大奥特曼打小怪兽

Rockchip RK3399 - TPL/SPL方式加载uboot

目录

开发板： `NanoPC-T4` 开发板
`eMMC` ： `16GB`
`LPDDR3` ： `4GB`
显示屏： `15.6` 英寸 `HDMI` 接口显示屏
`u-boot` ： `2017.09`

`NanoPC-T4` 开发板，主控芯片是 `Rockchip RK3399` ， `big.LITTLE` 大小核架构，双 `Cortex-A72` 大核(`up to 2.0GHz`) + 四 `Cortex-A53` 小核结构(`up to 1.5GHz`)； `Cortex-A72` 处理器是 `Armv8-A` 架构下的一款高性能、低功耗的处理器。

我们接着上一节，介绍《 `Rockchip` 处理器启动支持的两种引导方式》：

- `TPL/SPL` 加载：使用 `Rockchip` 官方提供的T `PL/SPL` U-boot（就是我们上面说的小的 `uboot` ），该方式完全开源；
- 官方固件加载：使用 `Rockchip idbLoader` ，来自 `Rockchip rkbin` 项目的 `Rockchip DDR` 初始化 `bin` 和 `miniloader bin` ，该方式不开源；

这一节我们将介绍采用 `TPL/SPL` 方式，如何编译源码以及烧录程序到 `eMMC` ，从而完成 `uboot` 的启动。

回到顶部

一、 uboot

uboot 通常有三种：

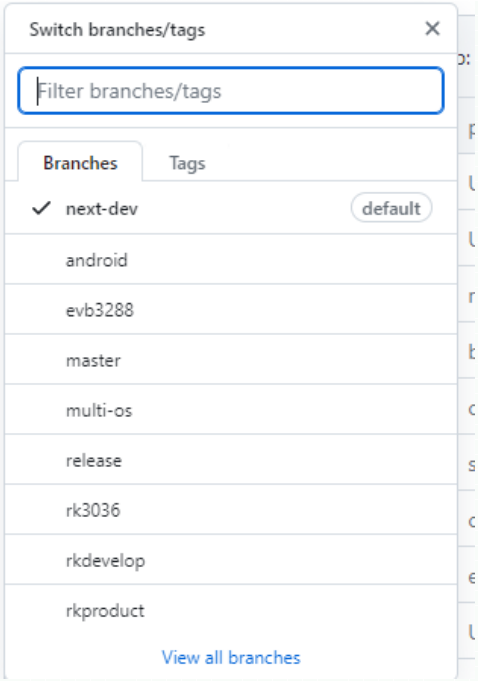
- uboot 官方源码： https://github.com/u-boot/u-boot ， uboot 官方源码是由 uboot 官方维护，支持非常全面的芯片，但对具体某款开发板支持情况一般；
- 半导体厂商瑞芯微官方源码： https://github.com/rockchip-linux/u-boot ，半导体厂商基于 uboot 官方源码进行修改，对自家的芯片进行完善的支持，针对某款处理器支持情况较好；
- 开发板友善之家官方源码： https://github.com/friendlyarm/uboot-rockchip ，开发板厂商基于半导体厂商维护的 uboot ，对自家的开发板进行板级支持，针对某款开发板支持情况较好；

我们不要上来就去移植 uboot 官方的源码，一般来说 uboot 官方的代码不做任何改动，是无法在我们板子上直接运行的。我们先去把 Rockchip 官方提供的 2017.09 版本的 uboot 代码下载下来，编译之后看看能不能运行，如果可以的话，再去参考 Rockchip 官方的 uboot 去移植最新版本的 uboot 。

这里有一点需要补充的是： Rockchip 官方提供的 uboot 2017.09 版本做了大量的改动，尤其是引导内核启动上，为了支持多种内核镜像加载方式，对 uboot 源码进行了大量修改，所以要求我们烧录的内核镜像也要按照官方指定的格式调整否则无法被 uboot 正确引导。

1.1 下载源码

我们可以在 Rockchip 的 github 上下载到芯片厂商提供的 u-boot 源码，如下图所示：



这里我们下载的是最新的 next-dev 分支的代码：

```
root@ubuntu:/work/sambashare/rk3399# git clone https://github.com/rockchip-linux/u-boot
```

这里我是下载到 /work/sambashare/rk3399 路径下的，这个路径后面专门存放与 rk3399 相关的内容。

进入到 u-boot 文件夹里，这就是我们需要的 uboot 的源码了，后面就可以进行二次开发了；

```
root@ubuntu:/work/sambashare/rk3399/u-boot# cd ..
root@ubuntu:/work/sambashare/rk3399# cd u-boot/
root@ubuntu:/work/sambashare/rk3399/u-boot# ls -l
总用量 488
drwxr-xr-x   2 root root   4096 5月   7 20:00 api
drwxr-xr-x  14 root root   4096 5月   7 20:00 arch
drwxr-xr-x 181 root root   4096 5月   7 20:00 board
drwxr-xr-x   6 root root   4096 5月   7 20:00 cmd
drwxr-xr-x   5 root root   4096 5月   7 20:00 common
-rw-r--r--   1 root root   2260 5月   7 20:00 config.mk
drwxr-xr-x   2 root root  69632 5月   7 20:00 configs
drwxr-xr-x   2 root root   4096 5月   7 20:00 disk
drwxr-xr-x  10 root root  12288 5月   7 20:00 doc
drwxr-xr-x   3 root root   4096 5月   7 20:00 Documentation
```

@超超加油 有，csdn，不过那个只会同步一次，文章都不是最新的...
--大奥特曼打小怪兽

5. Re:Rockchip RK3566 - orangepi-build脚本分析
博主有其他平台账号同步发文章吗
--超超加油

公告&打赏

```
drwxr-xr-x  56 root root   4096 5月   7 20:00 drivers
drwxr-xr-x   2 root root   4096 5月   7 20:00 dts
drwxr-xr-x   2 root root   4096 5月   7 20:00 env
drwxr-xr-x   4 root root   4096 5月   7 20:00 examples
drwxr-xr-x  12 root root   4096 5月   7 20:00 fs
drwxr-xr-x  32 root root  16384 5月   7 20:00 include
-rw-r--r--   1 root root   1863 5月   7 20:00 Kbuild
-rw-r--r--   1 root root  14162 5月   7 20:00 Kconfig
drwxr-xr-x  14 root root   4096 5月   7 20:00 lib
drwxr-xr-x   2 root root   4096 5月   7 20:00 Licenses
-rw-r--r--   1 root root  12587 5月   7 20:00 MAINTAINERS
-rw-r--r--   1 root root  56469 5月   7 20:00 Makefile
-rwxr-xr-x   1 root root  19845 5月   7 20:00 make.sh
drwxr-xr-x   2 root root   4096 5月   7 20:00 net
-rwxr-xr-x   1 root root   1640 5月   7 20:00 pack_resource.sh
drwxr-xr-x   5 root root   4096 5月   7 20:00 post
-rw-r--r--   1 root root     34 5月   7 20:00 PREUPLOAD.cfg
-rw-r--r--   1 root root 189024 5月   7 20:00 README
drwxr-xr-x   6 root root   4096 5月   7 20:00 scripts
-rw-r--r--   1 root root     17 5月   7 20:00 snapshot.commit
drwxr-xr-x  12 root root   4096 5月   7 20:00 test
drwxr-xr-x  16 root root   4096 5月   7 20:00 tools
```

需要注意的是：尽量不要下载 `release` 分支，最初我也下载了这个分支，这里分支默认配置有问题。其中配置项： `CONFIG_ROCKCHIP_SPL_RESERVE_IRAM=0x50000` ，表示 `SPL为ATF` 预留了 `0x50000` 大小的内存空间，这个地址范围是从 `0x00000008` 开始的，而 `0x50008` 之后是 `SPL` 代码。程序运行后， `SPL` 从 `eMMC` 加载 `bl31_0x00040000.bin` 到内存地址 `0x40000` 时，由于文件也比较大，直接覆盖了原有的 `SPL` 代码，导致程序直接卡死。当然修复这个问题也很简单，配置 `CONFIG_ROCKCHIP_SPL_RESERVE_IRAM=0` 即可。

1.2 配置 `uboot`

`uboot` 的编译分为两步：配置、编译。单板的默认配置在 `configs` 目录下，这里我们直接选择 `configs/evb-rk3399_defconfig` ，这是 `Rockchip` 评估板的配置：

▶ View Code

因此执行如下命令，生成 `.config` 文件：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# make evb-rk3399_defconfig V=1
```

输出如下（忽略编译器警告信息）：

```
root@ubuntu:/work/sambashare/rk3399/u-boot#  make evb-rk3399_defconfig V=1
make -f ./scripts/Makefile.build obj=scripts/basic
  cc -Wp,-MD,scripts/basic/.fixdep.d -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer
rm -f .tmp_quiet_recordmcount
make -f ./scripts/Makefile.build obj=scripts/kconfig evb-rk3399_defconfig
  cc -Wp,-MD,scripts/kconfig/.conf.o.d -Wall -Wstrict-prototypes -O2 -fomit-frame-point
  cat scripts/kconfig/zconf.tab.c_shipped > scripts/kconfig/zconf.tab.c
  cat scripts/kconfig/zconf.lex.c_shipped > scripts/kconfig/zconf.lex.c
  cat scripts/kconfig/zconf.hash.c_shipped > scripts/kconfig/zconf.hash.c
  cc -Wp,-MD,scripts/kconfig/.zconf.tab.o.d -Wall -Wstrict-prototypes -O2 -fomit-frame-
  cc  -o scripts/kconfig/conf scripts/kconfig/conf.o scripts/kconfig/zconf.tab.o
scripts/kconfig/conf  --defconfig=arch/../configs/evb-rk3399_defconfig Kconfig
#
# configuration written to .config
#
```

配置主要分为三个步骤：

- 第一步：执行 `make -f ./scripts/Makefile.build obj=scripts/basic` ，编译生成 `scripts/basic/fixdep` 工具；
- 第二步：执行 `make -f ./scripts/Makefile.build obj=scripts/kconfig evb-rk3399_defconfig` ，编译生成 `scripts/kcofig/conf` 工具；
- 第三步：执行 `scripts/kconfig/conf --defconfig=arch/../configs/evb-rk3399_defconfig Kconfig` ， `scripts/kconfig/conf` 根据 `evb-rk3399_defconfig` 生成 `.config` 配置文件；

这里会在当前路径下生成 `.config` 文件，这实际上是一个配置文件，这个文件是怎么生成的呢，实际上就是根据 `configs/evb-rk3399_defconfig` 文件以及我们 `make menuconfig` 看到的那些默认配置（或者说是各个目录下的 `Kconfig` 文件中有效的 `default` 项）生成的。

在进行 `make` 编译的时候，会根据这个文件生成 `include/config/auto.conf` 文件，同时在顶层 `Makefile` 会引入 `auto.conf` 文件：

```
ifeq ($(dot-config),1)
# Read in config
-include include/config/auto.conf
```

这样在执行 `make` 编译过程中，就可以根据 `include/config/auto.conf` 中的宏的定义编译不同的库文件。

1.2.1 配置串口波特率

`uboot` 中默认的调试串口波特率是 `1500000` ，有很多的调试终端不支持 `1.5M` 的波特率，我们可以把波特率重新配置下，在 `u-boot` 文件夹下输入命令： `make menuconfig` 配置；

```
Device Drivers  --->
      Serial drivers  --->
           (150000) Default baudrate
```

注意： 波特率数值如果无法删除，按 `CTRL+` 回车键尝试。

也可以直接编辑 `.config` 配置项：

```
CONFIG_BAUDRATE=1500000
```

1.2.2 配置 `uboot` 启动倒计时

如果在 `uboot` 启动倒计时结束之前，没有按下任何键，将会执行那么将执行也就是 `bootcmd` 中配置中的命令， `bootcmd` 中保存着默认的启动命令。

```
(5) delay in seconds before automatically booting
```

也可以直接编辑 `.config` 配置项：

```
CONFIG_BOOTDELAY=5
```

保存文件，输入文件名为 `evb-rk3399_defconfig` ，在当前路径下生成 `evb-rk3399_defconfig` ：存档：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# mv evb-rk3399_defconfig ./configs/
```

注意：如果需要配置生效，需要使用 `make distclean` 清除之前的配置，重新执行配置命令。

更多内容可以参考我之前写的有关 `s3c2440 uboot` 配置的文章：《 `make smdk2410_defconfig` 配置分析》，虽然 `SoC` 不同，但是 `make` 配置流程是一样的。

1.2.3 开启调试信息

在 `uboot` 启动时，如果我们想打印更加详细的信息，可以在 `include/configs/evb_rk3399.h` 中加入如下宏：

```
#define DEBUG
```

后面测试发现，设置了这个虽然可以输出一些调试信息，但是确无法进入 `uboot` 命令行，因此这个非特殊场景，尽量不要设置。

1.3 编译 `uboot`

执行 `make` 命令，生成 `u-boot` 文件:

```
root@ubuntu:/work/sambashare/rk3399/u-boot# make ARCH=arm CROSS_COMPILE=arm-linux-
```

编译过程我们会发现有一些错误，这些错误的处理我们在文章的最后单独介绍。

更多内容可以参考我之前写的有关 `s3c2440 uboot` 编译的文章：《 `make` 编译正向分析之顶层目标依赖》，虽然 `SoC` 不同，但是 `make` 编译流程是一样的。

1.4 image 镜像

成功编译之后，就会在 uboot 源码的根目录下产生多个可执行二进制文件以及编译过程文件，这些文件都是 u-bootxxx 的命名方式。这些文件由一些列名为 .xxx.cmd 的文件生成， .xxx.cmd 这些文件都是由编译系统产生的用于处理最终的可执行程序的。

在 uboot 根录下生成文件有：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ll u-boot*   Sys*
-rw-r--r-- 1 root root  153740 5月  14 10:30 System.map
-rwxr-xr-x 1 root root 6872736 5月  14 10:30 u-boot*
-rw-r--r-- 1 root root  931504 5月  14 10:30 u-boot.bin
-rw-r--r-- 1 root root   15808 5月  14 10:30 u-boot.cfg
-rw-r--r-- 1 root root    9996 5月  14 10:30 u-boot.cfg.configs
-rw-r--r-- 1 root root   51685 5月  14 10:30 u-boot.dtb       # 设备树
-rw-r--r-- 1 root root  931501 5月  14 10:30 u-boot-dtb.bin   # 等同u-boot.bin
-rw-r--r-- 1 root root  932864 5月  14 10:30 u-boot-dtb.img   # 等同u-boot.img
-rw-r--r-- 1 root root  932864 5月  14 10:30 u-boot.img
-rw-r--r-- 1 root root    1304 5月  14 10:30 u-boot.lds
-rw-r--r-- 1 root root  800454 5月  14 10:30 u-boot.map
-rwxr-xr-x 1 root root  879816 5月  14 10:30 u-boot-nodtb.bin*
-rwxr-xr-x 1 root root 2529568 5月  14 10:30 u-boot.srec*
-rw-r--r-- 1 root root  300850 5月  14 10:30 u-boot.sym
```

其中：

- u-boot ： 这个文件是编译后产生的 ELF 格式的最原始的 uboot 镜像文件，后续的文件都是由它产生的 .u-boot.cmd 这个命令脚本描述了如何产生；
- u-boot-nodtb.bin ： 这文件是使用编译工具链的 objcopy 工具从 u-boot 这个文件中提取来的，它只包含可执行的二进制代码。就是把 u-boot 这个文件中对于执行不需要的节区删除后剩余的仅执行需要的部分。由 .u-boot-nodtb.bin.cmd 这个命令脚本产生；
- u-boot-dtb.bin ： 在 u-boot-nodtb.bin 尾部拼接上设备树后形成的文件。由 .u-boot-dtb.bin.cmd 这个命令脚本产生；
- u-boot.bin ： 就是把 u-boot-dtb.bin 重命名得到的。由 .u-boot.bin.cmd 这个命令脚本产生；
- u-boot.img ： 在 u-boot.bin 开头拼接一些信息后形成的文件。由 .u-boot.img.cmd 这个命令脚本产生；
- u-boot-dtb.img ： 在 u-boot.bin 开头拼接一些信息后形成的文件。由 .u-boot-dtb.img.cmd 这个命令脚本产生；
- u-boot.srec ： S-Record 格式的镜像文件。由 .u-boot.srec.cmd 这个命令脚本产生；
- u-boot.sym ： 这个是从 u-boot 中导出的符号表文件。由 .u-boot.sym.cmd 这个命令脚本产生；
- u-boot.lds ： 编译使用的链接脚本文件。由 .u-boot.lds.cmd 这个命令脚本产生；
- u-boot.map ： 编译的内存映射文件。该文件是有编译工具链的连接器输出的；
- System.map ： 记录 u-boot 中各个符号在内核中位置，但是这个文件是使用了 nm 和 grep 工具来手动生成的；

由于开启了 SPL 和 TPL 的，因此，在编译 uboot 时会额外单独编译 SPL 、 TPL ，编译产生的镜像文件就存放在 ./spl 、 ./tpl 目录下。这下面的镜像生成方式与 uboot 基本是一模一样的。

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ls tpl
arch   cmd     drivers  env  include  u-boot.cfg      u-boot-tpl      u-boot-tpl.dtb
board  common  dts      fs   lib      u-boot-spl.lds  u-boot-tpl.bin  u-boot-tpl-dtb.bi
root@ubuntu:/work/sambashare/rk3399/u-boot# ls spl
arch   cmd     drivers  env  include  u-boot.cfg  u-boot-spl.bin  u-boot-spl-dtb.bin  u
board  common  dts      fs   lib      u-boot-spl  u-boot-spl.dtb  u-boot-spl.lds      u
```

以 SPL 为例：

- u-boot-spl ： 这个文件是编译后产生的 ELF 格式的 SPL 镜像文件，后续的文件都是由它产生的 .u-boot-spl.cmd 这个命令脚本描述了如何产生；

- `u-boot-spl-nodtb.bin`： 这文件是使用编译工具链的 `objcopy` 工具从 `u-boot-spl` 这个文件中提取来的，它只包含可执行的二进制代码。就是把 `u-boot-spl` 这个文件中对于执行不需要的节区删除后剩余的仅执行需要的部分。由 `.u-boot-spl-nodtb.bin.cmd` 这个命令脚本产生；
- `u-boot-spl-dtb.bin`： 在 `u-boot-nodtb.bin` 尾部依次拼接上 `u-boot-spl-pad.bin` 和 `u-boot-spl.dtb` 后形成的文件。由 `.u-boot-spl-dtb.bin.cmd` 这个命令脚本产生；
- `u-boot-spl.bin`： 就是把 `u-boot-dtb.bin` 重命名得到的。由 `.u-boot-spl.bin.cmd` 这个命令脚本产生；
- `u-boot-spl.lds`： 编译使用的链接脚本文件。由 `.u-boot-spl.lds.cmd` 这个命令脚本产生；
- `u-boot-spl.map`： 编译 `SPL` 的内存映射文件；
- `u-boot-spl.dtb`： 这个是编译好的设备树二进制文件。就是 `./dts/dt.dtb` 重命名得到的。`./dts/dt.dtb` 来自于 `arch/arm/dts/stm32f769-eval.dtb` 重命名；

回到顶部

二、idbloader.img

我们基于 `uboot` 源码编译出 `TPL/SPL` ，其中 `TPL` 负责实现 `DDR` 初始化， `TPL` 初始化结束之后会回跳到 `BootROM` 程序， `BootROM` 程序继续加载 `SPL` ， `SPL` 加载 `u-boot.itb` 文件，然后跳转到 `uboot` 执行。

`idbloader.img` 是由 `tpl/u-boot-tpl.bin` 和 `spl/u-boot-spl.bin` 文件生成，这里我们需要使用到 `tools` 目录下的 `mkimage` 工具。

2.1 `tpl/u-boot-tpl.bin`

在 `u-boot` 目录下执行：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# tools/mkimage -n rk3399 -T rksd -d tpl/u-bo
Image Type:   Rockchip RK33 (SD/MMC) boot image
Init Data Size: 81920 bytes
```

其中：

- `-n rk3399` 将镜像文件的名称设置为 `rk3399` ；
- `-T rksd` 将映像类型指定为 `Rockchip SD` 卡启动映像；
- `-d tpl/u-boot-tpl.bin` 将生成的 `TPL` 镜像文件 `tpl/u-boot-tpl.bin` 指定为输入文件，而 `idbloader.img` 则指定为输出文件。

生成 `idbloader.img` 文件：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ll idbloader.img
-rw-r--r-- 1 root root 83968 5月  14 10:38 idbloader.img
```

2.2 `spl/u-boot-spl.bin`

将 `spl/u-boot-spl.bin` 合并到 `idbloader.img` ：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# cat spl/u-boot-spl.bin >> idbloader.img
root@ubuntu:/work/sambashare/rk3399/u-boot# ll idbloader.img
-rw-r--r-- 1 root root 210675 5月  14 10:39 idbloader.img
```

回到顶部

三、 `u-boot.idb`

`u-boot.itb` 实际上是 `u-boot.img` 的另一个变种，也是通过 `mkimage` 构建出来的，依赖于 `u-boot.its u-boot.dtb u-boot-nodtb.bin` 这三个文件。

`mkimage` 工具在 `u-boot` 源码目录下的 `tools` 目录中，不过由于 `u-boot` 官方原本的 `FIT` 功能无法满足实际产品需要，所以 `RK` 平台对 `FIT` 功能进行了适配和优化，所以自然对 `mkimage` 工具的源代码进行了修改、优化；所以对于 `RK` 平台硬件，如果使用 `FIT` 格式镜像，必须使用 `RK u-boot` 源码编译生成的 `mkimage` 工具，不可使用 `u-boot` 原版的 `mkimage` 工具。

在顶层 `Makefile` 文件中：

```
u-boot.itb: u-boot-nodtb.bin dts/dt.dtb $(U_BOOT_ITS) FORCE
	$(call if_changed,mkfitimage)
```

这里的 `mkfitimage` 被设置为了：

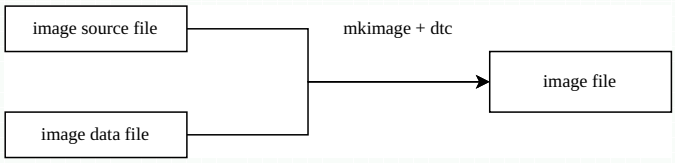
```
cmd_mkfitimage = $(objtree)/tools/mkimage $(MKIMAGEFLAGS_$(@F)) -f $(U_BOOT_ITS) -E $@
	$(if $(KBUILD_VERBOSE:1=), >$(MKIMAGEOUTPUT))
```

3.1 `FIT` 介绍

`FIT` 是 `flattened image tree` 的简称，它采用了 `device tree source filse (DTS)` 的语法，生成的 `image` 文件也和 `dtb` 文件类似（称做 `itb` ）。

注意：这一小节只是补充关于 `FIT` 的相关知识，并不是编译 `u-boot.itb` 的步骤。

3.1.1 生成步骤



其中 `image source file(.its)` 和 `device tree source file(.dts)` 类似，负责描述要生成的 `image file` 的信息。 `mkimage` 和 `dtc` 工具，可以将 `.its` 文件以及对应的 `image data file` ，打包成一个 `image file` 。

3.1.2 `image source file` 语法

`image source file` 的语法和 `device tree source file` 完全一样，只不过自定义了一些特有的节点，包括 `images` 、 `configurations` 等。说明如下：

3.1.2.1 `images` 节点

指定所要包含的二进制文件，可以指定多种类型的多个文件，例如 `u-boot.its` 中的包含了1个 `standalone image` 、5个 `firmware image` 、1个 `fdt image` 。每个文件都是 `images` 下的一个子 `node` ，例如：

```
atf@1 {
        description = "ARM Trusted Firmware";
        data = /incbin/("bl31_0x00040000.bin");
        type = "firmware";
        arch = "arm64";
        os = "arm-trusted-firmware";
        compression = "none";
        load = <0x00040000>;
        entry = <0x00040000>;
};
```

可以包含如下的关键字：

- `description` ：描述，可以随便写；
- `data` ：二进制文件的路径，格式为 `/incbin/("path/to/data/file.bin")` ；
- `type` ：二进制文件的类型， `standalone` ， `firmware` ， `kernel` ， `ramdisk` ， `flat_dt` 等；
- `arch` ：平台类型， `arm` ， `arm64` ， `i386` 等；
- `os` ：操作系统类型， `linux` 、 `vxworks` 等；
- `compression` ：二进制文件的压缩格式， `SPL` 、或 `uboot` 会按照执行的格式解压；
- `load` ：二进制文件的加载位置， `SPL` 、或 `uboot` 会把它 `copy` 对应的地址上；
- `entry` ：二进制文件入口地址，一般 `kernel image` 需要提供， `uboot` 会跳转到该地址上执行；
- `hash` ：使用的数据校验算法。

具体可以参考：

- 《 doc/uImage.FIT/source_file_format.txt 》;
- 《 RK3568 开发笔记整理之 FIT Image 》。

3.1.2.2 configurations 节点

可以将不同类型的二进制文件，根据不同的场景，组合起来，形成一个个的配置项， u-boot 在 boot 的时候，以配置项为单位加载、执行，这样就可以根据不同的场景，方便的选择不同的配置。下面是 u-boot.its 中的配置节点：

```
configurations {
        default = "config";
        config {
                description = "Rockchip armv8 with ATF";
                rollback-index = <0x0>;
                firmware = "atf@1";
                loadables = "uboot", "atf@2" , "atf@3" , "atf@4" , "atf@5" , "atf@6" ;
                fdt = "fdt";
                signature {
                        algo = "sha256,rsa2048";
                        padding = "pss";
                        key-name-hint = "dev";
                        sign-images = "fdt", "firmware", "loadables";
                };
        };

};
```

这里只包含了1种配置，默认配置项由 default 指定，当然也可以在运行时指定。

3.1.3 编译

FIT image 文件的编译过程很简单，根据实际情况，编写 image source file 之后（假设名称为 u-boot.its ），在命令行使用 mkimage 工具编译即可：

```
./tools/mkimage [-D dtc_options] [-f fit-image.its|-f auto|-f auto-conf|-F] [-b <dtb> [
           <dtb> file is used with -f auto, it may occur multiple times.
          -D => set all options for device tree compiler
          -f => input filename for FIT source
          -i => input filename for ramdisk file
          -E => place data outside of the FIT structure
          -B => align size in hex for FIT structure and header
          -b => append the device tree binary to the FIT
          -t => update the timestamp in the FIT
```

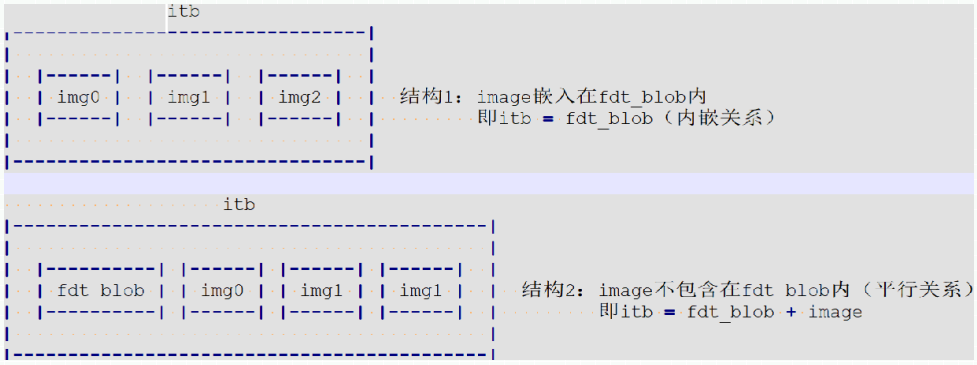
其中：

- -D ：指定 DTC (Device Tree Compiler) 编译器的选项;
- -f ：指定需要编译的 image source file ，并在后面指定需要生成的 image 文件（一般以 .itb 为后缀，例如 u-boot.itb ）;
- -i ：指定用于创建 FIT 镜像的 RAM disk 文件名;
- -E ：将 image data file 放置在FIT结构外的选项;
- -B ：指定 FIT 结构和头的对齐大小;
- -b ：支持将一个或多个设备树二进制文件附加到 FIT 文件中（可使用 -b <dtb> 多次指定）;
- -t ：更新 FIT 文件中的时间戳;

其中 -E 这个字段比较重要，它会影响生成的 itb 的文件布局;

- 如果没有指定该选项，其生成的 itb 文件格式和 dts 文件编译生成的 dtb 文件布局一样，包括 data 属性指定的 /incbin/("bl31_0x00040000.bin") 文件也会以二进制数据格式的形式放到 FIT 结构内;
- 如果指定了该选项，会为 data 属性指向的文件扩充 data-offset （指定文件的偏移，这个偏移是以 FIT 结构结束地址下一扇区起始地址开始计算的，即相对于 fdt_blob 末尾的位置偏移量）、以及 data-size （指定文件的大小）属性，而在 data 属性指向的二进制数据文件将会被追加到 FIT 结构的尾部（也是扇区对齐）；下文我们分析的 itb 文件布局格式就是这种;

有关这两种布局效果具体如下图所示：



我们可以采用如下方式生成 `u-boot.itb` 文件：

```
./tools/mkimage  -f  u-boot.its -E u-boot.itb
```

`u-boot.itb` 生成后，也可以使用 `mkimage` 命令查看它的信息：

```
tools/mkimage -l u-boot.itb
```

3.1.4 `itb` 文件布局

编译生成 `u-boot.itb` 文件的 `u-boot.its` 源文件如下：

▶ View Code

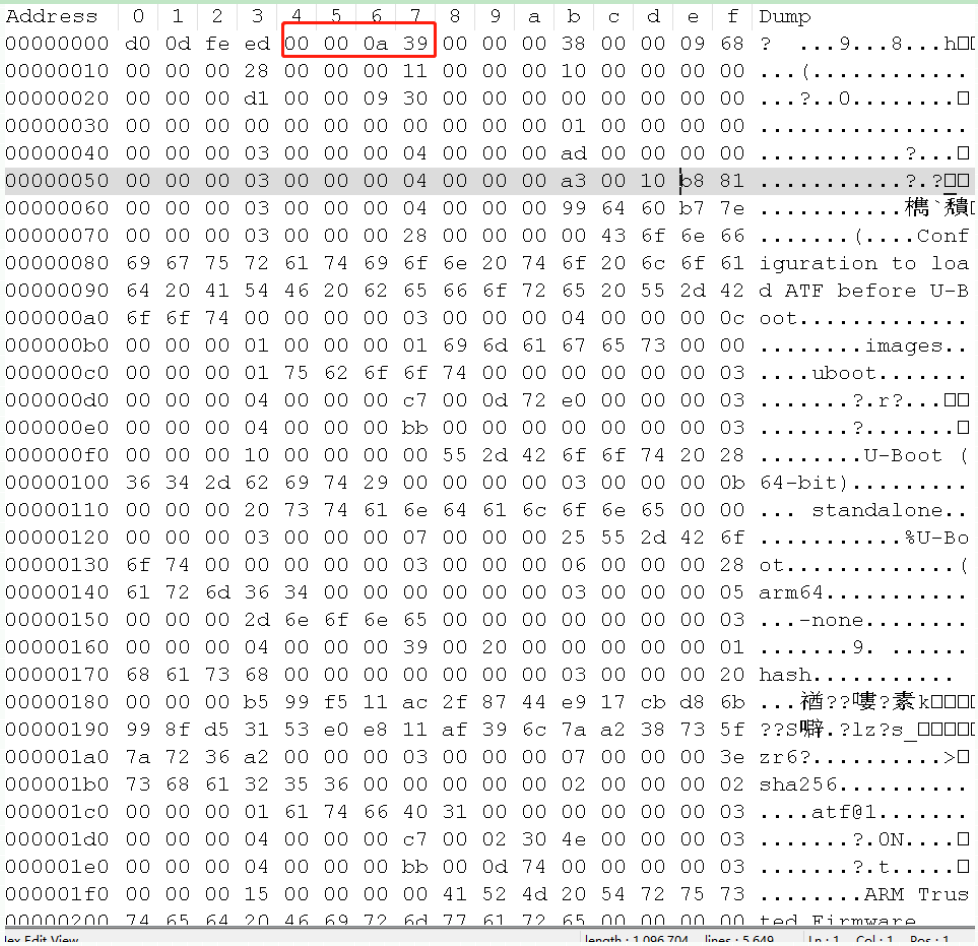
注意：这里我们补充一下 `u-boot.its` 文件的来源，在 `Makefile` 中有如下配置：

```
# Boards with more complex image requirments can provide an .its source file
# or a generator script
ifneq ($(CONFIG_SPL_FIT_SOURCE),"")     # 指定了u-boot.its文件
U_BOOT_ITS = $(subst ",,$(CONFIG_SPL_FIT_SOURCE))
else
ifneq ($(CONFIG_SPL_FIT_GENERATOR),"")  # 走这里
U_BOOT_ITS := u-boot.its
$(U_BOOT_ITS): FORCE
        $(srctree)/$(CONFIG_SPL_FIT_GENERATOR) \
        $(patsubst %,arch/$(ARCH)/dts/%.dtb,$(subst ",,$(CONFIG_OF_LIST))) > $@  # 利用p
endif
endif
```

由于 `configs/evb-rk3399_defconfig` 中配置了 `CONFIG_SPL_FIT_GENERATOR` ，因此这里会通过 `python` 脚本生成 `u-boot.its` ；

```
CONFIG_SPL_FIT_GENERATOR="arch/arm/mach-rockchip/make_fit_atf.py"
```

编译生成的 `u-boot.itb` 文件，以 `16` 进制查看：



```
Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  Dump
00000000 d0 0d fe ed 00 00 0a 39 00 00 00 38 00 00 09 68 ?  ...9...8...h□[
00000010 00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00 ...(............
00000020 00 00 00 d1 00 00 09 30 00 00 00 00 00 00 00 00 ...?..0........□
00000030 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 ................
00000040 00 00 00 03 00 00 00 04 00 00 00 ad 00 00 00 00 ............?...□
00000050 00 00 00 03 00 00 00 04 00 00 00 a3 00 10 b8 81 ............?.?□□
00000060 00 00 00 03 00 00 00 04 00 00 00 99 64 60 b7 7e ............橋`積[
00000070 00 00 00 03 00 00 00 28 00 00 00 00 43 6f 6e 66 .......(....Conf
00000080 69 67 75 72 61 74 69 6f 6e 20 74 6f 20 6c 6f 61 iguration to loa
00000090 64 20 41 54 46 20 62 65 66 6f 72 65 20 55 2d 42 d ATF before U-B
000000a0 6f 6f 74 00 00 00 00 03 00 00 00 04 00 00 00 0c oot.............
000000b0 00 00 00 01 00 00 00 01 69 6d 61 67 65 73 00 00 ........images..
000000c0 00 00 00 01 75 62 6f 6f 74 00 00 00 00 00 00 03 ....uboot.......
000000d0 00 00 00 04 00 00 00 c7 00 0d 72 e0 00 00 00 03 .......?.r?...□□
000000e0 00 00 00 04 00 00 00 bb 00 00 00 00 00 00 00 03 .......?.......□
000000f0 00 00 00 10 00 00 00 00 55 2d 42 6f 6f 74 20 28 ........U-Boot (
00000100 36 34 2d 62 69 74 29 00 00 00 00 03 00 00 00 0b 64-bit).........
00000110 00 00 00 20 73 74 61 6e 64 61 6c 6f 6e 65 00 00 ... standalone..
00000120 00 00 00 03 00 00 00 07 00 00 00 25 55 2d 42 6f ...........%U-Bo
00000130 6f 74 00 00 00 00 00 03 00 00 00 06 00 00 00 28 ot.............(
00000140 61 72 6d 36 34 00 00 00 00 00 00 03 00 00 00 05 arm64...........
00000150 00 00 00 2d 6e 6f 6e 65 00 00 00 00 00 00 00 03 ...-none........
00000160 00 00 00 04 00 00 00 39 00 20 00 00 00 00 00 01 .......9. ......
00000170 68 61 73 68 00 00 00 00 00 00 00 03 00 00 00 20 hash........... 
00000180 00 00 00 b5 99 f5 11 ac 2f 87 44 e9 17 cb d8 6b ...禧??嚷?素k□□□[
00000190 99 8f d5 31 53 e0 e8 11 af 39 6c 7a a2 38 73 5f ??S嚼.?lz?s_□□□□[
000001a0 7a 72 36 a2 00 00 00 03 00 00 00 07 00 00 00 3e zr6?...........>□
000001b0 73 68 61 32 35 36 00 00 00 00 00 02 00 00 00 02 sha256..........
000001c0 00 00 00 01 61 74 66 40 31 00 00 00 00 00 00 03 ....atf@1.......
000001d0 00 00 00 04 00 00 00 c7 00 02 30 4e 00 00 00 03 .......?.0N....□
000001e0 00 00 00 04 00 00 00 bb 00 0d 74 00 00 00 00 03 .......?.t.....□
000001f0 00 00 00 15 00 00 00 00 41 52 4d 20 54 72 75 73 ........ARM Trus
00000200 74 65 64 20 46 69 72 6d 77 61 72 65 00 00 00 00 ted Firmware....
```

由于 `itb` 文件布局和 `dtb` 文件布局一样，所以我们按照 `dtb` 文件布局格式来解读。

其中地址范围 `0x00000000~0x00000027` 表示的是 `fdt_header` 结构体的成员信息：

- 地址 `0x00000000` ：对应 `magic` ，表示设备树魔数，固定为 `0xd00dfeed` ；
- 地址 `0x00000004` ：对应 `totalsize` ，表示源 `u-boot.its` 文件打包后在 `u-boot.itb` 中所占的大小，由于我们编译指定了 `-E` 属性，因此这里计算的是不包含 `image data file` 文件的大小，更准确的说应该是 `FIT` 结构的大小。从上图可知这个值为 `0x00000a39` ；
- 地址 `0x00000008` ：对应 `off_dt_struct` ，表示 `structure block` 的偏移地址，为 `0x00000038` ；
- 地址 `0x0000000c` ：对应 `off_dt_strings` ，表示s `trings block` 的偏移地址，为 `0x00000968` ；
- 地址 `0x00000010` ：对应 `off_mem_rsvmap` ：表示 `memory reservation block` 的偏移地址，为 `0x00000028` ；
- 地址 `0x00000014` ：对应 `version` ，设备树版本的版本号为 `0x11` ；
- 地址 `0x00000018` ：对应 `last_comp_version` ：向下兼容版本号 `0x10` ；
- 地址 `0x0000001C` ：对应 `boot_cpuid_phys` ：在多核处理器中用于启动的主 `cpu` 的物理 `id` ，为 `0x0` ；
- 地址 `0x00000020` ：对应 `size_dt_strings` ， `strings block` 的大小为 `0xd1` ；
- 地址 `0x00000024` ： 对应 `size_dt_struct` ， `structure block` 的大小为 `0x00000930` ；

其中地址范围 `0x00000028~0x00000037` 表示的是 `fdt_reserve_entry` 结构体的成员信息：

- 对应结构体 `fdt_reserve_entry` ：它所在的地址为 `0x28` ， `u-boot.its` 设备树文件没有设置 `/memreserve/` ，所以 `address = 0x0` ， `size = 0x0` ；

其中地址范围 `0x00000038~0x00000967` 表示的是 `structure block` 信息：

- 地址 `0x00000038` ：值 `0x00000001` 表示的是设备节点的开始；
- 地址 `0x0000003c` ：表示设备节点的名字，这里是根节点，所以为 `0x00000000` ；

- 地址 `0x00000040` ：值 `0x00000003` 表示的是开始描述设备节点的一个属性；
- 地址 `0x00000044` ：表示这个属性值的长度为 `0x04` ；
- 地址 `0x00000048` ：表示这个属性的名字在 `strings block` 的偏移量是 `0xad` ，找到 `strings block` 的地址 `0x00000968+0xad=0xA15` 的地方，可知这个属性的名字是 `version` ；
- 地址 `0x0000004c` ：这个 `version` 属性的值是0；

我们来看一下 `uboot` 节点，由于 `uboot` 属性比较多，这里我并没有截全：

```
000000c0  00 00 00 01 75 62 6f 6f 74 00 00 00 00 00 00 03  ....uboot.......
000000d0  00 00 00 04 00 00 00 c7 00 0d 72 e0 00 00 00 03  ........?.r?...□□
000000e0  00 00 00 04 00 00 00 bb 00 00 00 00 00 00 00 03  ........?.....□
000000f0  00 00 00 10 00 00 00 00 55 2d 42 6f 6f 74 20 28  ........U-Boot (
00000100  36 34 2d 62 69 74 29 00 00 00 00 03 00 00 00 0b  64-bit).........
```

其属性 `data-size` 地址范围在 `0x000000c0~0x000000db` ：

- 地址 `0x000000c0` ：值 `0x00000001` 表示的是设备节点的开始；
- 地址 `0x000000c4` ：表示设备节点的名字，这里是 `uboot` 节点，所以为 `uboot` ，占用8个字节；
- 地址 `0x000000cc` ：值 `0x00000003` 表示的是开始描述设备节点的一个属性；
- 地址 `0x000000d0` ：表示这个属性值的长度为 `0x04` ；
- 地址 `0x000000d4` ：表示这个属性的名字在 `strings block` 的偏移量是 `0xc7` ，找到 `strings block` 的地址 `0x00000968+0xc7=0xa2f` 的地方，可知这个属性的名字是 `data-size` ；
- 地址 `0x000000d8` ：这个 `data-size` 属性的值是 `0x0d72e0` ；

属性 `data-offset` 地址范围在 `0x000000dc~0x000000eb` ：

- 地址 `0x000000dc` ：值 `0x00000003` 表示的是开始描述设备节点的一个属性；
- 地址 `0x000000e0` ：表示这个属性值的长度为 `0x04` ；
- 地址 `0x000000e4` ：表示这个属性的名字在 `strings block` 的偏移量是 `0xbb` ，找到 `strings block` 的地址 `0x00000968+0xbb=0xa23` 的地方，可知这个属性的名字是 `data-offset` ；
- 地址 `0x000000e8` ：这个 `data-offset` 属性的值是 `0x00` ；需要注意的这个描述的就是 `uboot` 节点 `data` 属性指定的 `u-boot-nodtb.bin` 文件的偏移，偏移 `0x00` 是从 `FIT` 文件结束下一扇区地址开始计算；也就是 `0xa3f` 取下一扇区起始地址，即 `0x0c00` ；

需要注意的是： `data-offset` 、 `data-size` 这两个属性在 `u-boot.its` 文件 `uboot` 节点中是没有的，这俩应该是自动扩充的属性的属性，用来描述 `data` 属性指向的 `u-boot-nodtb.bin` 文件的信息。

我们验证一下 `u-boot-nodtb.bin` 文件大小 `0xd72e0 =881376` ，和命令看到的完全匹配：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ll u-boot-nodtb.bin
-rwxr-xr-x 1 root root 881376 5月  14 18:27 u-boot-nodtb.bin*
```

我们验证一下文件内容， `u-boot-nodtb.bin` 源文件内容：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# hexdump u-boot-nodtb.bin -n 16
0000000 000a 1400 201f d503 0000 0020 0000 0000
```

定位到 `u-boot.itb` 文件地址 `0x0c00` ：

```
00000c00  0a 00 00 14 1f 20 03 d5  00 00 20 00 00 00 00 00  ..... .?. .....□
00000c10  e0 72 0d 00 00 00 00 00  e0 72 0d 00 00 00 00 00  郾......郾......□
```

这样我们可以得到一个文件内容布局表：

地址范围	偏移	大小	内容	加载到内存地址
0x0000 0000 ~ 0x0000 0a39		0x0a39	FIT：存放u-boot.its 文件信息	

地址范围	偏移	大小	内容	加载到内存地址
0x0000 0c00~ 0x000d 7ee0	0x00（按扇区大小对齐）	0xd72e0	uboot：u-boot-nodtb.bin	0x00200000
0x000d 8000~ 0x000f b04e	0xd7400（按扇区大小对齐）	0x2304e	atf@1：bl31_0x00040000.bin	0x00040000
自己计算	0xfa600（按扇区大小对齐）	0x1f58	atf@2：bl31_0xff3b0000.bin	0xff3b0000
自己计算	0xfc600（按扇区大小对齐）	0x1000	at3@3：bl31_0xff8c0000.bin	0xff8c0000
自己计算	0xfd600（按扇区大小对齐）	0x1000	atf@4：bl31_0xff8c1000.bin	0xff8c1000
自己计算	0xfe600（按扇区大小对齐）	0x00	atf@5：bl31_0xff8c2000.bin	0xff8c2000
自己计算	0xfe600（按扇区大小对齐）	0xc9e5	fdt：u-boot.dtb	

这里我只列出了一部分地址范围，有兴趣自己去搞吧，太费劲了。

3.1.5 `fdtdump`

`fdtdump` 命令可以查看 `itb` 文件内容：

```
fdtdump uboot.itb
```

3.2 编译 `ATF`

因为 `rk399` 是 `arm64` ，所以我们还需要编译 `ATF (Arm trust firmware)` ， `ATF` 主要负责在启动 `uboot` 之前把 `CPU` 从安全的 `EL3` 切换到 `EL2` ，然后跳转到 `uboot` ，并且在内核启动后负责启动其他的 `CPU` 。

3.2.1 下载源码

下载《 `arm-trusted-fireware` 》到 `uboot` 同级目录：

```
root@ubuntu:/work/sambashare/rk3399# git clone https://github.com/ARM-software/arm-trus
```

3.2.2 编译 `ATF`

运行如下命令：

```
root@ubuntu:/work/sambashare/rk3399/arm-trusted-firmware# make CROSS_COMPILE=arm-linux-
```

编译后报出一个缺少 `arm-none-eabi-gcc` 工具链的错误：

```
  CC      plat/rockchip/rk3399/drivers/gpio/rk3399_gpio.c
  CC      plat/rockchip/rk3399/drivers/pmu/m0_ctl.c
  CC      plat/rockchip/rk3399/drivers/pmu/pmu.c
make -C plat/rockchip/rk3399/drivers/m0 BUILD=/work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0
make[1]: Entering directory '/work/sambashare/rk3399/arm-trusted-firmware/plat/rockchip/rk3399/drivers/m0'
  CC      src/dram.c
arm-none-eabi-gcc -g -mcpu=cortex-m0 -mthumb -Wall -O3 -nostdlib -mfloat-abi=soft -ffunction-sections -fdata-sections -fomit-frame-pointer -fno-com
clude/ -I../../include/shared/ -MMD -MT /work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0/dram.o -c src/dram.c -o /work/sambasha
9/arm-trusted-firmware/build/rk3399/release/m0/dram.o
make[1]: arm-none-eabi-gcc：命令未找到
Makefile:124: recipe for target '/work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0/dram.o' failed
make[1]: *** [/work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0/dram.o] Error 127
make[1]: Leaving directory '/work/sambashare/rk3399/arm-trusted-firmware/plat/rockchip/rk3399/drivers/m0'
plat/rockchip/rk3399/platform.mk:110: recipe for target '/work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0/rk3399m0.bin' failed
make: *** [/work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/m0/rk3399m0.bin] Error 2
```

安装该工具链并重新编译：

```
root@ubuntu:/work/sambashare/rk3399/arm-trusted-firmware# sudo apt-get install gcc-arm-
```

```
root@ubuntu:/work/sambashare/rk3399/arm-trusted-firmware# make CROSS_COMPILE=arm-linux-
```

最终编译出来的目标文件为： `build/rk3399/release/bl31/bl31.elf` ， 这个文件需要和编译出来的 `uboot` 一起打包成 `fit` 格式的镜像才能被 `SPL` 加载。

```
  AS      plat/common/aarch64/platform_helpers.S
  AS      plat/common/aarch64/crash_console_helpers.S
  PP      bl31/bl31.ld.S
  LD      /work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/bl31/bl31.elf

Built /work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/bl31/bl31.elf successfully

  OD      /work/sambashare/rk3399/arm-trusted-firmware/build/rk3399/release/bl31/bl31.dump
root@zhengyang:/work/sambashare/rk3399/arm-trusted-firmware#
```

3.3 生成 `u-boot.itb`

3.3.1 拷贝 `bl31.elf`

将 `bl31.elf` 拷贝到 `uboot` 根目录下：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# cp /work/sambashare/rk3399/arm-trusted-firm
```

3.3.2 编译

然后执行编译命令：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# make u-boot.itb ARCH=arm CROSS_COMPILE=arm-
```

这里提示我们缺少 `python` 依赖 `elffile` ，如下图所示：

```
arm-linux-ld: warning: u-boot has a LOAD segment with RWX permissions
  OBJCOPY u-boot-nodtb.bin
start=$(arm-linux-nm u-boot | grep __rel_dyn_start | cut -f 1 -d ' '); end=$(arm-linux-nm u-boot |
); tools/relocate-rela u-boot-nodtb.bin 0x00200000 $start $end
make[2]: 'arch/arm/dts/rk3399-evb.dtb' is up to date.
./"arch/arm/mach-rockchip/make_fit_atf.py" \
arch/arm/dts/rk3399-evb.dtb > u-boot.its
Traceback (most recent call last):
  File "./arch/arm/mach-rockchip/make_fit_atf.py", line 15, in <module>
    from elftools.elf.elffile import ELFFile
ImportError: No module named elftools.elf.elffile
Makefile:1003: recipe for target 'u-boot.its' failed
make: *** [u-boot.its] Error 1
root@zhengyang:/work/sambashare/rk3399/u-boot#
```

我们直接使用如下命令安装 `python` 依赖包（需要注意自己的 `python` 版，必须是 `2.7` 版本，使用 `pip2` 安装依赖）：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# pip2 install pyelftools
```

重新编译生成 `u-boot.itb` 文件：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ls -l u-boot.itb
-rw-r--r-- 1 root root 1095168 5月  14 10:40 u-boot.itb
```

如果出现1： `dtc: not found make` 错误，如下：

```
arm-linux-ld: warning: u-boot has a LOAD segment with RWX permissio
  OBJCOPY u-boot-nodtb.bin
start=$(arm-linux-nm u-boot | grep __rel_dyn_start | cut -f 1 -d '
ela u-boot-nodtb.bin 0x00200000 $start $end
make[2]: 'arch/arm/dts/rk3399-evb.dtb' is up to date.
./"arch/arm/mach-rockchip/make_fit_atf.py" \
arch/arm/dts/rk3399-evb.dtb > u-boot.its
  MKIMAGE u-boot.itb
sh: 1: dtc: not found
./tools/mkimage: Can't read u-boot.itb.tmp: Invalid argument
Makefile:1035: recipe for target 'u-boot.itb' failed
make: *** [u-boot.itb] Error 255
```

`tc` 是 `device-tree-compiler` 的缩写，即设备树编译器，说明系统中没有安装这个编译器，安装设备树编译器重新编译即可：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# sudo apt-get install device-tree-compiler
```

回到顶部

四、 `rkdeveloptool`

`rkdeveloptool` 是 `Rockchip` 提供的一个与 `Rockusb` 设备进行通信的工具，通过该工具我们可以将镜像文件下载到开发板的 `eMMC` 。它被认为是 `upgrade_tool` 的一个开源版本，只有很少区别。

要使用 `rkdeveloptool` 进行升级，首先要知道 `rkdeveloptool` 是基于什么情况下才会起作用的，是在 `SoC` 进入 `MASKROM` 模式后而且跟主机通过 `USB` 连接，因为这个时候主板的 `DDR` 并没有初始化，而升级过程是需要很大的内存空间的，所以升级之前第一步要做的就是执行 `rkdeveloptool db rkxx_loader_vx.xx.bin` （这个固件本质上也是 `idbloader.img` ），只不过这时候只是在内存中执行，如果不执行db命令的话其他的命令则无法执行因为没有做内存初始化工作。

4.1 下载源码

在 `/work/sambashare/rk3399` 目录下执行如下命令：

```
root@ubuntu:/work/sambashare/rk3399# git clone https://github.com/rockchip-linux/rkdeve
```

4.2 配置

首先安装 `libusb` 与 `udev` ，例如对于 `ubuntu` ：

```
root@ubuntu:/work/sambashare/rk3399# sudo apt-get install libudev-dev libusb-1.0-0-dev
```

切换到 `rkdeveloptool/` 目录进行配置：

```
root@ubuntu:/work/sambashare/rk3399# cd rkdeveloptool/
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# autoreconf -i
configure.ac:12: installing 'cfg/compile'
configure.ac:19: installing 'cfg/config.guess'
configure.ac:19: installing 'cfg/config.sub'
configure.ac:7: installing 'cfg/install-sh'
configure.ac:7: installing 'cfg/missing'
Makefile.am: installing 'cfg/depcomp'
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# ./configure
```

4.3 编译安装

运行如下命令：

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# make
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# make install
```

如果遇到如下编译错误：

```
./configure: line 4269: syntax error near unexpected token `LIBUSB1,libusb-1.0'
./configure: line 4269: `PKG_CHECK_MODULES(LIBUSB1,libusb-1.0)'
```

还需要安装 `pkg-config` 与 `libusb-1.0` ：

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# sudo apt-get install pkg-config libu
```

编译完成后，在当前路径下生成 `rkdeveloptool` 可执行文件：

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# ll rkdeveloptool
-rwxr-xr-x 1 root root 1059720 5月  11 19:56 rkdeveloptool*
```

4.4 `rk3399_loader_v1.27.126.bin`

由于 `SoC` 进入到 `MASKROM` 模式后，目标板子会运行 `Rockusb` 驱动程序。在 `MASKROM` 模式下，需要使用到 `DDR` ，因此需要下载固件进行 `DDR` 的初始化。

《 `Rockchip rkbin` 项目》提供了 `ddr.bin` 、 `usbplug.bin` 、 `miniloader.bin` 这三个包：

- `ddr.bin` ：等价于我们之前说的 `TPL` ，用于初始化 `DDR` ；
- `usbplug.bin` ： `Rockusb` 驱动程序，用于将程序通过 `usb` 下载到 `eMMC` ；
- `miniloader.bin` ： `Rockchip` 修改的一个 `bootloader` ，等价于我们之前说的 `SPL` ，用于加载 `uboot` ；

4.4.1 下载 `rkbin`

我们可以在 `Rockchip` 的 `github` 上下载到 `Rockchip rkbin` 项目，如下所示：

```
root@ubuntu:/work/sambashare/rk3399# git clone https://github.com/rockchip-linux/rkbin.
```

4.4.2 合并

在 `rkbin` 目录下执行如下命令，可以将 `ddr.bin` 、 `usbplug.bin` 、 `miniloader.bin` 这三个包合并：

```
root@ubuntu:/work/sambashare/rk3399# cd rkbin/
root@ubuntu:/work/sambashare/rk3399/rkbin# tools/boot_merger  /work/sambashare/rk3399/r
********boot_merger ver 1.2********
Info:Pack loader ok.
root@ubuntu:/work/sambashare/rk3399/rkbin# ll rk3399_loader_v1.27.126.bin
-rw-r--r-- 1 root root 465230 5月  11 20:06 rk3399_loader_v1.27.126.bin
```

可以根据自己的需求可以在 `./RKBOOT/RK3399MINIALL.ini` 修改 `ddr` 、 `usbplug` 、 `miniloader` ：

```
[CHIP_NAME]
NAME=RK330C
[VERSION]
MAJOR=1
MINOR=26
[CODE471_OPTION]
NUM=1
Path1=bin/rk33/rk3399_ddr_800MHz_v1.27.bin
Sleep=1
[CODE472_OPTION]
NUM=1
Path1=bin/rk33/rk3399_usbplug_v1.26.bin
[LOADER_OPTION]
NUM=2
LOADER1=FlashData
LOADER2=FlashBoot
FlashData=bin/rk33/rk3399_ddr_800MHz_v1.27.bin
FlashBoot=bin/rk33/rk3399_miniloader_v1.26.bin
[OUTPUT]
PATH=rk3399_loader_v1.27.126.bin
```

将 `rk3399_loader_v1.27.126.bin` 拷贝到 `rkdeveloptool` 路径下：

```
root@ubuntu:/work/sambashare/rk3399/rkbin# cp rk3399_loader_v1.27.126.bin  ../rkdevelop
```

回到顶部

五、烧录程序

烧录方法有两种：

- 一种是基于 `Rockchip` 的官方烧录工具 `RKDevTool` ；官方 `RKDevTool` 是基于 `recovery` 模式实现的，如果板子带有 `recovery` 按键，可以使用这种方式；
- 另外一种是在开发板上使用 `rkdeveloptool` 工具直接烧写 `eMMC` ；这里我们采用 `rkdeveloptool` 烧录的方式；

5.1 准备镜像

我们需按照之前的流程得到了 `idbloader.img` 、 `u-boot.itb` 文件，由于我们这里不进行内核和根文件系统的烧录，所以暂时不需要准备这俩。

按照 `Rockchip` 官方要求将 `idbloader.img` 烧录到 `eMMC` 的 `0x40` 扇区， `u-boot.itb` 烧录到 `0x4000` 扇区。

我们需要将 `idbloader.img` 、 `u-boot.itb` 拷贝到 `rkdeveloptool` 路径下：

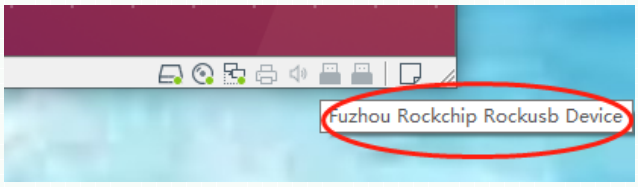
```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# cp ../u-boot/u-boot.itb ./
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# cp ../u-boot/idbloader.img ./
```

5.2 进入 `MASKROM` 升级模式

NanoPC-T4 开发板如需进入 MASKROM 升级模式，需要进入如下操作：

- 将开发板连接上电源，并且连接 Type-C 数据线到 PC ；
- 按住 BOOT 键再长按 Power 键开机（保持按下 BOOT 键5秒以上），将强制进入 MASKROM 模式。

一般电脑识别到 USB 连接，都会发出声音。或者观察虚拟机右下角是否突然多个 USB 设备：右键点击链接；



5.3 烧录

使用下载引导命令去使目标机器初始化 DDR与运行 usbplug （初始化 DDR 的原因是由于升级需要很大的内存，所以需要使用到 DDR ）；

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# rkdeveloptool db rk3399_loader_v1.27
Downloading bootloader succeeded.
```

由于 BootROM 启动会将 rk3399_loader_v1.27.126.bin 将在到内部 SRAM 中，然后跳转到 ddr.bin 代码进行 DDR 的初始化， ddr.bin 执行之后会回跳到 BootROM 程序， BootROM 程序继续加载 usbplug.bin ，由 usbplug.bin 完成程序的下载以及烧录到 eMMC 。

如果接上串口的话，执行完这一步可以看到如下输出信息：

▶ View Code

使用 wl 命令烧写镜像到目标机器的 eMMC ，需要注意的是访问 DDR 所需的所有其他命令都应在使用 db 命令之后才能使用；

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# rkdeveloptool wl 0x40 idbloader.img
Write LBA from file (100%)
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# rkdeveloptool wl 0x4000 u-boot.itb
Write LBA from file (100%)
```

在烧写镜像完成后使用 rd 命令重启目标机器：

```
root@ubuntu:/work/sambashare/rk3399/rkdeveloptool# rkdeveloptool rd
Reset Device OK.
```

需要注意的是：如果这个时候你也烧录了内核程序，执行完 rd 命令后是无法进入 uboot 命令的的，这里会直接启动内核。

回到顶部

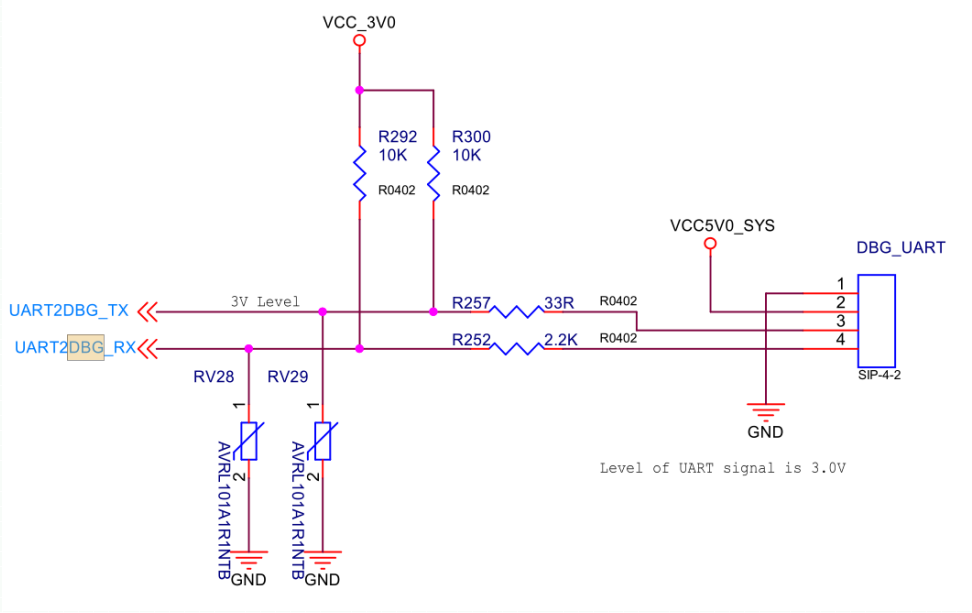
六、测试

6.1 串口连接

使用准备好的 USB 转串口适配器和连接线(需另购)，连接开发板，我使用的开发板提供了 DEG_UART 的4个引脚，其使用的是 RK3399 的 UART2 ；

引脚	开发板接口	USB转串口
1	GND	GND
2	VCC 5V	VCC
3	UART2DBG_TX	RX
4	UART2DBG_RX	TX

其电路原理图如下：



需要注意的是：这四根线都要接上，我只接 `TX` 、 `RX` 串口输出数据都是乱码。

6.2 `MobaXterm`

这里我使用的串口调试工具是 `MobaXterm` ，选择串口端口，设置波特率为 `1500000` ，8位数据位，1位停止位。

6.3 上电

重启开发板，通过串口打印输出：

▶ View Code

在倒计时执行完之前，按 `CTRL+C` 即可进入 `uboot` 命令行。

需要注意的是：有时候按 `CTRL+C` 并没有进 `入uboot` 命令行，可能串口有问题，没有接收到输入内容么？试着将串口重新连接，切换端口号试试。如果实在不行，可以尝试修改 `common/autoboot.c` 函数 `__abortboot` ，将 `if (ctrlc())` 修改为 `if (tstc())` ，这样按下任何键都可以进入 `uboot` 命令行。或者干脆函数返回1。

6.3.1 查看环境变量

在命令行输入 `print` ，查看所有环境变量：

▶ View Code

其中比较重要的环境变量有：

- `bootcmd` ：通过 `CONFIG_BOOTCOMMAND` 设置，用来启动内核的命令；

- `bootdelay`：通过`CONFIG_BOOTDELAY`设置，`uboot`启动倒计时，默认值为`5s`，只有设置了`bootcmd`，它才有用；
- `baudrate`：通过`CONFIG_BAUDRATE`设置，波特率默认为`1500000`；
- `ipaddr`：通过`CONFIG_IPADDR`设置，`IP`地址；可以不设置，使用`dhcp`命令来从路由器获取`IP`地址；
- `serverip`：通过`CONFIG_SERVERIP`设置，服务器`IP`地址；也就是`ubuntu`主机`IP`地址，用于调试代码；
- `netmask`：通过`CONFIG_NETMASK`设置，子网掩码；
- `gatewayip`：通过`CONFIG_GATEWAYIP`设置，网关地址；
- `ethaddr`：通过`CONFIG_ETHADDR`设置，网卡地址；如果设置了`CONFIG_NET_RANDOM_ETHADDR`此宏的话就会随机分配网卡物理地址；
- `bootargs`：通过`CONFIG_BOOTARGS`设置，启动参数；

这些配置信息大部分配置在`include/configs/evb_rk3399.h`、`include/configs/rk3399_common.h`、`include/configs/rockchip-common.h`、.`config`文件中。

6.3.2 内核启动命令

这里我们重点看一下启动内核的命令：

```
bootcmd=boot_android ${devtype} ${devnum};boot_fit;bootrkp;run distro_bootcmd;
bootcmd_dhcp=run boot_net_usb_start; if dhcp ${scriptaddr} ${boot_script_dhcp}; then so
bootcmd_mmc0=setenv devnum 0; run mmc_boot
bootcmd_mmc1=setenv devnum 1; run mmc_boot
bootcmd_pxe=run boot_net_usb_start; dhcp; if pxe get; then pxe boot; fi
bootcmd_usb0=setenv devnum 0; run usb_boot
distro_bootcmd=for target in ${boot_targets}; do run bootcmd_${target}; done
```

咦，看到这里是不是很奇怪，我记得我们在学习`Mini2440`的时候，`bootcmd`配置的很简单：

```
nand read 0x30000000 kernel; bootm 0x30000000
```

只是把`uImage从Nand Flash`内核分区加载到内存，然后直接`bootm <Legacy uImage addr>`，即可启动内核。这里咋搞了一大堆启动相关的命令呢?

实际上`Rockchip`为了支持从各种外部设备启动，同时也为了支持各种启动镜像格式，采用了《`Distro Bootcmd`启动机制》。更多细节可以参考《树莓派`4B (rpi4b)`引导`ubuntu`分析.md》。

6.3.3 查看板子信息

在命令行输入`bdinfo`，查看板子信息；

```
=> bdinfo
arch_number = 0x00000000
boot_params = 0x00000000
DRAM bank   = 0x00000000
-> start    = 0x00200000
-> size     = 0xF7E00000
baudrate    = 115200 bps
TLB addr    = 0xF7FF0000
relocaddr   = 0xF5DA0000
reloc off   = 0xF5BA0000
irq_sp      = 0xF3D87A10
sp start    = 0xF3D87A10
FB base     = 0x00000000
Early malloc usage: 14b0 / 4000
fdt_blob = 00000000f3d87a20
```

6.3.4 查看版本信息

在命令行输入`version`，查看板子信息；

```
=> version
U-Boot 2017.09-gef1dd65-dirty #root (May 14 2023 - 12:08:32 +0800)
```

```
arm-linux-gcc (Arm GNU Toolchain 12.2.Rel1 (Build arm-12.24)) 12.2.1 20221205
GNU ld (Arm GNU Toolchain 12.2.Rel1 (Build arm-12.24)) 2.39.0.20221210
```

6.4 设置环境变量

通常环境变量是存放在外部存储设备中， `uboot` 启动的时候会将环境变量读取 `DDR` 中。所以使用命令 `setenv` 修改的 `DDR` 中的环境变量值，修改以后要使用 `save` 命令将修改后的环境变量保存到 `eMMC` 中，否则的话 `uboot` 下一次重启会继续使用以前的环境变量值。

6.4.1 设置 `ip`

我们设置以下环境变量：

```
=> set ipaddr 192.168.0.105
Unknown command 'set' - try 'help'
=> setenv ipaddr 192.168.0.105
=> setenv serverip 192.168.0.200
=> setenv netmask 255.255.255.0
=> setenv gatewayip 192.168.0.1
```

这里 `192.168.0.200` 是我主机 `ubuntu` 的 `ip` 地址，而 `192.168.0.105` 是我为开发板设置的 `ip` 地址。

6.4.2 保存

进行保存：

```
=> saveenv
Saving Environment to nowhere...
```

看这意思，应该是我们没有设置环境变量的保存位置，参考之前写的这篇文章：《设置环境变量保存位置》，我们执行 `make menuconfig` ，配置：

```
Environment --->
    Select the location of the environment (Environment is not stored)  --->
           (X) Environment in an MMC device
```

需要注意的是环境变量默认存储在 `eMMC` 地址 `0x3f8000` ，大小： `0x8000` ；需要注意的是不要覆盖了其他分区。

```
CONFIG_ENV_SIZE=0x8000
CONFIG_ENV_OFFSET=0x3f8000
```

按照之前的步骤，重新编译，烧录程序即可。再次设置环境变量，保存：

```
=> setenv ipaddr 192.168.0.105
=> setenv serverip 192.168.0.200
=> setenv netmask 255.255.255.0
=> setenv gatewayip 192.168.0.1
=> saveenv
Saving Environment to MMC...
Writing to MMC(0)... done
```

如果断电重启或者通过 `reset` 命令重启，你会发现环境变量并没有保存成功。这个主要是因为 `common/board_r.c` 文件中 `init_sequence_r` 数组将 `initr_mmc` 放在了 `initr_env` 之后，导致初始化环境变量的时候 `eMMC` 还未初始化，修改方法，就是经将 `initr_mmc` 方法放到 `initr_env` 方法之前：

```
#ifdef CONFIG_MMC
        initr_mmc,
#endif

#ifndef CONFIG_USING_KERNEL_DTB
        /* init before storage(for: devtype, devnum, ...) */
        initr_env,
#endif
```

6.4.3 测试

接着我们测试一下开发板网络是否能用，这里我们直接 `ping` 服务器地址：

```
=> ping 192.168.0.200
ethernet@fe300000 Waiting for PHY auto negotiation to complete. done
Speed: 100, full duplex
Using ethernet@fe300000 device
host 192.168.0.200 is alive
```

我们通过 `tftp` 服务器向开发板内存写入数据测试，比如我们向内存 `0x30001000` 写入一个 `s3c2440-smdk2440.dtb` 文件：

```
=>  tftp 0x30001000 s3c2440-smdk2440.dtb
ethernet@fe300000 Waiting for PHY auto negotiation to complete. done
Speed: 100, full duplex
Using ethernet@fe300000 device
TFTP from server 192.168.0.200; our IP address is 192.168.0.105
Filename 's3c2440-smdk2440.dtb'.
Load address: 0x30001000
Loading: #
         1.4 MiB/s
done
Bytes transferred = 7328 (1ca0 hex)
```

当然如果你有内核镜像，可以直接烧录内核，并通过 `bootm` 命令启动内核。

6.5 `mmc` 操作指令

6.5.1 `mmc list`

在命令行输入 `mmc list` 命令用于来查看当前开发板一共有几个 `MMC` 设备：

```
=> mmc list
dwmmc@fe320000: 1
sdhci@fe330000: 0 (eMMC)
```

我们看到既有 `dwmmc` ，也有 `sdhci` ，在我使用的开发板中，设备树描述信息中： `dwmmc` 代表 `SD/MMC` 、 `sdhci` 代表 `eMMC` ，无论是 `SD还` 是 `eMMC` 都是 `MMC` 的一种。后面的数字代表设备号，比如 `sdhci` 设备号为0。

6.5.2 `mmc info`

默认会将 `eMMC` 设置为当前设备，如果需要查看 `eMMC` 信息，运行如下命令：

```
=> mmc info
Device: sdhci@fe330000
Manufacturer ID: 15
OEM: 100
Name: AJNB4
Timing Interface: HS400
Tran Speed: 150000000
Rd Block Len: 512
MMC version 5.1
High Capacity: Yes
Capacity: 14.6 GiB
Bus Width: 8-bit DDR
Erase Group Size: 512 KiB
HC WP Group Size: 8 MiB
User Capacity: 14.6 GiB WRREL
Boot Capacity: 4 MiB ENH
RPMB Capacity: 4 MiB ENH
```

从上图中可以看到 `MMC` 设备版本为 `5.1` ， 1 `4.6GiB` (`eMMC` 为 `16GB`)，速度为 `150000000Hz=150MHz` ， 8 位宽的总线。

6.5.3 `mmc dev`

如果我们需要切换到 `SD` ，则可以通过如下命令：

```
mmc dev [dev] [part]
```

[`dev`] 用来设置要切换的 `MMC` 设备号， [`part`]是分区号。如果不写分区号的话默认为分区0。

使用如下命令切换到 `SD` 卡：

```
=> mmc dev 1
CMD_SEND:0
switch to partitions #0, OK
mmc1 is current device
```

从上图可以看出，切换到 `SD` 卡成功， `mmc1` 为当前的 `MMC` 设备，输入命令 `mmc info` 即可查看 `SD` 卡的信息，结果如图下所示:

```
=> mmc info
Device: dwmmc@fe320000
Manufacturer ID: 9f
OEM: 5449
Name: 00000
Timing Interface: Legacy
Tran Speed: 52000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.9 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
```

从上图可以看出 `SD` 卡版本为 `3.0`，容量为 `14.9GiB` ，速度为 `52000000Hz=52MHz` ， 4位宽的总线。

6.5.4 `mmc rescan`

`mmc rescan` 命令用于扫描当前开发板上所有的 `MMC` 设备，包括 `eMMC` 和 `SD` 卡，输入 `mmc rescan` 即可。

6.5.5 `mmc part`

有时候 `SD` 卡或者 `eMMC` 会有多个分区，可以使用命令 `mmc part` 来查看其分区，比如查看 `eMMC` 的分区情况，输入如下命令：

```
=> mmc dev 0
switch to partitions #0, OK
mmc0(part 0) is current device
=> mmc part
Partition Map for MMC device 0  --   Partition Type: EFI
Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000040      0x00001f7f      "loader1"
        attrs:  0x0000000000000000
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        guid:   8ef917d1-5bd0-2c4f-9a33-b225b21cf919
  2     0x00004000      0x00005fff      "loader2"
        attrs:  0x0000000000000000
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        guid:   77877125-2374-ad40-8b02-609e37971c59
  3     0x00006000      0x00007fff      "trust"
        attrs:  0x0000000000000000
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        guid:   b4b84b8a-f8ae-0443-b5af-3605b195c4c9
  4     0x00008000      0x0003ffff      "boot"
        attrs:  0x0000000000000004
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        guid:   35219908-db08-c643-9133-2213191e57ff
  5     0x00040000      0x01d1efde      "rootfs"
        attrs:  0x0000000000000000
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        guid:   b921b045-1df0-41c3-af44-4c6f280d3fae
```

从上图中可以看出，此时 `eMMC` 有5个分区：

- 扇区 `0x00000040 ~ 0x00001f7f` 为第一个分区；存放的是 `idbloader.img` ；
- 扇区 `0x00004000 ~ 0x00005fff` 为第二个分区；存放的是 `u-boot.itb` ；
- 扇区 `0x00006000~ 0x00007fff` 为第三个分区；存放的是 `trust.img` ；
- 扇区 `0x00008000~ 0x0003ffff` 为第四个分区；存放的是 `boot.img` ；

- 扇区 `0x00040000~ 0x01d1efde` 为第五个分区；存放的是 `rootfs.img` ；

6.5.6 `mmc read`

`mmc read` 命令用于读取 `mmc` 设备的数据，命令格式如下：

```
mmc read addr blk# cnt
```

`addr` 是数据读取到 `DDR` 中的地址， `blk` 是要读取的块起始地址，一个块是 `512` 字节，这里的块和扇区是一个意思，在 `MMC` 设备中我们通常说扇区， `cnt` 是要读取的块数量（注意这里默认是16进制）。

比如从 `eMMC` 的第 `0x8000` 个扇区开始，读取1个扇区的数据到 `DRAM` 的 `0x10000000` 地址处，命令如下：

```
=> mmc read 0x10000000 0x8000 1

MMC read: dev # 0, block # 32768, count 1 ...
1 blocks read: OK
```

6.5.7 `mmc write`

要将数据写到 `MMC` 设备里面，可以使用命令 `mmc write` ，格式如下：

```
mmc write addr blk# cnt
```

`addr` 是要写入 `MMC` 中的数据在 `DDR` 中的起始地址， `blk` 是要写入 `MMC` 的块起始地址， `cnt` 是要写入的块大小（注意这里默认是 `16` 进制），一个块为 `512` 字节。

我们可以使用命令 `mmc write` 来升级 `uboot` ，也就是在 `uboot` 中更新 `uboot` 。这里要用到 `tftp` 命令，通过 `tftp` 命令将新的 `u-boot.itb` 下载到开发板的 `DDR` 中，然后再使用命令 `mmc write` 将其写入到 `MMC` 设备中。

我们就来更新一下 `eMMC` 中的 `uboot` ，输入命令：

```
=> setenv ipaddr 192.168.0.105
=> setenv serverip 192.168.0.200
=> tftp 0x10000000 u-boot.itb
=> mmc erase 0x4000 0x2000
=> mmc write 0x10000000 0x4000 0x2000
=> mmc read 0x10000000 0x4000 1

MMC read: dev # 0, block # 16384, count 1 ... 1 blocks read: OK
=> md.b 0x10000000 0x40   # 查看写入的数据
10000000: d0 0d fe ed 00 00 0a 39 00 00 00 38 00 00 09 68    .......9...8...h
10000010: 00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00    ...(............
10000020: 00 00 00 d1 00 00 09 30 00 00 00 00 00 00 00 00    .......0........
10000030: 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00    ................
```

6.5.8 `mmc erase`

如果要擦除 `MMC` 设备的指定块就是用命令 `mmc erase` ，命令格式如下：

```
mmc erase blk# cnt
```

回到顶部

七、 `uboot` 编译错误处理

7.1 `-Werror`

由于 `Makefile` 在编译的时候配置 `-Werror` ，将所有 `warning` 视为 `error` ，直接搜索代码：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# grep "\-Werror" * -nR
Makefile:363:KBUILD_CFLAGS      += -fshort-wchar -Werror
Makefile:618:KBUILD_CFLAGS   += $(call cc-option,-Werror=date-time)
scripts/Kbuild.include:117:     $(CC) -Werror $(KBUILD_CPPFLAGS) $(KBUILD_CFLAGS) $(1)
scripts/Kbuild.include:122:     $(CC) -Werror $(KBUILD_CPPFLAGS) $(KBUILD_CFLAGS) $(1)
scripts/gcc-stack-usage.sh:10:cat <<END | $@ -Werror -fstack-usage -x c - -c -o $TMP >/
tools/buildman/builderthread.py:429:                # We could avoid this by using -Wer
```

`KBUILD_CFLAGS` 是 `CC` 编译选项：

```
KBUILD_CFLAGS   += $(call cc-option,-Werror=date-time)          # 618行
```

在 `scripts/Kbuild.include` 定义了 `cc-option` 函数，函数用于检测 `$(CC)` 是否支持给定的选项。比如注释中的例子的意思：如果交叉编译工具 `$(CC)` 支持 `cc-optionl` 函数的参数一表示的选项(也就是指 `-marm`)，那么 `cc-option` 函数的返回就是该选项(指 `-marm`)，否则返回的是 `call` 函数的参数二表示的选项。

```
# cc-option
# Usage: cflags-y += $(call cc-option,-march=winchip-c6,-march=i586)

cc-option = $(call try-run,\
        $(CC) -Werror $(KBUILD_CPPFLAGS) $(KBUILD_CFLAGS) $(1) -c -x c /dev/null -o "$$
```

由于编译选项配置了 `-Werror`，因此在编译源代码的时候，会将所有 `warning` 视为 `error`。

因此我们可以通过修改 `Makefile`、以及 `scripts/Kbuild.include` 文件，将 `-Werror` 修改为 `-Wno-error` 从而达到禁用 `-Werror` 的目的。

关闭 `-Werror` 可能会导致程序中隐藏的问题，因此建议在调试和测试期间使用 `-Werror` 来确保代码质量。一旦您已经验证了程序的正确性，就可以将其关闭，并在开发过程中保持高水平的代码质量。

回到顶部

八、 脚本方式

由于我们每次修改程序后，重新编译步骤比较麻烦，这里我们可以将这步骤编写成一个 `shell` 脚本，这样每次执行就比较容易了。

8.1 自动构建脚本

在 `uboot` 根目录下创建一个 `build.sh` 文件：

```
#!/bin/bash

# 接收第一个参数 支持 ''|'config'|'clean'
step=$1
# 接收 V=1  支持编译输出详细信息
V=$2
cmd=${V%=*}

if [[ ${cmd} = 'V' ]]; then
    V=${V#*=}
fi

if [[ ${step} == "config" ]];then
    echo '----------------config evb-rk3399_defconfig------- -------------'
    make evb-rk3399_defconfig V=${V}
fi

if [[ -z ${step} ]];then
    echo "---------------1. compile uboot------------------------------- "
    make ARCH=arm CROSS_COMPILE=arm-linux- V=${V}
    echo "---------------2. mkimahe idbloader----------------------------"
    tools/mkimage -n rk3399 -T rksd -d tpl/u-boot-tpl.bin idbloader.img
    cat spl/u-boot-spl.bin >> idbloader.img
    echo "---------------3. make itb-------------------------------------"
    make u-boot.itb ARCH=arm CROSS_COMPILE=arm-linux-
    cp ./u-boot.itb ../rkdeveloptool
    cp ./idbloader.img ../rkdeveloptool
fi

if [[ ${step} == "clean" ]];then
    echo "-----------------clean----------------------------------------"
    make clean
    make distclean
fi
```

然后给文件赋予可执行权限：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# chmod +x build.sh
```

8.1.1 `clean`

执行如下命令进行清理：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./build.sh clean
```

8.1.2 配置

执行如下命令进行 `uboot` 配置：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./build.sh config
----------------config evb-rk3399_defconfig------- -------------
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
```

8.1.2 构建

执行如下命令进行 `uboot` 编译、生成 `dbloader.img` 、 `u-boot.itb` 文件，并拷贝到 `rkdeveloptool` 目录下：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./build.sh
```

如果需要输出编译详情信息，追加 `V=1` 参数即可。

8.2 官方构建脚本

实际上 `u-boot` 文件夹下有个 `make.sh` ，这是官方提供的一个自动构建的脚本，这个脚本代码比较多，毕竟官方考虑的比较全嘛，这里我们先不去解读，等后面有时间了再来解读。

▶ View Code

在命令行输入:

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./make.sh help
```

查看具体编译指令，如下图：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./make.sh help

Usage:
        ./make.sh [board|sub-command]

         - board:        board name of defconfig
         - sub-command:  elf*|loader|trust|uboot|--spl|--tpl|itb|map|sym|<addr>
         - ini:          ini file to pack trust/loader

Output:
         When board built okay, there are uboot/trust/loader images in current director

Example:

1. Build:
        ./make.sh evb-rk3399               --- build for evb-rk3399_defconfig
        ./make.sh firefly-rk3288           --- build for firefly-rk3288_defconfig
        ./make.sh EXT_DTB=rk-kernel.dtb    --- build with exist .config and external dt
        ./make.sh                          --- build with exist .config
        ./make.sh env                      --- build envtools

2. Pack:
        ./make.sh uboot                    --- pack uboot.img
        ./make.sh trust                    --- pack trust.img
        ./make.sh trust <ini>              --- pack trust img with assigned ini file
        ./make.sh loader                   --- pack loader bin
        ./make.sh loader <ini>             --- pack loader img with assigned ini file
        ./make.sh --spl                    --- pack loader with u-boot-spl.bin
        ./make.sh --tpl                    --- pack loader with u-boot-tpl.bin
        ./make.sh --tpl --spl              --- pack loader with u-boot-tpl.bin and u-bo
```

```
3. Debug:
        ./make.sh elf                       --- dump elf file with -D(default)
        ./make.sh elf-S                     --- dump elf file with -S
        ./make.sh elf-d                     --- dump elf file with -d
        ./make.sh elf-*                     --- dump elf file with -*
        ./make.sh <no reloc_addr>           --- unwind address(no relocated)
        ./make.sh <reloc_addr-reloc_off>    --- unwind address(relocated)
        ./make.sh map                       --- cat u-boot.map
        ./make.sh sym                       --- cat u-boot.sym
```

修改 `make.sh` 设置交叉编译工具路径：

```
RKBIN_TOOLS=../rkbin/tools
CROSS_COMPILE_ARM32=/usr/local/arm/12.2.1/bin/arm-none-linux-gnueabihf-
CROSS_COMPILE_ARM64=/usr/local/arm/12.2.1/bin/aarch64-none-linux-gnu-
```

同时将 `select_toolchain` 函数以下代码：

```
CROSS_COMPILE_ARM64=$(cd `dirname ${CROSS_COMPILE_ARM64}`; pwd)"/aarch64-linux-gnu-"
```

修改为：

```
CROSS_COMPILE_ARM64=$(cd `dirname ${CROSS_COMPILE_ARM64}`; pwd)"/aarch64-none-linux-gnu
```

第一次编译的时候， `uboot` 目录下并没有 `.config` 配置文件，需要指定默认的配置文件。

如果编译使用的 `defonfig` 文件为 `configs/evb-rk3399_defconfig` ，则可以直接使用如下命令来编译：

```
root@ubuntu:/work/sambashare/rk3399/u-boot# ./make.sh evb-rk3399
```

参考文章

[1] `RK3399-Linux`

[2] `rk3399` **移植** `u-boot`

[3] `Firefly-Rk339` **U-Boot 使用**

[4] `__attribute__((packed))` **)**

[5] 移植 `U-Boot` **思路和实践 | 基于** `RK3399`

[6] `RK3399` **——裸机大全**

[7] `U-Boot` **之一 零基础编译** `U-Boot` **过程详解、** `Image` **镜像介绍及使用说明、** `DTB` **文件使用说明**

[8] 嵌入式 `ARM64 uboot 2022.01` **移植**

[9] 深度探索 `uboot`

[10] `uboot 2021.10` **源码分析(启动流程)**

[11] `Uboot 2017.01 SPL` **中的** `image_loader`

[12] `imx8mq - u-boot-spl` **启动分析**

[13] `RK3399-SD` **卡** `linux` **系统制作(** `uboot,kernel` **内核,根文件**

亲爱的读者和支持者们，自动博客加入了打赏功能，陆陆续续收到了各位老铁的打赏。在此，我想由衷地感谢每一位对我们博客的支持和打赏。你们的慷慨与支持，是我们前行的动力与源泉。

日期	姓名	金额
2023-09-06	*源	19
2023-09-11	*朝科	88
2023-09-21	*号	5
2023-09-16	*真	60