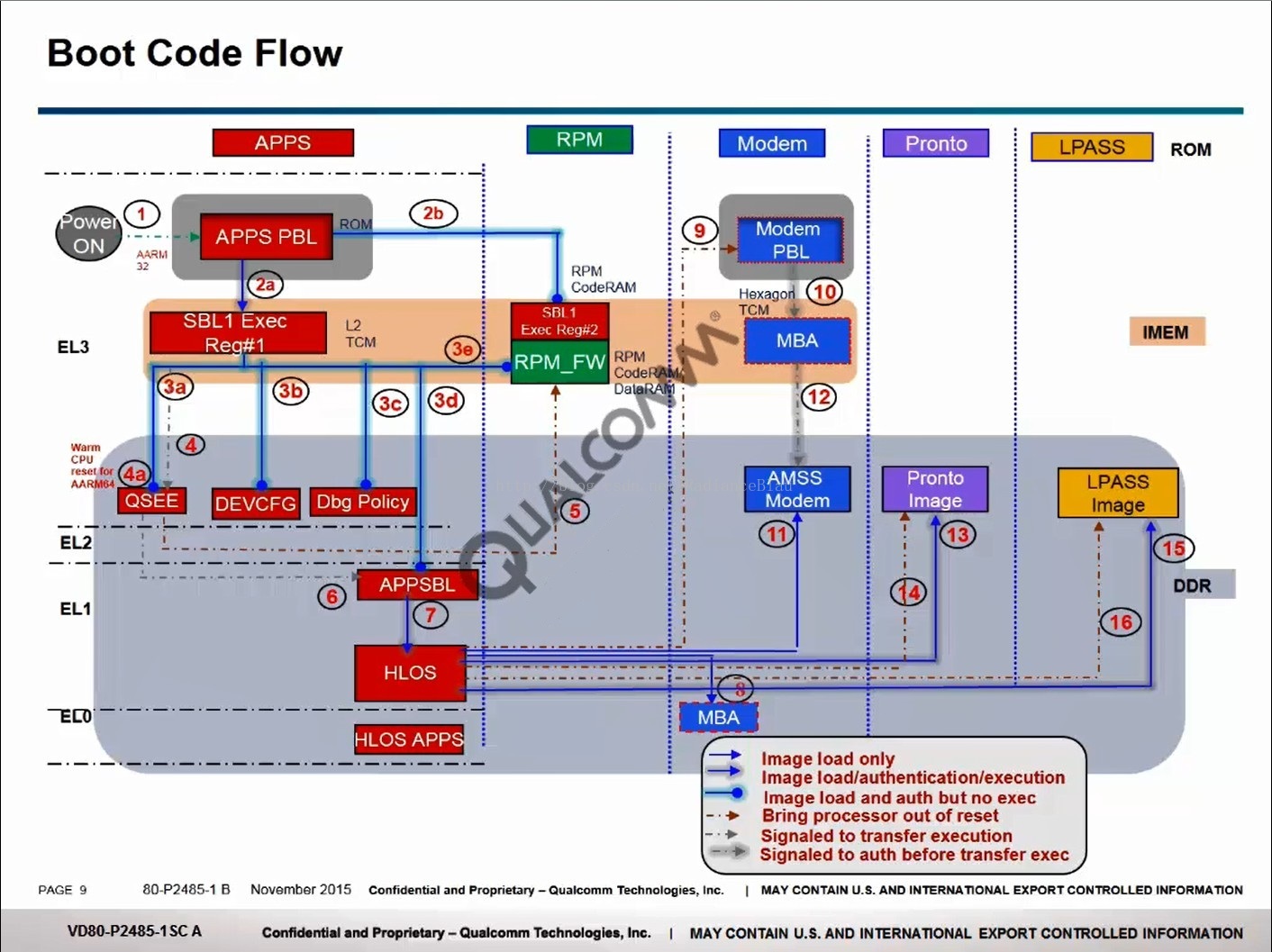
**linux驱动由浅入深系列：PBL-SBL1-(bootloader)LK-Android启动过程详解之一（高通MSM8953启动实例）**

翻译 2017年06月14日 14:59:47

* 标签：
* [linux](http://so.csdn.net/so/search/s.do?q=linux&t=blog) /
* [boot](http://so.csdn.net/so/search/s.do?q=boot&t=blog) /
* [qualcomm](http://so.csdn.net/so/search/s.do?q=qualcomm&t=blog) /
* [msm8953](http://so.csdn.net/so/search/s.do?q=msm8953&t=blog) /
* [PBL](http://so.csdn.net/so/search/s.do?q=PBL&t=blog)
* 4042

对于嵌入式工程师了解芯片启动过程是十分有必要的，在分析、调试各种问题的时候都有可能涉及到这方面的知识。同时这部分知识也是比较复杂的，因为其中涉及到芯片内部架构，启动各个阶段软件代码执行顺序，启动模式等等。下面以比较常用的Qualcomm MSM8953芯片的启动过程为例，进行宏观分析（大部分翻译了高通的手册^-^），下一篇文章进行代码分析。

[](https://img-blog.csdn.net/20170614194452081?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvUmFkaWFuY2VCbGF1/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center)

**处理器核心**

可以看到MSM8953中的处理器有5个，分别为：

1，  APPS           Cortex A53 core，运行android

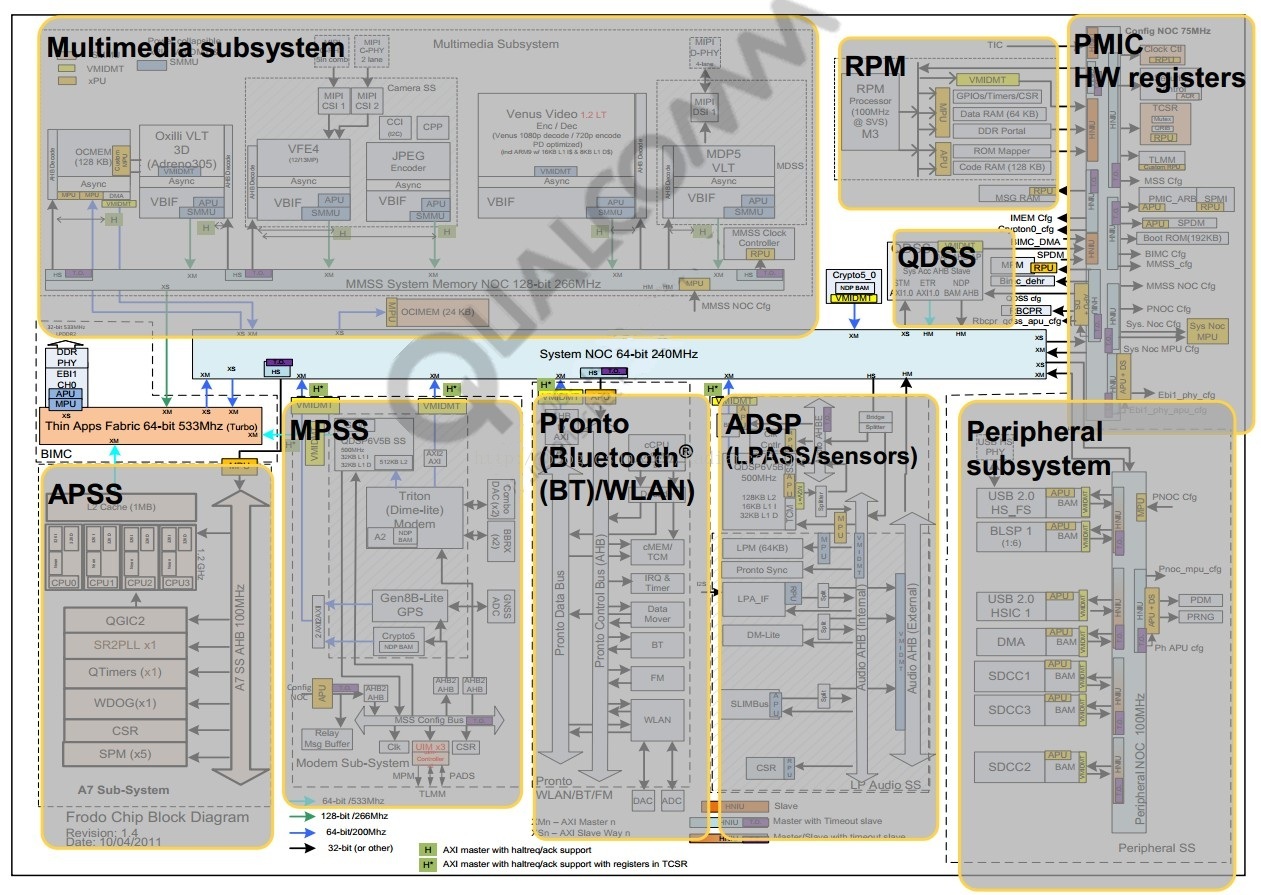
2，  RPM(Resource Power Manager)      CortexM3 core，主要用于低功耗应用

3，  Modem(MSS\_QDSP6)        高通自有指令集处理器，处理3G、4G通信协议等

4，  Pronto(WCNSS) 处理wifi相关代码

5，  LPASS        音频相关

对映下面芯片硬件结构图中各个处理器的框图来看就很清晰了。

[](https://img-blog.csdn.net/20170621123255672?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvUmFkaWFuY2VCbGF1/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center)

**启动相关image介绍**

1，PBL(Primary Boot Loader)   位于rom中，是芯片上电后执行的真正第一行代码，在正常启动流程中会加载SBL1。如果启动异常会虚拟出9008端口用于紧急下载（短接板子上的force\_boot\_from\_usb引脚(MSM8953 为gpio37)到1.8v可以强制进入紧急下载模式）。

2，SBL1(Second BootLoader stage 1) 位于eMMC中，由PBL加载，初始化buses、DDR、clocks等，会虚拟出9006端口，用于不能开机时dump ram

3，QSEE/TrustZone  安全相关，如fuse

4，DEVCFG OEM配置信息（如OEMLock）

6，  Debug Policy 调试相关

7，  APPSBL      即为BootLoader，目前使用LK（littlekernel）

8，HLOS(High LevelOperating System) 即为Linux/Android

9，Modem PBL  即为Modem处理器的PBL

10，MBA(Modem BootAuthenticator) Modem处理器启动鉴权

**启动流程**

1，  系统上电，使MSM8953从上电复位开始运行。

2，  在Cortex A53中运行的PBL会加载：

        a，  从启动设备（如eMMC）加载SBL1 segment1到L2（即为TCM）

        b，  加载SBL1 segment2到RPM处理器的RAM中。

3，  SBL1 segment1会初始化DDR，然后完成如下加载：

        a，  从启动设备加载QSEE image到DDR

        b，  从启动设备加载DEVCFG image到DDR

        c，  从启动设备加载Debug Policy image到DDR

        d，  从启动设备加载HLOS APPSBL image到DDR

        e，  从启动设备加载RPMfirmware image到RPM的RAM中。

4，  SBL1移交运行控制权给QSEE。QSEE建立安全运行环境，配置xPU，支持fuse。

        a，  SBL1运行在AArch32（译者注：名词相关知识见文末“附件介绍一”）模式，而QSEE运行在AArch64模式。为了切换到AArch64模式，SBL1会启动重映射器，操作RMR寄存器，然后触发warm-reset，QSEE就能够运行在AArch64模式了。

5，  QSEE通知RPM启动RPM 固件的执行。

6，  QSEE移交运行控制权给HLOS APPSBL。

        a，  APPSBL只能在AArch32模式开始运行。

        b，  这时AArch32的运行模式切换是在EL3/Monitor模式（译者注：名词相关知识见文末“附件介绍二”）完成的。通过查看APPSBL的ELF头能够得知其需要运行在32位指令集架构下。EL3/Monitor模式改变到32位模式，然后再启动APPSBL。

7，  APPSBL加载、验证kernel。APPSBL通过SCM调用改变到HLOS kernel需要的AArch64模式。这和之前LK直接跳转到kernel运行是不同的。

8，  HLOS kernel通过PIL加载MBA到DDR

9，  HLOS kernel对Hexagon modem DSP进行解复位。

10，Modem PBL继续它的启动。

11，HLOS kernel 通过PIL加载AMSS modemimage到DDR

12，Modem PBL验证MBA然后跳转到MBA。

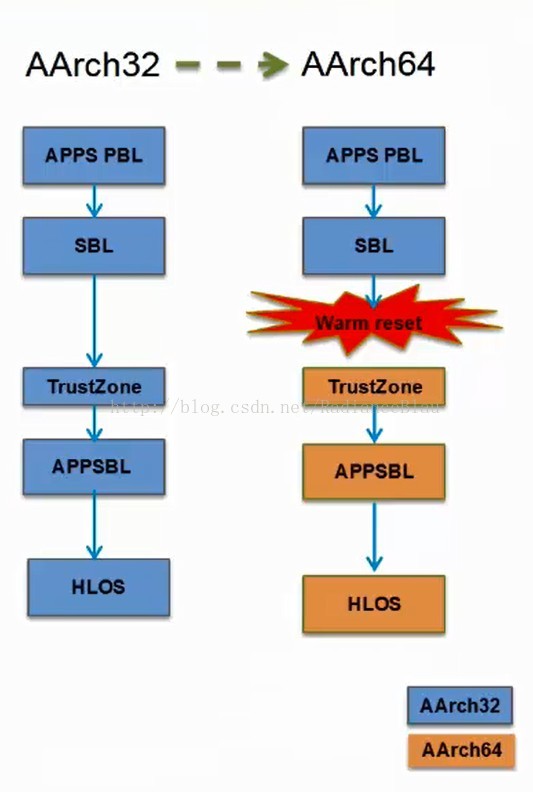
13，HLOS通过PIL加载WCNSS(Pronto)image到DDR

14，HLOS对WCNSS(Pronto)进行解复位以便Prontoimage开始执行。

15，HLOS通过PIL加载LPASS image到DDR

16，HLOS对LPSAA进行解复位以便LPASSimage开始执行。

下面是流程的简化图，其中区分了AArch32和AArch64位的QSEE/TrustZone



**关于eMMC和DDR的初始化时间问题**

从上面的描述中已经可以看清，为避免迷惑，在分离出来看看：

1，PBL中是含有eMMC驱动的，有访问eMMC的能力，自身运行在MCU内部SRAM中。

2，除了PBL程序的img在MCU片内ROM外，其余img均存储在eMMC中。

3，PBL首先从eMMC加载SBL1到L2（内部缓存并非DDR），SBL1同样运行在片内SRAM。

4，有SBL1初始化DDR各种时序后，DDR自此可用（eMMC一直可用）

5，再由SBL1加载其余各个img到DDR，然后按照linux的正常顺序启动^-^....

**附加介绍一：AArch64、AArch32**

AArch64是ARMv8架构的一种执行状态。

为了更广泛地向企业领域推进，需要引入 64位构架。同时也需要在 ARMv8架构中引入新的 AArch64执行状态。AArch64不是一个单纯的 32位 ARM构架扩展，而是 ARMv8内全新的构架，完全使用全新的 A64指令集。这些都源自于多年对现代构架设计的深入研究。更重要的是， AArch64作为一个分离出的执行状态，意味着一些未来的处理器可能不支持旧的 AArch32执行状态。虽然最初的 64位 ARM处理器将会完全向后兼容，但我们大胆且前瞻性地将 AArch64作为在 ARMv8处理器中唯一的执行状态。我们在这些系统中将不支持 32位执行状态，这将使许多有益的实现得到权衡，如默认情况下，使用一个较大的 64K大小的页面，并会使得纯净的 64位 ARM服务器系统不受遗留代码的影响。立即进行这种划分是很重要的，因为有可能在未来几年内将出现仅支持 64位的服务器系统。没有必要在新的 64位架构中去实现一个完整的 32位流水线，这将会提高未来 ARM服务器系统的能效。这样回想起来， AArch64作为在 Fedora ARM项目中被支持的 ARM构架是一个很自然的过程： armv5tel、armv7hl、aarch64。新的架构被命名为：aarch64，这同 ARM 自己选择的主线命名方式保持一致，同时也考虑到了 ARM架构名与 ARM商标分开的期望。

ARMv8-A 将 64位架构支持引入 ARM架构中，其中包括：

* 64 位通用寄存器、SP（堆栈指针）和 PC（程序计数器）
* 64 位数据处理和扩展的虚拟寻址

两种主要执行状态：

* AArch64 - 64 位执行状态，包括该状态的异常模型、内存模型、程序员模型和指令集支持
* AArch32 — 32 位执行状态，包括该状态的异常模型、内存模型、程序员模型和指令集支持

这些执行状态支持三个主要指令集

* A32（或 ARM）：32 位固定长度指令集，通过不同架构变体增强部分 32 位架构执行环境现在称为 AArch32。
* T32 (Thumb) 是以 16 位固定长度指令集的形式引入的，随后在引入 Thumb-2 技术时增强为 16 位和 32 位混合长度指令集。部分 32 位架构执行环境现在称为 AArch32。
* A64：提供与 ARM 和 Thumb 指令集类似功能的 32 位固定长度指令集。随 ARMv8-A 一起引入，它是一种 AArch64 指令集。   
  ARM ISA 不断改进，以满足前沿应用程序开发人员日益增长的要求，同时保留了必要的向后兼容性，以保护软件开发投资。在 ARMv8-A 中，对 A32 和 T32 进行了一些增补，以保持与 A64 指令集一致。

**附件介绍二：Exception Level**

• ARMv8定义EL0-EL3共 4个Exception Level来控制PE的行为.

|  |
| --- |
| ELx（x<4），x越大等级越高，执行特权越高 |
| 执行在EL0称为非特权执行 |
| EL2 没有Secure state，只有Non-secure state |
| EL3 只有Secure state，实现EL0/EL1的Secure 和Non-secure之间的切换 |
| EL0 & EL1 必须要实现，EL2/EL3则是可选实现 |

 Exception Level 与Security

|  |  |
| --- | --- |
| Exception Level | |
| EL0 | Application |
| EL1 | [**Linux**](http://lib.csdn.net/base/linux) kernel- OS |
| EL2 | Hypervisor (可以理解为上面跑多个虚拟OS) |
| EL3 | Secure Monitor(ARM Trusted Firmware) |
| Security | |
| Non-secure | EL0/EL1/EL2, 只能访问Non-secure memory |
| Secure | EL0/EL1/EL3, 可以访问Non-secure memory & Secure memory,可起到物理屏障安全隔离作用 |

关于ARMv8架构的知识，推荐如下博客：

http://blog.csdn.net/forever\_2015/article/details/50285865