# 蛋蛋读UFS之一：UFS简介

我们知道，我们电脑由三大件组成：CPU，内存和硬盘。CPU用以计算和控制，内存用以临时存储程序运行时所需的数据（掉电数据丢失），而硬盘用以长久保存数据（掉电数据不丢失）。

我们每天使用的手机，其本质是一个移动的小型计算机，同样由三大件组成：CPU，内存和存储设备。其中的存储设备相当于电脑的硬盘，用以长久保存手机上的数据，比如视频、照片、音乐、系统等数据。

电脑的硬盘有机械硬盘（HDD）和固态硬盘（SSD），前者是机械存储设备，存储介质是磁盘；而后者是电子存储设备，存储介质是闪存。我们不可能在小小的手机中塞入一个机械设备，所以手机上的存储设备只能是电子存储设备，存储介质也都是闪存。

现在是人手一个手机的时代，手机成了人们身体的一部分，一天不带手机，就感觉缺少了什么，吃嘛嘛不香。

IMG_256IMG_256

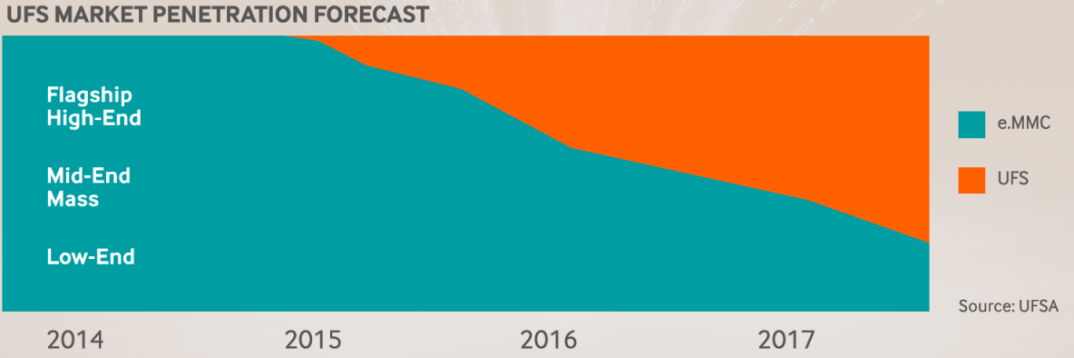
因此，人们对手机的要求也越来越高: 速度要快，容量要大，流畅不卡顿…

为了让手机更快，手机厂商使用更快、更多核的CPU，加大系统内存（4GB不够用6GB，6GB不够用8GB），使用更快的存储设备。无论是电脑还是手机，三驾马车（CPU，内存和存储设备）中，跑得最慢的就是存储设备了。CPU和内存的快步向前，促使最慢的存储设备也需要努力跟上，不然再快的CPU和再大容量的内存，你的手机用起来还是让你觉得不爽。

近年来，由于闪存技术的应用和发展，无论是电脑上的硬盘，还是手机中的存储设备，都在变得越来越快。

电脑上，从HDD 到SSD，从SATA SSD到PCIe SSD，硬盘是越来越快；

手机上，从SD卡，到eMMC卡，再到UFS卡，存储卡的速度也是越来越快。现在一般手机配的是eMMC，旗舰高端手机配的是UFS。



我们这个系列的主角UFS已登场。为什么UFS是主角？为什么我要带大家去了解UFS? 因为，UFS将是未来一段时间内手机存储的主流，我们有必要去了解UFS以及其相关的技术。

那么，什么是UFS？Universal Flash Storage，通用闪存存储。它有两个意思，一是指手机存储接口协议，类似SATA，PCIe/NVMe；二是使用该协议的存储设备。后面文章出现UFS，读者请根据上下文理解。

为什么说UFS是手机存储的未来？无他，快也！

大家感受一下：

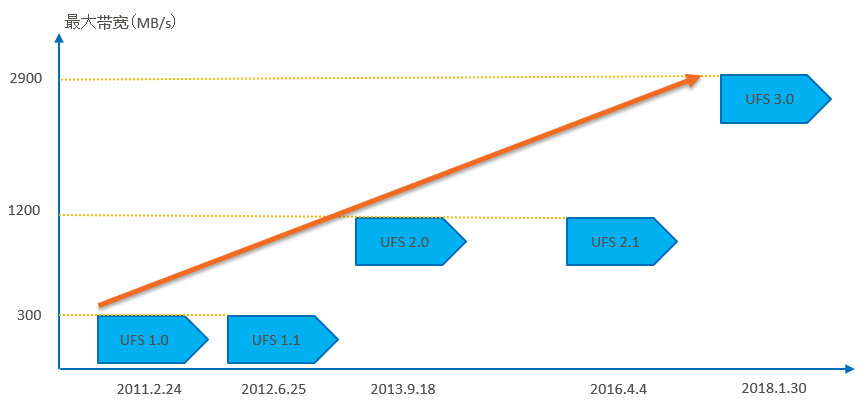
|  |  |  |
| --- | --- | --- |
| **UFS版本** | **2.0/2.1** | **3.0** |
| 单通道最大带宽（Mbps） | 5830.4 | 11660.8 |
| 最大通道数 | 2 | 2 |
| 最大有效带宽（MB/s） | 1081 | 2163 |

**注**：最大有效带宽去除了协议开销和8/10编码开销，二进制速率。

UFS最新标准是UFS3.0，于2018年1月30日发布。它最大带宽可以达到2163MB/s！4倍SATA3.0的速度 （600MB/s），超过PCIe3.0x2的速度（2GB/s单向速度）。

不过，目前市面上的UFS产品还是UFS2.0/2.1，其最大带宽1081MB/s，也是秒杀一般的SSD。

UFS协议是JEDEC（[www.jedec.org](https://www.jedec.org/)）组织制定的，三星、海力士、东芝等公司力捧。下面是UFS协议的发展历程：



我们可以看到，UFS协议一直在大踏步的朝着更高更快的目标前进。

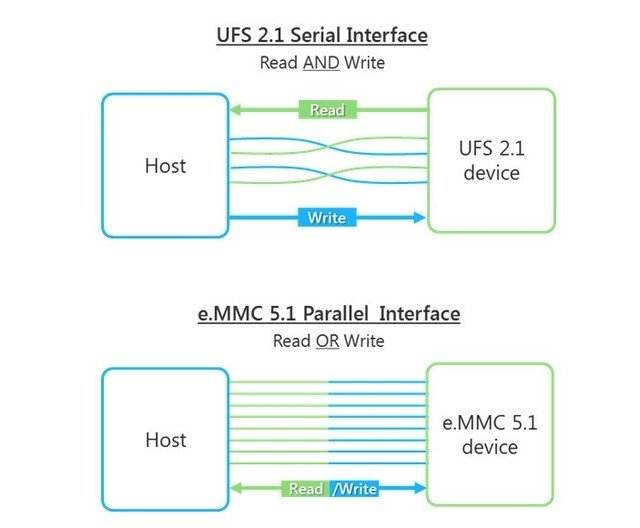
UFS为什么能那么快？

首先，它在数据信号传输上，使用的是差分串行传输。这是UFS快的基础。所有的高速传输总线，如SATA，PCIe，SAS，都是串行差分信号。串行，可以使用更快的时钟（时钟信息可以嵌在数据流中）；差分信号，即用两根信号线上的电平差表示0或者1。与单端信号传输相比，差分信号抗干扰能力强，能提供更宽的带宽（跑得更快）。打个比方，假设用两个信号线上电平差表示0和1，具体来讲，差值大于0，表示1，差值小于0，表示0。如果传输过程中存在干扰，两个线上加了近乎同样大小的干扰电平，两者相减，差值几乎不变，你大爷还是你大爷。但对单端信号传输来说，就很容易受干扰，比如0-1V表示0,1-3V表示1，一个本来是0.8V的电压，加入干扰，变成1.5V，相当于0变成1，数据就出错了，你大妈已经不是你大妈了。抗干扰能力强，因而可以用更快的速度进行数据传输，从而能提供更宽的带宽了。

UFS的前辈是eMMC，使用的是并行数据传输。并行最大的问题是速度上不去，因为一旦时钟上去，干扰就变大，信号完整性无法保证。

其次，UFS和PCIe一样，支持多通道数据传输，目前最多支持两个通道。多通道可以让UFS在成本、功耗和性能之间做取舍。

还有，它是全双工工作模式，就是读写可以并行。它的前辈eMMC是半双工，读写不能同时进行。



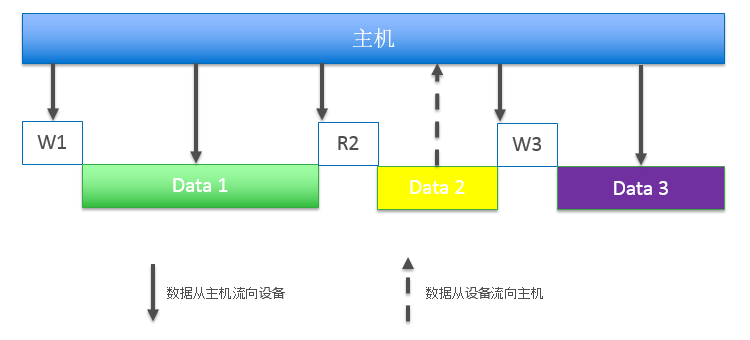
要让UFS速度快，这些基础设施是必须的。但要充分利用底层高速数据传输通道，还需要上层数据传输协议配合。就好比我们现在有一条又宽敞又平坦的高速公路，我们需要一辆高速的汽车行驶在上面。你如果让一辆拖拉机在上面跑，高速公路算是白修了。

UFS协议上层，怎样来充分发挥底层速度快的优势呢？

UFS支持命令队列，就是主机一下可以发很多个命令下去，然后UFS设备支持并行和乱序执行，谁先完成谁先返回状态。这种命令处理方式叫做异步命令处理。而它的前辈eMMC，是不支持命令队列的，命令一个一个执行，或者一包一包（每个包里面含有若干个命令）执行，前面命令没有执行完成，后面的命令是不能发下去的。这种命令处理方式叫做同步命令处理。

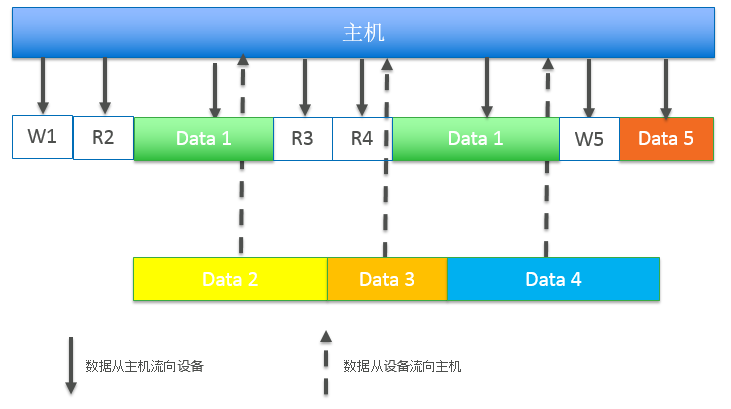
我们来比较一下“全双工+异步命令处理”和“半双工+同步命令处理”两者命令处理方式和命令执行效率。

* **半双工+同步**



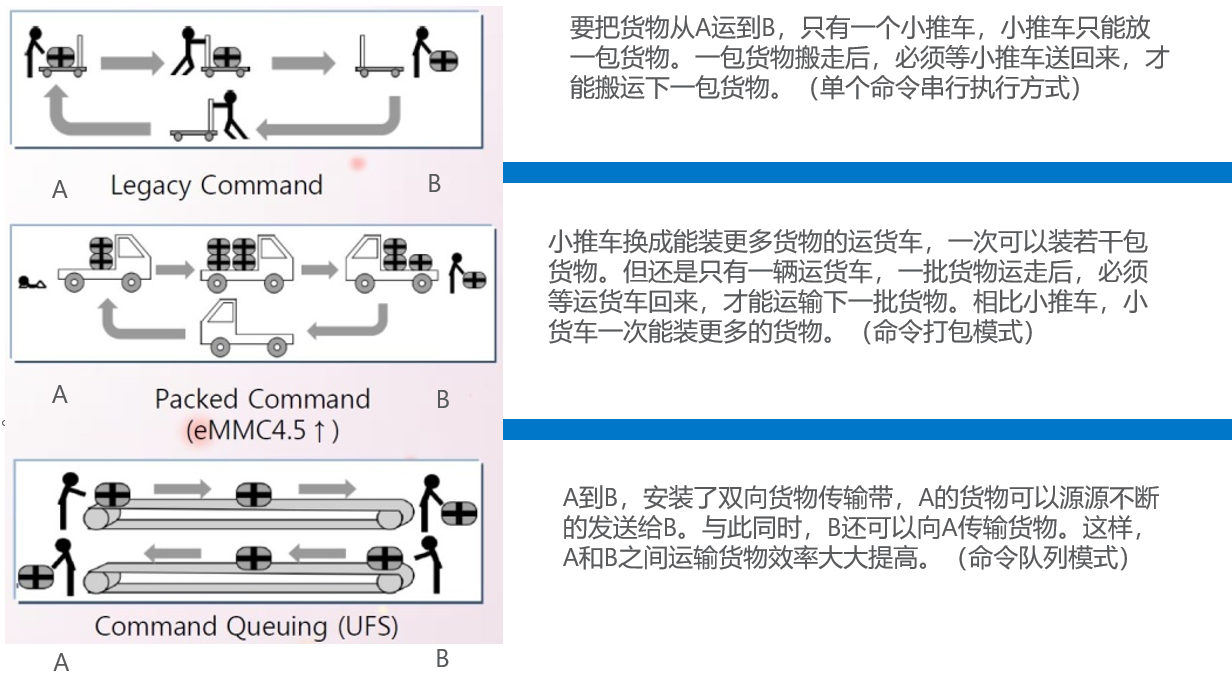
主机发了一个写命令W1给设备，然后主机把数据写到设备；由于是同步传输模式，命令处理是一个一个处理的，所以在发读命令R2之前，必须等前一个写命令W1完成；同样，在发送写命令W3之前，必须等R2命令完成。

* **全双工+异步**



由于支持命令队列，主机一下可以发若干个命令给设备，如上图，主机一下发了一个写命令W1和读命令R2给设备。设备可以并行处理这两个命令，由于协议支持全双工操作，主机传输写命令W1的数据给设备的同时，设备也可以把读命令R2的数据返回给主机。后面命令R3，R4，W5…的处理方式类似。

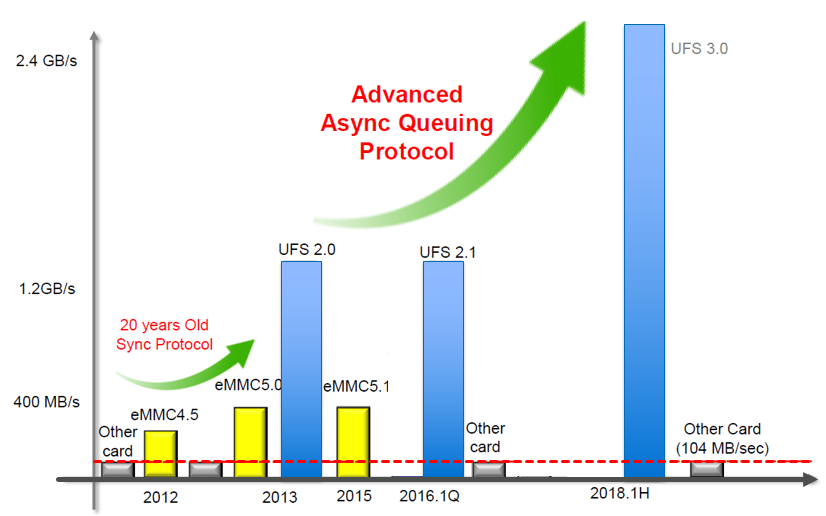
再形象一点，我们以搬运货物的例子来比较一下eMMC和UFS命令执行方式：



现在的手机，应用非常丰富，你要一边斗地主，一边听歌，还要聊微信，多线程操作。由于全双工和命令队列的存在，UFS处理命令的效率大大提高，给用户极好的体验。

前面我们拿UFS和eMMC做了几个对比，但我好像忘了说什么是eMMC。有人可能懵逼，什么是eMMC？

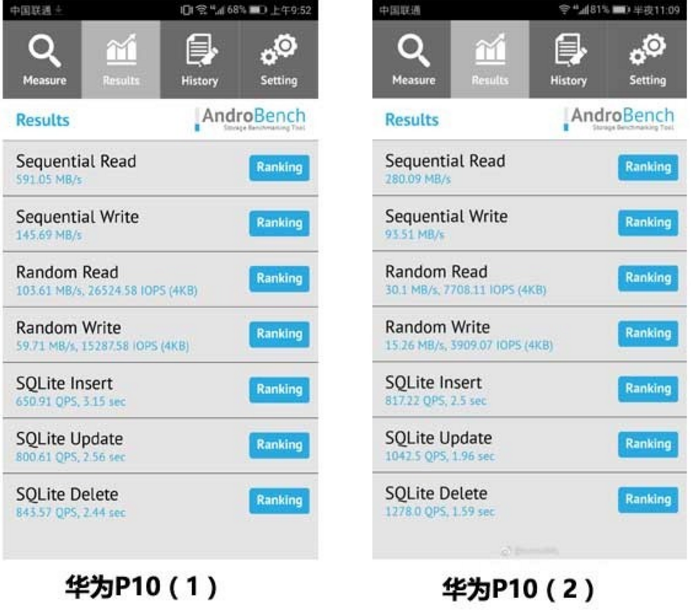
eMMC，Embedded Multi Media Card，和UFS一样，也是JEDEC制定的移动存储协议，它是UFS前一代协议标准。



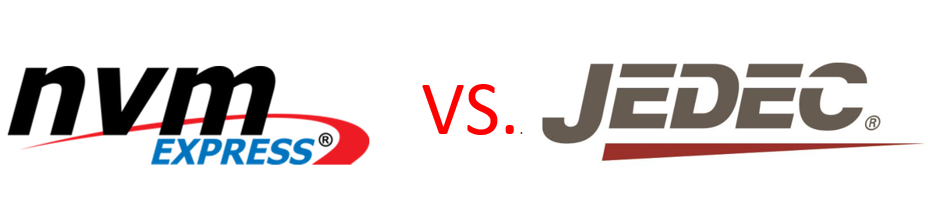
eMMC最新标准是2015年发布的eMMC5.1，最高速度是400MB/s。JEDEC已经有了UFS，不确定会不会再发布新的eMMC标准。毕竟，并行传输的eMMC由于受限于物理信号，速度想要有个质的飞跃是不太现实。

行文至此，让我不由的想起去年那事件。

同一款手机，有人UFS和eMMC混着卖，手机还卖一个价钱，真是无德！UFS和eMMC，速度差异那么大（见下图，[来自网络](http://mobile.it168.com/a2017/0428/3118/000003118367.shtml)），价格能一样吗？你系统再优化，能把eMMC顺序读写速度优化到401MB/s？别扯什么用户体验，用户都被耍猴了，体验还能好？水能载舟，亦能覆舟，不要太得意忘形。



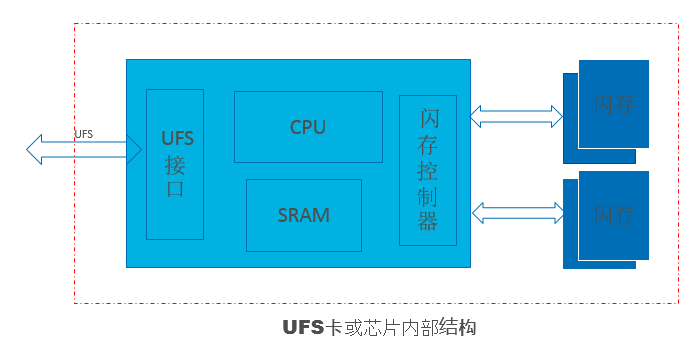
如果说eMMC是手机中的HDD，那么UFS就是手机中的SSD。UFS取代eMMC成为主流手机存储协议，这是毫无疑问的。不过，UFS一统天下的道路上还有一个拦路虎，那就是NVMe。有人说，NVMe不是SSD的协议标准吗？没错，不过，我要提醒大家的是，苹果现在手机中存储协议是NVMe而不是UFS。在短期，UFS和NVMe会分别在安卓和苹果手机中存在。长期来说，UFS和NVMe是二分天下，还是合二为一，我们只能拭目以待了。



在本章结束前，给大家看看UFS（BGA形式）的实物图：



大小如大拇指手指盖大小。麻雀虽小，五脏俱全。UFS存储芯片内部封装了UFS控制器和闪存阵列，和SSD结构很相似。不过和SSD相比，由于它的容量更小，因此闪存die比较少，闪存的通道数也少。另外，出于功耗和成本考虑，UFS芯片一般是不带DRAM的架构。



UFS芯片内部设计与实现不是我们本系列的重点，本系列后续文章将专注于UFS协议。

# 蛋蛋读UFS之二：UFS协议栈

任何一种接口或者协议，都是由一个完整的协议栈组成的。UFS也不例外。



UFS定义了一个完整的协议栈。从上到下，依次为应用层、传输层、数据链路层和物理层。UFS使用MIPI（Mobile Industry Processor Interface ，移动产业处理器接口）联盟的UniPro作为数据链路层和MIPI的M-PHY作为物理层，两者合起来称之为互连层（UFS InterConnect Layer）。与之相比，PCIe接口只定义了下三层（如下图），没有应用层。只有加上上层NVMe，才构成一个完整的SSD通讯协议。



目前UFS没有定义自己的命令（没有UFS Native Command Set），使用的命令是简化的SCSI命令（基于SBC和SPC），由INCITS T10组织定义的。关于SCSI相关协议，大家可以参看相应的spec。

四层中，只有传输层是JEDEC自己定义的。所以，UFS四层中有三层是别人的，命令层是T10的，数据链路层和物理层是MIPI的，传输层是JEDEC自己的。JEDEC移花接木的水平真是高。不由的想到一个广告：“我们不生产水，我们只是大自然的搬运工！”



UFS至今已经有五个版本，每层的版本也不尽相同。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **协议层** |  | **规范** | **组织** | **UFS 1.1** | **UFS 2.0** | **UFS 2.1** | **UFS 3.0** |
| 应用层 | SCSI | SPC-4 SBC-3  SAM-5 | T10 | Rev. 27  Rev. 24  Rev. 05 | Rev. 27  Rev. 24  Rev. 05 | Rev. 27  Rev. 24  Rev. 05 | Rev. 27  Rev. 24  Rev. 05 |
| 传输层 | UTP | UFS | JEDEC | UFS 1.1 | UFS 2.0 | UFS 2.1 | UFS 3.0 |
| 互联层（数据链路层+物理层） | UIC | UniPro | MIPI | 1.41 | 1.6 | 1.6 | 1.8 |
| M-PHY | MIPI | 2.00 | 3.0 | 3.0 | 4.1 |

我们依次来看看这几层。

* **UFS应用层**

应用层包括UFS命令集、设备管理器（Device Manager）和任务管理器（Task Manager）。应用层处于整个协议栈的最高层，所有的命令或者请求都来源于该层。它是最高统帅，所有的战术和策略都是它制定的，然后真正去冲锋陷阵的是将军和士兵（应用层下面的传输层和内联层）。

**命令集**

如前所述，目前UFS没有定义自己的命令，使用简化的SCSI命令。

其中包括一些SPC（SCSI Primary Commands）命令：

|  |
| --- |
| **命令名字** |
| INQUIRY |
| MODE SELECT (10) |
| MODE SENSE (10) |
| REPORT LUNS |
| READ BUFFER |
| TEST UNIT READY |
| WRITE BUFFER |
| SECURITY PROTOCOL IN |
| SECURITY PROTOCOL OUT |

和一些SBC（SCSI Block Commands）命令：

|  |
| --- |
| **命令名字** |
| FORMAT UNIT |
| PRE-FETCH (10) / PRE-FETCH (16)\* |
| READ (6) / READ (10) / READ (16)\* |
| READ CAPACITY (10) / READ CAPACITY (16) |
| REQUEST SENSE |
| SEND DIAGNOSTIC |
| UNMAP |
| WRITE (6) / WRITE (10) / WRITE (16)\* |
| START STOP UNIT |
| SYNCHRONIZE CACHE (10) / CACHE (16)\* |
| VERIFY (10) |
| 注: \* 表示可选命令 |

UFS除了定义基本的读写命令，也有trim命令（UNMAP），还有其它一些命令。我们不打算深入其中。

**设备管理器**

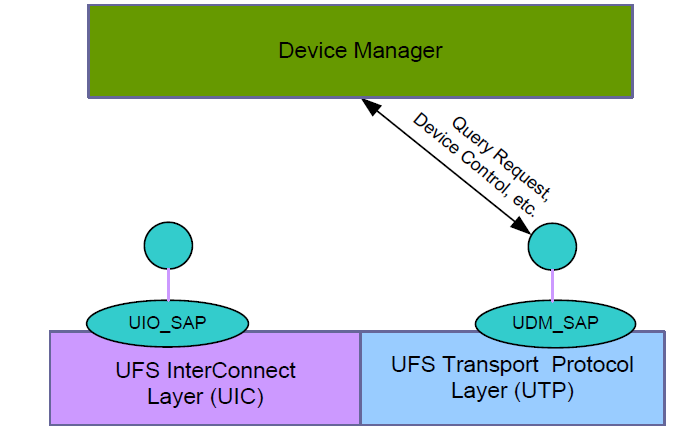
顾名思义，设备管理器用以管理UFS设备。

设备管理器有两个功能：一是处理设备级操作，二是管理设备级配置。

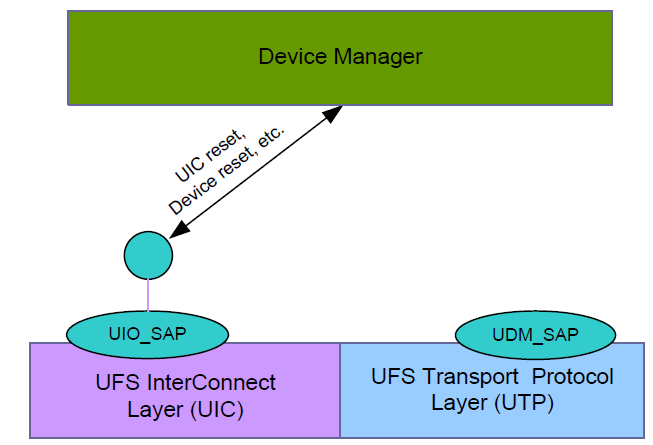
前者包括管理设备功耗、设置数据传输相关参数、使能/禁止设备后台操作（Background Operation）以及其它设备相关操作。

后者通过维护和存储一系列的描述符（Descriptor，后面有章节介绍），通过诸如Query请求修改或获取设备的配置信息。

从UFS层次架构图来看，设备管理器既可以通过下层的传输层为其服务（通过UDM\_SAP）：



设备管理器也可以绕过传输层（通过UIO\_SAP），直接管理与控制互联层：

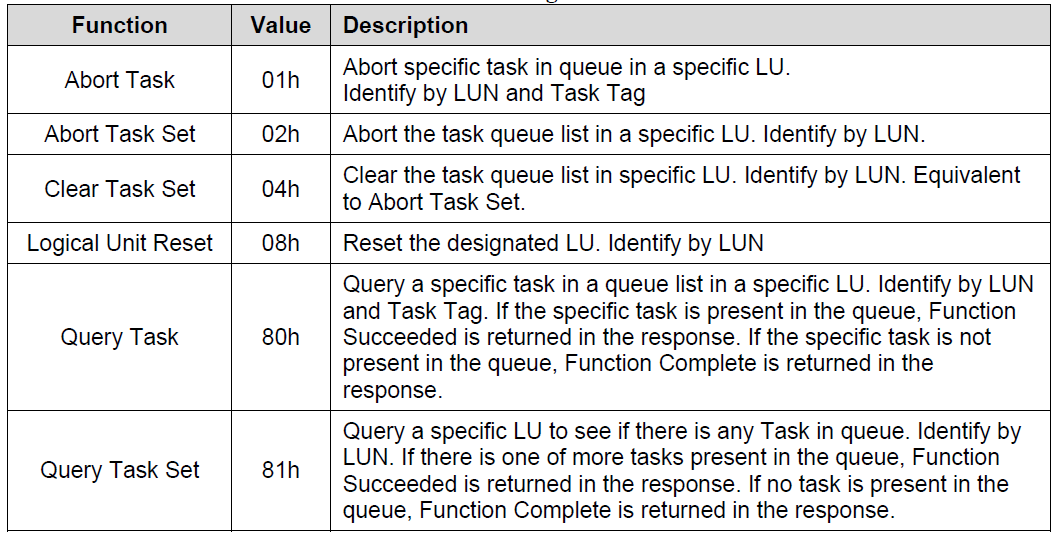


设备管理器可以通过互联层提供的接口（UIO\_SAP），使用一系列的原语（Primitive）直接控制操作互联层（UIC）。这些原语包括重启设备、重启互联层、让物理层进入和退出休眠模式（Hibernate）等原语。

总之，设备管理器既可以走常规渠道（通过传输层，以数据包UPIU的形式），也可以走快速通道（发送UIC能理解的命令，原语的形式）管理和操作设备。

**任务管理器**

任务管理器用以管理命令队列中的命令。比如任务管理器可以发Abort命令，终止之前发下去的命令。它也可以清空命令队列中的所有命令。具体如下：



当某个命令超时时，系统可能发Abort命令把这个命令终止掉。

* **UFS传输层**

传输层为它上面的应用层服务。当传输层收到应用层命令或者请求后，它会产生UPIU(UFS Protocol Information Unit)，把命令块或者请求封装成固定格式的数据结构，然后交由下层传到接收端的传输层。和命令相关的数据、状态，也有相应的UPIU数据包。UPIU是主机和设备进行信息交换的基本数据单元。

UPIU，和SATA中的FIS，PCIe中的TLP，是同一层次的东西，上层命令或者数据都是通过此类数据包封装起来，然后传输到接收端。

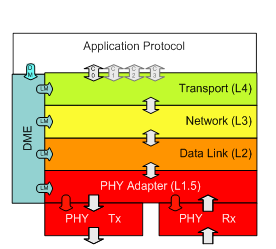
如果说应用层是统帅的话，传输层可以认为是将军了。

下一章节为专门介绍UPIU，这里就不细讲。

* **UFS互联层**

UFS互联层包括MIPI UniPro和M-PHY，分别充当UFS数据链路层和物理层的角色。数据链路层负责主机和设备的链接，物理层传输实实在在的物理信号。

UniPro其实不仅仅只定义了数据链路层，它本是也是一个比较完整的协议栈，如下图所示：



传输层（L4）支持多设备之间的双向连接，但UFS只支持CPort0；

网络层（L3）支持通过设备ID寻址多达128个设备，但由于UFS是点到点传输，所以无需网络层；

数据链路层（L2）支持流控、CRC生成和校验、重传机制等，UFS利用了UniPro的数据链路层为主机和设备之间通讯提供可靠的连接。

物理层（M-PHY）使用8/10编码、差分信号串行数据传输。数据传输分高低速模式，每种模式下又有几种不同的速度档。

关于MIPI UniPro和M-PHY，读者可以看相关的spec，这里不细讲。

本章对UFS协议栈做了简单介绍，下一章将会对传输层发起的UPIU进行详细的介绍。

# 蛋蛋读UFS之三：UFS数据包UPIU

UFS中流淌的数据包叫做UPIU（UFS Protocol Information Unit，UFS协议信息单元），它是固定格式的数据结构，用以传输应用层发来的命令或者请求，以及跟它们相关的数据或者状态信息。它就是SATA中的FIS，PCIe中的TLP。

我们看看UFS中命令或请求是怎么执行的。



UFS采用“客户-服务器”或者说主从的命令架构，UFS主机（Client，命令发起者，Initiator，他们都是一个意思）发送命令或者请求（Request）给UFS设备（服务器，Target），然后UFS设备执行命令并返回命令状态（Response）。

一个命令或者请求的执行包含下面几个阶段：

命令阶段

数据阶段（可选）

状态阶段

**命令阶段**：主机发起命令或请求给设备，这是“因”；

**数据阶段**：传输跟命令相关的数据，比如读写命令，都涉及到数据的传输；有些命令不涉及数据的传输，所以这个阶段并不是总是存在的，跟具体命令和请求相关。

**状态阶段**：设备执行完命令，必须给主机返回命令执行状态信息。这个是“果”，必不可少的。在PCIe中，有Posted和Non-posted的TLP。对前者，命令执行者无需返回命令执行状态给命令发起者，对后者，命令执行者必须返回状态给命令发起者。对UFS来说，它的命令总是non-posted，即**设备必须返回命令状态给主机**。

在命令执行过程中，无论是处在哪个阶段，UFS主机和设备间都是通过UPIU进行信息的交互。

* UFS主机通过命令或者请求UPIU发命令请求给设备；
* UFS主机或者设备通过UPIU传输数据；
* UFS设备通过UPIU返回命令状态信息给主机。

下面我们看看UFS当中都有哪些UPIU。

* **命令或者请求UPIU**

前一章看到，应用层包括UFS命令、设备管理器和任务管理器三个模块，传输层根据不同模块发来的命令或者请求，分别产生不同类型的UPIU。

**UFS命令模块**发送简化版本的SCSI命令，当传输层收到命令请求后，它会生成：**COMMAND UPIU，**把命令封装起来。

应用层通过**任务管理器**来管理任务队列，比如终止（Abort）和查询命令队列中的命令。当传输层收到来自任务管理器中的请求后，它会生成：**TASK MANAGEMENT REQUEST UPIU，**把请求封装起来。

UFS通过**设备管理器**来管理UFS设备，比如设置和查询UFS设备的配置（Configuration）。当传输层收到来自设备管理器发来的请求后，它会生成：**QUERY REQUEST UPIU，**把请求封装起来。

|  |  |  |  |
| --- | --- | --- | --- |
| **应用层模块** | **对应的UPIU** | **传输方向** | **作用** |
| SCSI命令 | **COMMAND UPIU** | 主机到设备 | 主机发送SCSI命令给设备 |
| 任务管理器 | **TASK MANAGEMENT REQUEST UPIU** | 主机到设备 | 主机管理命令队列中的命令，比如终止或者查询设备命令队列中的命令 |
| 设备管理器 | **QUERY REQUEST UPIU** | 主机到设备 | 主机通过设备管理器查询、配置、管理设备。 |

* **数据传输相关UPIU**

当主机发送了类似读命令给设备之后，设备需要返回数据给主机，设备通过**DATA IN UPIU**向主机传输数据。

当主机发送了类似写命令给设备之后，主机需要往设备写数据，主机通过**DATA OUT UPIU**向设备传输数据。

UFS的主机是个暖男，它在向设备写数据的时候，会考虑到设备这个时候能不能接收数据（因为设备可能这个时候没有足够的空间接收主机数据），它在向设备发了写命令之后，不会立刻把数据传输给设备，而是在那里等设备的通知。当设备准备好接收数据，以及接收多少数据，设备通过**READY TO TRANSFER UPIU （RTT）**告知主机。当主机接收到该RTT后，才开始按照RTT的信息传输数据。至于每次传输数据的多少，RTT中包含这信息，主机根据RTT进行传输。

所以，主机只有在收到设备的RTT，才能发**DATA OUT UPIU！**

注意，读命令无需这种机制。因为设备从闪存中获得数据后，是设备控制数据的传输。对主机来说，它在发读命令之前，已经准备好足够的空间用以接收数据，所以不存在主机没有空间接收数据的情况。

|  |  |  |
| --- | --- | --- |
| **数据传输相关的UPIU** | **传输方向** | **作用** |
| **DATA IN UPIU** | 设备到主机 | 设备传输数据给主机 |
| **DATA OUT UPIU** | 主机到设备 | 主机写数据到设备。主机只有收到RTT后，才能往设备写数据。 |
| **READY TO TRANSFER UPIU** | 设备到主机 | 同步。处理写命令时，设备告诉主机可以传数据以及传多少数据。 |

* **状态UPIU**

前面看到，主机有三种请求：SCSI命令，任务管理器发出的Task Management Request，以及设备管理器发出的Query request。针对不同的命令或者请求，设备在执行完相应的任务后，分别返回对应的状态UPIU给主机。

|  |  |  |
| --- | --- | --- |
| **设备响应UPIU** | **传输方向** | **作用** |
| **RESPONSE UPIU** | 设备到主机 | 设备返回命令执行状态 |
| **TASK MANAGEMENT RESPONSE UPIU** | 设备到主机 | 设备返回任务管理请求执行状态 |
| **QUERY RESPONSE UPIU** | 设备到主机 | 设备返回设备管理器的Query请求执行状态 |

* **其它UPIU**

除了以上常规的UPIU，还有其它一些UPIU作为他用。

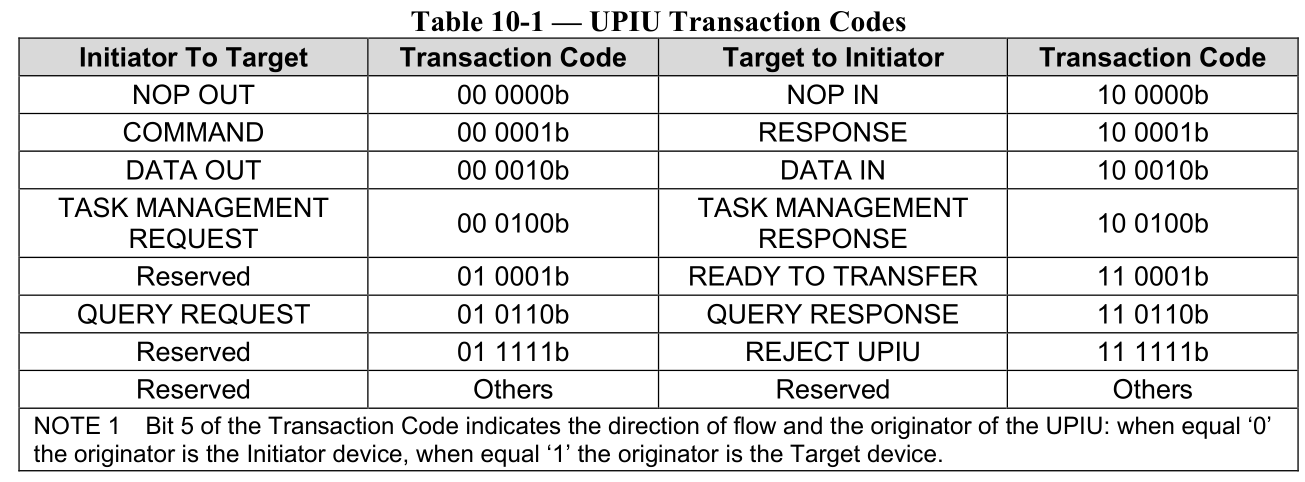
设备上电后，主机检测是否与之连接，会发**NOP OUT UPIU**给设备。我们平时想看看跟某个电脑或者网站能否连接上，会发一个ping命令。NOP OUT UPIU跟ping命令作用类似。

当设备收到NOP OUT UPIU后，会返回**NOP IN UPIU**。主机收到该UPIU后，确认与设备连接，然后可以进行后续操作。

最后一个UPIU就是**REJECT UPIU**。当设备收到一个无效的UPIU时，它会发REJECT UPIU拒绝无效的UPIU。

|  |  |  |
| --- | --- | --- |
| **辅助UPIU** | **传输方向** | **作用** |
| **NOP OUT UPIU** | 主机到设备 | 主机ping设备，查询主机是否与设备相连 |
| **NOP IN UPIU** | 设备到主机 | 设备响应NOP OUT UPIU。主机收到该UPIU后，确认主机与设备相连。 |
| **REJECT UPIU** | 设备到主机 | 设备收到无效的UPIU，发该UPIU给主机 |

* **UPIU汇总**

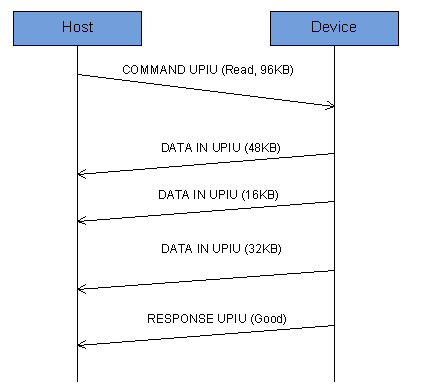


偷个懒，我就直接把UFS spec这张表贴这里。数了数，一共12个UPIU。经过我之前的解释，读者现在应该清楚每个UPIU的作用了。

**读写命令中UPIU交互例子**

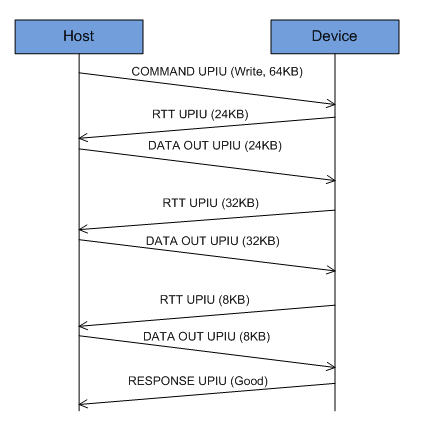
前面我们都是单个来看UPIU，现在我们以读写命令为例，看看他们是如何组合完成命令处理的。

首先是一个“主机往设备读取96KB数据”的例子。



首先，主机发送读96KB数据的命令给设备，然后设备执行命令，分了三批把数据返回给主机，最后返回命令执行状态给主机。

然后是一个“主机往设备写64KB数据”的例子。



主机发送写64KB数据的命令给设备，然后在那里等设备响应。很快，设备说，你可以传24KB数据下来了，于是主机写24KB数据给设备；接着，设备又来通知说可以继续传32KB数据，主机照做。最后，设备通知说可以把最后8KB数据也传过来，主机于是写最后8KB数据。最后，主机收到设备命令执行完成的响应。

我们看到，主机必须等收到RTT后才能启动数据传输！

# 蛋蛋读UFS之四：UPIU数据包格式

UPIU是命令、数据和状态信息传输的载体，是UFS协议栈的灵魂。UPIU是有固定格式的数据包，我们分析数据包格式，有助于我们更深的理解UPIU以及整个UFS协议。这一章我们看看UPIU数据包的格式。

每个UPIU都有一个12字节的Header，再加上跟每个UPIU相关的域。一个UPIU（包括Header）最小为32字节，最大为65600字节。

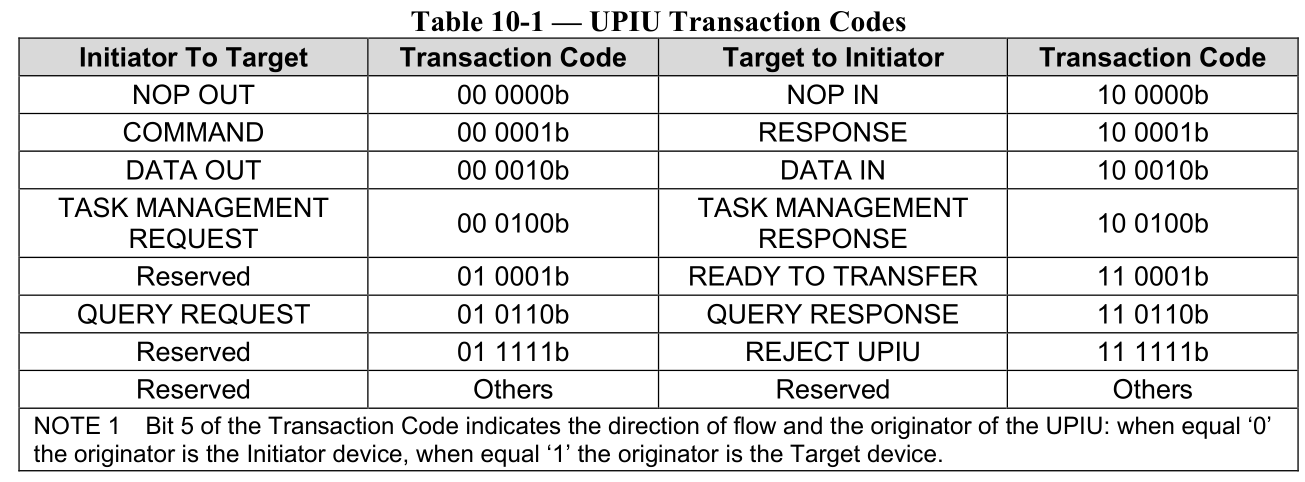


我们看通用的Header，具体如下：

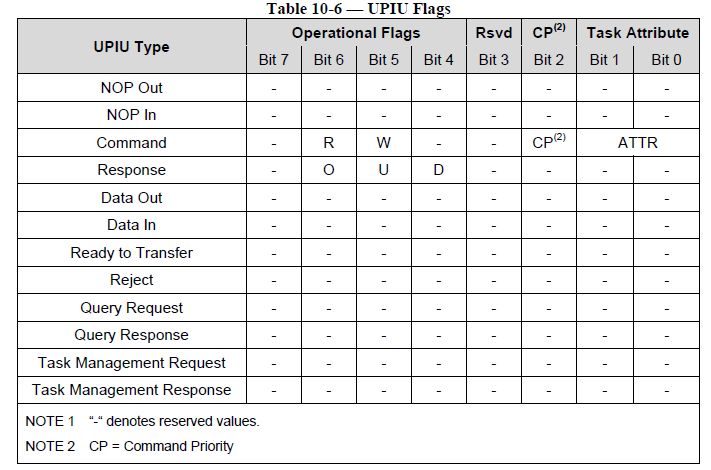


我们看看其中的一些域。

1. **Transaction Type**：就是指定该UPIU是前面12个UPIU中的哪一个,具体如下：



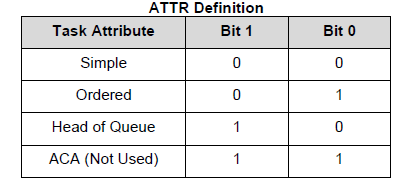
1. **Flags**：只对命令和其响应的UPIU有用，指定命令的属性。



**R**: 如果该比特置起来，说明该命令是读命令；

**W**: 如果该比特置起来，说明该命令是写命令；

**ATTR**: 命令属性域。UFS命令有simple ，ordered 和Head of Queue命令。



那么，这些命令有什么不一样呢。

**Simple command：**就是一般的命令，设备收到这样的命令无需特别处理，一般谁先到谁先执行。

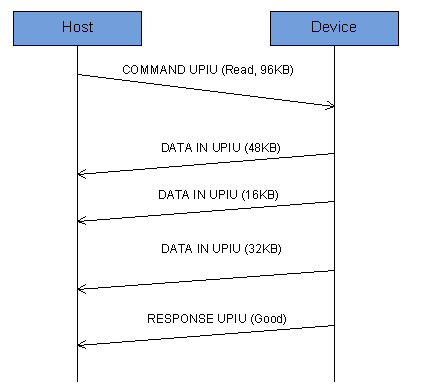
**Ordered command**：设备收到这样的命令，应该把该命令之前的命令都处理完，才能处理该命令。（明星出场，先清个场。）

**Head of Queue command**：设备收到该命令后，放到命令队列的头部，立刻执行。（又见插队，这个没有上过幼儿园吧，连基本的排队意识都没有。）

**CP**: 表示命令的优先级。1为高优先级，0为低优先级。注意，该比特只适合简单命令（simple command）。

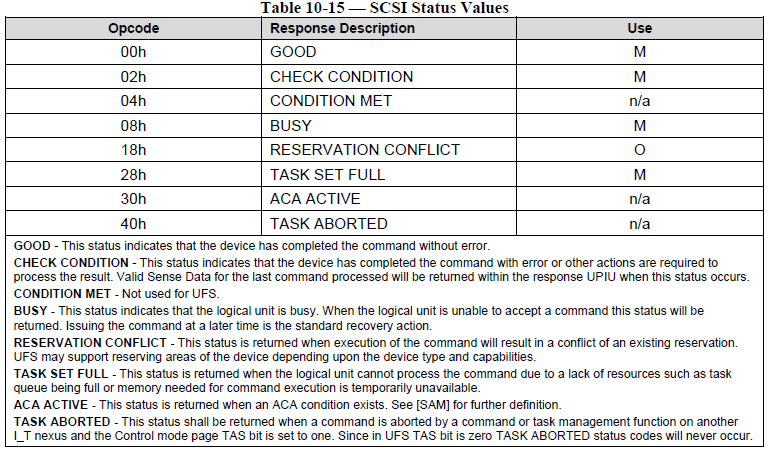
1. **LUN**: Logical Unit Number。UFS上层协议来自SCSI，它继承了LU的概念，即把存储物理空间划分成若干个逻辑空间，每个逻辑空间都是从LBA 0开始，用LUN标识。主机在发命令或者请求时，应该在命令中指定该命令是发给哪个LU。LUN用以寻址。UFS的LU和NVMe中的Namespace一个概念。
2. **Task Tag：**UFS支持命令队列，主机可以同时发送很多个命令给设备。为区分这些命令或者请求，主机需要为每个命令贴上标签Tag。然后跟这个命令或者请求相关的数据UPIU和状态UPIU，都具有跟这个命令UPIU一样的Tag。

举例：



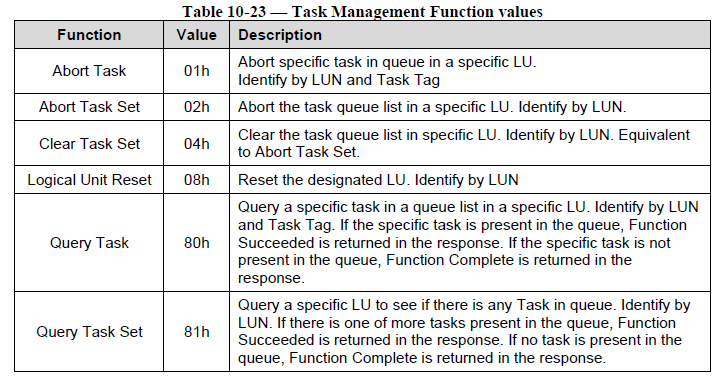
对这个读命令来说，COMMAND UPIU、所有的DATA IN UPIU和RESPONSE UPIU都具有同一个task tag。

1. **Command Type**：命令类型。UFS预期有三类命令：一是简化的SCSI命令，二是UFS自己原生的命令，三就是用户自定义命令。目前UFS的命令都是从别人家（SCSI）借来的，自己一个命令也没有制定。如用户无自定义命令，该域就是0（SCSI命令）。
2. **Initiator ID**： 主机的ID，手机系统中一般一个主机连接一个UFS设备，所以主机ID一般为0。
3. **Response**：设备告知主机命令或请求执行是否成功。
4. **Status：**设备返回命令执行状态。对SCSI命令的状态信息，UFS有如下状态：

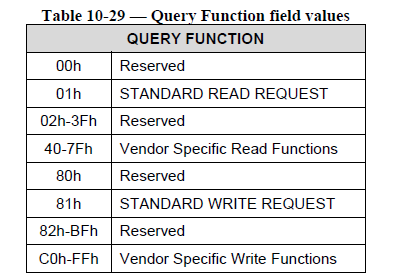


1. **Query Function, Task Manag. Function** ：指定具体Query和Task Management功能。

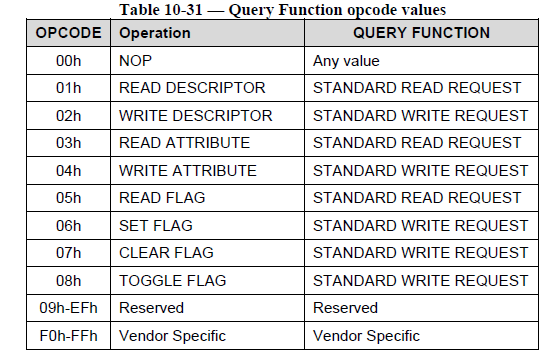
**任务管理器有如下功能（Function）**：



**设备管理器有如下功能：**



总的来说，就是读写设备属性（Attributes）、标识（flags）和描述符（descriptors）。



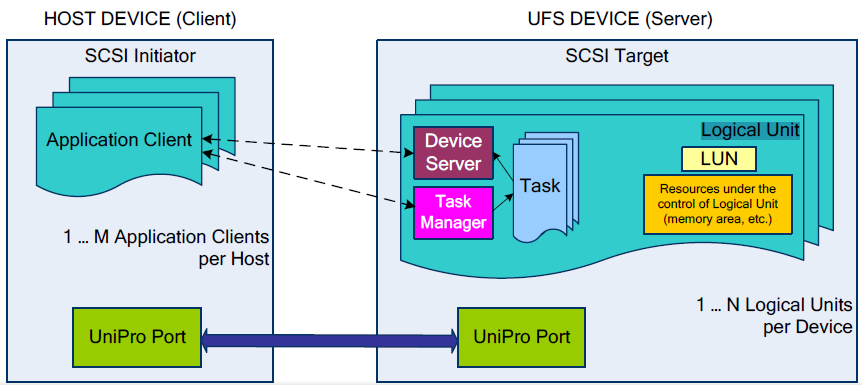
关于设备属性、标识和描述符，后面有专门章节讲述。

1. **Device Information**：设备信息。该域往往跟该命令或者请求无关，属于设备夹带私货。因为UFS主机和设备是主从关系，如果UFS主机没有向设备发命令或者请求，UFS设备是不能主动向主机报告设备状况的。如果UFS设备有特殊事件发生，它可以趁返回RESPONSE UPIU的时候把事件顺带告诉主机。所以该域只对RESPONSE UPIU有效。

以上是UPIU头的基本信息，这个是所有UPIU都具有的。除此之外，每个UPIU有它独有的其它信息，UFS spec上都有介绍，读者可以自行阅读。

# 蛋蛋读UFS之五：逻辑单元（LU）

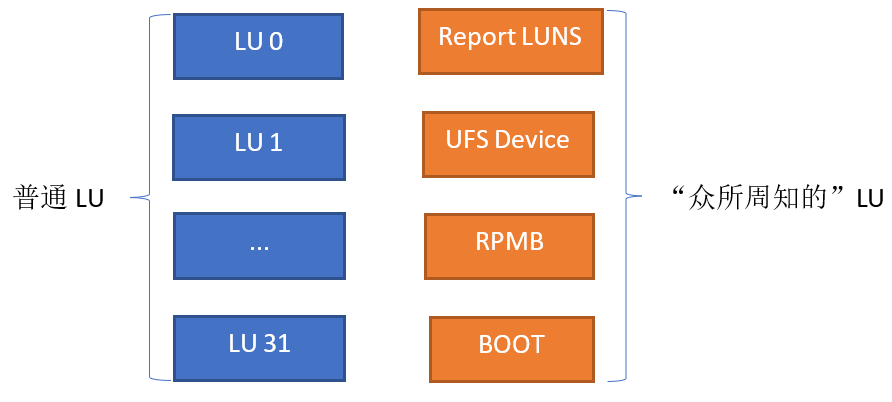
熟悉NVMe的朋友知道，NVMe里面有namespace的概念，就是把SSD物理空间划分成若干个逻辑地址空间。在UFS的世界里，它也有这个特性。UFS设备的物理存储空间可以有若干个独立的逻辑地址空间，我们把逻辑地址空间叫做LU，即Logical Unit，俗称“撸”。前面看到，在每个UPIU的Header中，有个LUN（Logical Unit Number）的域，就是标识该UPIU关联的命令或者请求的目标逻辑单元。每个LU的地址空间是独立的，主机在发命令或者请求给设备的时候，须通过LUN指定目标逻辑单元。



如上图所示，UFS设备有若干个LU，每个LU接收主机发过来的命令或者请求，这些命令或者请求可来自应用层的SCSI模块、设备管理器或者任务管理器。每个LU都是独立的，“独立”表现在下面几个方面：

* 逻辑地址空间是独立的，都是从LBA 0开始；
* 逻辑块大小可以不同，可以为4KB，8KB...；
* 可以有不同的安全属性，比如可以设置不同的写保护属性；
* 每个LU可以有自己的命令队列；
* 不同的LU可以存储不同的数据，比如有的LU存储系统启动代码，有的LU存储普通的应用数据，有的LU存储用户特殊数据...
* 。。。

UFS2.1中可以有最多32个普通LU和“四大名撸”（四个Well known LU，众所周知的LU）。

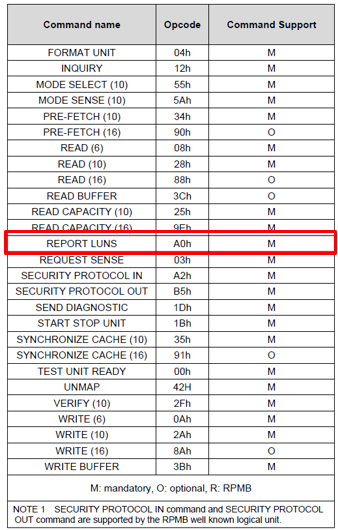


普通LU的逻辑块大小至少是4KB，但RPMB LU逻辑块大小为256B。至于什么是RPMB LU，后面再讲。

普通LU我觉得没有什么好讲的，就是分别用来存储用户数据的。我们主要来讲讲“四大名撸”。

* **Report LUNS LU**

Report LUNS主要用来代表设备向主机汇报设备LU清单。主机想知道设备LU的支持情况，就需要发命令或者请求给该LU。UFS其中有个命令“Report LUNS” （和该LU名字一样）用来访问Report LUNS。



* **UFS Device LU**

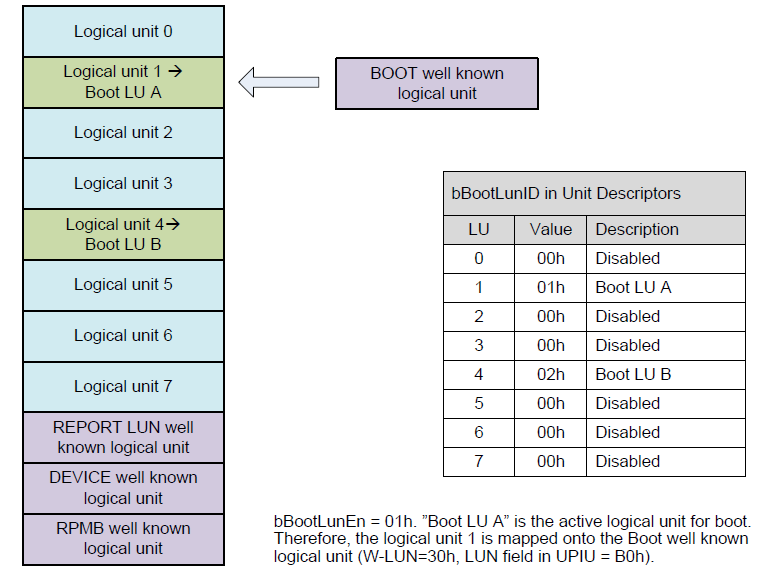
UFS设备的法人。当UFS主机不针对某个具体LU，而是对整个UFS设备发命令的时候，UFS Device LU就成为该命令接收的对象，比如格式化UFS设备（FORMAT UNIT命令）、切换UFS设备的功耗模式（START STOP UNIT命令）等等。

* **BOOT LU**

顾名思义，就是用来存储启动代码的LU。不过，BOOT LU本身是不存储启动代码的，它只是个虚拟的LU，启动代码物理上是存储在普通LU上的。

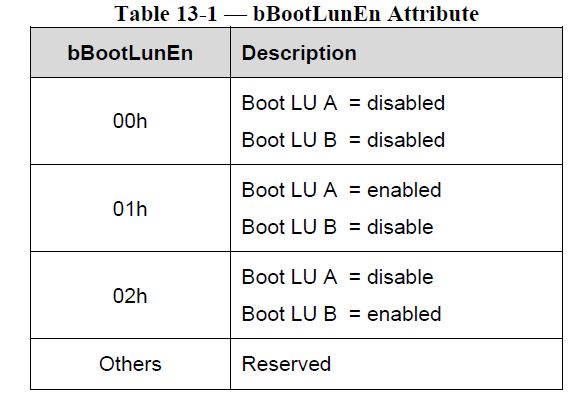
有两个Boot LU，LU A和LU B，可以用来存储不同启动代码（比如一个新，一个旧），但在启动过程中，只有一个是活跃的（Active）的。32个普通LU中的任意一个可以配成Boot LU A或者Boot LU B。

举例说明：



在上例中，LU 1 充当Boot LU A，LU 4 充当Boot LU B。由于有两份启动代码，分别保存在LU 1和LU 4，那启动的时候读取哪一份呢？

主机启动时，首先应该通过设备管理器，发送Query 请求给设备，获取一个叫做“bBootLunEn ”的属性，该属性标识当前活跃（Active）的Boot LU。



在上例中，bBootLunEn = 01，说明Boot LU A是当前活跃的Boot LU，因此主机会从LU 1上读取启动代码完成系统的启动。

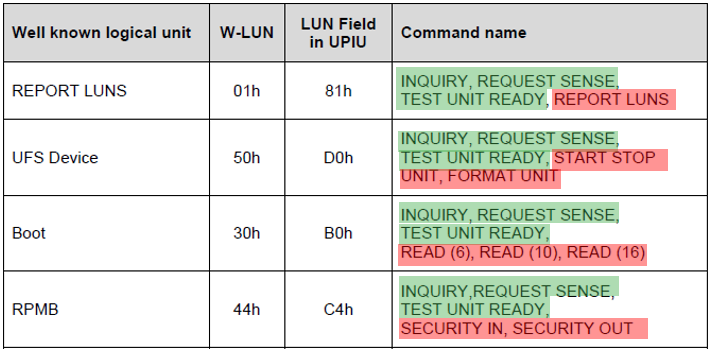
值得一提的是，Boot LU不是必须的。如果系统的启动代码不是存储在UFS设备上，那么Boot LU就不需要，因此bBootLunEn = 0。

* RPMB LU

在UFS里，有这么一个LU，主机往该LU写数据时，UFS设备会校验数据的合法性，只有特定的主机才能写入；同时，主机在读取数据时，也提供了校验机制，保证了主机读取到的数据是从该LU上读的数据，而不是攻击者伪造的数据。这个LU就是RPMB LU。

关于RPMB，后面有专门章节介绍，这里不多说。

“四大名撸” 每个LU分工明确，分别执行不同的任务。下面把 “四大名撸” 能接收的命令列一下：



他们能接收一些通用的命令（如上图绿色命令），还有只有该LU能执行的命令（如红色命令），具体命令可查看Spec。

# 蛋蛋读UFS之六：UFS设备初始化和启动

这一节讲讲UFS初始化。

初始化和启动包括三个阶段：部分初始化，加载启动代码（可选）和初始化完成。

* **部分初始化阶段**

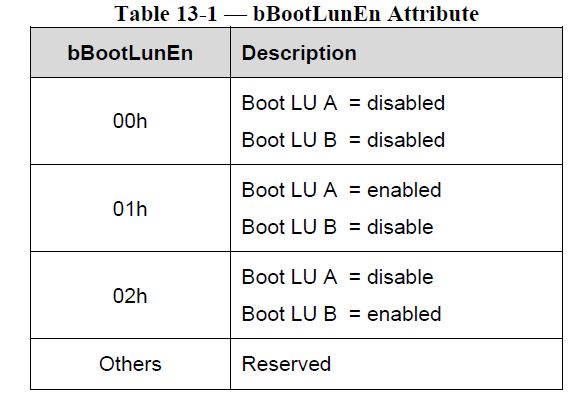
这个阶段开始于上电或者设备重启，它涉及到整个UFS栈的初始化。

这个初始化阶段完成后，整个物理层（M-PHY）和数据链路层（UniPro）应该被初始化好，传输层可以和主机交互Read命令和“TEST UNIT READY”命令（主机发该命令给设备，查询设备是否准备好），主机也可以通过设备管理器访问设备描述符（Device Descriptor），获取设备配置信息。

* **加载启动代码**

如果启动代码不是存储在UFS设备上，则没有这一阶段。主机怎么知道启动代码是不是存储在UFS设备上呢？

经过前一阶段的初始化，主机可以访问设备描述符，获得“bBootLunEn ”属性，读取该属性可以知道启动代码是否在UFS设备上，以及具体在哪个Boot LU 上面。



如果bBootLunEn = 01h或者02h，说明启动代码存储在UFS设备上。由于Boot LU是映射到普通的LU上的，要读取启动代码，还需要知道Boot LU和存储启动代码LU的映射。主机可以通过读取单元描述符（Unit Descriptor）知道，比如：



查找到具体存储代码的LU，主机就可以读取该LU获得启动代码。

* **初始化完成**

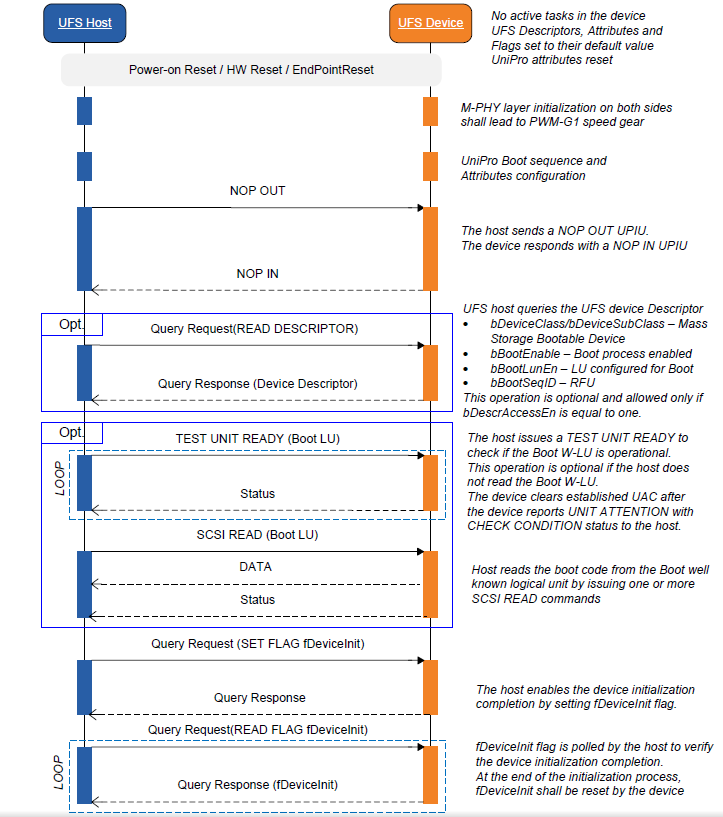
当主机完成前面两个阶段，主机会通过设备管理器，给设备设置fDeviceInit = 1，这是一个标志（flag），用以初始化。 主机设置了该标志后，然后就一直在那查询该标志的值。

与此同时，UFS设备的固件继续完成自己的初始化，当设备完成初始化，认为可以响应主机任何命令或者请求时，就清掉fDeviceInit，即fDeviceInit = 0。

当主机查询到fDeviceInit =0，就可以发任何UFS协议中的任何命令或请求给UFS设备。

至此，整个UFS初始化和启动完成。

下图具体展示了UFS初始化和启动过程（可选的，Opt）：



再解释一下这个启动过程：

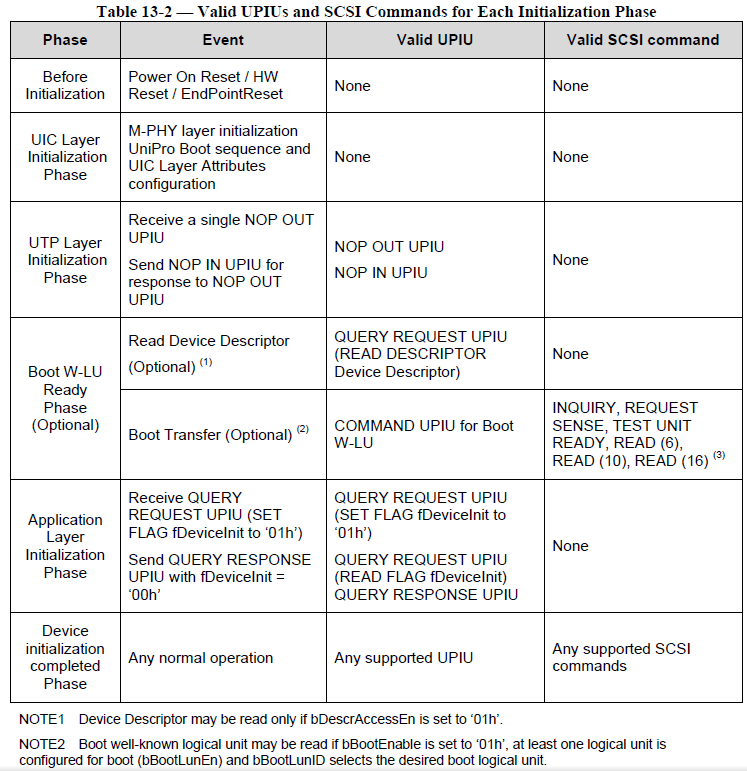
主机给设备上电或者重启设备，然后主机和设备端的物理层和数据链路层完成初始化，之后主机ping设备（通过NOP OUT UPIU），确认设备双方连接正常。设备收到NOP OUT UPIU，应该回NOP IN UPIU，表明双方连接没有问题。

加载启动代码可选（上图蓝色方框中的步骤）。主机读取UFS设备描述符，如果bDescrAccessEn = 0，设备描述符不可访问，那么，即使启动代码存储在UFS设备上，我们也无法在该阶段加载启动代码，因为诸如bBootEnable和bBootLunEn之类的信息无法获取，主机就无法知道存储代码存储在哪个LU上。因此，如果bDescrAccessEn = 0，加载启动代码阶段不应该放在这里，而是在设备彻底初始化好后。

加载启动代码阶段，主机通过读取设备描述符，获得启动代码在哪个LU上，然后发个试探性命令“TEST UNIT READY”给该LU，查看该LU是否准备好。如果Boot LU准备好，主机就通过发READ命令给设备，加载启动代码。

然后，主机设置fDeviceInit = 1，然后一直轮询该标志，一旦fDeviceInit 变成0，标志UFS设备初始化完成。

最后，再把设备初始化过程中，双方交互的内容做个总结：

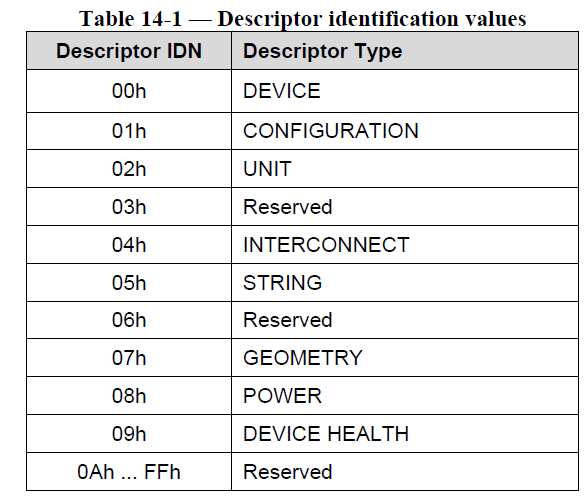


# 蛋蛋读UFS之七：描述符、标识和属性

UFS中也有吉祥三宝，那就是描述符（Descriptors）、标识（Flags）和属性（Attributes），主机通过这三宝，来控制与管理UFS设备。

* **描述符**

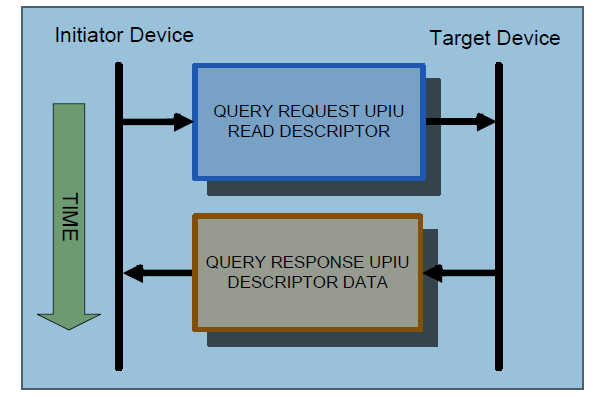
描述符是一块或者一页参数用以描述一个UFS设备，比如，UFS有整个UFS设备的描述符 （Device Descriptor），UFS设备的配置描述符（Configuration Descriptor），每一个LU还有其描述符（Unit Descriptor），等等。下面是UFS里面所有种类的描述符。



除了配置描述符和OEM\_ID字符串描述符，所有的描述符都是只读的，即UFS设备一旦出厂，主机是不能对它进行修改。

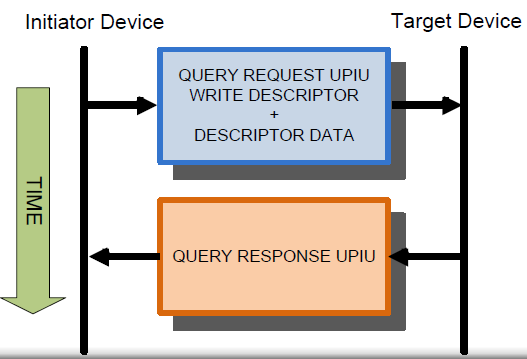
还记得Query Request UPIU吗？主机是通过设备管理器来访问这些描述符的。

**主机读描述符：**



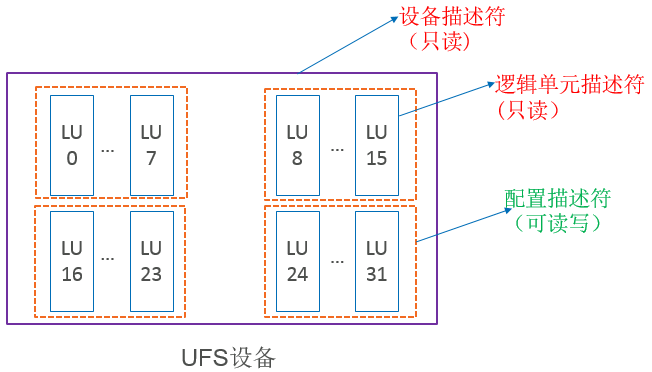
主机通过发Query Request UPIU给UFS设备，然后设备通过Query Response UPIU返回描述符数据。

**主机写描述符：**



主机如果想更改配置，可以写**配置描述符**。主机要写入的数据包含在Query Request UPIU中，一旦UFS设备更新完，返回Query Response UPIU。

我们简单过一下上面的这些描述符，更加详细的描述大家可以自行看UFS spec。



UFS设备只有一个，所以只有一个设备描述符；

一共有32个普通LU，每个LU有一个逻辑单元描述符，所以最多有32个逻辑单元描述符；

每8个LU有一个配置描述符，所以一共最多4个配置描述符。

1. **设备描述符**

设备描述符就是描述整个UFS设备属性的描述符，这些参数在UFS设备出厂时就由厂家设置好，主机对它只可读不可写。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **设备描述符** | | | | | |
| **偏移量** | **大小** | **名字** | **出厂默认设置** | **可配置？** | **说明** |
| 00h | 1 | bLength | 40h | 否 | 设备描述符大小。固定为64字节。 |
| 01h | 1 | bDescriptorIDN | 00h | 否 | 设备描述符类型编号 |
| … | … | … | … | … | … |
| 06h | 1 | bNumberLU | 00h | 是 | 普通LU的个数。 |
| 07h | 1 | bNumberWLU | 04h | 否 | Well Known LU的个数，UFS2.1中有“四大名撸”。 |
| 08h | 1 | bBootEnable | 00h | 是 | 标识是否支持从UFS设备启动。如果该位为0，表示不支持UFS设备启动，在初始化过程时，就没有加载启动代码的阶段。（见前一章） |
| 09h | 1 | bDescrAccessEn | 00h | 是 | 标识设备在部分初始化（Partial）后，主机能否访问设备描述符。 |
| … | … | … | … | … | … |
| 12h | 2 | wManufactureDate | 设备相关 | 否 | 设备出厂日期，比如August 2010 = 0810h |
| 14h | 1 | iManufactureName | 设备相关 | 否 | 设备生产厂家名称。这里只是一个索引，索引到包含厂家名称的字符串中。 |
| 15h | 1 | iProductName | 设备相关 | 否 | 设备产品名称。同样是个索引，索引到包含产品名称的字符串中。 |
| 16h | 1 | iSerialNumber | 设备相关 | 否 | 设备序列号。同样是个索引，索引到包含产品序列号的字符串中。 |
| ... | ... | ... | ... | ... | ... |

上面只是截取了一部分设备描述符数据结构内容，有关完整的设备描述符内容，大家可以看spec。

前面说了设备描述符是只读属性，为什么我们看到设备描述符里的有些项是可配置的呢？设备描述符的确只可能读不可写，但是主机通过写配置描述符（主机可写），然后这些项的变化就反映到设备描述符里来了。

UFS设备只有一个，所以一共只有一个设备描述符。

1. **逻辑单元（LU）描述符**

逻辑单元描述符用来描述某个具体LU的特性和能力，比如该LU逻辑块大小、该逻辑块是不是存有启动代码、该逻辑块内存类型等等。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **逻辑单元描述符** | | | | | |
| **偏移量** | **大小** | **名字** | **出厂默认设置** | **可配置？** | **说明** |
| 00h | 1 | bLength | 23h | 否 | 逻辑单元描述符大小。固定为35字节。 |
| 01h | 1 | bDescriptorIDN | 02h | 否 | 逻辑单元描述符类型编号 |
| ... | ... | ... | ... | ... | ... |
| 03h | 1 | bLUEnable | 00h | 是 | 逻辑单元使能。  1：该逻辑单元使能。  0：该逻辑单元禁止。 |
| 04h | 1 | bBootLunID | 00h | 是 | Boot LUN ID:  00h：该逻辑单元没有映射到任何Boot LU；  01h：该逻辑单元映射到Boot LU A;  02h：该逻辑单元映射到Boot LU B.  其它：保留。 |
| 05h | 1 | bLUWriteProtect | 00h | 是 | 标识该逻辑单元是否写保护。 |
| 06h | 1 | bLUQueueDepth | 设备相关 | 否 | 每个逻辑单元有个命令队列，该域指明命令队列深度。  0：没有自己的命令队列，和其它LU共享命令队列；  1-255：该LU自己的队列深度 |
| ... | ... | ... | ... | ... | ... |
| 08h | 1 | bMemoryType | 00h | 是 | 指定LU内存类型：  00h：存储普通数据  01h：存储系统代码  02h：挥发性存储类型  03-06h：分别对应增强型存储类型1-4。对应增强型数据，在闪存上，可能要存储在SLC上确保其存储可靠性。 |
| ... | ... | ... | ... | ... | ... |
| 0Ah | 1 | bLogicalBlockSize | 0Ch | 是 | 逻辑块大小，默认为4KB，用户可配置逻辑块大小。 |
| ... | ... | ... | ... | ... | ... |

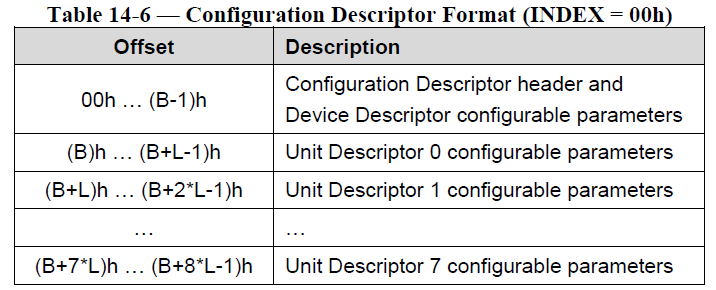
对于逻辑单元描述符中可配置的项，主机可以通过写配置描述符进行相应的更改。

1. **配置描述符**

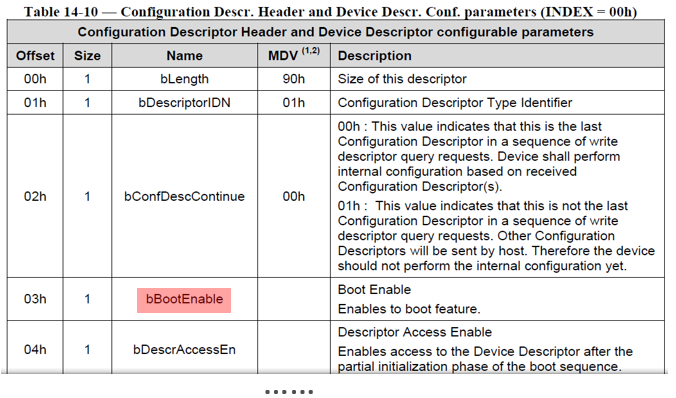
用户想对UFS做一些配置，或者使能/禁止一些feature， 可以通过写配置描述符达到目的。这些项的更新会反映到设备描述符或者逻辑单元描述符上。注意只有在属性（Attribute）bConfigDescrLock = 0时才可以写配置描述符，即配置描述符没有被锁住，配置描述符才能写，否则也是只读的。

UFS2.1有32个普通的LU，每8个LU有个配置描述符，所以一共有4个描述符。

拿出一个配置描述符来看看它的格式。

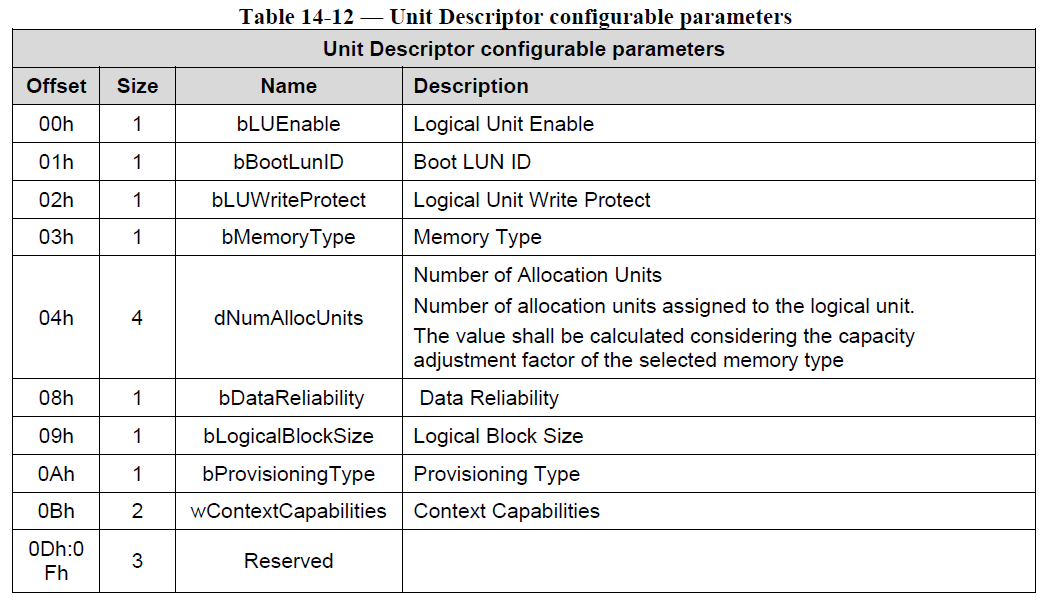


该配置描述符，包含了设备描述符可配置的参数和LU 0-7中可配置的参数。



举例来说，bBootEnable是设备描述符中一个可配置的项。它在出厂设置时bBootEnable = 0，用户在使用UFS设备时，把启动代码存放在UFS设备上，因此，为使能Boot feature, 用户须通过写配置描述符把该比特置起来：bBootEnable = 1。然后，主机在读取设备描述符的时候，会看到bBootEnable 变成了1。

除了配置UFS设备，配置描述符还可以对每一个LU进行配置。



比如，主机可以通过写配置描述符，使能某个LU，或者设置某个LU的逻辑块大小，以及其它和LU相关的配置。

UFS中还有其它的一些描述符，这里就不一一细看。

我们接下里看看另一宝：标志（flags）。

* **标志（flags）**

UFS中的标志其实就是一些开关，布尔型，非0即1，打开或者关闭。这些标志可以用来使能或者禁止UFS设备的一些功能、模式或者状态。

在UFS2.1协议中，一共有以下一些标志：

|  |  |
| --- | --- |
| **名字** | **说明** |
| fDeviceInit | 用以设备初始化。主机在加载启动代码后设置该标志，设备初始化完毕后清除该标志。 |
| fPermanentWPEn | 该标志使能所有配置为永久保护的逻辑单元的写保护，一旦设置为永久写保护后，该标志不能被清除。  0：永久写保护禁止；1：永久写保护使能。 |
| fPowerOnWPEn | 该标志使能所有配置为上电写保护的逻辑单元的写保护，一旦设置为永久写保护后，该标志不能被清除。  0：上电写保护禁止；1：上电写保护使能。 |
| fBackgroundOpsEn | 0：UFS设备被禁止后台操作（比如垃圾回收操作）；  1：UFS设备允许运行后台操作。 |
| fDeviceLifeSpanModeEn | 0：设备寿命模式（？可查看UFS spec）禁止；  1：设备寿命模式（？可查看UFS spec）使能； |
| fPurgeEnable | 0：Purge操作（把垃圾数据从闪存中擦除掉）被禁止；  1：Purge操作使能； |
| fPhyResourceRemoval | ☹ 看UFS spec |
| fBusyRTC | Busy Real Time Clock:  0: 设备没有执行跟RTC相关的内部操作；  1：设备正在执行跟RTC相关的内部操作。  当设备正在执行跟RTC相关的内部操作，建议这个时候主机不要打扰设备，不要发命令给设备。 |
| fPermanentlyDisableFwUpdate | 0: 设备固件可以被更新；  1：禁止更新设备固件。 |

主机也是通过设备管理器的Query Request UPIU来读取或者写标志。

* **属性（Attributes）**

如果说flags是布尔类型，那么属性就是C语言中的枚举类型。属性的值不仅仅是0或者1，它是一定数字范围的。属性可以表示设备的一些状态，比如当前设备后台任务的状态。有些属性，主机只可读，有些属性，主机可以写。

主机也是通过设备管理器的Query Request UPIU来读取或者写属性。

|  |  |
| --- | --- |
| **名字** | **说明** |
| bBootLunEn | 标识从哪个Boot Lun加载启动代码。  00h：禁止从UFS设备启动；  01h：从Boot LU A启动；  02h：从Boot LU B启动；  其它：保留。 |
| bCurrentPowerMode | 当前功耗模式。  00h：Idle;  10h: Pre-Active;  11h: Active;  20h: Pre-Sleep;  22h: Sleep;  30h: Pre-PowerDown;  33h: PowerDown;  其它：保留。 |
| bActiveCCLevel | Active ICC级别定义了在UFS设备在工作功耗（Active）模式下允许的最大电流消耗：  00h：最低Active ICC级别；  …  0Fh: 最高Active ICC级别；  其它：保留。 |
| bOutOfOrderDataEn | 乱序数据传输使能。  00h：禁止乱序数据传输；  01h：允许乱序数据传输，即一个命令当中的逻辑块数据，可以不按LBA的顺序传输。  其它：保留。 |
| bBackgroundOpStatus | UFS设备后台操作状态。  00h: 不需要后台操作；  01h：需要，但不急（Critical）；  02h：需要，对性能有影响；  03h：迫切需要做后台操作；  其它：保留。 |
| bPurgeStatus | UFS设备清除操作（把设备里无效的数据清除）状态。  00h：空闲，清除操作被禁止；  01h：UFS设备正在执行清除操作；  02h：UFS设备清除操作被host提前终止了；  03h：UFS设备成功完成清除操作；  04h：清除操作失败，因为逻辑单元的命令队列非空；  05h：清除操作一般性失败；  其它：保留 |
| bMaxDataInSize | 指定 DATA IN UPIU中最大传输数据大小（设备向主机传输数据）。该值以512字节为单位，不能超过bMaxInBufferSize。 |
| bMaxDataOutSize | 指定 DATA OUT UPIU中最大传输数据大小（主机向设备传输数据）。该值以512字节为单位，不能超过bMaxOutBufferSize。设备向主机发READY TO TRANSFER UPIU时，请求的数据量不能超过bMaxDataOutSize。 |
| dDynCapNeeded | Dynamic Capacity Needed.  UFS支持动态调整某个LU的容量。该值用以指定需要某个LU的物理空间减小多少容量。 |
| bRefClkFreq | 参考时钟频率。  0h: 19.2MHz;  1h: 26MHz;  2h: 38.4MHz;  3h: 52MHz;  其它：保留。 |
| bConfigDescrLock | 配置描述符锁。  0h：配置描述符没有锁住，主机可以读写配置描述符；  1h：配置描述符锁住了，主机不能写配置描述符（只读）；  其它：保留。 |
| bMaxNumOfRTT | 允许设备发的最多的Outstanding RTTs个数，不能超过bDeviceRTTCap。 |
| wExceptionEventControl | 异常事件控制（Exception Event Control）。  Bit 0: 用以使能/禁止动态容量调整事件；  Bit 1: 用以使能/禁止系统池事件；  Bit 2: 用以使能/禁止紧急后台事件；  Bit 3-15: 保留。 |
| wExceptionEventStatus | 异常事件状态（Exception Event Status）。  Bit 0: 标识是否需要动态调整容量，比如当闪存坏块出现很多时，动态的把用户容量缩小；  Bit 1: 用以标识系统池是否耗尽；  Bit 2: 用以标识是否有紧急后台操作；  Bit 3-15: 保留。 |
| ... | ... |

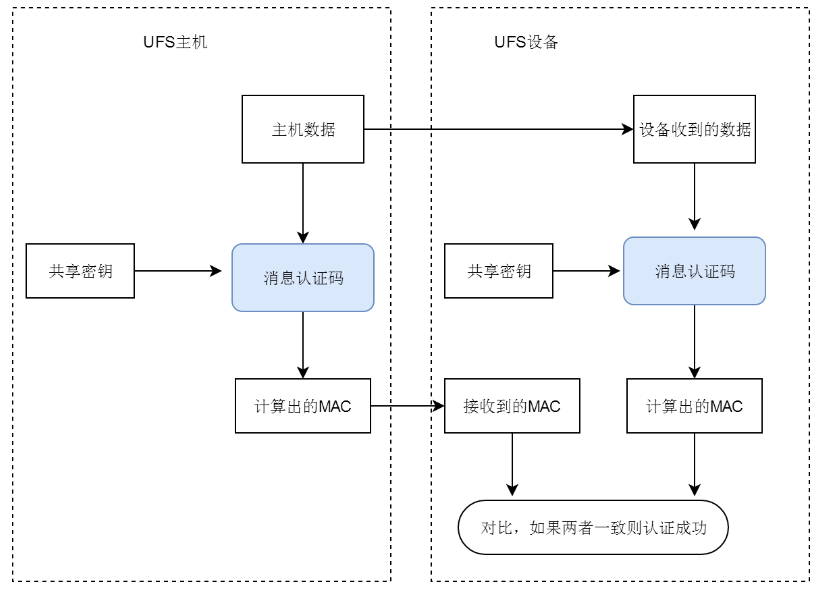
# 蛋蛋读UFS之八：RPMB

在UFS里，有这么一个LU，主机往该LU写数据时，UFS设备会校验数据的合法性，只有特定的主机才能写入；同时，主机在读取数据时，也提供了校验机制，保证了主机读取到的数据是从该LU上读的数据，而不是攻击者伪造的数据。这个LU就是RPMB（Replay Protected Memory Block）LU，四大“名撸”（四个Well Known LU）之一。

有些人家里有保险箱，用以存放他们认为重要的东西，比如现金、存折、房产证、情书等。输入密码，打开密码箱，然后放东西进去；取的时候，首先需要密码打开保险箱，然后把东西取出。没有密码，老婆是万万看不到老公和他初恋之间的情书的。RPMB就像是手机里的密码箱，用户可以把一些重要数据存储其中。

我们来看看RPMB这个保险箱。

UFS主机通过认证（authenticated）的方式访问RPMB LU。下图展示了数据写过程：



1. 首先，UFS主机和UFS设备共享密钥，该密钥在UFS设备出厂时就保存在UFS设备；
2. UFS主机在发送主机数据给UFS设备前，会用该密钥和哈希算法生成消息认证码(Message Authentication Code, MAC)；
3. UFS主机把主机数据连同MAC一起发给UFS设备；
4. UFS设备把收到的主机数据和共享密钥在本地重新计算MAC，然后把计算出的MAC和收到的MAC做对比，如果一致，则认证成功,写入到闪存；否则,拒绝该笔数据的写。

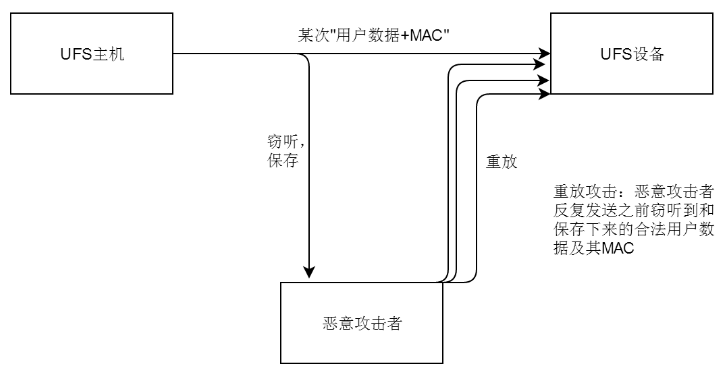
UFS使用HMAC（Hash-based Message Authentication Code）SHA-256算法生成消息认证码。HMAC运算利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。关于HMAC具体算法，可参看<https://en.wikipedia.org/wiki/HMAC>，我们这里不深入。

消息认证码本质是哈希值。哈希的一个特点是，即使只改变原数据一比特数据，两者的哈希值也是完全不同的。如果恶意攻击者在数据传输过程中篡改了用户数据，那么UFS设备根据收到的数据和共享密钥生成的MAC肯定与接收到的MAC不一样，认证通不过，数据就不会写入UFS设备。

这里的前提是共享密钥不能被恶意攻击者获取，否则，恶意攻击者完全可以模拟主机行为：把自己的恶意数据和共享密钥生成MAC，然后把恶意数据和其对应的MAC发送给UFS设备。UFS设备会认证成功,恶意数据被写入。所以，请保管好你的密码!

但是，恶意攻击者是狡猾的，即使他没有办法获得你的密钥，它还是有办法对你进行攻击的。

恶意攻击者监听到UFS主机和UFS设备之间某次数据传输，得到“主机数据 + MAC”，然后该恶意攻击者重复发送该“主机数据 + MAC”给UFS设备，由于“主机数据 + MAC”是合法的，认证通过，UFS设备就会接收该数据并写到闪存。恶意攻击者如果一直重复发这些数据给UFS设备，UFS设备RPMB LU将会被写爆！这就是重放攻击, Replay Attack。



RPMB的全名是：Replay Protection Memory Block，它的名字暗示了RPMB是能抵御重放攻击的。那么RPMB是怎么对付重放攻击的呢？

UFS维护了一个写计数（Write Counter），初始化为0。UFS设备每次成功处理完一个RPMB写命令，写计数加一。主机在往设备写入数据前，获得该计数。然后把用户数据和该计数一起做MAC计算。这样，即使恶意攻击者窃听到某次合法的“用户数据 + MAC”，往设备写入时，由于写计数发生变化，它无法生成写计数改变之后的MAC值，因此就无法一直重复往设备写入某次合法的“用户数据 + MAC”。魔高一尺，道高一丈，正义终战胜邪恶！

上面就是RPMB数据安全性背后的原理。下面再回到UFS RPMB协议上来。

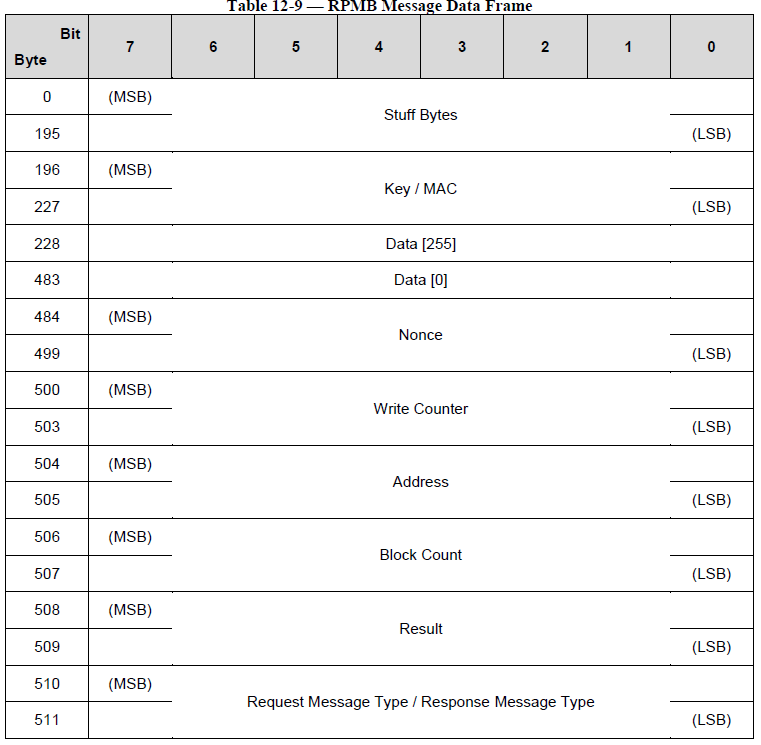
UFS2.1中，RPMB LU最小逻辑空间为128KB，最大为16MB。它的逻辑块大小为256B（普通LU逻辑块大小一般为4KB）。应用层不是通过普通的Read/Write命令读/写RPMB上的数据，而是通过SECURITY PROTOCOL OUT/IN命令来访问RPMB的。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **逻辑空间大小** | **逻辑块大小** | **访问命令** | **安全性** | **存储数据类型** |
| **RPMB LU** | 128KB -16MB | 256B | Security Protocol OUT/IN | 数据认证 | 防篡改的重要数据 |
| **普通LU** | 无限制 | 一般为4KB | Read/Write | 无 | 普通用户数据 |

UFS主机在访问设备RPMB时，是通过下面消息交互完成的。

|  |  |  |
| --- | --- | --- |
| **请求（Request）** | **响应（Response）** | **说明** |
| Authentication Key programming request | Authentication Key Programming response | 用在写认证密钥 |
| Write Counter read request | Write Counter read response | 用在读取写计数 |
| Authenticated data write request | Authenticated data write response | 用在写认证数据 |
| Authenticated data read request | Authenticated data read response | 用在读认证数据 |
| Result read request | NA | 读取操作结果 |
| Secure Write Protect Configuration Block write request | Secure Write Protect Configuration Block write response | 用于写安全写保护配置块 |
| Secure Write Protect Configuration Block read request | Secure Write Protect Configuration Block read response | 用于读安全写保护配置块 |

每条消息包含一条或者若干条消息数据帧。消息数据帧大小是512字节，具体如下：



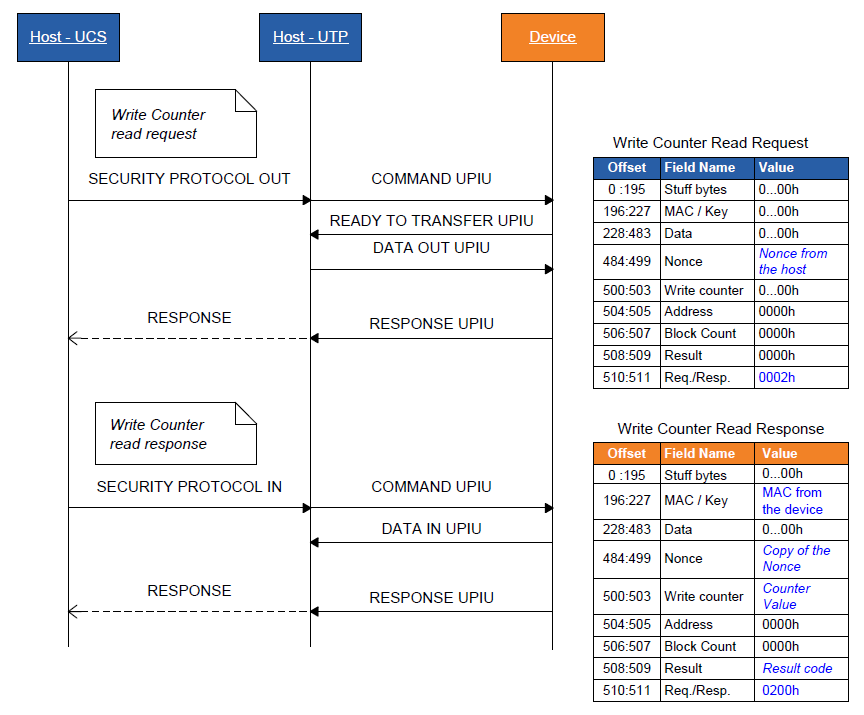
从中，我们看到：

* 认证密钥（Key）是32字节；
* UFS2.1使用SHA-256计算MAC，就是任意长度的数据，产生的MAC值总是256比特，即MAC大小为32字节。
* 逻辑块数据大小为256字节。
* 写计数（Write Counter）为4字节，当该值涨到0xFFFF FFFF，它就保持不动，不会继续增长了。
* Address，RPMB的逻辑地址，同LBA。两个字节，最多表示65536个逻辑块，每个逻辑块大小为256字节，因此RPMB逻辑空间最大为16MB.
* Block Count，逻辑块数，即指定读写多少个逻辑块。
* Result，RPMB操作结果（状态）。

下面举几个RPMB操作例子来理解上面的消息：

* **主机读取写计数**

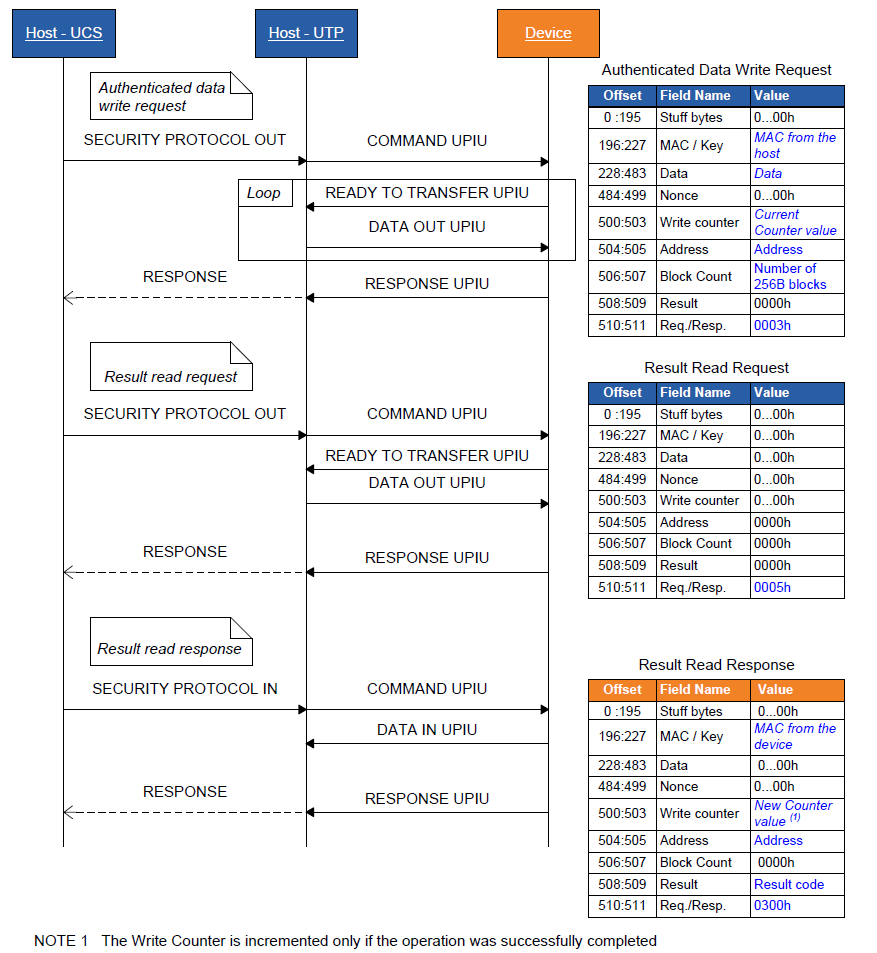
如前所述，写计数的目的是抵御重放攻击。写计数是UFS设备管理和维护的，UFS设备递增该计数。主机在写数据时，需要知道该计数，然后加上用户数据，一起计算MAC。



命令层发SECURITY PROTOCOL OUT/IN命令读取写计数，然后传输层生成相应的UPIU进行主机与设备之间的交互，具体见上图。

* **主机写认证数据**

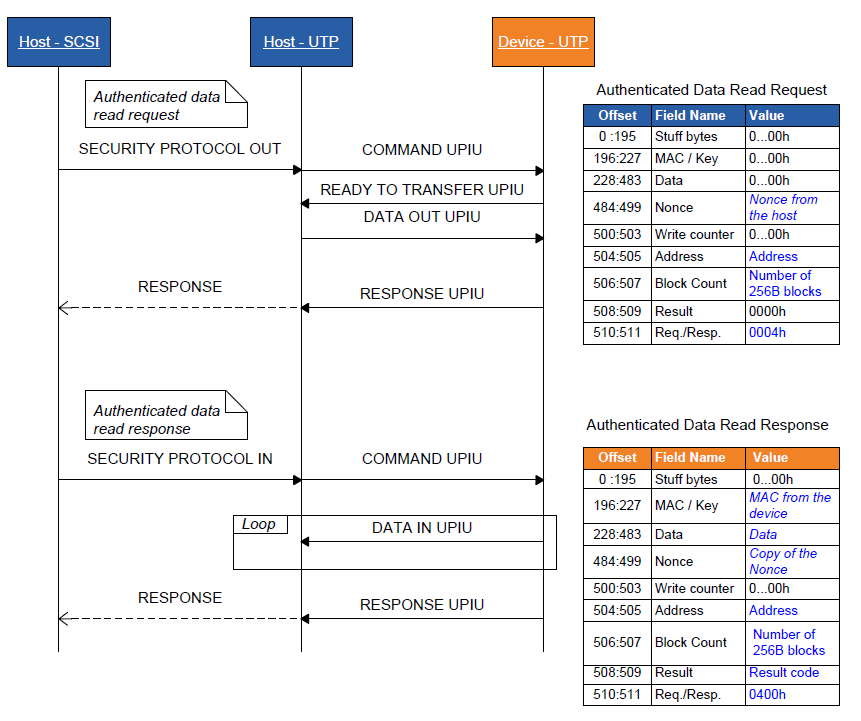
主机命令层通过SECURITY PROTOCOL OUT命令把用户数据和对应的MAC发送给设备，然后通过SECURITY PROTOCOL OUT请求获取前面数据写结果，最后通过SECURITY PROTOCOL IN读取写结果。写结果中包含新的写计数，这样下次主机利用新的写计数计算MAC。注意，只有本次写认证数据成功，设备才会递增该计数。



* **主机读认证数据**

首先，主机通过SECURITY PROTOCOL OUT命令发送读取认证数据请求给设备，然后发送SECURITY PROTOCOL IN命令读取数据。

注意，主机读取数据也是需要认证的。在设备端，UFS设备会计算MAC，然后主机端根据MAC认证该数据。这样可以防止恶意攻击者在数据传输过程中（从设备到主机），用恶意数据更换原始数据。



RPMB提供了认证访问方式和抵御重放攻击的机制，保证了存储在RPMB LU上数据的安全。因此，用户可以把一些敏感和重要的信息写在RPMB上。在实际应用中，它通常用于存储一些有防止非法篡改需求的数据，例如手机上指纹支付相关的公钥、序列号等敏感信息。

# 蛋蛋读UFS之九：UFS数据安全

前面提到RPMB使用认证机制和抗重放攻击机制保障数据不被黑客攻击，除此之外，UFS还有其它一些手段来保护用户数据安全，这一章节我们来关注UFS数据安全。

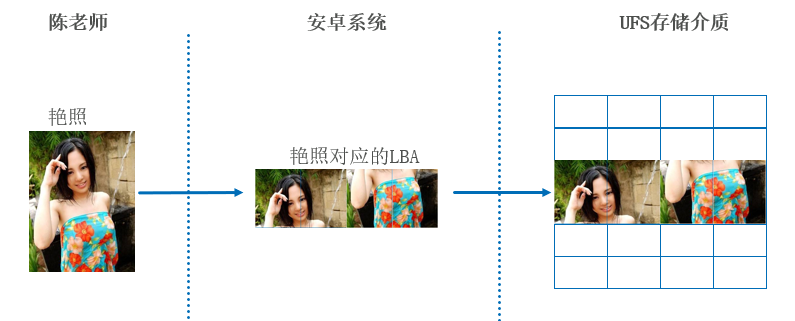


UFS设备用来存储个人或者企业的数据信息，UFS设备需要这样一种机制，就是必要时，数据能永久从设备（闪存）删除，这样就能防止别有用心的人通过反向工程获取你的数据。

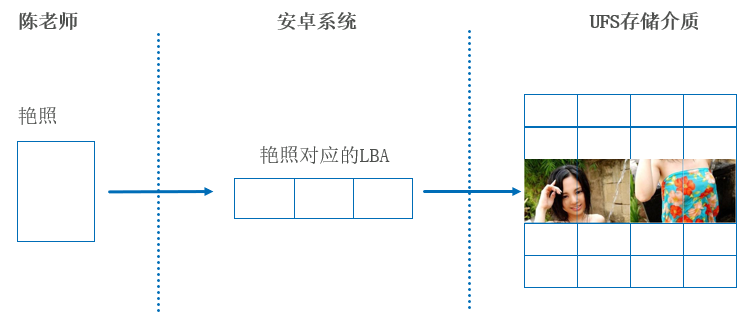
话说，陈老师吸取了上次教训，每次拍照后，事后“阅后即焚”。陈老师想：这样就没有人看到我们的照片了吧。陈老师很是得意。

没有想到，不久后网上又爆出陈老师新的“艳照门”事件。陈老师很是纳闷，我不是明明都删除了吗？？

我们帮陈老师分析一下为什么删了的照片还能被修手机的人弄出来。



手机文件系统把陈老师拍的照片数据用逻辑块管理，然后把这些逻辑块写到UFS设备的存储介质（如上图所示）。陈老师删除照片，删除的只是逻辑块数据，存在UFS存储介质上的数据还在原地，如下图所示：

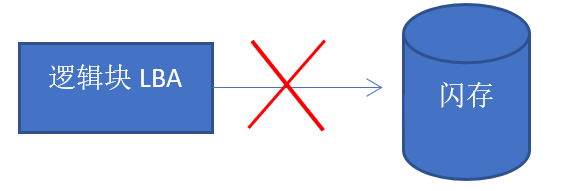


修手机的一看，这不陈老师吗？修手机的知道肯定能发现什么，嘴角不禁露出一丝不易觉察的笑。他从文件中没有找到照片，有点失望。小子吃一堑长一智呀！但猎奇的心不会让他轻易放弃的，有经验的他盯上了存储卡。功夫不负有心人，他从存储介质里把照片弄出来！

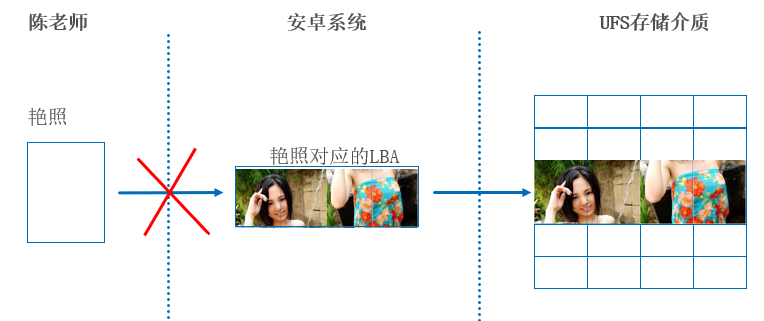
拍照不容易，且拍且珍惜。我们看看如何避免陈老师的悲剧。

* **擦除操作（Erase Operation）**

注意，这个“擦除”操作不是擦除存储介质，不是闪存层面的擦除操作，而是UFS层面的擦除操作。数据写在闪存上，UFS设备内部有个逻辑地址到物理地址的映射，擦除操作通过切断这种映射，主机就不能获得擦除掉的数据。



当陈老师删除照片时，它切断了用户直接访问照片的途径：



当陈老师删除照片后，手机系统会通过发送UNMAP命令（就是通常说的TRIM）来告诉设备这些照片数据无效。设备收到该命令后，然后根据逻辑单元描述符中的**bProvisioningType**来确定执行具体操作。

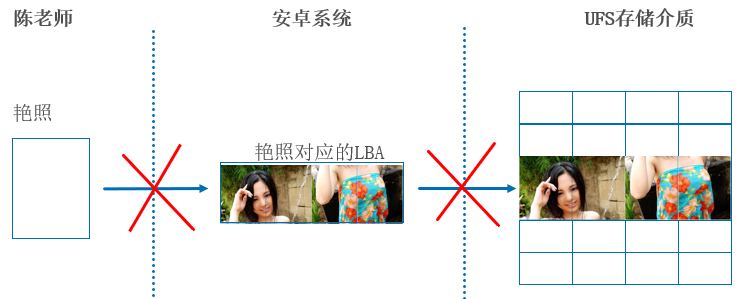
**bProvisioningType**：

**00h**: Thin Provisioning is disabled (default)

**02h**: Thin Provisioning is enabled and TPRZ = 0 (Discard)

**03h**: Thin Provisioning is enabled and TPRZ = 1 (Erase)

即当**bProvisioningType** = 03h时，设备执行擦除操作，即切断逻辑地址到物理空间的映射。



一个逻辑块如果被擦除，那么主机访问这个逻辑块时，设备必须返回全0数据给主机。

注意，这个“擦除”操作不是擦除存储介质，只是主机让设备切断逻辑地址到物理地址的映射，因此不保证照片数据从闪存介质删除。但是，由于UFS设备知道该照片数据已经删除（没有逻辑块到物理空间的映射），在后续垃圾回收时，这些被删掉的数据很大概率会从介质上擦除掉。

* **舍弃操作（Discard Operation）**

和擦除操作类似，主机通过发送UNMAP命令来执行舍弃操作。当bProvisioningType = 02h时，设备执行舍弃操作。

舍弃操作和擦除操作的区别：主机访问一个被舍弃的逻辑块，可能获得任何数据，甚至包括舍弃前的数据，而擦除操作是主机获得全0数据。也就是说，对删除的照片，如果UFS设备执行的是舍弃操作，那么主机还可能获得原图片；如果UFS设备执行的是擦除操作，主机不可能再获得原照片。

但不管是舍弃操作还是擦除操作，都不能保证照片从存储介质上删除。像修手机这样的人，它不走寻常路（通过手机系统），直接操作闪存的话，还是有可能把删除的照片找回来。

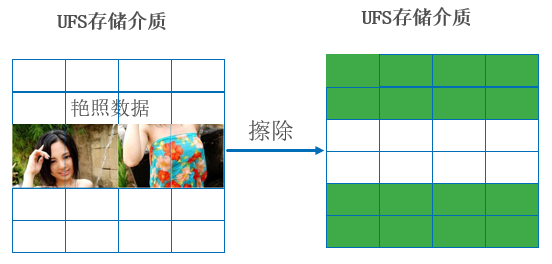
陈老师看到这里，急了，难道我以后再也不能拍照了吗？？

别急，小陈！你先坐下，听我慢慢讲。

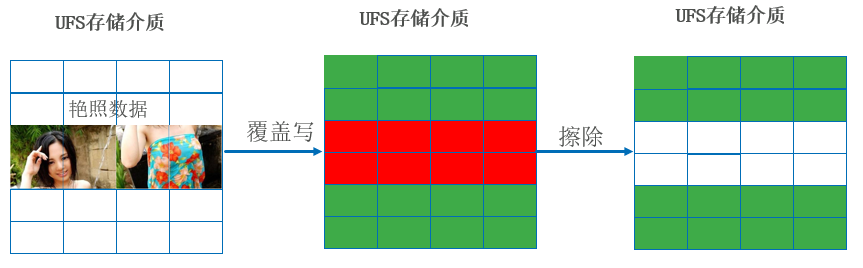
* **安全清除（Secure Removal）**

陈老师，有三种可选策略用以安全清除数据，你造吗？

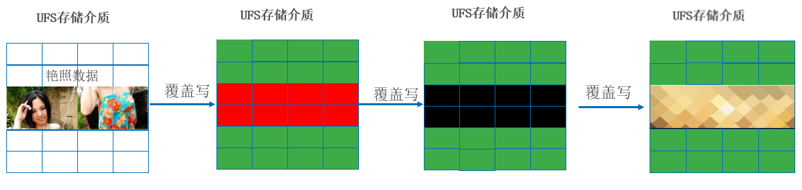
1. 设备控制器擦除（Erase）要被删除的逻辑块所对应的物理地址空间；



1. 设备控制器用单一字符覆盖写要被删除的逻辑块所对应的物理地址空间，然后擦除设备；



1. 设备控制器用单一字符、字符补码和随机字符，依次覆盖写要被删除的逻辑块所对应的物理地址空间。



又是覆盖写，又是擦除，照片是彻底从闪存中删除了。

陈老师听到这里，情绪缓和下来，终于是坐了下来。

* **清除操作（Purge Operation）**

清除操作是针对垃圾数据(比如陈老师删除的照片），让这些数据不仅不能通过正规渠道（操作系统）访问，还让这些数据无法从存储介质中获取，彻底把垃圾数据从UFS设备清除掉。

前面所说的擦除和舍弃操作，都是主机通过命令层的UNMAP命令来实施的。而清除操作则是主机通过设备管理器的Query功能来告诉设备的。

这里涉及到一个重要的标志（flag）和一个重要的属性（Attribute），分别是发PurgeEnable和bPurgeStatus，前者用以使能/禁止清除操作，后者用以设备向主机提供清除操作的状态信息。

fPurgeEnable:

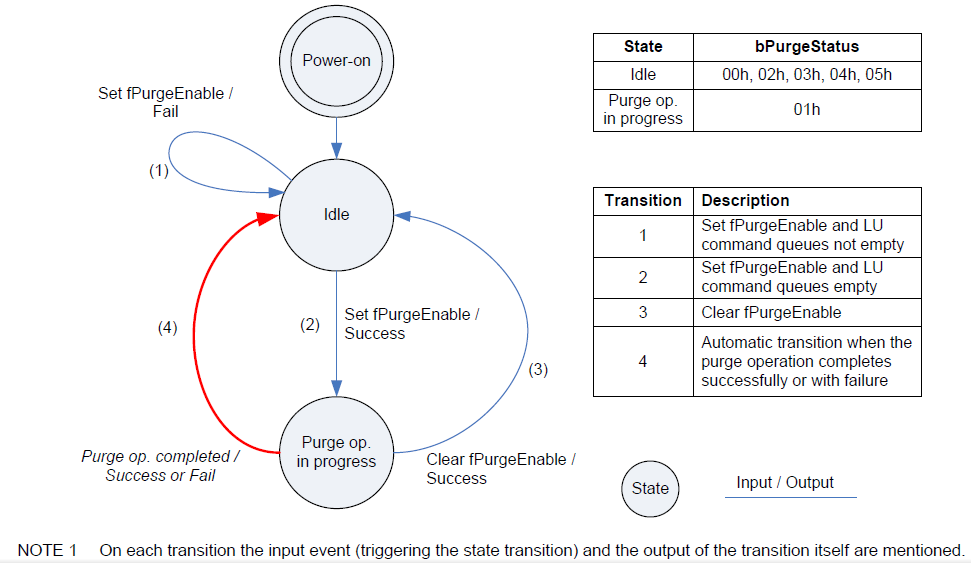
1. 上电或者重启，该标志位0；
2. 主机通过设置或者清除该标志，使能或禁止清除操作；
3. 只有当所有逻辑单元的命令队列空的时候，主机才能设置该标志为1使能清除操作；
4. 当UFS设备执行完清除操作或者发生错误，该标志会被清零；
5. 如果主机想终止设备执行清除操作，可以通过清除该标志达到目的。

bPurgeStatus：

|  |  |
| --- | --- |
| **bPurgeStatus的值** | **状态** |
| 00h | 清除操作被禁止 |
| 01h | 设备正在执行清除操作 |
| 02h | 清除操作被主机提前终止 |
| 03h | 清除操作成功完成 |
| 04h | 清除操作失败因为逻辑单元的命令队列非空 |
| 05h | 清除操作一般性错误 |

主机为了让设备执行清除操作，主机通过QUERY REQUEST UPIU设置fPurgeEnable = 1。如果当前逻辑单元的命令队列中没有任何命令，设备会执行清除操作。一旦设备开始执行清除操作，它不会响应主机发来的任何命令。如果这个时候主机需要让设备紧急响应命令，主机首先应该通过QUERY REQUEST UPIU设置fPurgeEnable = 0来提前终止设备的清除操作，然后再发送命令。

下图是清除操作的状态机图：



UFS设备在执行清除操作时，对那些垃圾数据，有以下几种处理方式：

1. 默认是把这些垃圾数据从闪存空间擦除掉；
2. 或者先用单个字符（比如全A）覆盖写，然后再擦除；
3. 抑或先用单个字符（比如A）覆盖写，然后用它的补码（比如5）覆盖写，最后用随机字符覆盖写；
4. 最后还可以使用用户自定义的方式处理。

这些手段前面已经介绍过。

* **格式化设备（Wipe Device）**

主机通过发送FORMAT UNIT命令格式化所有的逻辑单元（RPMB LU除外）。不过，对那些写保护的逻辑单元，FORMAT UNIT命令会失败。

FORMAT UNIT的命令对象是Device well know LU，它格式化除RPMB之外所有无写保护的逻辑单元。

FORMAT UNIT会切断逻辑块到物理空间的映射。但如果要让数据彻底从设备上清除，UFS设备还需要执行Purge操作，这样数据才能彻底删除。

* **写保护**

前面都是千方百计的清除数据，但有时候UFS设备需要保护写的数据。

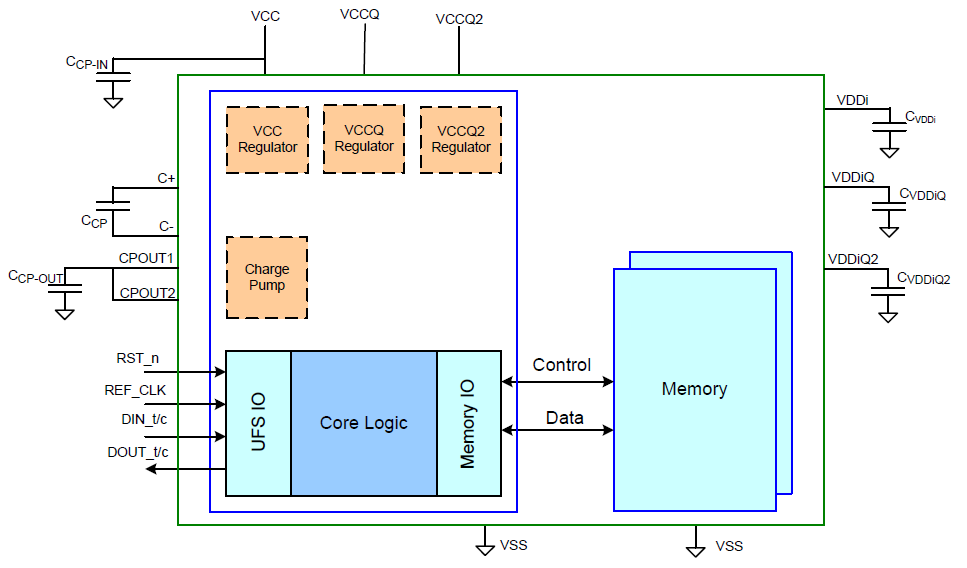
每个逻辑逻辑单元（除了RPMB）有写保护属性。写保护包括永久写保护和上电写保护，前者的意思是说，一旦该逻辑单元写保护使能，将终生是写保护（不能改回去了）；而后者写保护只对某次上电有效，如果设备重上电或者重启，写保护将失效。

最后总结一下UFS数据安全机制：

1. 安全擦除（本章重点讲述）；
2. 写保护（本章讲述）；
3. RPMB，提供数据认证和抗重放攻击机制保护一些重要数据，请参看RPMB章节。

# 蛋蛋读UFS之十：UFS电源管理

UFS是手机存储设备，因此对功耗要求很高。我们来看看UFS的电源管理。

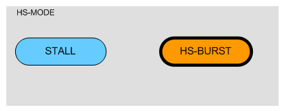


三个供电电压，VCC,VCCQ和VCCQ2，分别给UFS设备模块供电。UFS设备主要包括三部分：前端UFS接口（M-PHY），UFS控制器和闪存介质（图中的Memory模块）。VCC给闪存介质供电，VCCQ一般给闪存输入输出接口和UFS控制器供电，VCCQ2一般给M-PHY或其它一些低电压模块供电。

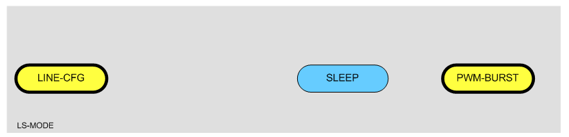
UFS2.1中，三者电压值为：

|  |  |  |
| --- | --- | --- |
|  | **最小电压（V）** | **最大电压（V）** |
| VCC | 2.7 | 3.6 |
| 1.70 | 1.95 |
| VCCQ | 1.1 | 1.3 |
| VCCQ2 | 1.70 | 1.95 |

我们知道，UFS协议采用MIPI的M-PHY作为物理层和UniPro作为其数据链路层。M-PHY有高速模式（High Speed Mode, HS-MODE）和低速模式 （Low Speed Mode, LS-MODE）。其中，高速模式下，M-PHY有两种状态：STALL和HS-BURST。



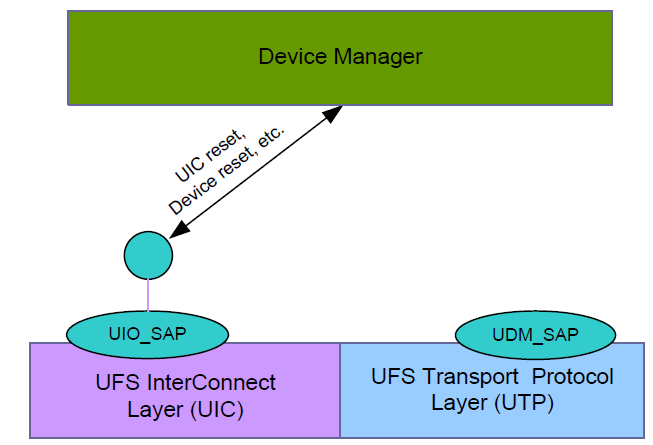
低速模式下，M-PHY有三种状态：LINE-CFG，SLEEP和PWM-BURST。



当链路上没有数据传输时，M-PHY会自动切换到STALL或者SLEEP状态下，这两种状态为省电状态。

除此之外，M-PHY还有一种更加省电的状态，那就是HIBERN8 （Hibernate，休眠状态），这种状态下，M-PHY极为省电。UFS主机和UFS设备不可能一直交互数据，总有闲下来的时候。当UFS主机没有读写UFS设备，它会让彼此链路进入休眠状态，即HIBERN8。那UFS主机如何通知M-PHY切换到休眠状态呢？

前面提到，设备管理器可以略过传输层，直接管理与控制互联层：



主机设备管理器可以通过原语（Primitive）直接与UFS互联层（UIC，即MIPI的UniPro和M-PHY）通信。除了上图中所示的reset原语，UFS还包括让UIC进入和退出休眠的原语：DME\_HIBERNATE\_ENTER和DME\_HIBERNATE\_EXIT。

这是UFS主机和设备之间链路的省电模式，对UFS设备来说，链路只是整个UFS设备的一部分。一个UFS设备是否省电，除了看其链路，还需要考虑UFS控制器、存储介质等是否省电，即看整个UFS设备是否有好的电源管理。

UFS定义了4种基本功耗模式：Active，Idle，Power Down和Sleep（简称AIDS），外加3个过渡功耗模式：Pre-Active, Pre-Sleep和Pre-PowerDown，一共是7种功耗模式。非常4+3！

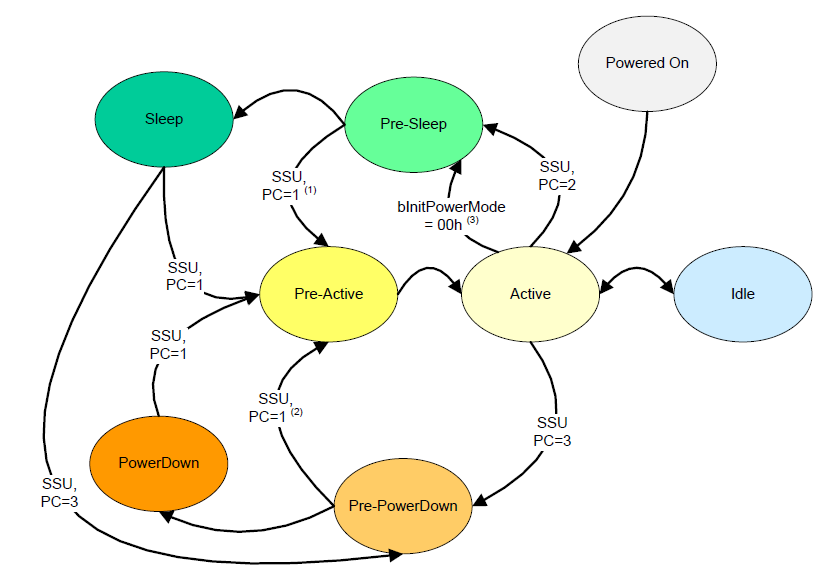
**Active模式**：UFS设备在执行命令或者做后台任务（Background Operation）时处于这种状态；

**Idle模式**：UFS设备空闲时，即既没有来自UFS主机的命令，自身也没有后台任务需要处理，设备就处于该状态；

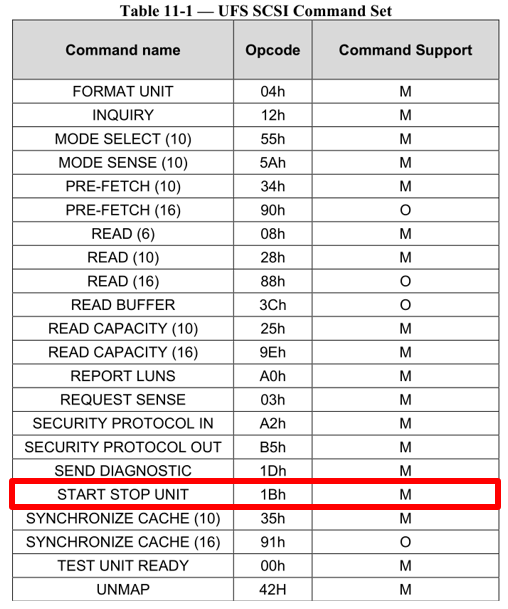
**Sleep模式**：闲得瞌睡了。睡眠模式下，VCC电源可能被切断（取决UFS设备设计）。VCC一般给闪存供电，即切断闪存供电。

**Power Down模式**：掉电模式下，所有电源供电VCC, VCCQ和VCCQ2都可能被掐断（取决UFS设备设计），该模式是最省电的功耗模式了。

这些模式之间的转换如下图：



我们看到，触发模式之间转换的很多是SSU，那么什么是SSU? SSU是Start Stop Unit的缩写，它是UFS协议中的一个基本命令，主机用它来切换UFS设备的功耗模式。



具体命令可以参看UFS spec。

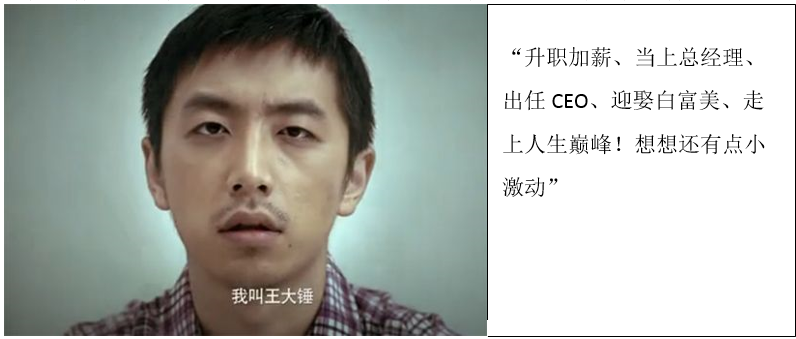
注意，UFS设备的这些功耗状态，和前面说的M-PHY接口的STALL，SLEEP或者HIBERN8状态是独立的，两者没有必然联系。比如，当前M-PHY处于HIBERN8状态，UFS设备可以处于以上状态中的任何一种，比如UFS设备可以是处于Active状态，没有要求说你休眠了我也得跟着休眠。

一个优秀的员工，不是老板push一下，然后才往前走一步，而是能主动的去承担一些任务。一个好的UFS设备，不是等着主机发功耗切换命令来进入省电模式，而是自己能主动做一些事情来省电。

下面就是一个优秀UFS设备需要具备的素质。

比如，UFS刚上电时，UFS进入Active状态，一段时间如果没有来自主机的命令，自己内部也没有后台任务要处理，UFS设备将进入Idle状态。Idle意味着无事可做，这时候主机也没有发任何SSU命令要求UFS设备进入指定的状态（老板也没有叫你去做什么），好的UFS设备，这个时候就要想想怎么去省电。举例来说，如果当前M-PHY处于HIBERN8状态，说明主机目前不会访问UFS设备，因此，UFS设备可以做一些节能设计：比如把当前UFS设备的软硬件上下文保存到闪存，然后切断所有电源以达到省电目的。待M-PHY接口退出HIBERN8状态，UFS设备上电，然后把软硬件上下文加载运行。

老板没有叫你去干活，你主动的去把活干了，这样的员工哪个老板不喜欢？



省电和用户体验（命令响应快慢）其实是个矛盾。因为如果UFS设备休眠了，它就不能及时的响应主机的命令，因为它需要先退出休眠（比如需要把休眠之前保存的上下文重新加载，这往往比较花时间），然后再响应主机命令。睡觉是个技术活，在追求最大节能的同时，还要兼顾用户体验。