH:/home/<路径>/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin/
```

4. 就行了，可以编译 uboot、imx-atf 等等。

使用 Linaro GCC 编译器

正点原子的文章`【正点原子】I.MX6U嵌入式Linux驱动开发指南V1.5.1`在`4.3.1.2 小节`里说到个别版本能编译通过但是不能运行，多换换版本试试。

到 Linaro Releases 下载适合的编译器，使用方法与上面类似。

关于编译器的选择

除了要注意 Linux 主机平台和目标平台要选对以外，还要注意选择支持操作系统的版本（并支持硬件浮点），这个版本支持 Linux 的进程等 API 的编译，由于 uboot 没有用到这些 Linxu API，所以这个版本的编译器也可以编译 uboot，可以说是全流程编译，所以为了方便选用这个版本的编译器：

- 对于 “使用 GUN-A 官方编译器”：带有 “bare-metal” 的为不支持操作系统的，除此之外的可以选择；
- 对于 “基于 yocto 的 bitbake”：选择 “构建应用层工具链” 一节去产生编译器。

分析 imx-mkimage

注：米尔是使用的 imx-mkimage 是旧版的（2020 上半年的），这里介绍的 imx-mkimage 是 2021.5 最新下载的，源码上有细微差别，但使用上没有区别（大概）。

- 先进入 imx-mkimage 目录，查看帮助：`make help`；
- 查看 针对 SOC=iMX8MM 平台 有哪些选项：打开具体平台目录下的 Makefile 文件 `vim ./iMX8M/soc.mak`，
    - 其中， flash_ddr4_evk ，代表：flash_ddr4_evk_no_hdmi，使用 firmware-imx 里面的 ddr4 固件；
    - 其中， flash_evk ，代表：flash_evk_no_hdmi，使用 firmware-imx 里面的 lpddr4 固件；
    - 其中， flash_evk_emmc_fastboot ，代表：flash_evk_no_hdmi_emmc_fastboot，使用 firmware-imx 里面的 lpddr4 固件，多加一个叫 emmc_fastboot 的东西，暂不清楚；
    - 其中，flash_evk_flexspi，使用 lpddr4 ，用于 imx8mm 的 M4 内核，把 M4 固件下载到 qspi；
    - 其中，flash_ddr4_evk_flexspi，使用 ddr4 ，用于 imx8mm 的 M4 内核，把 M4 固件下载到 qspi；米尔提供的的 imx-mkimage 在 soc.mak 中加了一句 `flash_ddr4_evk_qspi:  flash_ddr4_evk_flexspi`，然后使用 `make SOC=iMX8MM flash_ddr4_evk_qspi` 编译，效果一样。
    - 使用到的 firmware-imx 里面的固件（共四个，两个 dmem，两个 imem），需要提前放在 iMX8MM 目录中。
- 分析 imx-mkimage 中 iMX8MM 文件夹内的 soc.mak 文件的编译流程：

DDR4 的情况：

1. 输入 `make SOC=iMX8MM flash_ddr4_evk` 之后，解析为 `flash_ddr4_evk: flash_ddr4_evk_no_hdmi`；
2. 而 `flash_ddr4_evk_no_hdmi` 依赖 `u-boot-spl-ddr4.bin u-boot-ddr4-evk.itb` 这两个文件；
3. 前者 `u-boot-spl-ddr4.bin` 使用 firmware-imx 里面的 ddr4 固件合成，要提前移动到 iMX8MM 文件夹内；这里注意当 平台是 iMX8MP 时候，需要移动名字带 `_202006` 尾缀的，具体看 soc.mak 文件源码，其他平台无需移动带这个尾缀的；
4. 后者 `u-boot-ddr4-evk.itb` 使用设备树 .dtb 文件合成，依赖一个 .dtb 的设备树文件，imx-mkimage 只识别 全名为 `fsl-imx8mm-ddr4-evk.dtb` 或者 `imx8mm-ddr4-evk.dtb` 的文件。

LPDDR4 的情况：

1. 输入 `make SOC=iMX8MM flash_evk` 之后（或者使用 `flash_evk_emmc_fastboot`），解析为 `flash_evk: flash_evk_no_hdmi`；
2. 而 `flash_evk_no_hdmi` 依赖 `u-boot-spl-ddr.bin u-boot.itb` 两个文件；
3. 前者 `u-boot-spl-ddr.bin` 使用 firmware-imx 里面的 lpddr4 固件合成，要提前移动到 iMX8MM 文件夹内，对于 imx8mm ，无需名字带 `_202006` 尾缀；
4. 后者 `u-boot.itb` 使用设备树 .dtb 文件合成，依赖一个 .dtb 的设备树文件，只识别名为 `fsl-imx8mm-evk.dtb` 或者 `imx8mm-evk.dtb` 的设备数文件。

移植过程

下面以 NXP 提供的 uboot-imx 2019.04 中的 imx8mm 芯片为例。参考 "i.MX_Porting_Guide.pdf" 手册和 "MYD-C8MMX_Linux软件开发指南_V2.0.pdf" 手册为主，以及其他数十个 网络文章/视频 为辅。

1. 板级支持文件创建/修改：
    - 在 `board/<公司名>/<板子名>` 中找到一个与自己板子最相近的板子，`<板子名>` 就是与要移植的板子最相近的现成的板子目录；
    - 分别复制 `board/<公司名>/<板子名>/` 和 `board/<公司名>/common/` 到 `board/<新公司名>/<新板子名>/` 和 `board/<新公司名>/common/` 下面；
    - 修改 `<板子名>_.c` 名字为 `<新板子名>.c`，`<新板子名>` 里面可以带有标识尾缀 `_imx8mm_ddr4`，即 SoC 型号；修改 `board/<新公司名>/<新板子名>/` 下的 Makefile 文件，将其中的 `<板子名>.o` 修改为 `<新板子名>.o`；以下都按照 ddr4 版本来讲。
2. Kconfig 文件配置：
    - 在 `arch/arm/mach-imx/imx8m/Kconfig` 文件增加如下内容：

```
...
config TARGET_IMX8MM_EVK
    bool "imx8mm LPDDR4 EVK board"
    select IMX8MM
    select IMX8M_LPDDR4

config TARGET_IMX8MM_DDR4_EVK
    bool "imx8mm DDR4 EVK board"
    select IMX8MM
    select IMX8M_DDR4
...

以上是原有内容，照葫芦画瓢加下面的内容：
config TARGET_<新板子名（全大写）>              在下一个步骤会用到
    bool "<新板子名的全名描述 + board>"        板子上电，uboot 打印信息
    select IMX8MM
    select IMX8M_DDR4                          在这里区分使用 LPDDR4 还是 DDR4

endchoice
.................
source "board/<新公司名>/<新板子名>/Kconfig"   在下一个步骤会修改的文件的索引
```

    - 修改 `board/<新公司名>/<新板子名>/Kconfig` 文件，添加如下内容：

```
if TARGET_<新板子名（全大写）>   与上面的 "config TARGET_<新板子名（全大写）>"的 config 后面的一致
config SYS_BOARD
    default "<新板子名>"        代表 board/<新公司名>/ 下的新板子名文件夹
config SYS_VENDOR
    default "<新公司名>"        代表 board/ 下的新公司名文件夹
config SYS_CONFIG_NAME
    default "<新板子名>"        代表 nclude/configs/ 目录下的新板子名的头文件

source "board/<新公司名>/common/Kconfig"
endif
```

3. include 文件创建/修改：复制 `include/configs/<板子名>.h` 为一个新的 `include/configs/<新板子名>.h`，将 .h 文件的宏定义修改：

```
#ifndef __IMX8MM_EVK_H
#define __IMX8MM_EVK_H
改为
#ifndef __<新板名 全大写>_IMX8MM_H
#define __<新板名 全大写>_IMX8MM_H
...
#endif
```

   还可以 添加/修改 功能裁剪/选择 和 参数定义 的宏，功能定义宏的前缀 "`CONFIG_`"，参数定义宏的前缀 "`CFG_`"；

4. 板级设备树文件创建/修改：若新板子的原理图与原板子的高度一致，就比较好办，举例，复制 `arch/arm/dts/imx8mm.dtsi` 为一个新的 `arch/arm/dts/xxx-imx8mm.dtsi`（包括 imx8mm 系列共有的硬件外设的描述，一般不对其做改动），复制 `arch/arm/dts/imx8mm-evk.dts` 为一个新的 `arch/arm/dts/xxx-imx8mm-ddr4.dts`（板级设备树文件，按需修改，一般无需大改）。接着修改目录下的设备树 Makefile，增加内容如下：

```
...
imx8mp-evk.dtb \    插入下面一句
xxx-imx8mm-ddr4.dtb
...
```

   有关设备树的详细介绍请看《【主线剧情 番外02】设备树》。

5. 新板子配置文件创建/修改：
   - 复制 `configs/<板子名>_defconfig` 文件为一个新的 `configs/<新板子名>_defconfig`；
   - 在这里区分使用 LPDDR4 还是 DDR4，对于 uboot-imx 里面，`imx8mm_evk_defconfig` 用 lpddr4 和 emmc（大概对应于官方 EVKB 评估版），`imx8mm_ddr4_evk_defconfig` 用 ddr4 且支持 nand 和 emmc（大概对应于官方 EVK 评估版），这两个文件内容还有好几处不一样。
   - 修改新的配置文件如下：

```
CONFIG_TARGET_<新板子名（全大写）>=y    与上面的 <新板子名（全大写）> 一致
#CONFIG_TARGET_..._IMX8MM=y     原来板子的配置选项注释/删掉

CONFIG_DEFAULT_DEVICE_TREE="xxx-imx8mm"          在上一步的 arch/arm/dts/ 里面的新增加的设备树文件名字
                                                 To set the filename of the device tree source
                                                 编译最后会生成 arch/arm/dts/xxx-imx8mm-base.dtb 设备树文件
CONFIG_DEFAULT_FDT_FILE="xxx-imx8mm-ddr4.dtb"    生成扁平设备树文件名字，与上面的名字相同，加上尾缀 .dtb 即可

其他的配置选项适当修改，加注释失能，加尾缀"=y"使能
```

- 为了支持 uuu 工具（imx 系列 SoC USB 下载工具），查看其的手册 UUU.pdf（18 页），需要配置文件必须含有以下：

```
# To talk with uuu, uboot need enable fastboot. fastboot need auto run when detect boot from USB.
CONFIG_CMD_FASTBOOT=y
CONFIG_USB_FUNCTION_FASTBOOT=y
CONFIG_USB_GADGET=y
CONFIG_USB_GADGET_DOWNLOAD=y
CONFIG_USB_GADGET_MANUFACTURER="FSL"
CONFIG_USB_GADGET_VENDOR_NUM=0x0525
CONFIG_USB_GADGET_PRODUCT_NUM=0xa4a5
CONFIG_CI_UDC=y # UDC need change according system, - some system use CONFIG_USB_DWC3, some use CONFIG_USB_CDNS3
CONFIG_FSL_FASTBOOT=y
CONFIG_FASTBOOT=y
CONFIG_FASTBOOT_BUF_ADDR=0x83800000 # Address need change according - system, generally it can be the same as ${LOADADDR}
CONFIG_FASTBOOT_BUF_SIZE=0x40000000
CONFIG_FASTBOOT_FLASH=y
CONFIG_FASTBOOT_FLASH_MMC_DEV=2
CONFIG_EFI_PARTITION=y
CONFIG_ANDROID_BOOT_IMAGE=y

# If use SPL, SDP need be enabled.
CONFIG_SPL_USB_HOST_SUPPORT=y
CONFIG_SPL_USB_GADGET_SUPPORT=y
CONFIG_SPL_USB_SDP_SUPPORT=y
CONFIG_SDP_LOADADDR=0x40400000 # Address need change according - system, choose free memory
```

6. 编译：

```
make distclean
make <新板子名>_defconfig
make
```

产生所需的文件：

- u-boot-nodtb.bin；

    - spl / u-boot-spl.bin；
    - arch / arm / dts/ xxx-imx8mm.dtb。
7. 调试/调整 u-boot 源码直到其可以在板子上正常启动，串口（串口输出和交互正常）、命令行（u-boot 基本命令）、网口（能 dhcp 获得 IP，能 ping 通，能使用 ftfp 或者 nfs 命令）和 FLASH（能正常读和存 u-boot 的环境变量） 都可以正常使用。

重点修改 `include/configs/<新板子名>.c`、`include/configs/<新板子名>.h`、`arch/arm/dts/xxx-imx8mm-base.dts`、`configs/<新板子名>_defconfig` 文件；具体可用 `diff -yB <文件1> <文件2>` 命令比对芯片原厂和开发板厂的两个版本的 u-boot 哪里有改动，可以在 PC 电脑上面使用 Vscode 查看 uboot 源码并修改，然后进行编译和下载调试。具体对于网口、EMMC/SD、IO 等的修改可以参考《【正点原子】I.MX6U嵌入式Linux驱动开发指南》、《【百问网】嵌入式Linux应用开发完全手册_韦东山全系列视频文档全集》等手册，学习思想，触类旁通。

u-boot 编译流程

- 首先编译 `/$(CPU)/start.S`，对于不同的 CPU，还编译 `cpu/$(CPU)` 下的其他文件；
- 然后，对于平台/开发板相关的每个目录、每个通用目录都使用它们各自的 Makefile 生成相应的库；
- 将前两部分产生的 .o .a 文件根据 u-boot.lds 进行链接；得到最终文件。

更多 u-boot 源码分析等内容看上面 "u-boot 启动流程"、"u-boot 移植要点" 章节内容。

u-boot 图形化配置

在 uboot 目录里面先执行 `make <板名>_defconfig`，再执行 `make menuconfig`。方向键移动光标，enter 进入子菜单，空格键打开/关闭一个选项，双击 tab 键返回上一个目录。

执行 `make <板名>_defconfig` 之后会把配置信息写入 `.config` 文件，再执行 `make menuconfig`，menuconfig 通过读取 `.config` 文件来显示，经过 menuconfig 配置并保存，实际是修改了 `.config` 文件，此时执行 make 命令进行编译会用 `.config` 文件（在 `make <板名>_defconfig` 的基础上再经过 menuconfig 修改之后的最终的配置文件）的配置信息进行编译；如果执行 `make distclean` 会删除 `.config` 文件；所以通过 menuconfig 配置的是临时的，经过清理后就不存在了。

目标 menuconfig 依赖 scripts/kconfig/mconf，后者会调用 uboot 根目录下的 Kconfig 文件开始构建图形化配置界面，此 Kconfig 里面的 mainmenu 就是主菜单，有关此菜单的 Kconfig 语法可以参考《【正点原子】I.MX6U嵌入式Linux驱动开发指南》的 `34.2.2 Kconfig 语法简介` 一节以及 `34.3 添加自定义菜单` 一节。

构建图形化配置界面更多参考：

- Linux内核编译——Uboot_笑傲江湖-CSDN博客；
- uboot图形化配置浅析 - lzd626 - 博客园 (cnblogs.com)；
- ..

添加自定初始化代码

以下还是根据 imx8mm_evk 板子而言，不同板子函数的位置会有变动。

- 接着前文的工作，只需修改 `board/<公司名>/<板子名>/<板子名>.c` 文件及其对应的 .h 文件，在其中的 `board_early_init_f()`、`board_init()` 或者 `board_late_init()` 函数中添加自己要加的驱动代码。尽量只改这两个函数即可，其他初始化函数不用多动。

  各个初始化函数的说明（以下是按照执行顺序排列，以下函数都在 `board/<公司名>/<板子名>/<板子名>.c` 文件里面）：

    - The `board_early_init_f()` function is called at the very early phase if you define CONFIG_BOARD_EARLY_INIT_F. You can put the UART/SPI-NOR/NAND

IOMUX setup function, which requires to be set up at the very early phase；这个函数被调用在第一次初始化阶段，初始化看门狗、串口时钟等。
    - The `board_init()` function is called between `board_early_init_f` and `board_late_init`. You can do some general board-level setup here. If you do not define CONFIG_BOARD_EARLY_INIT_F, do not call printf before the UART setup is finished. Otherwise, the system may be down；第二次初始化阶，段推荐在这里改；
    - The `board_late_init()` function is called later. To debug the initialization code, put the initialization function into it；第二次初始化阶，推荐在这里改。
- 改变打印板子名：
    - u-boot logo：打开 `uboot-imx/tools/Makefile`，执行 `:/LOGO_BMP` 搜索“LOGO_BMP”，分析 Makefile 可知 默认使用 `$(srctree)/$(src)/logos/denx.bmp` 的图片，如果存在 `logos/$(BOARD).bmp` 或者 `logos/$(VENDOR).bmp` 就使用，可以直接在此按需替换。具体的显示 logo 的函数在 `uboot /board/esd/common/` 目录下的 `lcd.c` 文件中，大约在 81 行左右。屏幕的初始化配置在官方源码的 `<board name>.c` 文件里面。因为 uboot 启动的时间很短，一般都使用 Linux 的 logo，所以 uboot 中可以将 logo 图片删掉，或者在 `<board name>.c` 里面将显示有关的代码屏蔽掉。
    - 增加 uboot 额外的版本信息，在 uboot 目录里面先执行 `make <板名>_defconfig`，再执行 `make menuconfig`，在 `Console` 里面的 `Board specific string ... version string` 里面填入板子名和版本字符串。
    - 对于 imx8mm：
        - 在 `arch/arm/dts/imx8mm-ddr4-evk.dts` 里面的 model 改板名，板子上电后 uboot 的 `Model:` 会显示这里的信息；`board/board-info.c` 里面的 `show_board_info()` 会调用读取这个 `model`；
        - 在 `<板名>.c` 里的 `board_late_init()`，有一个设置 uboot 的 `board_name` 环境变量；
    - 对于其他 imx8 系列：
        1. 在 `<板子名>.c` 中，`board_early_init_f()` 的 `init_sequence_f[]` 里面有一个 `checkboard()` 函数；
        2. 直接一点的方法：delete `identify_board_id()` inside `checkboard()` and replace `printf("Board: ");` with `printf("Board: i.MX on <custom board>\n");`；
        3. 间接方法，The identification can be detected and printed by implementing the __print_board_info() function according to the identification method on the custom board，查看一下 `identify_board_id()` 和 `__print_board_info()` 这两个函数的内容，进行修改。
- 调试：在串口初始化之后的程序中可以添加 `printf()` 函数打印调试信息，初始化串口函数默认在 `board_early_init_f()` 里面。

添加自定义命令

u-boot 添加自定义命令。

添加命令行菜单界面

- UBOOT通用菜单menu的实现_leochen_career的专栏-CSDN博客;
- uboot中的快捷菜单的制作说明-小超hide-ChinaUnix博客;
- 如何在U-Boot中添加自定义命令_HowieXue 薛永浩的博客-CSDN博客;
- ...

合成最后的二进制文件

这里还是针对 nxp i.mx8mm 而言，armv8 的 SoC 应该都大同小异。

看 i.MX_Linux_Users_Guide.pdf 手册的 4.1.1 Bootloader 章节：On i.MX 8M SoC, the second program loader (SPL) is enabled in U-Boot. SPL is implemented as the first-level bootloader running on TCML (For i.MX 8M Nano and i.MX 8M Plus, the first-level bootloader runs in OCRAM). It is used to initialize DDR and load U-Boot, U-Boot DTB, Arm trusted firmware, and TEE OS (optional) from the boot device into the memory. After SPL completes loading the images, it jumps to the Arm trusted firmware BL31 directly. The BL31 starts the optional BL32 (TEE OS) and BL33 (U-Boot) for continue booting kernel.

In imx-boot, the SPL is packed with DDR Firmware together, so that ROM can load them into Arm Cortex-M4 TCML or OCRAM (only for i.MX 8M Nano and i.MX 8M Plus). The U-Boot, U-Boot DTB, Arm Trusted firmware, and TEE OS (optional) are packed into a FIT image, which is finally built into imx-boot.

看 i.MX_Porting_Guide.pdf 手册 5.3 OP-TEE booting flow 章节：On Arm V8, Arm has a specified preferred way to boot Secure Component with the Arm Trusted Firmware (ATF). The ATF first loads the OP-TEE OS. The OP-TEE OS initializes the secure world. Then, the ATF loads U-Boot that modifies the DTB on the fly to add a specific node to load Linux TEE drivers. Then, the Linux OS is booted.

需要的组件：(U-Boot, Arm Trusted Firmware, DDR firmware)

- imx-atf（Arm trusted firmware）：ATF 主要负责 Non-secure 环境和 secure 环境的切换（编译后产生所需的一个 bl31.bin 文件）。
- imx-uboot： nxp 提供的 适合 imx 系列的 u-boot（编译之后产生 u-boot-nodtb.bin、u-boot-spl.bin 和 设备树 .dtb 三个所需文件）。
- system controller firmware：直接提供的一个二进制文件，启动时由 spl 或者 uboot 调用，用以初始化 DDR。
- imx-mkimage：生成目标镜像的工具。

简短的说：

编译 u-boot，产生 u-boot-nodtb.bin、spl/u-boot-spl.bin、arch/arm/dts/xxx-imx8mm-base.dtb 文件后，再编译 imx-atf，再利用 imx-mkimage 镜像合成工具把所有编译好的文件连接整合成最后适合 imx8mm 的 u-boot 的二进制文件 flash.bin。然后可在 linux 主机 上使用 dd 命令烧入 sd 卡的存放 u-boot 的分区里面。或者使用根文件生成软件如 buildroot 或 yocto 合成最后的 sd/emmc 镜像文件。

具体步骤：参考 "环境构建" 小节 的第 4 ~ 6 步 "使用 imx-mkimage 链接合成所有文件生成最后二进制文件"。

其他补充说明

- 比较两个文件的异同，可用 `diff -yB 文件1 文件2` 命令，以显示全部源文和标出不同之处显示。
- Makefile 中，`obj-y += xx.o xx.o` 在编译时，只编译带有 `obj-y` 的；比如 `obj-$(CONFIG_MX6) += mx6/`，若 `CONFIG_MX6` 为 `y`，则会编译 `mx6/` 目录下的内容，否则不会。
- 在哪个目录添加了文件，若想其被编译，就在其所在目录的 Makefile 文件中添加 `obj-y += <文件名>.o`；`COBJS-$(CONFIG_CMD_MMC) += cmd_mmc.o`，这种是用变量控制的，若 `CONFIG_CMD_MMC` 变量为 `y`，则会添加编译，否则不会。
- 添加初始化代码：
    - 在 `include/configs/<板子名>.h` 里面添加初始化代码相关的宏定义；
    - 在 `board/<公司名>/<板子名>/<板子名>.c` 里，在 `board_xxx_init()` 函数里面添加对应的初始化函数，比如 `board_eth_init()` 里面添加 `dm9000_initialize()` 初始化函数。
- 增加个串口输出特定内容：

  在 `/common/board_f.c` 中加：`\#include <debug_uart.h>`；

  在 `void board_init_f(ulong boot_flags)` 中加一句 `printascii("uboot runnig.\r\n");`。
- 不同家的 SoC 和同一家但不同系列的 SoC 的启动流程会不同，有的 SoC 的 DDR 等初始化在 DCD 段，有的在 SPL 段，在用芯片公司提供的 u-boot 编译的时候 会用 mkimage 根据已经提供的默认的 .cfg 文件进行正确的连接组合产生 uboot 镜像/二进制文件，具体还是要多看官方手册，和理解性的看《【主线剧情01】ARM IMX6ULL 基础学习记录》，只看这里是容易迷糊的。
- i.mx8m 系列启动 Linux 所需的要素：
    - imx-boot (built by imx-mkimage), which includes SPL, U-Boot, Arm Trusted Firmware, DDR firmware；
    - HDMI firmware (only supported by i.MX 8M Quad)；
    - Linux kernel image；
    - A device tree file (.dtb) for the board being used；
    - A root file system (rootfs) for the particular Linux image。
- 如何理解曾博所说的“看国外数学教材提高智商”？ - 知乎 (zhihu.com)，其中说到 "国内大学很多教材简直魔幻，是“你只有学会了才看得懂”，然后学生是“需要看教材才学得会”，反复套娃死循环。 "；
- 写一个操作系统内核有多难？大概的内容、步骤是什么？ - 知乎 (zhihu.com)，其中说到 "人家通俗易懂，很有诚意，比起那些复制粘贴强行装B的书不知道要强多少倍。。。 "；
- NXP i.MX 8M Mini平台Linux系统启动时间优化 | i2SOM涯兔核；
- ...

署名

- 编辑整理：Github 页，知乎页
- 发表时间：始于 2021.5 且无终稿
- 首发平台：知乎 & Github
- 遵循协议：CC-BY-NC-SA 4.0
- 其他说明：

1. 本文件是“瞰百易”计划的一部分，尽量遵循 “二项玻”定则，致力于与网络上碎片化严重的现象泾渭分明（这中二魂…）！
2. 本文系广泛撷取、借鉴和整理，适合刚入门的人阅读和遵守，也适合已经有较多编程经验的人参看。如有错误恭谢指出！
3. 转载请注明作者及出处。整理不易，请多支持。