

# 目录

- · 一、SoC启动流程
  - 。 1.1 BootROM介绍
    - 1.1.1 初始化硬件
    - 1.1.2 加载程序到SRAM
  - 。 1.2 SPL介绍
    - 1.2.1 方案一
    - 1.2.2 方案二
  - 1.3 启动流程
- 二、RK3399 地址空间分布
  - 2.1 地址映射
  - 2.2 系统启动
- 三、Rockchip引导流程
  - 3.1 启动阶段
    - 3.1.1 idbloader.img
    - 3.1.2 u-boot.img
    - 3.1.3 u-boot.itb
    - 3.1.4 trust.img
  - 3.2 引导流程
    - 3.2.1 TPL/SPL方式
    - 3.2.1 官方固件方式
- 四、安装交叉编译工具链
  - 4.1 下载
  - 4.2. 安装

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。

回到顶部

# 一、SoC启动流程

## 1.1 BootROM介绍

通常来说,SoC厂家都会做一个ROM在SoC的内部,这个ROM很小,里面固化了上电启动的代码(一经 固化,永不能改,是芯片做的时候,做进去的);这部分代码呢,我们管它叫做BootROM,也叫作一级



创作不易,喜欢的话,请考虑支

昵称: 大奥特曼打小怪兽 园龄: 6年6个月 粉丝: 894 关注: 12 +加关注

# 积分与排名

积分 - 525869 排名 - 1223

# 随笔分类 (493)

bigdata(4) deep learning(38) H3(1) java(14) javascript(1) linux alsa(15) linux blk(7) linux debug(1) linux drm(13) linux dts(13) linux gpio(6)

linux embedded environment(6)

linux gpu(2) linux gui(2) Linux i2c(7) linux interrupt(5) linux network(5) linux ota(4) linux rootfs(15)

linux shell(2) 更多

启动程序。

## 1.1.1 初始化硬件

芯片上电后先接管系统的是SoC厂家的BootROM,它要做些什么事情呢?初始化系统,CPU的配置,关闭看门狗,初始化时钟,初始化一些外设(比如 USB Controller、MMC Controller,Nand Controller等);

#### 1.1.2 加载程序到SRAM

当我们拿到一款新的SoC时,都会进行电路原理图设计,我们一般会在芯片外挂一些存储设备(eMMC、Nand、Nor、SDCard等)和内存(SDRAM、DDR等)电路图绘制好了。我们接着会绘制电路板,制作出板子。

有了板子还不行,我们还得往里面烧写程序。这个烧写程序,其实就是将可执行的二进制文件写到外部的存储设备上(eMMC、Nand、SD等)。系统上电启动的时候,会将他们读到内存中执行。

前面我们说了,上电后先接管系统的是SoC厂家的BootROM,其它可执行的程序(u-boot、Kernel)都放(烧写)到了外部存储器上;那么BootROM的代码除了去初始化硬件环境以外,还需要去外部存储器上面,将接下来可执行的程序读到内存来执行。

既然是读到内存执行,那么这个内存可以不可以是我们板载的 DDR呢?理论上是可以的,但是,SoC厂家设计的DDR控制器呢,一般会支持很多种类型的DDR设备,并且会提供兼容性列表,SoC厂家怎么可能知道用户PCB上到底用了哪种内存呢?所以,直接把外部可执行程序读到DDR显然是不太友好的,一般来说呢,SoC都会做一个内部的小容量的SRAM,BootROM将外部的可执行程序从外部存储器中读出来,放到SRAM去执行;

好了,现在我们引出了SRAM,引出了BootROM;那么 BootROM从具体哪个存储器读出二进制文件呢? SoC厂家一般会支持多种启动方式,比如从eMMC读取,从SDCard读取,从Nand Flash读取等等;上电的时候,需要告诉它,它需要从什么样的外设来读取后面的启动二进制文件;

一般的设计思路是,做一组Bootstrap Pin,上电的时候呢?BootROM去采集这几个IO的电平,来确认要从什么样的外部存储器来加载后续的可执行文件;比如呢, $2 \uparrow$  IO,2'b00 表示从Nand启动,2'b01表示从eMMC启动,2'b10 表示从SDCard启动等等;

当BootROM读到这些值后,就会去初始化对应的外设,然后来读取后面要执行的代码;这些IO一般来说,会做成板载的拨码开关,用于调整芯片的启动方式;

这里,读取烧写的二进制的时候呢,需要注意一些细节,比如SoC厂家告诉你,你需要先把SDCard初始 化称为某种文件系统,然后把东西放进去才有效,之类的;因为文件系统是组织文件的方式,并不是裸 分区;你按照A文件系统的方式放进去,然后SoC的BootROM也按照A文件系统的方式读出来,才能够达 成一致;

如果你对Mini2440这款开发板足够了解的话,你应该知道其采用的SoC型号为s3c2440,其内部有一个4kb的SRAM。其有两种启动方式:

- 采用Nor Flash启动,0x00000000就是2MB Nor Flash实际的起始地址,由于uboot程序一般只有几百k b,可以全部烧录到Nor Flash中,因此uboot程序完全可以在Nor Flash中运行,没有拷贝到SDRAM中运行的必要;
- 采用Nand Flash启动,片内4KB的SRAM被映射到了0x000000000,s3c2440的BootROM会自动把Nand Flash中的前4kb代码数据搬到内部SRAM中运行,那么问题来了,假设4KB代码运行到最后,我想继续运行Nand Flash剩余的代码怎么办?为了解决这个问题,uboot引入了SPL,全称Secondary Progra m Loader。

注意:无论是Nor Flash还是Nand Flash都是外挂在s3c2440上的的存储设备。

## 1.2 SPL介绍

前面说了,芯片上电后BootROM会根据Bootstrap Pin去确定从某个存储器来读可执行的二进制文件到SRAM并执行;理论上来说,这个二进制文件就可以是我们的u-boot.bin文件了;也就是BootROM直接加载u-boot.bin;

理论上是这样的,但是这里有一个问题,就是SRAM很贵,一般来说,SoC的片上SRAM都不会太大,一般4KB、8KB、16KB...256KB不等;但是呢,u-boot 编译出来却很大,好几百KB,放不下,就像我上面说的s3c2440的例子那样。

# 放不下怎么办? 有两种办法:

- 假设片内SRAM为4KB, uboot的前4KB程序实现uboot的重定位,即将uboot拷贝到SDRAM中运行;
- 做一个小一点的boot程序,先让BootROM加载这个小的程序,后面再由这个小boot去加载uboot;

# 1.2.1 方案一

比如,我们的uboot有400KB,SRAM有4KB,外部SDRAM有64MB:如果使用第一种方案的话,uboot的前面4KB被加载进入SRAM执行,uboot被截断,我们就需要保证在uboot的前4KB代码,把板载的SDRAM初始化好,把整个uboot拷贝到SDRAM,然后跳转到SDRAM执行;

#### 随笔档案 (350)

2024年8月(2) 2024年7月(7) 2024年6月(6) 2024年4月(1) 2024年3月(4) 2023年2月(5) 2024年1月(2) 2023年11月(9) 2023年10月(4) 2023年9月(10) 2023年8月(1) 2023年6月(7) 2023年6月(7) 2023年5月(14)

告&打赏

## 阅读排行榜

更多

2023年4月(7)

2023年3月(7)

2023年2月(12)

2023年1月(1)

2022年10月(3)

- 1. 第六节、双目视觉之相机标定(69635)
- 2. 第三十七节、人脸检测MTCNN和人 脸识别Facenet(附源码)(51305)
- 3. 第十九节、基于传统图像处理的目标 检测与识别(HOG+SVM附代码)(41929) 4. 第七节、双目视觉之空间坐标计算(4 1754)
- 5. 第九节、人脸检测之Haar分类器(403 19)

## 推荐排行榜

- 1. 第七节、双目视觉之空间坐标计算(14)
- 2. 第十一节、Harris角点检测原理(附源码)(11)
- 3. 第九节、人脸检测之Haar分类器(10)
- 4. 第三十三节,目标检测之选择性搜索-Selective Search(10)
- 5. 第三十七节、人脸检测MTCNN和人 脸识别Facenet(附源码)(8)

# 最新评论

1. Re:第二十五节,初步认识目标定位、特征点检测、目标检测 @大奥特曼打小怪兽 谢谢博主回复o( ̄▽ ̄)ブ...

--au3h2o

2. Re:第二十五节,初步认识目标定位、特征点检测、目标检测 @au3h2o 吴恩达的视频...

--大奥特曼打小怪兽

3. Re:第二十五节,初步认识目标定位、特征点检测、目标检测 写的真好,另外麻烦问一下作者,这个ppt是哪里的呀,有出处吗,谢谢分享

--au3h2o

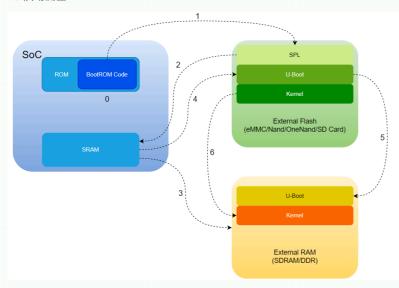
4. Re:Rockchip RK3566 - orangepi-buil d脚本分析

比如我们之前介绍的Mini2440开发板从Nand Flash启动时,uboot程序就是采用的这种实现方式: <u>Mini2440之uboot移植之实践NAND启动</u>。

#### 1.2.2 方案二

第二种方案的话,我们做一个小的uboot,这个uboot就叫做SPL(Secondary Program Loader),它很小很小(小于SRAM大小),它先被BootROM加载到SRAM运行,那么这个SPL要做什么事情呢?最主要的就是要初始化内存控制器,然后将真正的大u-boot从外部存储器读取到SDRAM中,然后跳转到大uboot。

## 1.3 启动流程



## 如上图所示:

- (0)上电后,BootROM开始执行,初始化时钟,关闭看门狗,关Cache,关中断等等,根据Bootstrap Pin来确定启动设备,初始化外设;
- (1) 使用外设驱动,从存储器读取SPL;
- (2) SPL被读到SRAM 执行,此刻,控制权以及移交到我们的SPL 了;
- (3) SPL初始化外部SDRAM;
- (4) SPL使用驱动从外部存储器读取uboot并放到SDRAM;
- (5) 跳转到SDRAM中的uboot执行;
- (6) 加载内核;

# 实际情况中,还需注意很多问题:

- 编译阶段的链接地址,是否需要地址无关?
- SPL的代码和uboot的代码是否有重合的地方?如果有,是否意味着SPL执行过的,跳转到uboot又要在执行一次?
- 具体情况下,需要配置哪些硬件? 怎么配置?

# 回到顶部

# 二、RK3399 地址空间分布

## 2.1 地址映射

RK3399支持从内部BootROM启动,并且支持通过软件编程进行地址重映射。重映射是通过 SGRF\_PMU\_CON0[15]控制的,当重映射设置为0时,地址0xFFFF0000被映射到BootROM,当重映射设 置为1时,0xFFFF0000被映射到片内SRAM。 @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-buil d脚本分析

博主有其他平台账号同步发文章吗--超超加油

		FF35_0000 p									
	After_REMAP	FF35_0000	Reserved (64K)	FF6F_0000	DMAC1 (64K)						
FFFF_0000/	INTMEM0 (192K)	FF34_0000	PMUSGRF	FF6E_0000	DMAC0 (64K)						
FF8C_0000		FF33_0000	(64K) PMUGRF	FF6D_0000	(64K) Reserved						
FFFD_0000	BOOTROM (32K)	FF32_0000	(64K)	FF6C_0000	(64K)						
		FF31_0000	PMU (64K)		MAILBOX0 (64K)						
	Before_REMAP		Reserved (640K)	FF6B_0000	DCF (64K)	FF87_0000	STMIER6~11(6ch) (32K)				
	INTMEMO (192K)	FF27_0000	TS-ADC (64K)	FF6A_0000	EFUSE0	FF86_8000	STIMER0~5(6ch)				
FF8C_0000	BOOTROM	FF26_0000	Reserved	FF69_0000	(64K)	FF86_0000	(32K) TIMER6~11(6ch)				
FFFF_0000,	/ (32K)	FF21_0000	(320K) SPIS	FF68_0000	(64K)	FF85_8000	(32K)	FF9A_0000	Reserved	ı	
		FF20_0000	(64K)		IEP (64K)		TIMER0~5(6ch) (32K)		(64K)		
FEE0_0000	Reserved (1M)		SPI4 (64K)	FF67_0000	VIDEO_DECODER (64K)	FF85_0000	WDT0 (32K)	FF99_0000	HDCP2.2 (32K)		
FED0_0000	DP	FF1F_0000	SPI2 (64K)	FF66_0000	VIDEO_ENCODER	FF84_8000	WDT1	FF98_8000	Reserved (64K)		
FEC0_0000	(1M) Reserved	FF1E_0000	SPI1	FF65_0000	(64K) Reserved	FF84_0000	(32K) TYPEC_PHY1	FF97_8000	eDP	FFFF_FFFF	BOOTROM/
FEA0_0000	(2M)	FF1D_0000	(64K)	FF64_0000	(64K)	FF80_0000	(256K)	FF97_0000	(32K) DSI HOST1	FFFF_0000	INTMEMO (64KB)
FE90_0000	USB3.0/2.0_OTG1 (1M)	FF1C_0000	SPI0 (64K)	FF63_0000	DFI_MONITOR (64K)	FF7C_0000	TYPEC_PHY0 (256K)	FF96_8000	(32K)		Reserved (64KB)
	USB3.0/2.0_OTG0 (1M)	FF1C_0000	UART3 (64K)		CIC (64K)	7770_0000	TCPD1 (64K)	FF96_0000	DSI_HOSTO (32K)	FFFE_0000	BOOTROM (64KB)
FE80_0000	DEBUG (4MB)	FF1B_0000	UART2 (64K)	FF62_0000	Reserved (1984K)	FF7B_0000	TCPD0		HDMI (128K)	FFFD_0000	SDMAC1 (64K)
FE40_0000	USB2.0_HOST1	FF1A_0000	UART1	FF43_0000	PWM(4CH)	FF7A_0000	(64K) INTR_ARB1	FF94_0000	носрими	FFFC_0000	SDMAC0
FE3C_0000	(256K)	FF19_0000	(64K) UARTO	FF42_0000	(64K) Reserved	FF79_C000	(16K)	FF93_0000	(64K) ISP1	FFFB_0000	(64K)
FE38_0000	USB2.0_HOST0 (256K)	FF18_0000	(64K)	FF3F 0000	(192K)		INTR_ARBO (16K)	FF92_0000	(64K)	FFFA_0000	EFUSE1 (64K)
- 1	HSIC PHY (64K)		Reserved (64K)		12C8 (64K)	FF79_8000	GPIO4 (32K)	FF91_0000	ISP0 (64K)		Reserved (3712KB)
FE37_0000	HSIC (192K)	FF17_0000	12C7 (64K)	FF3E_0000	12C4 (64K)	FF79_0000	GPIO3 (32K)	_	VOP_BIG (64K)	FFC0_0000	CCI500 (1M)
FE34_0000	eMMC	FF16_0000	1206	FF3D_0000	1200	FF78_8000	GPIO2	FF90_0000	VOP_LIT (64K)	FFB0_0000	Service NoC
FE33_0000	(64K) SDMMC	FF15_0000	(64K) 12CS	FF3C_0000	(64K) INTMEM1	FF78_0000	(32K) GRF	FF8F_0000	INTMEMO	FFA9_0000	(448k) Service NoC
FE32_0000	(64K)	FF14_0000	(64K)	FF3B_0000	(64K)	FF77_0000	(64K)	FF8C_0000	(192K)	FFA8_C000	(16k)
FE31_0000	SDIO (64K)	FF13 00	12C3 (64K)	FF3A_0000	Reserved (64K)	FF76_0000	CRU (64K)	FF8B_8000	CRYPTO1 (32K)	FFA8_8000	DDRC1 (16k)
- 1	GMAC (64K)	FF13_0000	12C2 (64K)		MAILBOX1 (64K)		PMUCRU (64K)		CRYPTOO (32K)		Service NoC (16k)
FE30_0000	Reserved (3MB)	FF12_0000	12C1 (64K)	FF39_0000	WDT2 (64K)	FF75_0000	Reserved (64K)	FF8B_0000	1252(8CH) (64K)	FFA8_4000	DDRC0 (16k)
FE00_0000	PCIe	FF11_0000	SAR-ADC	FF38_0000	UART4	FF74_0000	GPIO1	FF8A_0000	1251(8CH)	FFA8_0000	Service NoC
F800_0000	(96MB)	FF10_0000	(64K) Reserved	FF37_0000	(64K) PMUTIMERO~1	FF73_0000	(64K) GPIO0	FF89_0000	(64K) 12S0(8CH)	FFA5_0000	(192k)
	DDR	FF00_0000	(1MB)	FF36_0000	(64K)	FF72_0000	(64K)	FF88_0000	(64K)	FF98_0000	Reserved (640k)
	(4G-128M)	*****	GIC500 (2MB)		SPI3 (64K)		Reserved (192K)	FF87_0000	SPDIF (64K)	FF9A 0000	GPU (64K)
0000_0000		FEE0_0000		FF35_0000		FF6F_0000					

## 从这张图我们可以看到在进行重映射前:

- 0x0000 0000 ~ 0xF800 0000: 为DDR内存空间;
- 0xFF8C 0000~0xFF98 0000: 片内SRAM内存空间,一共192KB;
- 0xFFFF 0000~ 0xFFFF 8000: 为BootROM内存空间,一共32KB;
- 其它空间: 用于一些特定功能;

# 如果进行了地址重映射:

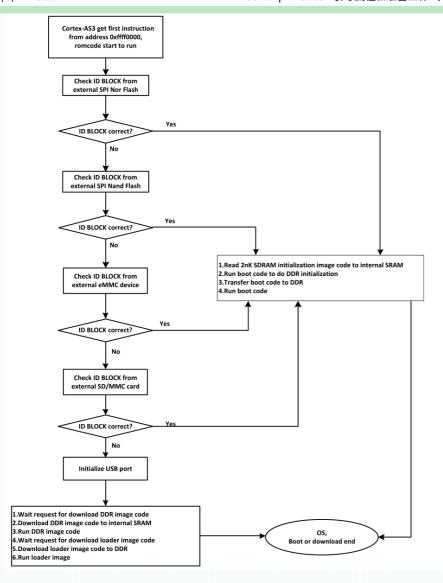
- BootROM被映射到地址0xFFFD 0000;
- 片内SRAM被映射到地址0xFFFF 0000;

# 2.2 系统启动

RK3399提供从片外设备启动系统,如serial nand or nor flash、eMMC、SD/MMC卡。当这些设备中的启动代码没有准备好时,还可以通过USB OTG接口将系统代码下载到各个外设存储中。

所有引导代码都将存储在内部BootROM中。其中支持以下功能:

- (1) 支持安全启动模式和非安全启动模式;
- (2) 支持系统从以下设备启动;
  - SPI接口;
  - eMMC接口;
  - SD/MMC卡;
- (3) 支持系统代码通过USB OTG下载;
- 以下是存储在BootROM中的启动代码的整个启动过程:



从图中可以得到以下几个结论:

- (1) 上电后, A53核心从0xffff0000这个地址读取第一条指令,这个内部BootROM在芯片出货的时候已经由原厂烧写;
- (2) 然后依次从Nor Flash、Nand Flash、eMMC、SD/MMC获取ID BLOCK,ID BLOCK正确则启动,都不正确则从USB端口下载;
  - 如果eMMC启动,则先读取SDRAM(DDR)初始化代码到内部SRAM,由于SRAM只有192KB,因此最多只能读取那么多,然后初始化DDR,再将eMMC上的代码(剩下的用户代码)复制到DDR运行;
  - 如果从USB下载,则先获取DDR初始化代码,下载到内部SRAM中,然后运行代码初始化DDR,再获取loader代码(用户代码),加载到DDR中并运行;

回到顶部

# 三、<u>Rockchip引导流程</u>

针对不同的解决方案,Rockchip提供了两种不同的启动加载程序方法,其步骤和生成的镜像文件也是完全不同的。

- TPL/SPL加载:使用Rockchip官方提供的<u>TPL/SPL</u> <u>U-boot</u>(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,它由<u>Rockchip rkbin project</u>的Rockchip ddr init bin和miniloa der bin组合而成,该方式不开源;

需要注意的是: 并不是所有平台都支持这两种启动加载程序方法。

上面我们介绍了SPL,那什么是TPL?实际上将我们上面所说的SPL初始化SDRAM等硬件工作的部分独立出去,就是TPL。那么我们总结一下:

• TPL是Targer Program Loader,就是芯片级的初始化过程,这个时候的代码都是基于芯片平台的部分,它在启动过程中进行DDR初始化和一些其他的系统配置,以便后续的SPL能够正确地运行;

• SPL是Secondary Program Loader,它从存储设备中读取trust(如ATF/OP-TEE)和uboot二进制文件,将它们加载到系统内存中并运行它们,进而启动完整的操作系统;

TPL和SPL的区别在于它们的职责不同。TPL主要负责初始化系统硬件,而SPL负责加载和运行其它软件组件,如trust和uboot。此外,在一些特殊情况下,如加密启动或安全启动模式下,TPL还可能执行其他额外的任务。

## 3.1 启动阶段

Rockchip处理器启动可以划分为5个阶段:

++   Boot     stage     number	Terminology #1	+   Actual   program   name	Rockchip     Image     Name	Image   Location  (sector)	
	Primary Program Loader	ROM code     	BootROM		
2	Secondary Program Loader (SPL)	U-Boot   TPL/SPL   	idbloader.img      	0×40       	pre-loader
3	-	   U-Boot   	u-boot.itb     uboot.img	0×4000     	including u-b only used wit
		ATF/TEE 	trust.img	0×6000	only used wit
4	-	kernel	boot.img	0x8000	
5	-	ı   rootfs +	rootfs.img   	0×40000   +	

当我们讨论从eMMC/SD/U盘/网络启动时,它们涉及到不同的概念:

- 第一阶段始终在BootROM中,它加载第二阶段并可能加载第三阶段(当启用SPL\_BACK\_TO\_BROM 选项时);
- 从SPI闪存启动意味着第二阶段和第三阶段固件(仅限SPL和U-Boot)在SPI闪存中,第四/五阶段在其他位置;
- 从eMMC启动意味着所有固件(包括第二、三、四、五阶段)都在eMMC中;
- 从SD Card启动意味着所有固件(包括第二、三、四、五阶段)都在SD Card中;
- 从U盘启动意味着第四和第五阶段的固件(不包括SPL和U-Boot)在磁盘中,可选地仅包括第五阶段;
- 从Net/TFTP启动意味着第四和第五阶段的固件(不包括SPL和U-Boot)在网络上。

## 启动阶段涉及到了多个镜像文件:

- 阶段一中的BootROM这个是SoC厂商提供的,我们不用关心;
- 阶段二方式需要提供一个idbloader.img,这个我们后面具体说说;
- 阶段三实际上就是uboot的镜像文件了,这里又搞出了两种,uboot.img(还需要搭配trust.img)和u-boot.itb(这个是因为它已经把ATF打包进去了);这两个文件里面除了都包含u-boot.bin原始二进制文件,又放了点其他东西,可以被idbloader.img识别,然后加载,这个我们后面具体说说;
- 阶段四和阶段五是内核镜像和根文件系统;

这里我们具体说一下阶段二,阶段三涉及到的几个镜像文件。

# 3.1.1 idbloader.img

idbloader.img文件是一个Rockchip格式的预加载程序,在SoC启动时工作,它包含:

- 由Rockchip BootROM知道的IDBlock 头;
- DDR初始化程序,由BootROM加载到SRAM,运行在SRAM内部;
- 下一级加载程序,由BootROM加载并运行在DDR上;

## 3.1.2 u-boot.img

u-boot.bin是uboot源码编译后生成的原始二进制映像,可以直接烧录到设备的闪存中。而u-boot.img则是通过mkimage工具在u-boot.bin基础上增加了一个头部信息,这个头部信息可能也包括一些额外的数据,例如启动参数和内核映像地址等。

因此,通过使用u-boot.img而不是u-boot.bin,可以使引导ROM更容易地识别uboot映像,并更好地指导uboot在设备上正确启动。

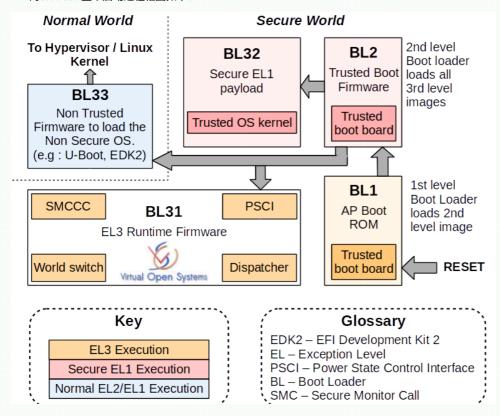
#### 3.1.3 u-boot.itb

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,里面除了u-boot.dtb和u-boot-nodtb.bin这两个uboot源码编译出来的文件之外,还包含了bl31.elf、bl32.bin、tee.bin等ARM trust固件。其中bl31.elf是必须要有的,bl32.bin、tee.bin是可选的,可以没有。

#### 3.1.4 trust.img

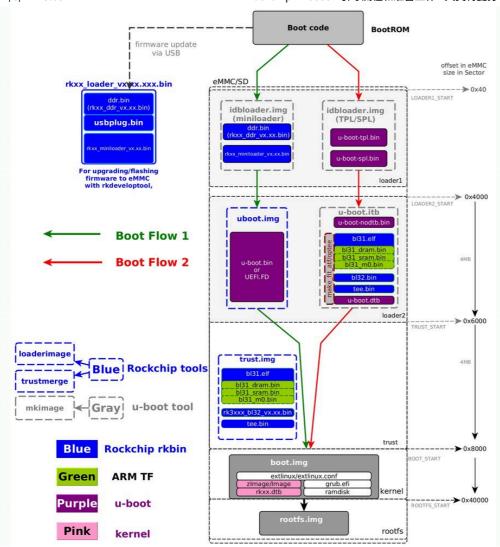
因为RK3399是ARM64,所以我们还需要编译ATF (ARM Trust Firmware),ATF主要负责在启动uboot之前把CPU从安全的EL3切换到EL2,然后跳转到uboot,并且在内核启动后负责启动其他的CPU。

ATF将系统启动从最底层进行了完整的统一划分,将secure monitor的功能放到了bl31中进行,这样当系统完全启动之后,在CA或者TEE OS中触发了smc或者是其他的中断之后,首先是遍历注册到bl31中的对应的service来判定具体的handle,这样可以对系统所有的关键smc或者是中断操作做统一的管理和分配。ATF的code boot整个启动过程框图如下:



## 3.2 引导流程

Rockchip提供了外部uboot加载的流程图,如下图示:



# 如上图所示:

- 引导流程1是典型的使用Rockchip miniloader的Rockchip引导流程;
- 引导流程2用于大多数SoC,使用U-Boot TPL进行DDR初始化,使用SPL加载加载u-boot.itb文件;

注1: 如果loader1具有多个阶段,则程序将返回到BootROM,BootROM将载入并运行到下一个阶段。例如,如果loader1是TPL和SPL,则BootROM将首先运行到TPL,TPL初始化DDR并返回到BootROM,BootROM然后将加载并运行到SPL。

注2:如果启用了trust,在安全模式(armv8中的EL3)下,loader1需要同时加载trust和U-Boot,然后运行到trust中,trust在非安全模式(armv8中的EL2)下进行初始化,并运行到U-Boot。

注3:对于trust(在trust.img或u-boot.itb中),armv7仅有一个带或不带TA的tee.bin,armv8具有bl31.elf并且可选包含bl32。

注4:在boot.img中,内容可以是Linux的zImage和其dtb,可以选择grub.efi,也可以是AOSP boot.img,ramdisk可选。

## 3.2.1 TPL/SPL方式

在TPL/SPL加载方式中,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,由SPL加载u-boot.itb文件。

TPL:被BootROM加载到内部SRAM,起始地址为0xff8c2000;结束地址不能超过0xff980000,所以TPL程序最大不能超过184KB;

SPL:被BootROM加载到DDR,起始地址为0x00000000;结束地址绝对不能超过0x00040000,因为0x00040000地址被用来加载bl31\_0x00040000.bin,因此SPL程序最大不能超过256KB:反汇编如下:

```
000000000000000008 < TEXT BASE>:
000000000000000010 < end ofs>:
                            .inst 0x0001c618; undefined
     14:
             00000000
                            udf
0000000000000018 <_bss_start_ofs>:
    18:
          00400000 .inst 0x00400000 ; undefined
             00000000
     1c:
                            udf
000000000000000020 <_bss_end_ofs>:
     20:
          004003c0 .inst 0x004003c0 ; undefined
             00000000
                           udf
00000000000000028 <reset>:
     28:
             1400010a
                            b
                                   450 <save_boot_params>
0000000000000002c <save_boot_params_ret>:
     2c:
            10007ea0
                           adr
                                   x0, 1000 <vectors>
```

## 这里我们具体说一下采用这种方式RK3399的启动流程:

- BootROM首先将eMMC中0x40扇区开始的184KB数据加载到片内SRAM中;由于TPL和SPL加在一起 是超过184KB的,所以无法全部加载到SRAM,但是把TPL全部加载到SRAM中还是绰绰有余的,这 里加载地址为0xff8c2000;
- BootROM跳转到0xff8c2000执行TPL代码,主要是DDR的初始化,当然还有一些其他硬件的初始化;需要注意的是,执行完TPL代码之后,会返回到BootROM程序,你把它当做汇编指令bl TPL那样会更好理解;
- BootROM加载SPL代码到DDR中,这里加载地址为0x00000000,然后跳转到地址0x000000000去执行,需要注意的是这个时候不会再返回到BootROM了,因此SPL会初始化eMMC并将eMMC中0x4000扇区的uboot加载到0x00200000地址处,然后跳转到该处执行uboot程序;

补充说明:上面描述的只是一个大概流程,当然中间SPL还会加载bl31.bin(bl32.bin、tee.bin非必须)去执行,但是这不是重点,所以就不做概述。

由于BootROM不是开源的,我们没法去研究BootROM源码,当然也我们可以修改common/spl/spl.c文件board\_init\_r函数在SPL代码执行时将地址0x000000000、0xff8c2000、0x40000000等地址数据打印出来(printf函数要加在boot\_from\_devices函数执行之后),和源二进制文件进行比对来验证自己的猜想:

```
board_init_r
addr 0x00000000 = 0x14000001  # 和u-boot-spl.bin前4字节匹配
addr 0x00000004 = 0x14000009  # 和u-boot-spl.bin文件偏移0x4处的4个字节匹配
addr 0x00000008 = 0x0  # 同样匹配
addr 0x00040000 = 0xaa0003f4  # 和bl31_0x00040000.bin文件前4字节匹配
addr 0x00050018 = 0xb8656883  # 和bl31_0x00040000.bin文件偏移0x10018处的4个字节匹配
addr 0xff8c2000 = 0x33334b52  # 这个地址数据和u-boot-tpl.bin有点对不上,可能后期被改变了?可以:
```

# 在该方式中,我们需要用到以下源代码:

- uboot源码:编译生成u-boot-spl.bin、u-boot-tpl.bin、u-boot-nodtb.bin、u-boot.dtb;
- ATF源码:编译生成bl31.elf;

## 通过编译和工具我们最终可以生成:

- idbloader.img: 由u-boot-spl.bin、u-boot-tpl.bin通过工具合并得到;
- u-boot.itb: 由bl32.elf、u-boot-nodtb.bin、u-boot.dtb、u-boot.its通过工具合并得到;

## 3.2.1 官方固件方式

在官方固件加载方式中,我们基于Rockchip rkbin官方给的ddr.bin、miniloader.bin来实现的;

- (1) 通过tools/mkimage将官方固件ddr, miniloader打包成BootROM程序可识别的、带有ID Block header的文件idbloader.img;
  - ddr.bin: 等价于上面说的TPL, 用于初始化DDR;

• miniloader.bin: Rockchip修改的一个bootloader,等价于上面说的SPL,用于加载uboot;

这个文件打包出来实际上也是超过192KB的,因此也是分为二阶段执行的。

(2) 通过tools/loaderimage工具将u-boot.bin打包成u-boot.img;其中u-boot.bin是由uboot源码编译生成;

补充说明:使用Rockchip miniloader的 idbloader 时,需要将u-boot.bin通过tools/loaderimage转换为可加载 的miniloader格式。

(3)使用Rockchip工具tools/trust\_merge将bl31.bin打包成trust.img;其中bl31.bin由ATF源码编译生成;

补充说明:使用Rockchip miniloader的idbloader 时,需要将bl31.bin通过tools/trust\_merge转换为可加载的 miniloader格式。

# 四、安装交叉编译工具链

既然我们想向开发板处理器中烧录程序,交叉编译工具是必不可少的。选择交叉编译工具这一步需要慎 重,我们首先要知道我们自己使用的开发板采用的ARM架构是哪一个?ARM目前总共发布了8种架构: ARMv1、ARMv2、ARMv3、ARMv4、ARMv5、ARMv6、ARMv7、ARMv8。

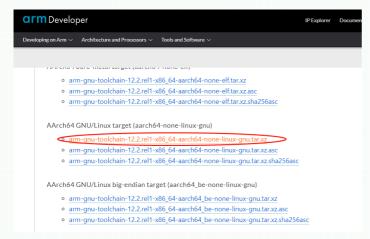
确认了ARM架构之后,选择支持该架构的交叉编译环境。可以参考GNU Arm Embedded Toolchain官网中 的文档和说明来确定march支持的版本。在ARM官方提供的文档中,可以查看到march选项支持的处理器 架构和对应的版本。例如,使用aarch64-linux-gnu-gcc -march=armv8-a命令编译代码时,表示编译针对于 Cortex-A53处理器。

除了GNU Arm Embedded Toolchain官网提供的交叉编译工具外, LINARO也提供了交叉编译工具,下载 地址<u>https://releases.linaro.org/components/toolchain/binaries/</u>。具体有什么差异可以参考:<u>ARM CPU汇总 &</u> 编译链选择。

## 4.1 下载

因此我们必须选择一个支持Armv8-A架构的交叉编译工具,即在Linux上编译ARM64 Linux程序,本文采 用GNU Arm Embedded Toolchain官网提供的交叉编译工具链。

这里我们直接选择最新版本的交叉编译工具:



arm-gnu-toolchain-12.2.rel1-x86\_64-aarch64-none-linux-gnu.tar.xz,该文件名称意为:在x86平台的linux主机 进行编译,生成的文件为aarch64平台可运行的文件,这里宿主机和目标平台都是64位机器。

如何您使用 LINARO提供的交叉编译工具,可以选择gcc-linaro-7.5.0-2019.12-x86 64 aarch64-linux-

注意: 最新版本可能存在各种坑,因此推荐您安装11.3版本。

复制下载地址,下载在/work/sambashare/tools/路径:

 ${\tt root@zhengyang:/work/sambashare/tools\#\ wget\ https://developer.arm.com/-/media/Files/downedia/Files/files$ 4.2. 安装 使用如下命令进行解压:

root@zhengyang:/work/sambashare/tools# tar -xvf arm-gnu-toolchain-12.2.rel1-x86\_64-aarcl

执行该命令,将把arm-linux-gcc 自动安装到/usr/loca/arm/arm-gnu-toolchain-12.2.rel1-x86\_64-aarch64-nonelinux-gnu目录。

```
root@zhengyang:/usr/local/arm# 11
总用量 24
drwxr-xr-x 6 root root 4096 5月 8 23:22 ./
drwxr-xr-x 20 root root 4096 5月 8 22:21 ../
drwxr-xr-x 7 root root 4096 3月 25 2009 4.3.2/
dr-xr-xr-x 8 root root 4096 7月 26 2010 4.4.3/
drwxr-xr-x 9 root root 4096 2月 12 2022 4.8.3/
drwxr-xr-x 9 802 802 4096 12月 11 07:16 arm-gnu-toolchain-12.2.rel1-x86_64-aarch64-nc
```

## 由于路径名太长,我们重命名:

```
root@zhengyang:/usr/local/arm# mv arm-gnu-toolchain-12.2.rel1-x86_64-aarch64-none-linux-
```

接下来配置系统环境变量,把交叉编译工具链的路径添加到环境变量PATH中去,这样就可以在任何目录下使用这些工具:

root@zhengyang:/work/sambashare/tools/usr/local/arm# vim /etc/profile

将解压目录下的bin目录添加至环境变量中:

```
export PATH=$PATH:/usr/local/arm/12.2.1/bin
```

注意: 如果配置了其它版本的交叉编译环境,需要将其屏蔽掉。

接下来使用以下命令使修改后的profile文件生效:

```
root@zhengyang:/usr/local/arm# source /etc/profile
```

然后,使用命令:aarch64-none-linux-gnu-gcc -v查看当前交叉编译链工具的版本信息:

```
root@zhengyang:/usr/local/arm# aarch64-none-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=aarch64-none-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/11.3.0/bin/../libexec/gcc/aarch64-none-linux-gnu/12.2
Target: aarch64-none-linux-gnu
Configured with: /data/jenkins/workspace/GNU-toolchain/arm-12/src/gcc/configure --target
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 12.2.1 20221205 (Arm GNU Toolchain 12.2.Rel1 (Build arm-12.24))
```

由于在/usr/local/arm/12.2.1/bin下没有arm-linux-gcc、arm-linux-ld、arm-linux-strip链接,所以我们进入bin路径:

```
root@zhengyang:/usr/local/arm# cd 12.2.1/bin/
```

## 创建自己创建软链接:

```
ln -s aarch64-none-linux-gnu-gcc arm-linux-gcc
ln -s aarch64-none-linux-gnu-ld arm-linux-ld
ln -s aarch64-none-linux-gnu-objdump arm-linux-objdump
ln -s aarch64-none-linux-gnu-objcopy arm-linux-objcopy
ln -s aarch64-none-linux-gnu-strip arm-linux-strip
ln -s aarch64-none-linux-gnu-cpp arm-linux-cpp
ln -s aarch64-none-linux-gnu-ar arm-linux-ar
ln -s aarch64-none-linux-gnu-as arm-linux-as
ln -s aarch64-none-linux-gnu-strings arm-linux-strings
ln -s aarch64-none-linux-gnu-readelf arm-linux-readelf
ln -s aarch64-none-linux-gnu-size arm-linux-size
ln -s aarch64-none-linux-gnu-c++ arm-linux-c++
ln -s aarch64-none-linux-gnu-gdb arm-linux-gdb
ln -s aarch64-none-linux-gnu-nm arm-linux-nm
ln -s aarch64-none-linux-gnu-g++ arm-linux-g++
```

然后,使用命令: arm-linux-gcc -v查看当前交叉编译链工具的版本信息:

```
root@zhengyang:/usr/local/arm/12.2.1/bin# arm-linux-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/12.2.1/bin/../libexec/gcc/aarch64-none-linux-gnu/12.2
Target: aarch64-none-linux-gnu
Configured with: /data/jenkins/workspace/GNU-toolchain/arm-12/src/gcc/configure --target
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 12.2.1 20221205 (Arm GNU Toolchain 12.2.Rel1 (Build arm-12.24))
```

## 参考文章

[1]rk3399移植 u-boot

[2]ARM CPU汇总 & 编译链选择

[3]<u>u-boot (3) —— spl</u>

[4] Rockchip Boot option

[5]U-Boot 之零 源码文件、启动阶段(TPL、SPL)、FALCON、设备树

[6]RK3399 TRM

[7]RK3399 Datasheet

[8]Rockchip RK3399TRM V1.3 Part1.pdf

[9]Rockchip RK3399TRM V1.3 Part2.pdf

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
2023-10-26	*通	9.9
2023-11-04	*慎	0.66
2023-11-24	*恩	0.01

创作不易,喜欢的话,请考虑支持一下(微信扫码)



分类: uboot , rk3399







