移动基带安全研究系列之一

概念和系统篇

阿里安全 谢君

一、背景

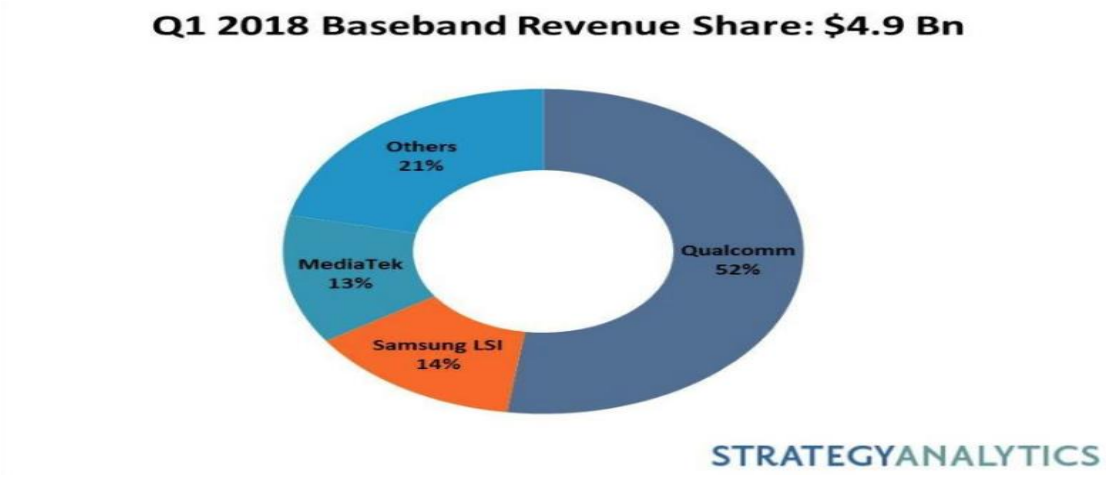
随着 5G 大浪潮的推进，未来万物互联将会有极大的井喷爆发的可能，而移动基带系统作为连接世界的桥梁，必将成为未来非常重要的基础设施，而基础设施的技术自主能力已经上升到非常重要的国家层面上的战略意义，从美国对待中国的通信产商华为的禁令就可以看得出基础技术的发展对一个国家的震慑，现今人类的生产生活已经离不开移动通信，未来也将会继续是引领人类科技的发展的重要媒介，人工智能，自动驾驶，物联网以及你所能想到的一切科技相关的发展都会与移动通信产生重要的联系，在此之上其安全性和可靠性将会成为人类所关心的重要问题，这也是笔者为了写这个系列文章的初衷，也希望更多的安全研究人员参与到基础设施的安全研究当中来，挖掘出更多的缺陷与隐患，完善未来的基础设施的安全。

二、概念和研究目的

3GPP 移动通信的标准化组织 3rd Generation Partnership Project，成立于上世纪末，主要职能是为了制订移动通信的技术标准，保证各个不同国家以及运营商在移动通信方面的兼容性，最常见的例子就是能够让我们的手机可以做到在不同的国家漫游使用。3GPP 所制定的移动通信技术标准涵盖了所有的 2/3/4/5G 通信相关的技术体系，产生了大量的技术文档供研究人员学习和参考，有兴趣的可以从 3GPP 的官方网站获取。

本系列文章研究对像是指 3GPP 定义的移动通信相关 2/3/4/5G 的基带软硬件和通信系统，例如手机的语音/短信/数据流量，以及物联网中使用的相关移动通信技术的端设备。基带系统本身是泛指无线通信系统里面的软/硬件和通信技术的集合体，例如蓝牙/Wi-Fi/GSM 都有基带系统，所以本系列文章所指的基带系统单指移动通信相关的 2/3/4/5G 技术相关的基带系统。

研究对象和目的：高通的基带芯片以及对 3GPP 定义的对通信协议栈的实现，基带系统是一个非常庞大且复杂的系统，包括软/硬件和通信技术的完美融合，所以具有相关设计能力的芯片产商很少，从 2018 年基带芯片的市场份额分布，高通是这个领域市场份额做的最大的芯片产商，高通是多个国内手机产商的供应商，例如小米，oppo/vivo 等，而华为现在已经有了自己基于海思芯片设计的基带系统，打破了国外基带芯片市场的垄断，现在华为的手机产品都是用的华为自产的海思基带芯片，不过软件系统还是基于人家的 VxWorks，不过刚刚华为发布了鸿蒙微内核系统，这个系统很有可能华为会把基带系统移植上去，现在应该加紧进行基带系统的移植工作，本身各家的基带系统都是非常封闭的，因为涉及各家的核心技术能力，加上移动通信的复杂性，研究的人也比较少，我研究的目的之一就是挖掘里面的一些设计逻辑，结合 3GPP 的协议的定义来更好的理解整个基带系统的实现，并且深入挖掘里面的攻击面以及如何更好的发现里面的安全问题。



上面图片来源

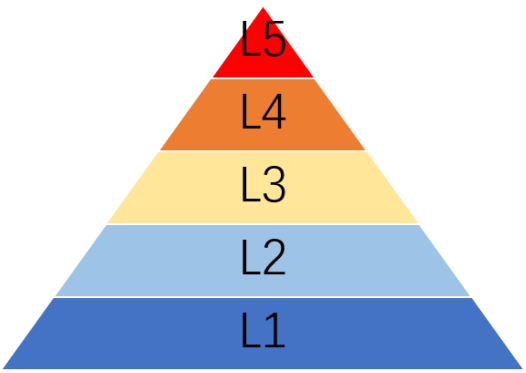
研究方法:

整个系列文章将会围绕高通基带系统对 3GPP 定义的协议栈的实现来挖掘里面的一些业务逻辑以及挖掘相关的攻击面来进行，所以我的研究方法会针对如下层次来进行。

1．操作系统
2．应用系统
3．3GPP 实现的协议栈
4．攻击面研究以及缺陷挖掘

封闭的基带系统需要大量的逆向工程的工作，来获取对基带系统行为的了解，逆向工程是安全研究者在挖掘未知的必备技能，什么时候需要逆向工程，在你无法获取目标研究对象的源代码和设计文档或者仅能够获取极少文档信息的情况下，想了解其目标对象的一些设计逻辑，原理和算法，这个时候你只能通过逆向工程这种合法手段来达到上面的目的。

逆向工程也分软件和硬件，现今的数字系统基本上都是通过软件来定义的，我们对于硬件的逆向工程就不展开讲了，有机会单独写出来，所以本文讨论的也基本上是软件层面上的逆向工程，而基带系统与硬件结合又是非常紧密的，所以对基带系统的逆向工程也需要硬件研究能力的支撑，逆向工程的难易程度也是分等级的，如下是我个人对逆向工程难易的理解，默认下面所有应用的固件都可以获取，通过研究工具的获取和研究的成本来分类。



没有任何芯片资料，包括指令架构，芯片手册，例如TI的电池芯片bqBMP架构，但是有固件

没有任何芯片资料，有编译器和反汇编器，有固件，例如泄露出来的TI的电池芯片bqBMP架构的开发工具链

有芯片架构信息，无芯片手册，例如运行在商业操作系统ThreadX上面的ACP设计的4G SoC的应用

有芯片架构信息，没有芯片手册，例如基于ARM Cortex-A7架构设计的LC1860芯片在Android上的应用

有芯片架构信息，有芯片手册，有固件无源代码，例如有stm32f108芯片在某设备应用的固件

而我们选择的研究对象高通的 MDM 系列芯片按我的理解难度应该在上图的 L3 的级别，非常有限的芯片信息的情况下。

三、高通基带硬件系统介绍

高通的基带硬件按照功能的不同分为两类：

1．MSM 系列　　（Mobile Station Modem）

2．MDM 系列　　(Mobile Data Modem)

MSM 系列主要是给手持移动通信设备使用，例如手机等

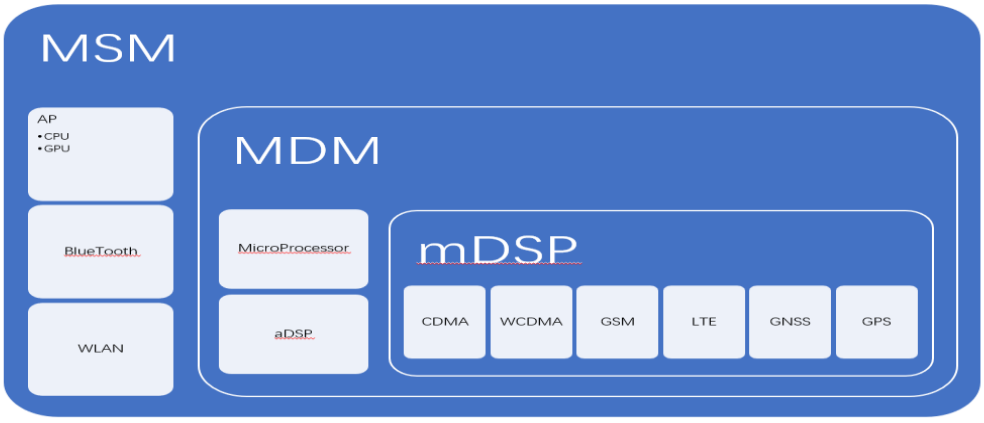
MDM 系列主要是给移动数据流量设备使用，车联网或其它物联网设备等

MSM 系列与 MDM 系列的区别

MSM 系列芯片包括应用处理器（Application Processor）和基带系统处理器(Baseband Processor)还有 Wi-Fi，蓝牙等,这个主要是提供整体的手机解决方案来给手机产商使用，Android 生态的大部分手机都是运行在高通的 MSM 系列的 SoC 之上，例如小米 5 手机搭载的高通骁龙系列 S820 的 SoC 就是 MSM8996 系列的芯片，应用处理器运行的是 Android 系统。

MDM 系列早期只包含(Baseband Processor)，主要是提供数据 modem 和语音的功能，苹果手机生态和车联网以及 4G 无线上网卡等应用中比较常见，比如 iPhone 8/8 Plus 和 iPhone X 都是配备的高通 MDM9655 的基带芯片，而宝马/奥迪车联网的 TBOX 则配备的 MDM6x00 系列的基带芯片,而 15 年生产的通用安吉星系统 TBOX 则采用的是 MDM9215 系列，为了能够提供更强大的业务逻辑能力，MDM 系列基带芯片 SoC 剥离了基带系统和业务系统，由两个 core 组成，比如 mdm9xxx 系列芯片包含一个 hexagon 的 DSP 基带处理核，以及一个 ARM Cortex-A 系列的核。

从功能上来说，MSM 系列的功能是包含了 MDM 系列的功能



所以高通的 MDM 系列的 Baseband Processor 并不是严格意义上的一块处理器，而是至少有 3 个 core。

1．一个基于 ARM 的微处理子系统

　a. ARM1136　　→ MDM6600

　b. ARM Cortex-A5 + Hexagon DSP　　→ MDM9215

2．一个基于高通 Hexagon QDSP 架构的 Modem DSP(mDSP)

3．一个基于高通 Hexagon QDSP 架构的 Application DSP(aDSP)

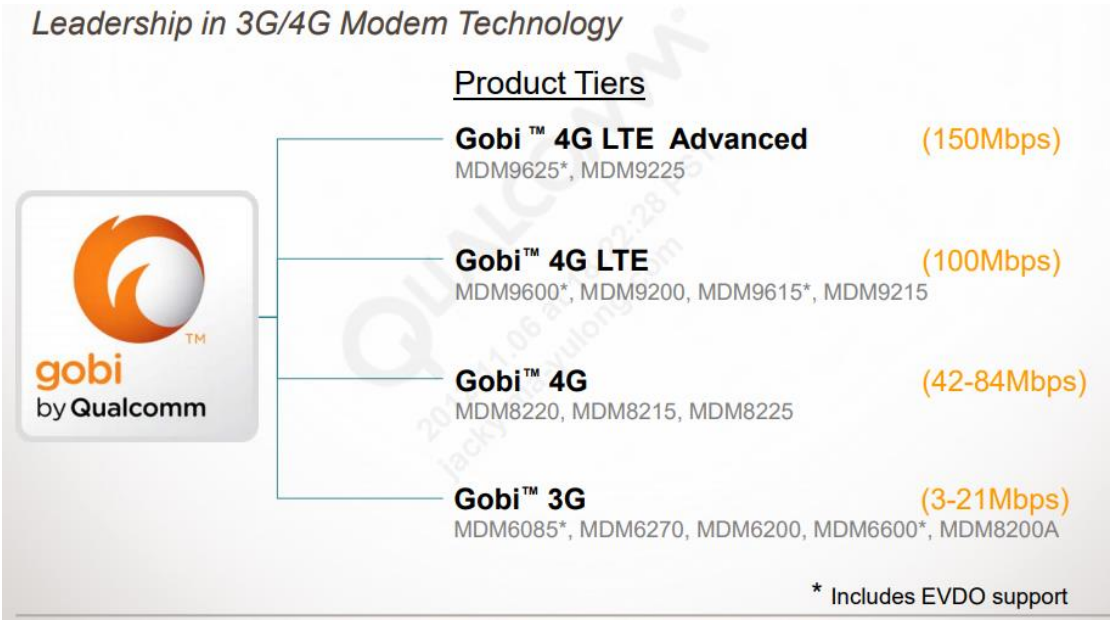
这 3 个 core 的主要功能如下：

1．这个基于 ARM 的微处理器属于基带系统的子系统（MDM6x00 基于 ARM1136 的架构，MDM9x15 系列基于 ARM Cortex-A5 以及新增了一个 hexagon DSP 处理器），它将协助 mDSP 和 aDSP 的初始化和与这两个 core 进行通信交互以及实现 3GPP 定义通信的所需的协议栈功能和算法，也可作为特定应用相关处理平台，例如在车联网中会将它作为 TBOX 的应用逻辑的处理器，MDM9x15 把 3GPP 协议栈的实现转移到了 hexagon DSP 上，而 MDM6x00 的 3GPP 协议栈的实现是在这个 ARM1136 上完成。

2．mDSP 的主要功能就是无线信号的调制与解调，在 3G 为代表的 MDM6x00 系列的 mDSP 主要实现 CDMA/WCDMA/GSM/GNSS 信号的调制与解调，在 4G 为代表的 MDM9x15 系列主要实现了包括 CDMA/WCDMA/GSM/LTE/GNSS 信号的调制与解调。

3．aDSP(Application DSP)，主要功能是实现与应用相关的信号调制与解调，例如语音信号的调制与解调（Audio DSP），常见的应用就是我们手机语音通话时编码与解码以及压缩就是通过这个 aDSP 来实现。

下图为高通 MDM 系列基带芯片的一些特性：

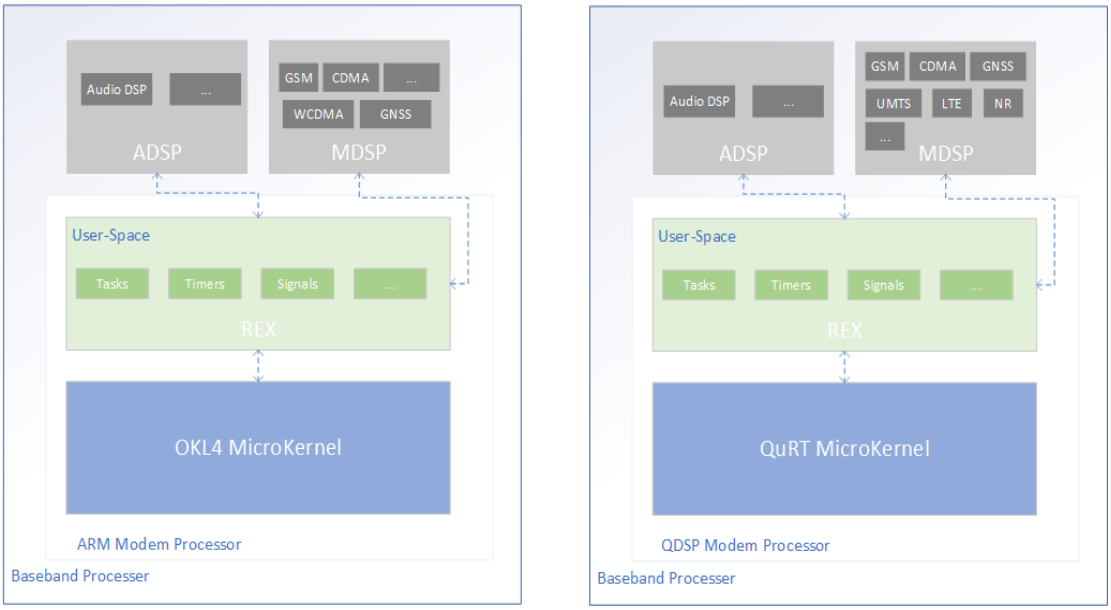


上面图片来源高通

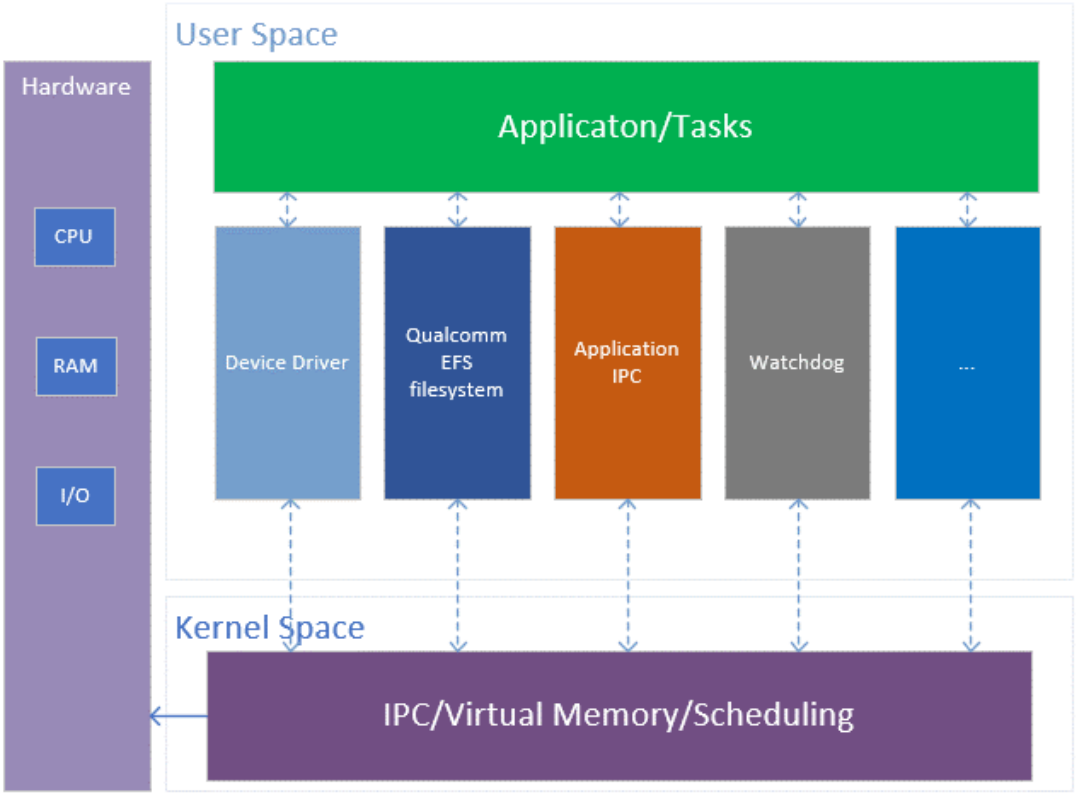
四、高通基带软件系统介绍

高通基带的软件系统从 2000 年左右就开始应用他们自己设计的嵌入式 rtos 系统 REX 来构建他们自己的手机基带应用系统 AMSS，而且基础的应用软件架构一直沿用至今，由于基带应用系统其复杂的特性以及大量的功能应用，为了保证其应用良好的移植性和兼容性，所以基带的底层系统采用精简的微内核系统 OKL4，这是一个开源的微内核系统，基于 ARM 的基带处理器都是采用的 OKL4 微内核，自从高通开发的新的 hexagon DSP 基带处理芯片后，一个名为 QuRT 嵌入式微内核系统因此而产生，这个 QuRT 前期也叫 Blast，它的出现应该是专门为 QDSPv6 架构的 DSP 处理器而开发的，

我们今天分析的 MDM6600 基带芯片是基于 OKL4 的微内核+REX AMSS 应用系统，而我们重点关注的其实也是运行在 REX 之上的 AMSS 应用，下图是整个基带系统的基于 ARM 和基于 hexagon QDSP 架构逻辑，未来 5G 应用还会继续沿用右边的架构。



微内核的好处在于，应用系统可以保持高度的可移植性，微内核系统只要满足基本的 IPC 通信机制，内存管理，CPU 调度机制即可，驱动文件系统等以及应用都可以在用户态来初始化完成，这对于需要支持多个硬件平台的高通来说无疑非常高效的做法，如下图是高通的系统架构。

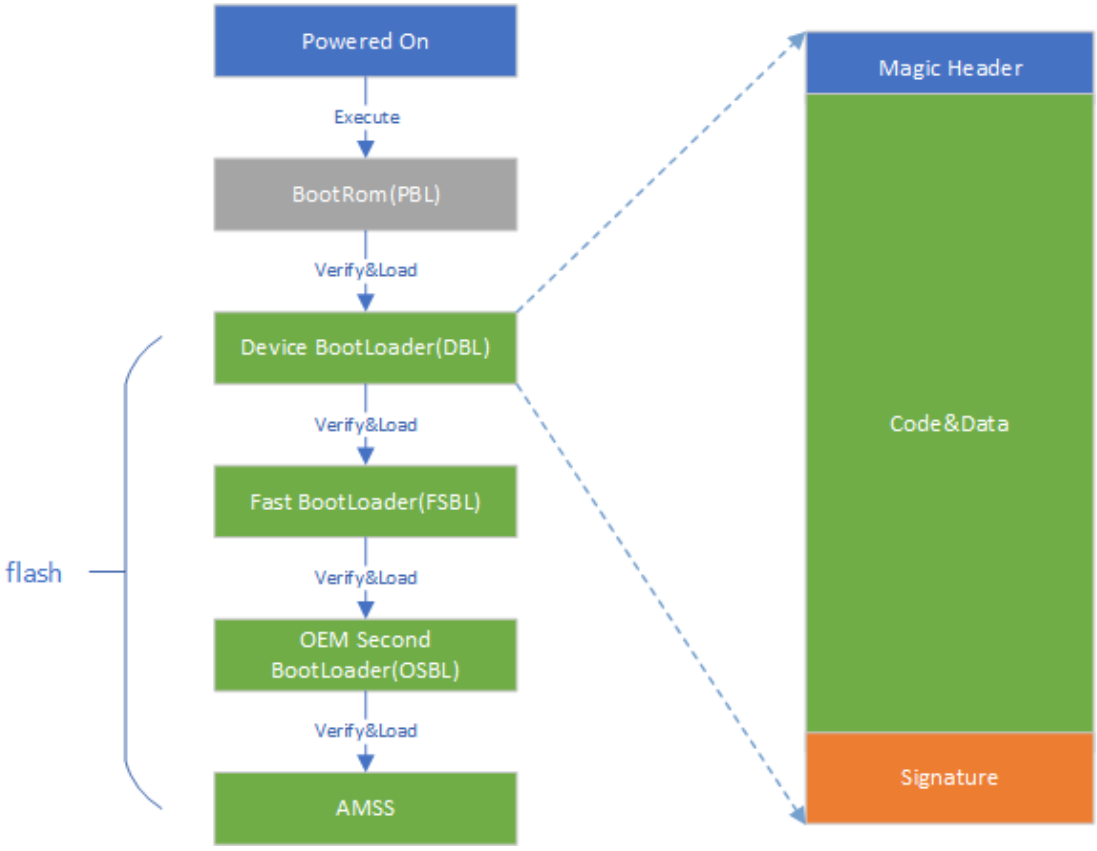


基带软件系统主要包括如下部分：

a. 启动管理
b. 内存管理
c. 文件系统
d. 定时器机制
e. 任务管理和 IPC 通信机制
f. 中断管理

a. 基带系统启动过程

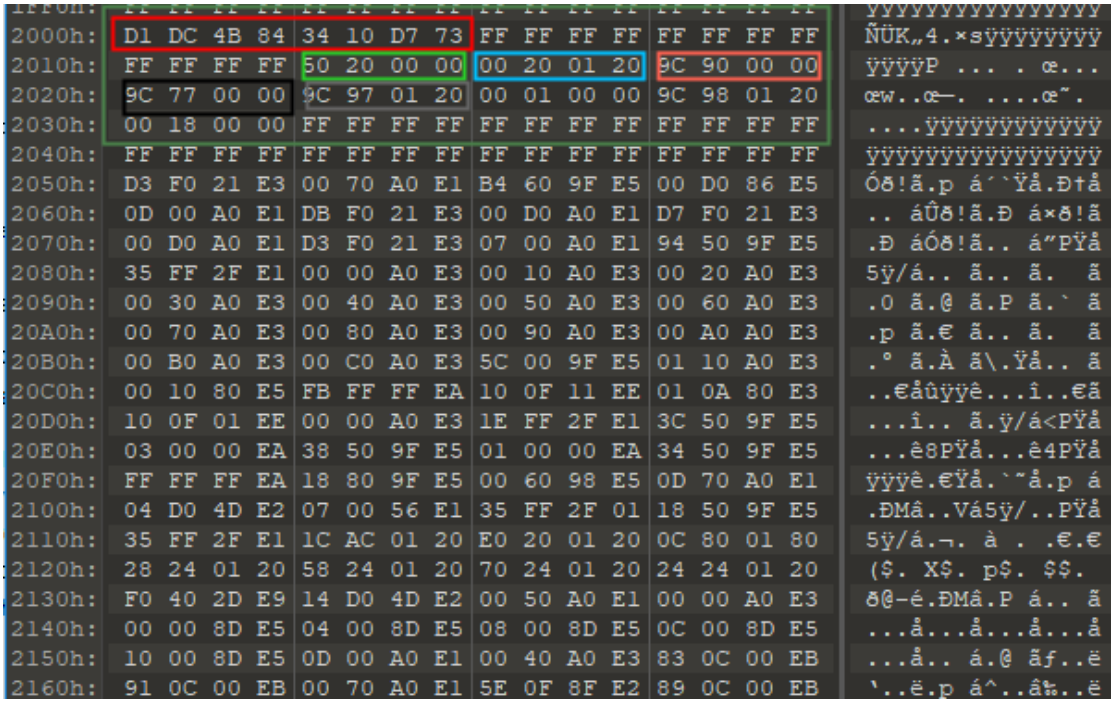
高通基带芯片很早就引入了 secure boot 的启动验证机制，来防止启动过程中运行的代码或数据被篡改，旨在安全可信计算，现在大部分高通系的手机都有这个功能，芯片上电后先被芯片的 BootRom 接管，该 BootRom 里面的代码不可篡改，里面存有 flash 控制器的基本读写功能，而且芯片的 OTP 区域可以存储产商授权的公钥证书，用于签名认证启动过程中需要认证的分区数据。以 MDM6600 芯片在某个车联网应用基带设备为例，它的启动过程如下：



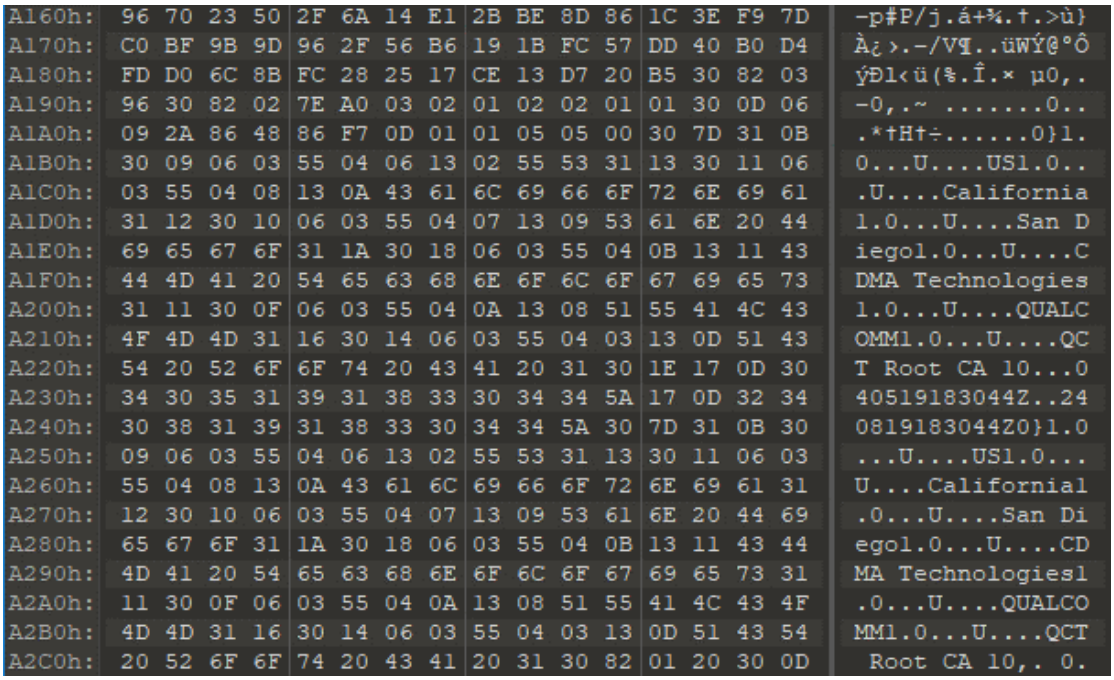
芯片上电后执行 BootRom 里面代码检测是否从 flash 启动，如果是从 flash 的第一个扇区读入数据到内存并搜索 secureboot 启动的 Magic Header，然后解析头部相应的数据结构，获取代码和数据的大小和偏移以及装载到内存的地址信息，签名/证书数据偏移和长度，如下图是 DBL 头部区域信息。

0x00 - CodeWord ("D1 DC 4B 84")
0x04 - Magic ("34 10 D7 73")
0x14 – Body start offset (0x2050)
0x18 - Loading address (0x20012000)

0x1C - Body size (Code + Signature + Certificate store size)
0x20 - Code size
0x24 - Signature address
0x28 - Signature length (256 bytes)
0x2C - Certificate store address
0x30 - Certificate store length



证书信息截图



当 BootRom 验证 DBL 代码和数据签名成功后，此后 DBL 的代码接管执行，然后搜索 MIBIB 分区表，获取各个分区的起始 block 信息，然后在相应的块去读取相应的数据，接着就是验证相应分区数据的签名，然后相应的分区代码接管，完成一系列的信任启

动链，DBL 验证成功后，验证 FSBL，然后是 OSBL，最后是 AMSS。

0x00 - CodeWord ("AA 73 EE 55")

0x04 - Magic ("DB BD 5E E3")

0x0C – Partition Nums (0xa)

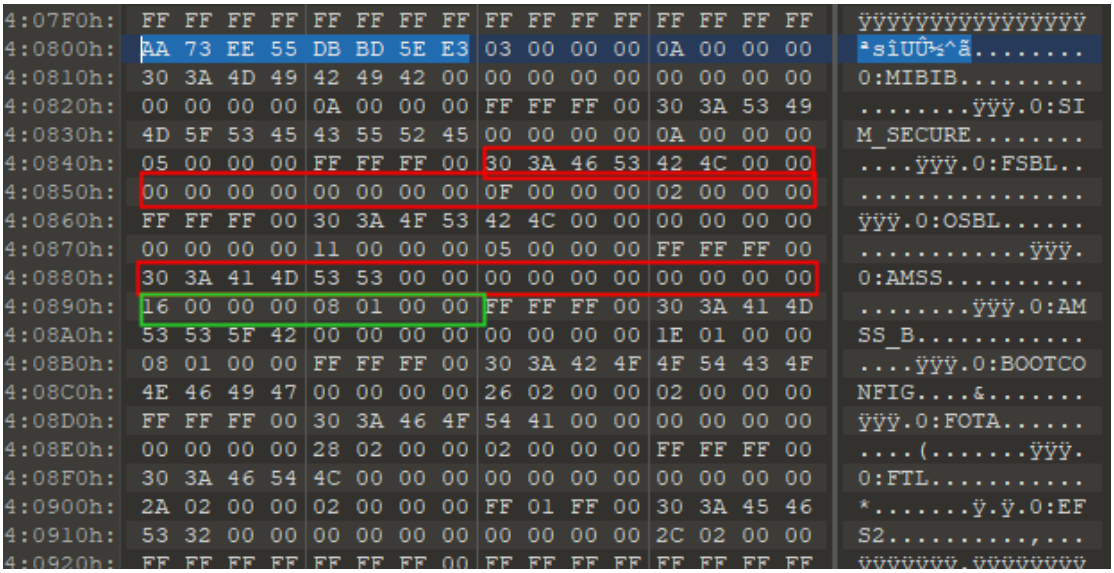
每个分区表信息长度 0x1c，例如

0x00 – 0x10 partition name (0:FSBL)

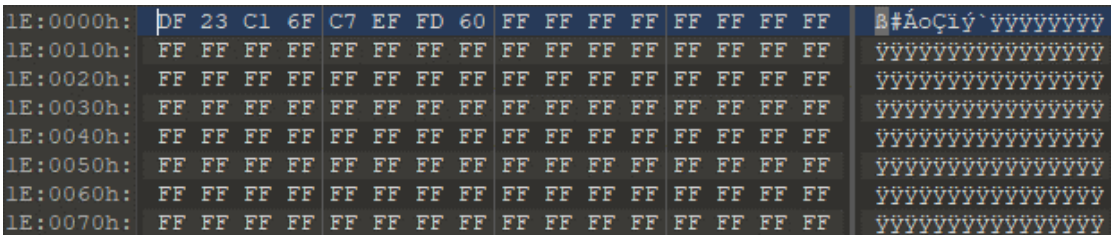
0x10 – Partition start block information (0x0f)

0x14 – Partition block length (0x2)

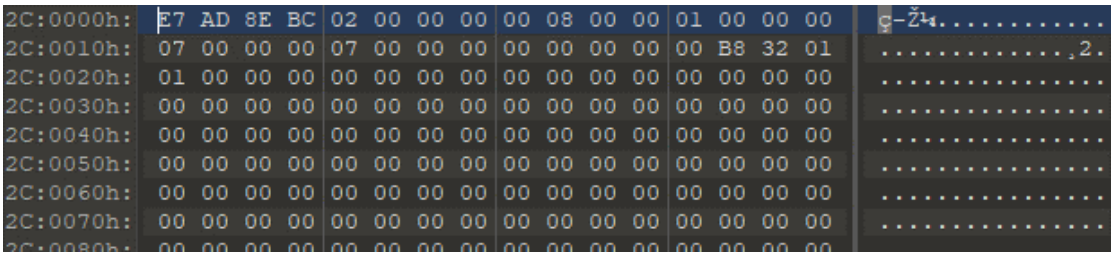
这里定义的每个页是 0x800 字节，每个块 block 有 64 个页，所以每个 block 的长度是 0x20000 字节，所以根据这个信息我们就可以定位这些分区的物理偏移信息。



例如 FSBL 的物理偏移为 0x20000*0xf=0x1e0000



AMSS 的物理偏移为 0x20000*0x16=0x2c0000



b. 基带系统内存管理

当基带系统的安全信任启动链验证完成后，最后系统被 AMSS 系统代码接管， AMSS 系统定义了代码执行的内核特权模式以及 AMSS 应用模式，设置页表（映射硬件外设

地址到页表中）并且开启 MMU(内存管理单元)，在某些敏感的内存地址区域通过 MPU 的特性来进行保护，只有特定权限的应用的可以访问，应用模式的代码想要进入内核态（例如 IPC 消息发送），可以通过设置的特权中断指令 SVC 进入内核态，下图就是进入特权 syscall 的中断向量表入口。

```
OAD:005A4258 PUSH            {R4-R11,LR}
OAD:005A425C MOV             R4, R1
OAD:005A4260 MOV             R1, #0
OAD:005A4264 MOV             R3, #0x2000
OAD:005A4268 MOV             R12, SP
OAD:005A426C MOV             SP, #0xFFFFFF00
OAD:005A4270 SVC             0x1400
OAD:005A4274 MOV             R0, R3
OAD:005A4278 POP             {R4-R11,PC}
OAD:F0025000 ; ---------------------------------------------------------------------------
OAD:F0025000 B               arm_reset_exception
OAD:F0025004 ; ---------------------------------------------------------------------------
OAD:F0025004 B               arm_undefined_inst_exception
OAD:F0025008 ; ---------------------------------------------------------------------------
OAD:F0025008 B               arm_swi_syscall
OAD:F002500C ; ---------------------------------------------------------------------------
OAD:F002500C B               arm_prefetch_abort_exception
OAD:F0025010 ; ---------------------------------------------------------------------------
OAD:F0025010 B               arm_data_abort_exception
OAD:F0025014 ; ---------------------------------------------------------------------------
OAD:F0025014 NOP
OAD:F0025018 B               arm_irq_exception
OAD:F002501C ; ---------------------------------------------------------------------------
OAD:F002501C MRS             R8, SPSR
OAD:F0025020 ANDS            R9, R8, #0xF
OAD:F0025024 TSTNE           R8, #0x80
OAD:F0025028 BEQ             arm_fiq_exception
```

通过初始化页表完成内核地址空间和外设硬件地址映射，开启 mmu，创建第一个 rootTask 后切入用户态空间，初始化用户态需要创建的应用与驱动，这里主要介绍应用层堆内存结构以及内存分配和回收算法。

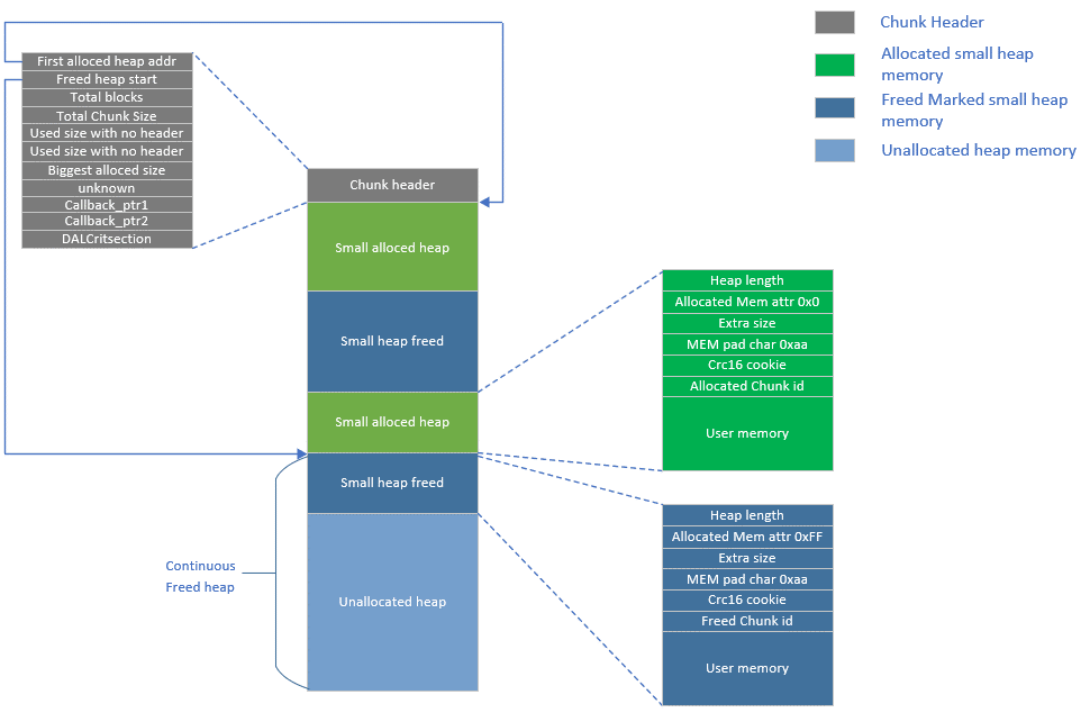
REX 系统堆内存分两种类型：

Big chunk（大堆）

small heap（小堆）

大堆在不同应用初始化的时候指定内存的起始地址与长度，而且根据应用功能的不同，分配方式也不同，小堆将会在大堆上进行分配使用，大堆由于给使用的应用不同，分配小堆的方式有所不同。

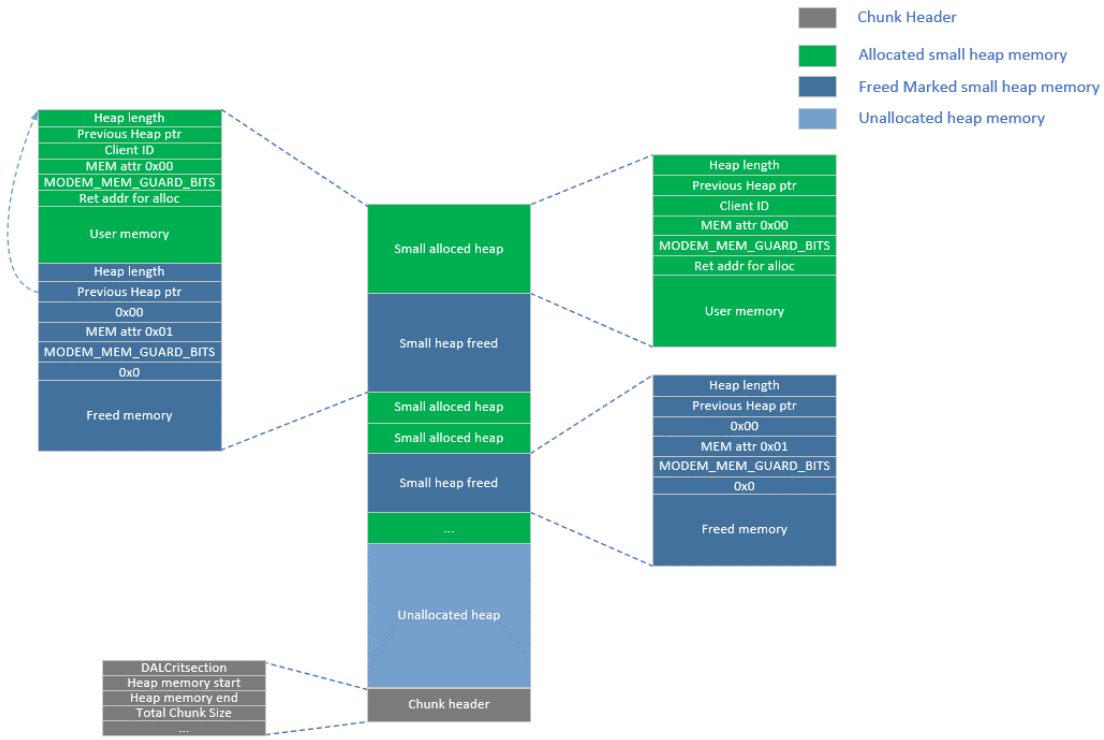
a．大堆类型 1,内存连续，分配小堆的方式是顺序分配，前面是分配好的小堆，后面是连续的空闲堆块，分配小堆只会在连续的空闲块上进行分配，例如前面多个分配好的小堆其中一个需要被释放后，只是把这个小堆的属性标记为 freed，但由于它后面的小堆到连续的空闲块中间有标记为已经分配属性，所以后续在分配小堆的过程中不会考虑这块已经被释放的内存，除非要释放的小堆内存和连续的空闲块紧挨着，下一次分配内存时才会从这个已经标记为释放的内存上进行分配，而是直接到后面的连续空闲块上进行分配，这样做的目的是为了分配和释放内存更高效，虽然牺牲了一些空间，结构如下图。



下图是这种 chunk 上分配小堆的状态信息示例

```
addr 0x221b070 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b0b0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b1b0 mem_len 0x40 attr 0xaa0c00ff memtag 0x2915126 status Freed
addr 0x221b1f0 mem_len 0x100 attr 0xaa0c00ff memtag 0x2915126 status Freed
addr 0x221b2f0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b330 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b430 mem_len 0x30 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b460 mem_len 0x80 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b4e0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b520 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b620 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b660 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b760 mem_len 0x50 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221b7b0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b7f0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b8f0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b930 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221ba30 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221ba70 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bb70 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bbb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bcb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bdb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221beb0 mem_len 0x180 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221c030 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c070 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c170 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c1b0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c2b0 mem_len 0x310 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221c5c0 mem_len 0xc0 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c680 mem_len 0x47cf0 attr 0xaa00ffff memtag 0x0 status Freed
cnt 0x5a
```

b．大堆类型 2，（modem chunk）,也是一个连续内存区域，但是 chunk header 在内存的底部，上部为分配小堆区域，分配顺序也是从上往下分配，小堆的头部数据结构中会指向上一个已经分配好的小堆，通过单向链表进行小堆内存的回溯，最上面的小堆回溯指针为空，但是它的内存分配算法跟上面的不同，就算要被释放的小堆内存和空闲块不挨着，但是它任能在下一次的堆内存申请中被重用，只要它的大小合适，而且小堆数据结构与类型 1 也不同，基本结构如下图。



Modem 使用大堆结构示例

```
addr 0x1d70a30 pre_mem 0x1d709d4 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70a8c pre_mem 0x1d70a30 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70ae8 pre_mem 0x1d70a8c mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70b44 pre_mem 0x1d70ae8 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70ba0 pre_mem 0x1d70b44 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70bfc pre_mem 0x1d70ba0 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70c58 pre_mem 0x1d70bfc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70cb4 pre_mem 0x1d70c58 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70d10 pre_mem 0x1d70cb4 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70d6c pre_mem 0x1d70d10 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70dc8 pre_mem 0x1d70d6c mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70df0 pre_mem 0x1d70dc8 mem_len 0x34 client_id 0x0 mem_attr freed ret 0x0
addr 0x1d70e24 pre_mem 0x1d70df0 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70e80 pre_mem 0x1d70e24 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70edc pre_mem 0x1d70e80 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70f38 pre_mem 0x1d70edc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70f94 pre_mem 0x1d70f38 mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70fbc pre_mem 0x1d70f94 mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70fe4 pre_mem 0x1d70fbc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d71040 pre_mem 0x1d70fe4 mem_len 0x230f34 client_id 0x0 mem_attr unallocated ret 0x0
cnt 0xed
```

我们可以看到 chunk 类型 1 和 chunk 类型 2 上面分配的小堆内存结构稍有不同，数据结构如下：

```
Small heap1{
  Uint32 size;//+0  分配内存空间的长度加上头部长度 0xc 字节
    Uint8    mem_flag;//+0x4  内存属性标志，0 表示已分配，0xff 表示释放掉的内存
    Uint8    extr_mem_flag;//+0x5  扩展内存属性标志，0 表示内存分配过，0xff 表示
                                    //内存空间，没有被使用过
    Uint8    mem_extra_size;//+0x6  额外分配的内存长度，为了内存 0x10 字节对齐
                                    //所额外增加的申请内存长度，必须小 0x10 字节
    Uint8    mem_pad_char;//+0x7  填充字节 0xaa
    Uint16 crc16_cookie;//+0x8   对传入的第三个参数的 crc16 计算的值
    Uint16 mem_id;//0x0a      内存标识,第四个参数传入
    Uint8 mem_buffer[size-0x0c];//+0xc  用户使用内存 buffer
}

Small modem heap{
    Uint32 size;// +0  分配内存空间的长度加上头部长度 0x10 字节
    Uint32 *pre_alloc_ptr;//+4  指向上一个分配好的小堆内存头部指针
    Uint8 client_id;//+8  申请内存的应用 id 值，modem 功能中定义了
                    //RRC/CM/SM/RLC/gstk/wms 等多个应用，这个 id 来标识申请内
                    //存的应用来自于哪里
    Uint8    mem_flag;//+0x9  内存属性标志，0 表示分配了，1 表示释放了，
                        //3 表示未使用
    Uint8    unknown_byte;//+0xa
    Uint8    mem_guard_bits;//+0xb modem 内存保护标志 0x6a
    Uint32    alloc_ret_addr;//+0xc  分配内存函数的下一条指令地址，目的是为了
                                    //确定执行内存分配行为的精确地址
    Uint8    mem_buf[0xsize-0x10]; //+0x10  供用户使用的内存 buffer
}
```

c. 基带系统文件系统

由于篇幅问题，我会对 Qualcomm 基带的文件系统 EFS 单独写一篇详细的分析文章。

d. 高通基带芯片定时器（Timer）

定时器是嵌入式芯片非常重要的组成部分，它在嵌入式操作系统的 CPU 调度和定时任务执行，以及精确的延时等待等操作中扮演着非常重要的角色，高通的基带芯片的定时器调度算法大体都差不太多，我们基于 ARM1136 架构的 MDM6600 基带芯片对定时器算法进行了深入分析。

MDM6600 的定时器是通过 Sleep Timer 控制器来实现的，它包含两个 16 位的 Timer0 和 Timer1，以及一个 32 位的 TimeTick 的计数器(counter)，它们的功能用途如下.

1．Timer0　供 watchdog 使用

2．Timer1　供 3G 的 wcdma 的功能模块使用

3．TimeTick 系统计数器，服务于系统的子任务模块创建的定时器任务的执行以及延时功能的使用

Timer0 应用于 watchdog 功能中，Watchdog 在实时嵌入式系统中扮演着非常重要的角色，它监控任务的正常运行，监控的任务必须定时喂狗（feed dog），watchdog 才认为你在正常工作，要不然就可能会直接 reset 系统，后续也会介绍它在基带里面具体监控的应用。

Timer1 将会在 3G WCDMA 应用中收发相关的定时中断中会详细介绍。

TimeTick 是一个 32 位的系统计数器，初始化后会从 0 开始计数，计数到 0xffffffff 后溢出到 0 后重新开始计数，主要功能如下：

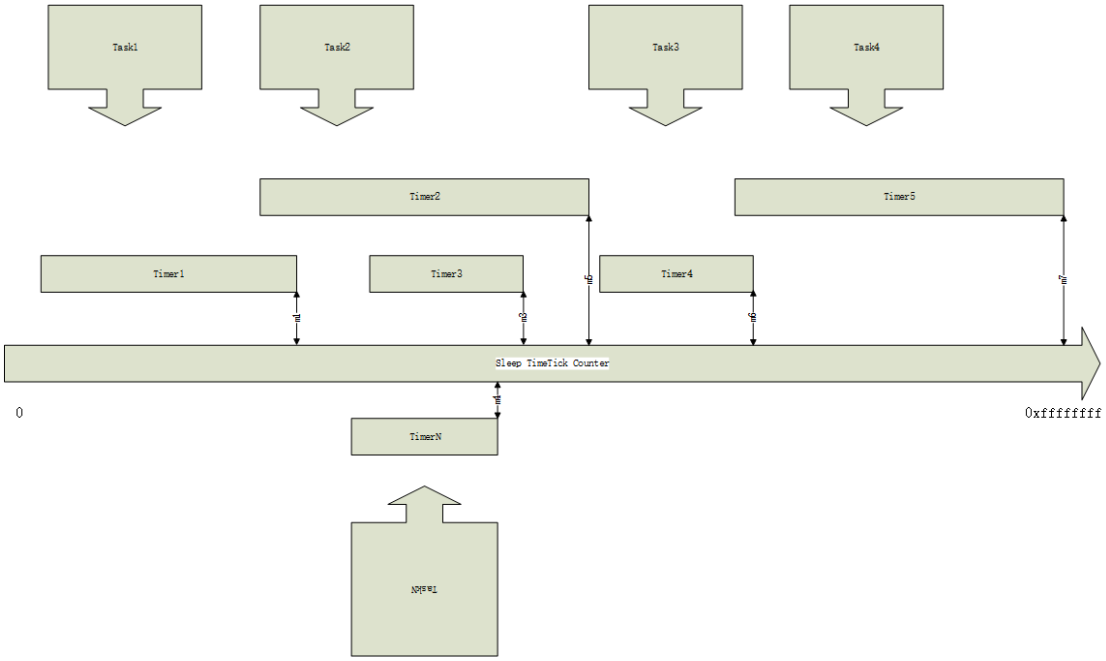
1．执行定时任务
   a. 执行一次
   b. 周期性执行
2．执行延时功能
   a. 延时等待

TimeTick 的时钟源为 32768Hz，这意味着这个计数器 1 秒钟会计数 32768 次，通过这个信息我们可以大致计算出从 0 计数到 0xffffffff 需要 36 个小时。

定时任务功能特性：

a. 通过设置 TimeTick 的 match value 来决定计数器计数到这个值后产生一个中断，中断里面可以处理相应的定时任务，以及设置新的 TimeTick match value。
b. 所有的定时任务都会存储在定时任务列表中，提供定时任务的插入，删除，暂停，唤醒执行等功能。

下图描绘了定时器任务执行的基本过程



在基带系统中存在多个应用任务，每个任务的执行都是依赖内核的 CPU 调度，常见的方式就是时间片和优先级切换让各个不同的任务有机会得到执行，而某些任务在运行过程中的某个时机可能会创建一个或者多个定时器任务，例如上图所示的任务 Task1 创建的定时器任务 Timer1，Task2 创建的定时器任务 Timer2 和 Timer3，处理这些任务的算法如下：

1．创建定时任务时，获取当前 TimeTick 的计数
2．把延时换算法成计数，比如 1 秒等于 32768 次计数

3．把当前 timetick 计数加上延时的计数值作为该定时任务中断触发的 match value
4．遍历所有定时任务，根据任务设置的定时任务中断触发的 match value 大小排序插入到定时任务列表
5．当 timetick 的计数到达某个定时任务的 Match value 的时候产生中断，中断处理例程 ISR 会通过向 DPC（Deferred Procedure Calls）发送执行定时任务的消息去执行该定时任务的例程函数，如果只是延时任务就不需要执行了，同时更新 timetick 的下一次中断产生的 match value，并把这个定时任务从定时任务列表中移除

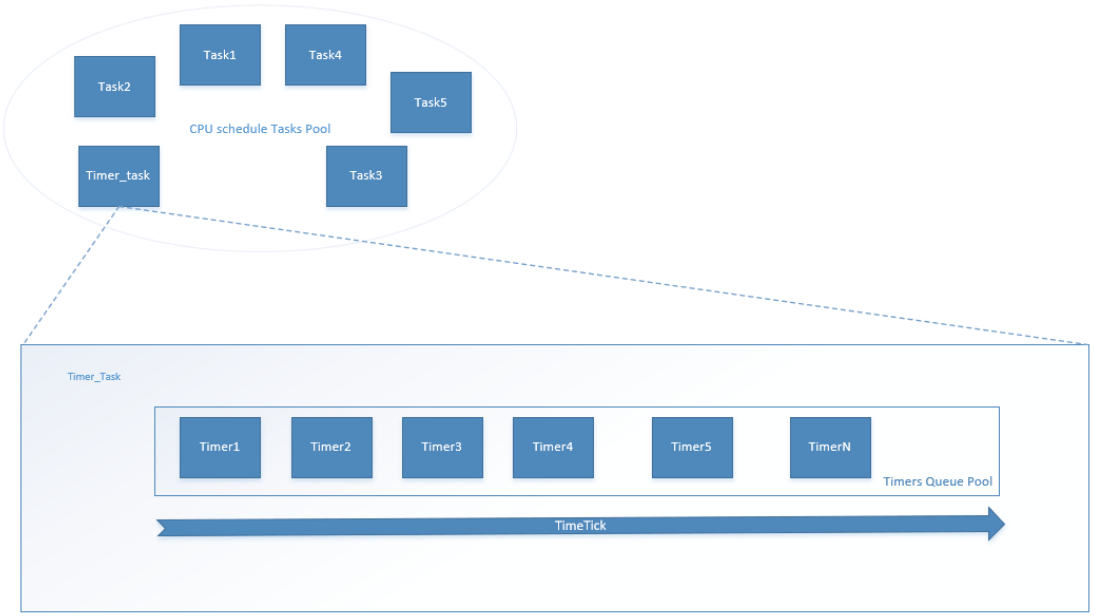
如上图举例:

应用任务 \ 定时任务/MV		
Task1	Timer1/M1	
Task2	Timer2/M5	Timer3/M3
Task3	Timer4/M6	
Task4	Timer5/M7	
TaskN	TimerN/M4	

按照时间推进过程，这些定时任务执行需要设置的 Match value 来产生中断的顺序依次是:

M1 → M4 → M3 → M5 →M6 → M7

所以在基带系统里面会有一个专们的定时器应用任务来管理维护其它应用任务产生的定时器任务的调度



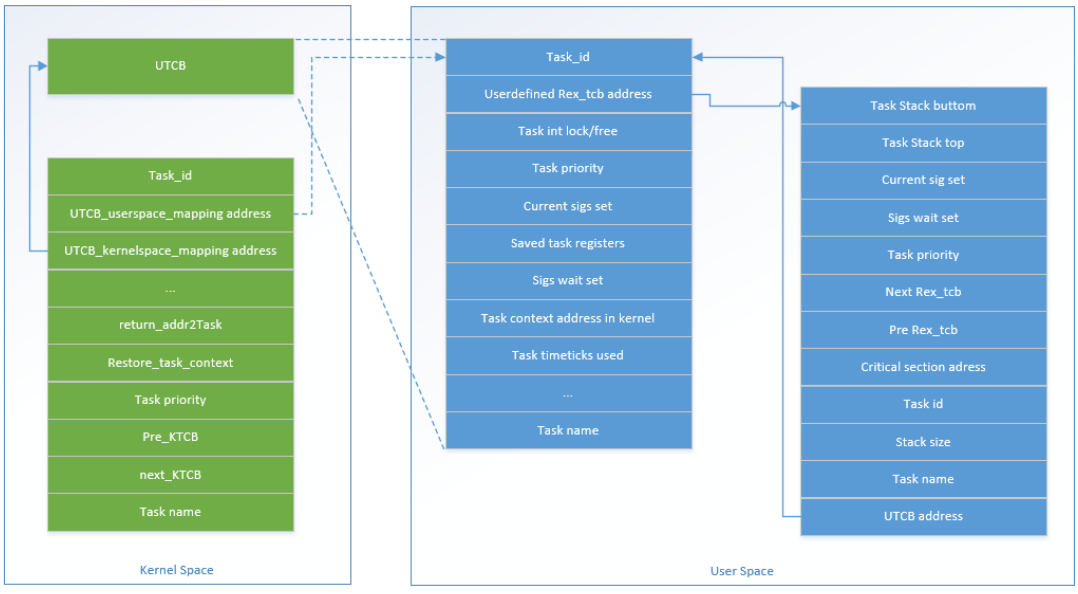
e. 任务管理和 IPC 通信机制

上面提到基带系统从内核态切入到应用态会创建第一个 rootTask 应用任务，这个任务有点类似 linux 系统里面的 init 进程，rootTask 接下来会创建应用权限很高的 DPC_task 任务（负责高实时异步任务执行），权限仅次于 IST（interrupt service threads,中断服务接管线程），然后是应用层的全局管理任务 main_task 将会启动，接下来业务所需的各种驱动相关的初始化和通信业务逻辑任务将在 main_task 任务中得以创建，例如中断接管服务相关的 IST(interrupt Service Threads)，定时器业务相关的

timer_task，qualcomm EFS 文件系统相关的 fs_task，任务监控相关的 watchdog_task，以及 GSM/UMTS 业务相关的通信层面的各个任务。

每个任务被创建时，REX 内核和用户态各自会维护一套数据结构，以及用户自定义的一套 TCB 结构：
内核态—>KTCB（Kernel Task Control Block）
用户态—>UTCB（User Task Control Block）
用户态—>REX_TCB(用户自定义 TCB 结构)

在内核态，cpu 通过 KTCB 来管理调度所有的任务，以及管理用户态任务在切换时存储任务的 context 信息。
内核态的 KTCB 列表包含 1 个 idle 内核线程,8 个 IRQ 和 1 个 FIQ 内核线程任务 KTCB 结构，以及每一个用户任务 UTCB 对应的在内核空间存储的 KTCB 结构。
在用户态，每一个任务都会通过 UTCB 结构存储任务信息供用户读写，并且该 UTCB 结构也会映射到内核空间供内核读写，而用户态的 REX_TCB 是供用户自定义的数据结构，用户可以自定义一些方便业务间通信的数据结构。
任务的几个重要的特性：
1. 内核态读取 0xf0000008 地址存储着当前活动任务的 KTCB 指针
2. 内核态 0xf001e000 存储着所有 KTCB 结构的列表
3. 在用户态读取 0xff000ff0 地址值可以获取当前活动任务的 UTCB 指针
KTCB，UTCB 和用户定义的 TCB 结构关系如下图：



从上图可知，UTCB 结构通过内存映射的方式会被内核态和用户态共同读写，utcb 通过 timetick 计数器来记录任务使用了多少 cpu 时间，为任务调度提供了很好的判断条件。

每个被创建的任务都包含一些信息，初始化时会存储在 UTCB 结构和用户定义的 TCB 结构中：

1. 任务的执行函数地址
2. 任务执行函数参数
3. 堆栈起始地址
4. 堆栈的长度
5. 任务优先级别
6. 存储用户 tcb 地址
7. 任务名称

任务创建函数定义类似结构如下，不同的版本可能会有一些变形：

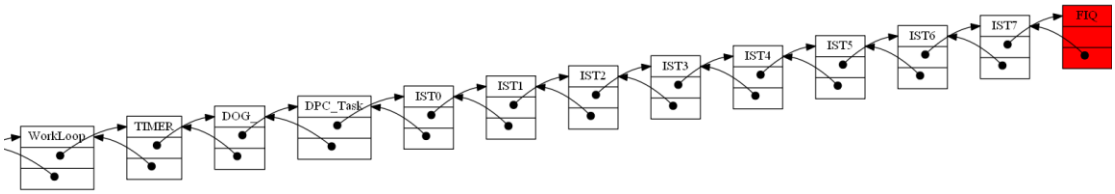
Void *createTask(void *utcb,void *task_func_ptr,uint32 stack_size,void *stack_buttom,void *stack_top,uint32 task_priority，void *pararm)

用户定义 tcb 结构是一个双向链表结构，每个用户 tcb 会把高于自己优先的任务插入到前链，低于自己优先级的任务插入到后链，所有的任务中中断接受任务中的 FIQ 任务的优先级是最高的，它用于快速处理来自于 fiq 中断请求。

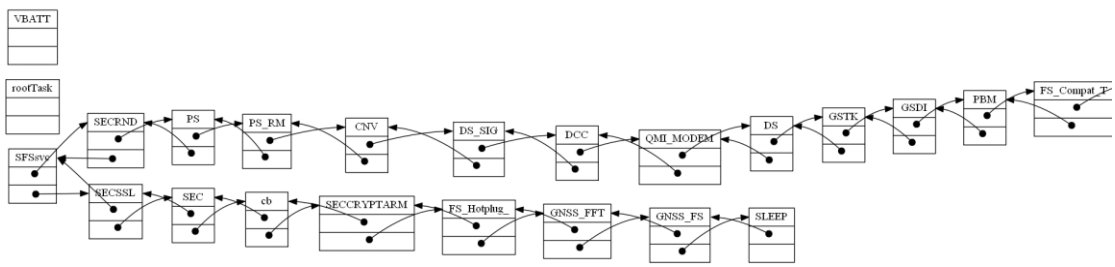
下图是枚举出的部分运行的任务列表：

```
task counts 96
utcb 0x140000 taskname rootTask tid 0x34001  priority 0x64 rex_tcb 0x22aaf84 pre_rextcb 0x0[none] next_rextcb 0x0[none]  timeused 0x0
utcb 0x140100 taskname DPC Task tid 0x38001  priority 0xf3 rex_tcb 0x232eac4 pre_rextcb 0x18ef3d4[IST0] next_rextcb 0x17e8020[DOG ]  timeused 0x0
utcb 0x140200 taskname Main Task tid 0x3c001  priority 0xc4 rex_tcb 0x185693c pre_rextcb 0x18927e8[GNSS_PGI] next_rextcb 0x1894a1c[GNSS_PP]  timeused 0x75d0
utcb 0x140300 taskname WorkLoop tid 0x40001  priority 0x64 rex_tcb 0x2217e7c pre_rextcb 0x17f63f4[UI] next_rextcb 0x18524d4[VS]  timeused 0xf
utcb 0x140400 taskname WorkLoop tid 0x44001  priority 0xed rex_tcb 0x2219c0c pre_rextcb 0x1818c5c[TIMER] next_rextcb 0x17eb8f0[SND]  timeused 0x0
utcb 0x140500 taskname IST0 tid 0x48001  priority 0xf6 rex_tcb 0x18ef3d4 pre_rextcb 0x18ef608[IST1] next_rextcb 0x232eac4[DPC Task]  timeused 0xeff
utcb 0x140600 taskname IST1 tid 0x4c001  priority 0xf7 rex_tcb 0x18ef608 pre_rextcb 0x18ef83c[IST2] next_rextcb 0x18ef3d4[IST0]  timeused 0x11f1
utcb 0x140700 taskname IST2 tid 0x50001  priority 0xf8 rex_tcb 0x18ef83c pre_rextcb 0x18efa70[IST3] next_rextcb 0x18ef608[IST1]  timeused 0x0
utcb 0x140800 taskname IST3 tid 0x54001  priority 0xf9 rex_tcb 0x18efa70 pre_rextcb 0x18efca4[IST4] next_rextcb 0x18ef83c[IST2]  timeused 0x0
utcb 0x140900 taskname IST4 tid 0x58001  priority 0xfa rex_tcb 0x18efca4 pre_rextcb 0x18efed8[IST5] next_rextcb 0x18efa70[IST3]  timeused 0x0
utcb 0x140a00 taskname IST5 tid 0x5c001  priority 0xfb rex_tcb 0x18efed8 pre_rextcb 0x18f010c[IST6] next_rextcb 0x18efca4[IST4]  timeused 0x0
utcb 0x140b00 taskname IST6 tid 0x60001  priority 0xfc rex_tcb 0x18f010c pre_rextcb 0x18f0340[IST7] next_rextcb 0x18efed8[IST5]  timeused 0x0
utcb 0x140c00 taskname IST7 tid 0x64001  priority 0xfd rex_tcb 0x18f0340 pre_rextcb 0x18f0574[FIQ] next_rextcb 0x18f010c[IST6]  timeused 0x0
utcb 0x140d00 taskname FIQ tid 0x68001  priority 0xfe rex_tcb 0x18f0574 pre_rextcb 0x232ce1c[none] next_rextcb 0x18f0340[IST7]  timeused 0x0
utcb 0x140e00 taskname TIMER tid 0x6c001  priority 0xee rex_tcb 0x1818c5c pre_rextcb 0x17e8020[DOG ] next_rextcb 0x2219c0c[WorkLoop]  timeused 0x71b
utcb 0x140f00 taskname SLEEP tid 0x70001  priority 0x2 rex_tcb 0x181b0c4 pre_rextcb 0x18b92ec[GNSS_FS] next_rextcb 0x232e090[none]  timeused 0x30
utcb 0x141000 taskname DOG  tid 0x74001  priority 0xf0 rex_tcb 0x17e8020 pre_rextcb 0x232eac4[DPC Task] next_rextcb 0x1818c5c[TIMER]  timeused 0xba
utcb 0x141100 taskname QDSP tid 0x78001  priority 0xde rex_tcb 0x17e9c88 pre_rextcb 0x18696dc[GSM L1] next_rextcb 0x18927e8[GNSS_PGI]  timeused 0x94b5
utcb 0x141200 taskname VOC tid 0x7c001  priority 0x84 rex_tcb 0x17ea6bc pre_rextcb 0x18cddf0[LOC_MIDDLEW] next_rextcb 0x181dd2c[ECALL_IVS]  timeused 0x165
utcb 0x141300 taskname SND tid 0x80001  priority 0xe9 rex_tcb 0x17eb8f0 pre_rextcb 0x2219c0c[WorkLoop] next_rextcb 0x188e5b4[GNSS_CC]  timeused 0x135
utcb 0x141400 taskname FS tid 0x84001  priority 0x59 rex_tcb 0x17edd58 pre_rextcb 0x18de728[IXFILE] next_rextcb 0x18e8350[FS Compat T]  timeused 0x70c
utcb 0x141500 taskname FS Compat T tid 0x88001  priority 0x50 rex_tcb 0x18e8350 pre_rextcb 0x17edd58[FS] next_rextcb 0x18d708c[PBM]  timeused 0x2
utcb 0x141600 taskname FS Hotplug  tid 0x8c001  priority 0xc rex_tcb 0x2534114 pre_rextcb 0x1848e38[SECCRYPTARM] next_rextcb 0x1898c50[GNSS_FFT]  timeused 0x50
utcb 0x141700 taskname IXFILE tid 0x90001  priority 0x5a rex_tcb 0x18de728 pre_rextcb 0x17ecb24[NV] next_rextcb 0x17edd58[FS]  timeused 0x1d
utcb 0x141800 taskname NV tid 0x94001  priority 0x5e rex_tcb 0x17ecb24 pre_rextcb 0x1819e90[TIME_IPC] next_rextcb 0x18de728[IXFILE]  timeused 0x956
utcb 0x141900 taskname CM tid 0x98001  priority 0x7a rex_tcb 0x17f31c0 pre_rextcb 0x17eef8c[DIAG] next_rextcb 0x18d6258[FTM]  timeused 0xe
utcb 0x141a00 taskname MMOC tid 0x9c001  priority 0xa2 rex_tcb 0x1854708 pre_rextcb 0x188894c[MN CNM MAIN] next_rextcb 0x188a380[GNSS_MC]  timeused 0x8
utcb 0x141b00 taskname UI tid 0xa0001  priority 0x67 rex_tcb 0x17f63f4 pre_rextcb 0x189ae84[GNSS_CD] next_rextcb 0x2217e7c[WorkLoop]  timeused 0x5
utcb 0x141c00 taskname CNV tid 0xa4001  priority 0x43 rex_tcb 0x18d92c0 pre_rextcb 0x182c830[DS_SIG] next_rextcb 0x182ea64[PS_RM]  timeused 0x8
utcb 0x141d00 taskname DIAG tid 0xa8001  priority 0x7c rex_tcb 0x17eef8c pre_rextcb 0x181baf8[ECALL_APP] next_rextcb 0x17f31c0[CM]  timeused 0x50
utcb 0x141e00 taskname SEC tid 0xac001  priority 0x27 rex_tcb 0x183c79c pre_rextcb 0x184d06c[SECSSL] next_rextcb 0x187e2b0[cb]  timeused 0x1b
utcb 0x141f00 taskname SECRND tid 0xb0001  priority 0x2c rex_tcb 0x18409d0 pre_rextcb 0x18263c8[PS] next_rextcb 0x1844c04[SFSsvc]  timeused 0x7
utcb 0x142000 taskname SECCRYPTARM tid 0xb4001  priority 0x1e rex_tcb 0x1848e38 pre_rextcb 0x187e2b0[cb] next_rextcb 0x2534114[FS Hotplug ]  timeused 0x21
utcb 0x142100 taskname SFSsvc tid 0xb8001  priority 0x2a rex_tcb 0x1844c04 pre_rextcb 0x18409d0[SECRND] next_rextcb 0x184d06c[SECSSL]  timeused 0x7
utcb 0x142200 taskname SECSSL tid 0xbc001  priority 0x28 rex_tcb 0x184d06c pre_rextcb 0x1844c04[SFSsvc] next_rextcb 0x183c79c[SEC]  timeused 0xb
utcb 0x142300 taskname UIM tid 0xc0001  priority 0x60 rex_tcb 0x18512a0 pre_rextcb 0x18524d4[VS] next_rextcb 0x1819e90[TIME_IPC]  timeused 0xeb0
utcb 0x142400 taskname GSDI tid 0xc4001  priority 0x4d rex_tcb 0x1858370 pre_rextcb 0x18d708c[PBM] next_rextcb 0x185a5a4[GSTK]  timeused 0x3c3
utcb 0x142500 taskname GSTK tid 0xc8001  priority 0x4c rex_tcb 0x185a5a4 pre_rextcb 0x1858370[GSDI] next_rextcb 0x181ff60[DS]  timeused 0x49b
utcb 0x142600 taskname VS tid 0xcc001  priority 0x61 rex_tcb 0x18524d4 pre_rextcb 0x2217e7c[WorkLoop] next_rextcb 0x18512a0[UIM]  timeused 0x25
```

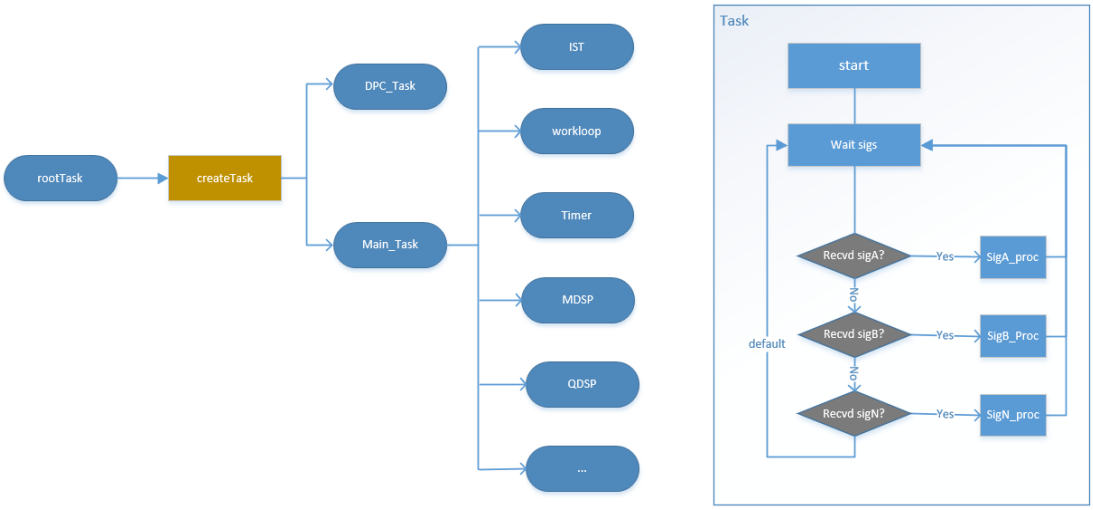
所有的任务通过优先级的高低，利用双向链表链接起来，如下图，FIQ 任务具有最高优先级。



而 sleep 任务具有最低运行优先级。



用户态的任务创建和运行流程如下图：



用户态任务运行特性：

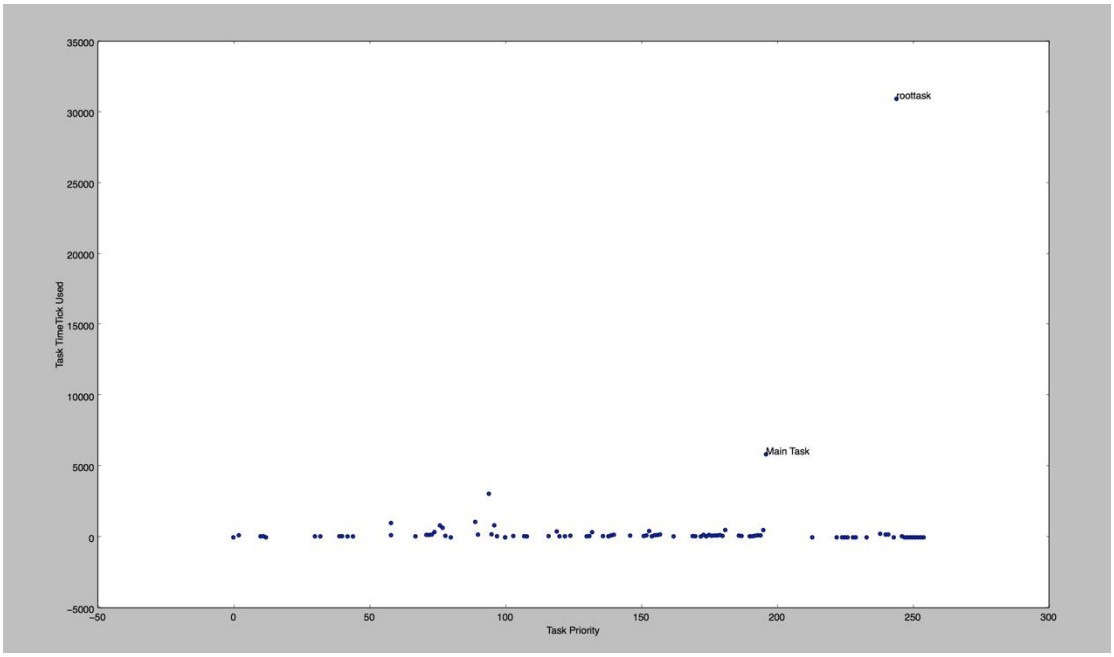
a. 每个被创建后的任务会被调度运行起来后，直至到等待信号的循环，阻塞接收消息，此时交出 cpu 执行权，切换执行任务。
b. 当某个任务接收到消息后，任务等待信号的循环返回，根据接受到信号去处理相应的例程，然后清除接受到的信号值，继续新一轮的信号等待。
c. 任务通过设置接受信号的掩码来设置多个信号处理例程，每个任务最多支持设置 32 个信号接受值。
d. 信号接受值和信号接受掩码会在 utcb 结构中设置。

任务调度机制：

a. 中断发生时，cpu 将调度到 IST 接管中断处理，因为 IST 的优先级比较高
b. 当任务等待消息阻塞时，任务主动交出 cpu 控制权
c. 应用任务都在等待时，rootTask 和 Main Task 接管 CPU，类似 idle loop
d. 当各个任务都有接受到消息时，根据任务的优先级和 cpu 使用时间进行调度

如下图系统初始化过程中的任务的切换过程以及 CPU 使用时间统计。

```
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x9d pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x1858204
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x128 pri 0xf4 to KTCB 0xf0307a10 0xf0306e00 [TIMER] CPU tick used 0x0 pri 0xee
switch KTCB 0xf0307a10 UTCB 0xf0306e00 [TIMER] CPU tick used 0x0 pri 0xee to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x1c009 0x18581ac
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x641 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0xec005 0x1858194
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x6d9 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x4c004 0x18581e4
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x752 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4
switch KTCB 0xf03028a0 UTCB 0xf0306200 [Main Task] CPU tick used 0x0 pri 0xc4 to KTCB 0xf0302cf0 0xf0304000 [roottask] CPU tick used 0x7a0 pri 0xf4
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x7a0 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x39 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x185822c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0xd72 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x39 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x185823c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0xdf3 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x39 pri 0xc4
switch KTCB 0xf03028a0 UTCB 0xf0306200 [Main Task] CPU tick used 0x39 pri 0xc4 to KTCB 0xf0302730 0xf0306300 [WorkLoop] CPU tick used 0x0 pri 0x64
switch KTCB 0xf0302730 UTCB 0xf0306300 [WorkLoop] CPU tick used 0x0 pri 0x64 to KTCB 0xf03078a0 0xf0306f00 [SLEEP] CPU tick used 0x0 pri 0x2
switch KTCB 0xf03078a0 UTCB 0xf0306f00 [SLEEP] CPU tick used 0x0 pri 0x2 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x18581ec
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0xe7c pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x18581fc
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0xf00 pri 0xf4 to KTCB 0xf0307730 0xf0311000 [DOG ] CPU tick used 0x0 pri 0xf0
switch KTCB 0xf0307730 UTCB 0xf0311000 [DOG ] CPU tick used 0x0 pri 0xf0 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x185822c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0xf9d pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x185823c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x1033 pri 0xf4 to KTCB 0xf03075c0 0xf0311100 [QDSP] CPU tick used 0x0 pri 0xde
switch KTCB 0xf03075c0 UTCB 0xf0311100 [QDSP] CPU tick used 0x0 pri 0xde to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x185822c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x10d2 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x185823c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x1159 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4
switch KTCB 0xf03028a0 UTCB 0xf0306200 [Main Task] CPU tick used 0x70 pri 0xc4 to KTCB 0xf0307450 0xf0311200 [VOC] CPU tick used 0x0 pri 0x84
switch KTCB 0xf0307450 UTCB 0xf0311200 [VOC] CPU tick used 0x0 pri 0x84 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x185822c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x11e8 pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x185823c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x126c pri 0xf4 to KTCB 0xf03072e0 0xf0311300 [SND] CPU tick used 0x0 pri 0xe9
switch KTCB 0xf03072e0 UTCB 0xf0311300 [SND] CPU tick used 0x0 pri 0xe9 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x3c005 0x185822c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x12ed pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4
    |-->IPC called in UTCB 0x140200 param 0x2c001 0x2c001 0x2c005 0x185823c
switch KTCB 0xf0302cf0 UTCB 0xf0304000 [roottask] CPU tick used 0x137a pri 0xf4 to KTCB 0xf03028a0 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4
switch KTCB 0xf03028a0 UTCB 0xf0306200 [Main Task] CPU tick used 0xb2 pri 0xc4 to KTCB 0xf0307450 0xf0311200 [VOC] CPU tick used 0x1f7 pri 0x84
```



我们可以看到，在系统初始化过程中，各个任务的初始化过程，cpu 使用时间都差不太多，因为初始化完了都处于阻塞状态了，只有 rootTask 和 Main Task 占有大量的 CPU 时间，因为 rootTask 需要负责大量的 KTCB 切换的通知操作，而且 Main Task 主动初始化那些应用任务。

IPC 任务间通信

IPC 通信是多任务协作通知和同步数据，非常重要的系统机制，在实时操作系统中应用广泛，对于无线通信复杂的状态机制以及低延时同步处理，IPC 通信起到了至关重要的作用。

从上图我们可知每个运行的任务都有独立运行环境，有自己的堆栈空间，当不同任务之间进行数据交换和同步的时候，这时候就需要用到 IPC 机制了，我们把用户任务的 rex_tcb 结构作为任务的唯一标示，用它与之不同的任务进行通信，这里用到了很重要的信号通知和等待信号通知的机制，从上面我们可知每个任务可以定义最多 32 个信号量来区分接收到的不同信号，然后根据接受到的不同信号进行相应的处理。

例如 A 任务需要告之 B 任务，是时候处理 B 任务里面的某个分支逻辑时，A 只需要设置 B 任务 rex_tcb 结构里面相应的信号值即可，当 B 任务被调度起来后的接受信号等待函数会返回取出 A 设置的信号值，然后 B 任务作相应的处理，该 IPC 通知机制在基带系统里面应用广泛，后续我也会提到。

f. 中断管理

基带系统在系统初始化过程中会初始化中断控制器，注册相应的中断服务例程，设置中断优先级，并且生效中断响应，在高通的 MDM6600 基带系统中设置了 8 个响应 IRQ 的 IST 任务，和 1 个响应 FIQ 的 IST 任务，优先级依次提升，FIQ 的 IST 任务具有最高的优先级别，因为在中断处理过程，可能会有更高优先级的中断产生，这时需要有高优先级的 IST 来接管响应来提升中断响应的实时性，由于中断是由硬件产生，而 IST 在应用

态，所以中断处理过程如下。

1．硬件中断产生 （物理层）
2．判断是否是 generic irq 还是 fiq （物理层）
3．进入到 irq exception 或者 fiq exception 向量表 （内核）
4．投递到相应中断处理分发器 （内核）
5．查询 IRQ 和 FIQ 的内核 KTCB 状态是否空闲 （内核）
6．通过 KTCB 结构找到相应的 IST 任务 （内核）
7．相应的 IST 接管中断，锁定该 IST，并查询中断号对应的 ISR （应用层）
8．执行 ISR 后，清除中断状态，解锁 IST，等待新的中断响应 （应用层）

五、结语

本文章的目的主要是为了对高通的基带系统有一个体系化的了解，操作系统作为承载业务系统的基础设施，了解其运行原理对于研究上层业务会有很大的帮助，由于高通的的基带系统非常封闭，研究需要大量的逆向工程的工作，记录了大量的笔记，无法一一整理发出，所以也有可能会有一些遗漏和不足，如果有熟悉的同学，也希望能够指出有错误的地方，便于改正，接下来系列的研究文章将针对高通基带对于 3GPP 定义的 GSM/UMTS/LTE，以及 5G 的实现上，并且在研究过程中，也会对比三星和华为在基带方面的实现和安全问题，并且挖掘其安全攻击面。

阿里安全　　谢君