

目錄

Introduction	1.1
Storage	1.2
Flash Memory	1.2.1
eMMC	1.2.1.1
eMMC 分区管理	1.2.1.1.1
eMMC 总线协议	1.2.1.1.2
eMMC 工作模式	1.2.1.1.3
eMMC CMDQ	1.2.1.1.4
eMMC RPMB	1.2.1.1.5
eMMC 设备寄存器	1.2.1.1.6
eMMC Commands	1.2.1.1.7
SD Card	1.2.1.2
UFS	1.2.1.3
NAND Flash	1.2.1.4
NOR Flash	1.2.1.5
NVMe	1.2.1.6
Linux SDVMMC Framework	1.2.2
Memory	1.3
SDRAM	1.3.1
SRAM	1.3.2
DRAM	1.3.3
DRAM Storage Cell	1.3.3.1
DRAM Memory Organization	1.3.3.2
DRAM Device	1.3.3.3
DRAM Timing	1.3.3.4
DRAM Devices Organization	1.3.3.5
Cache	1.3.4
CPU	1.4
ISA	1.4.1
Microarchitecture	1.4.2
Pipeline	1.4.2.1
Cache	1.4.2.2
Branch prediction	1.4.2.3
Superscalar	1.4.2.4
Out-of-order execution	1.4.2.5
Multiprocessing and multithreading	1.4.2.6
Register Renaming	1.4.2.7
VLIW	1.4.2.8
虚拟化	1.4.3

Linux Kernel Internals

Linux Kernel 相关的内容～

Storage

Flash Memory 简介

Flash Memory 是一种非易失性的存储器。在嵌入式系统中通常用于存放系统、应用和数据等。在 PC 系统中，则主要用在固态硬盘以及主板 BIOS 中。另外，绝大部分的 U 盘、SDCard 等移动存储设备也都是使用 Flash Memory 作为存储介质。

Flash Memory 的主要特性

与传统的硬盘存储器相比，Flash Memory 具有质量轻、能耗低、体积小、抗震能力强等的优点，但也有不少局限性，主要如下：

1. 需要先擦除再写入

Flash Memory 写入数据时有一定的限制。它只能将当前为 1 的比特改写为 0，而无法将已经为 0 的比特改写为 1，只有在擦除的操作中，才能把整块的比特改写为 1。

2. 块擦除次数有限

Flash Memory 的每个数据块都有擦除次数的限制（十万到百万次不等），擦写超过一定次数后，该数据块将无法可靠存储数据，成为坏块。

为了最大化的延长 Flash Memory 的寿命，在软件上需要做擦写均衡（Wear Leveling），通过分散写入、动态映射等手段均衡使用各个数据块。同时，软件还需要进行坏块管理（Bad Block Management，BBM），标识坏块，不让坏块参与数据存储。（注：除了擦写导致的坏块外，Flash Memory 在生产过程也会产生坏块，即固有坏块。）

3. 读写干扰

由于硬件实现上的物理特性，Flash Memory 在进行读写操作时，有可能会导致邻近的其他比特发生位翻转，导致数据异常。这种异常可以通过重新擦除来恢复。Flash Memory 应用中通常会使用 ECC 等算法进行错误检测和数据修正。

4. 电荷泄漏

存储在 Flash Memory 存储单元的电荷，如果长期没有使用，会发生电荷泄漏，导致数据错误。不过这个时间比较长，一般十年左右。此种异常是非永久性的，重新擦除可以恢复。

NOR Flash 和 NAND Flash

根据硬件上存储原理的不同，Flash Memory 主要可以分为 NOR Flash 和 NAND Flash 两类。主要的差异如下所示：

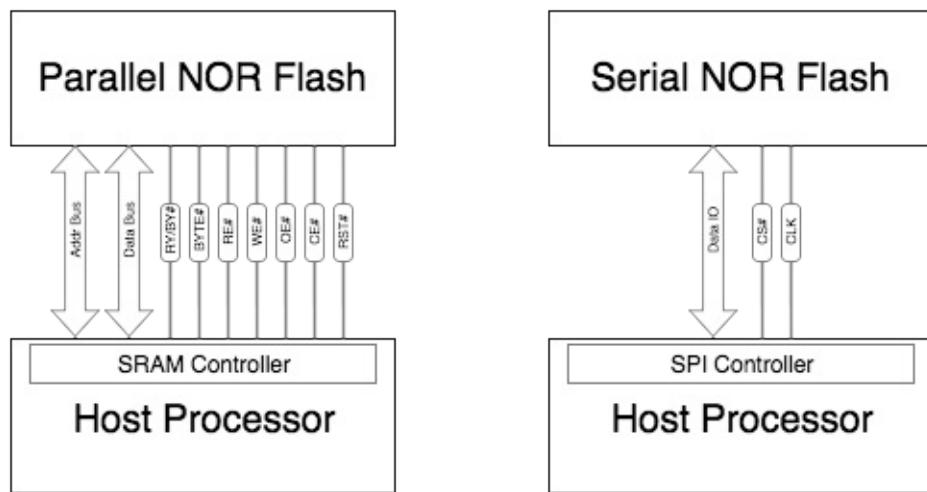
- NAND Flash 读取速度与 NOR Flash 相近，根据接口的不同有所差异；
- NAND Flash 的写入速度比 NOR Flash 快很多；
- NAND Flash 的擦除速度比 NOR Flash 快很多；
- NAND Flash 最大擦次数比 NOR Flash 多；
- NOR Flash 支持片上执行，可以在上面直接运行代码；
- NOR Flash 软件驱动比 NAND Flash 简单；
- NOR Flash 可以随机按字节读取数据，NAND Flash 需要按块进行读取。
- 大容量下 NAND Flash 比 NOR Flash 成本要低很多，体积也更小；

（注：NOR Flash 和 NAND Flash 的擦除都是按块块进行的，执行一个擦除或者写入操作时，NOR Flash 大约需要 5s，而 NAND Flash 通常不超过 4ms。）

NOR Flash

NOR Flash 根据与 CPU 端接口的不同，可以分为 Parallel NOR Flash 和 Serial NOR Flash 两类。

Parallel NOR Flash 可以接入到 Host 的 SRAM/DRAM Controller 上，所存储的内容可以直接映射到 CPU 地址空间，不需要拷贝到 RAM 中即可被 CPU 访问，因而支持片上执行。Serial NOR Flash 的成本比 Parallel NOR Flash 低，主要通过 SPI 接口与 Host 连接。



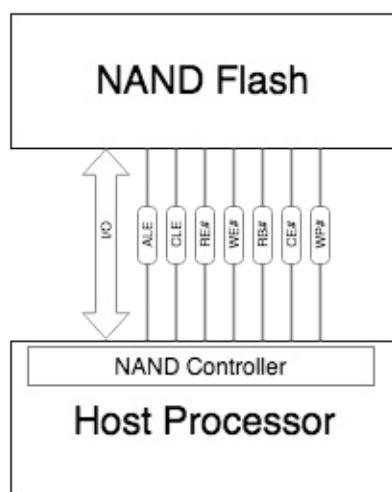
图片：Parallel NOR Flash 与 Serial NOR Flash

鉴于 NOR Flash 擦写速度慢，成本高等特性，NOR Flash 主要应用于小容量、内容更新少的场景，例如 PC 主板 BIOS、路由器系统存储等。

更多 NOR Flash 的相关细节，请参考 [NOR Flash 章节](#)。

NAND Flash

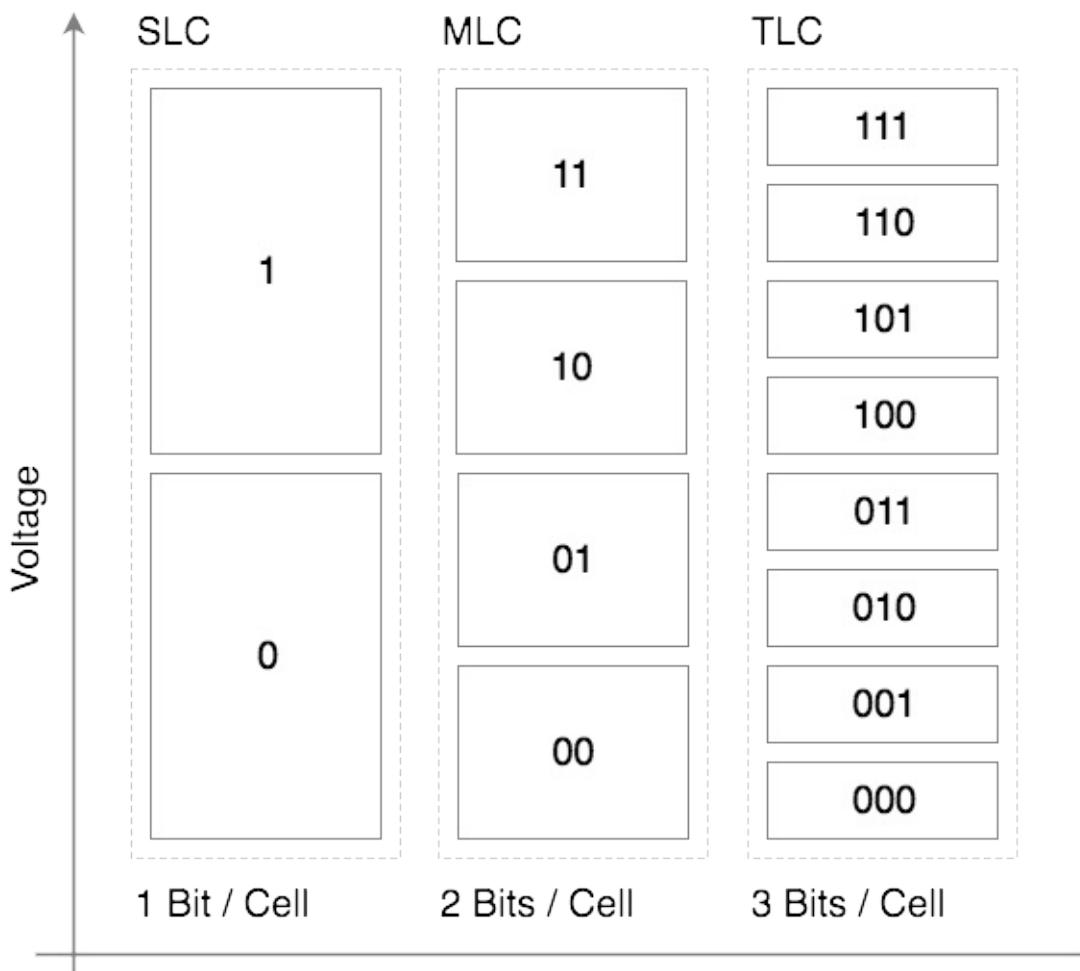
NAND Flash 需要通过专门的 NFI (NAND Flash Interface) 与 Host 端进行通信，如下图所示：



图：NAND Flash Interface

NAND Flash 根据每个存储单元内存储比特个数的不同，可以分为 SLC (Single-Level Cell)、MLC (Multi-Level Cell) 和 TLC (Triple-Level Cell) 三类。其中，在一个存储单元中，SLC 可以存储 1 个比特，MLC 可以存储 2 个比特，TLC 则可以存储 3 个比特。

NAND Flash 的一个存储单元内部，是通过不同的电压等级，来表示其所存储的信息的。在 SLC 中，存储单元的电压被分为两个等级，分别表示 0 和 1 两个状态，即 1 个比特。在 MLC 中，存储单元的电压则被分为 4 个等级，分别表示 00 01 10 11 四个状态，即 2 个比特位。同理，在 TLC 中，存储单元的电压被分为 8 个等级，存储 3 个比特信息。



图片：SLC、MLC 与 TLC

NAND Flash 的单个存储单元存储的比特位越多，读写性能会越差，寿命也越短，但是成本会更低。Table 1 中，给出了特定工艺和技术水平下的成本和寿命数据。

Table 1

	SLC	MLC	TLC
制造成本	30-35 美元 / 32GB	17 美元 / 32GB	9-12 美元 / 32GB
擦写次数	10万次或更高	1万次或更高	5000次甚至更高
存储单元	1 bit / cell	2 bits / cell	3 bits / cell

(注：以上数据来源于互联网，仅供参考)

相比于 NOR Flash，NAND Flash 写入性能好，大容量下成本低。目前，绝大部分手机和平板等移动设备中所使用的 eMMC 内部的 Flash Memory 都属于 NAND Flash。PC 中的固态硬盘中也是使用 NAND Flash。

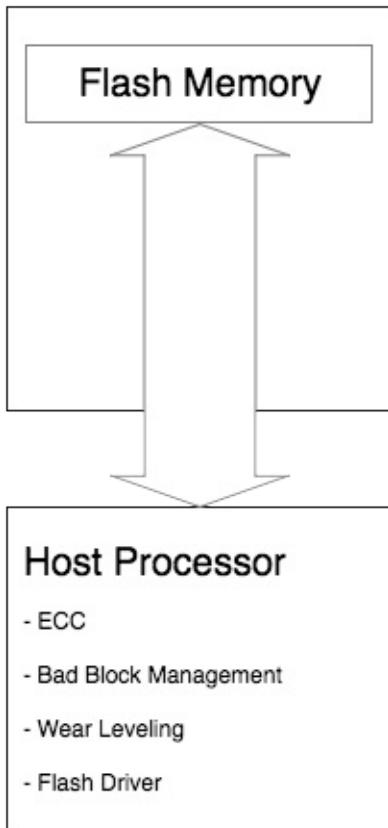
更多 NAND Flash 的相关细节，请参考 [NAND Flash](#) 章节。

Raw Flash 和 Managed Flash

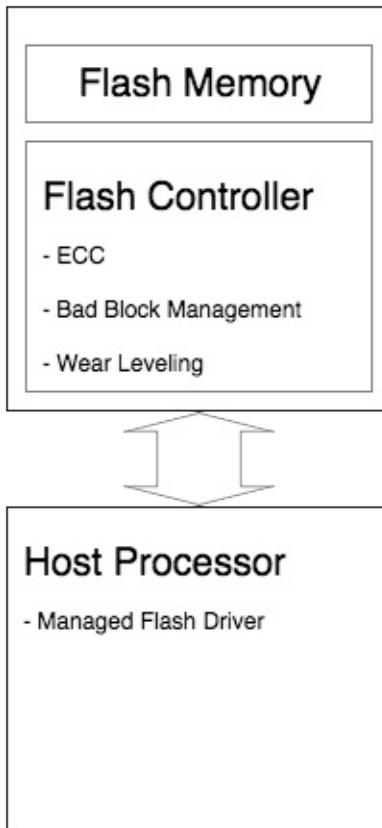
由于 Flash Memory 存在按块擦写、擦写次数的限制、读写干扰、电荷泄露等的局限，为了最大程度的发挥 Flash Memory 的价值，通常需要有一个特殊的软件层次，实现坏块管理、擦写均衡、ECC、垃圾回收等的功能，这一个软件层次称为 FTL (Flash Translation Layer)。

在具体实现中，根据 FTL 所在的位置的不同，可以把 Flash Memory 分为 Raw Flash 和 Managed Flash 两类。

Raw Flash



Managed Flash



图片：Raw Flash 和 Managed Flash

Raw Flash

在此类应用中，在 Host 端通常有专门的 FTL 或者 Flash 文件系统来实现坏块管理、擦写均衡等的功能。Host 端的软件复杂度较高，但是整体方案的成本较低，常用于价格敏感的嵌入式产品中。

通常我们所说的 NOR Flash 和 NAND Flash 都属于这类型。

Managed Flash

Managed Flash 在其内部集成了 Flash Controller，用于完成擦写均衡、坏块管理、ECC校验等功能。相比于直接将 Flash 接入到 Host 端，Managed Flash 屏蔽了 Flash 的物理特性，对 Host 提供标准化的接口，可以减少 Host 端软件的复杂度，让 Host 端专注于上层业务，省去对 Flash 进行特殊的处理。

eMMC、SD Card、UFS、U 盘等产品是属于 Managed Flash 这一类。

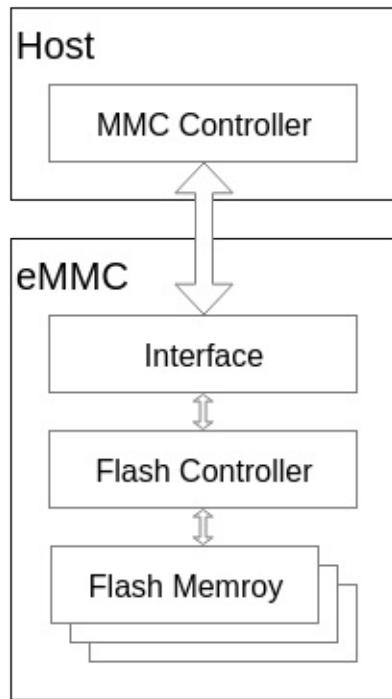
参考资料

1. NOR NAND Flash Guide: Selecting a Flash Storage Solution [PDF]
2. Wiki: Common Flash Memory Interface [Web]
3. Quick Guide to Common Flash Interface [PDF]
4. MICRON NOR Flash Technology [Web]
5. MICRON NAND Flash Technology [Web]
6. Wiki : 闪存 [Web]
7. Wiki : Flash File System [Web]
8. Wear Leveling in Micron® NAND Flash Memory [PDF]
9. Understanding Flash: The Flash Translation Layer [Web]
10. 谈NAND Flash的底层结构和解析 [Web]
11. 闪存基础 [Web]
12. Open NAND Flash Interface (ONFI) [Web]

eMMC 简介

eMMC 是 embedded MultiMediaCard 的简称。MultiMediaCard，即 MMC，是一种闪存卡（Flash Memory Card）标准，它定义了 MMC 的架构以及访问 Flash Memory 的接口和协议。而 eMMC 则是对 MMC 的一个拓展，以满足更高标准的性能、成本、体积、稳定、易用等的需求。

eMMC 的整体架构如下图片所示：



图片：eMMC 整体架构

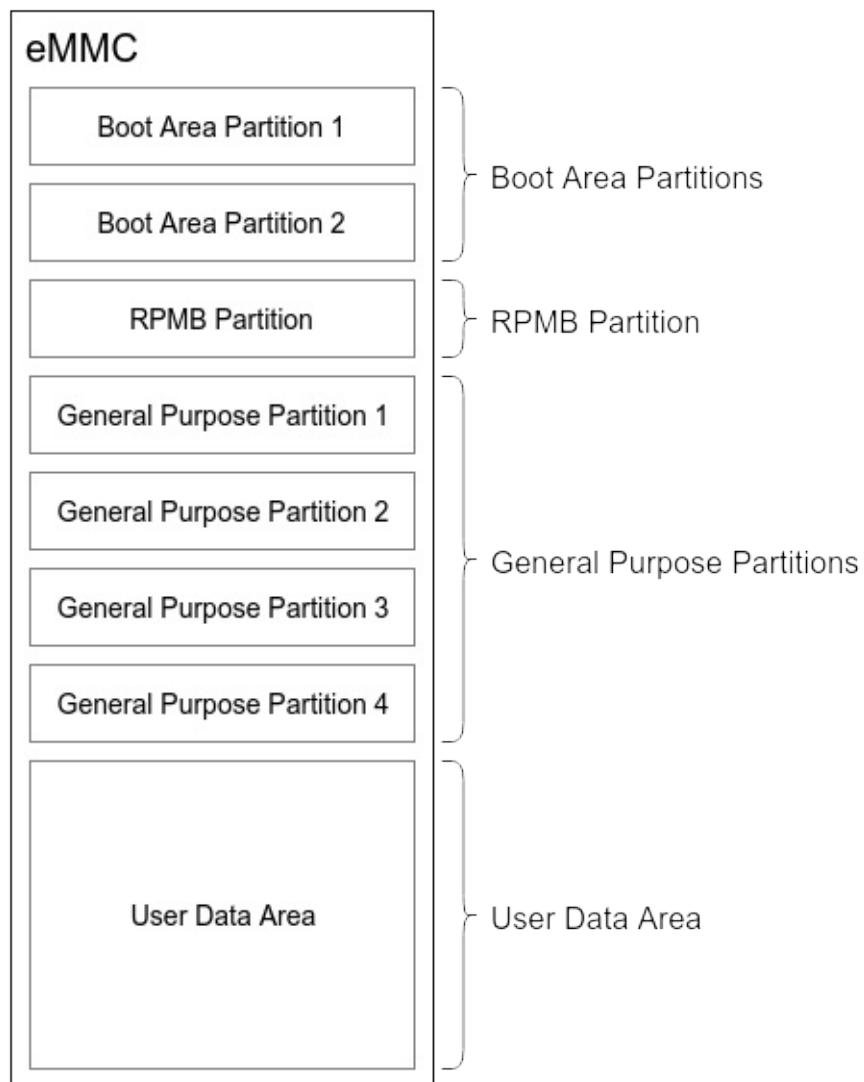
eMMC 内部主要可以分为 Flash Memory、Flash Controller 以及 Host Interface 三大部分。

Flash Memory

Flash Memory 是一种非易失性的存储器，通常在嵌入式系统中用于存放系统、应用和数据等，类似与 PC 系统中的硬盘。

目前，绝大部分手机和平板等移动设备中所使用的 eMMC 内部的 Flash Memory 都属于 NAND Flash，关于 NAND Flash 的更多细节可以参考 [Flash Memory](#) 章节。

eMMC 在内部对 Flash Memory 划分了几个主要区域，如下图所示：



图片：eMMC 内部分区

1. BOOT Area Partition 1 & 2

此分区主要是为了支持从 eMMC 启动系统而设计的。

该分区的数据，在 eMMC 上电后，可以通过很简单的协议就可以读取出来。同时，大部分的 SOC 都可以通过 GPIO 或者 FUSE 的配置，让 ROM 代码在上电后，将 eMMC BOOT 分区的内容加载到 SOC 内部的 SRAM 中执行。

2. RPMB Partition

RPMB 是 Replay Protected Memory Block 的简称，它通过 HMAC SHA-256 和 Write Counter 来保证保存在 RPMB 内部的数据不被非法篡改。

在实际应用中，RPMB 分区通常用来保存安全相关的数据，例如指纹数据、安全支付相关的密钥等。

3. General Purpose Partition 1~4

此区域则主要用于存储系统或者用户数据。General Purpose Partition 在芯片出厂时，通常是不存在的，需要主动进行配置后，才会存在。

4. User Data Area

此区域则主要用于存储系统和用户数据。

User Data Area 通常会进行再分区，例如 Android 系统中，通常在此区域分出 boot、system、userdata 等分区。

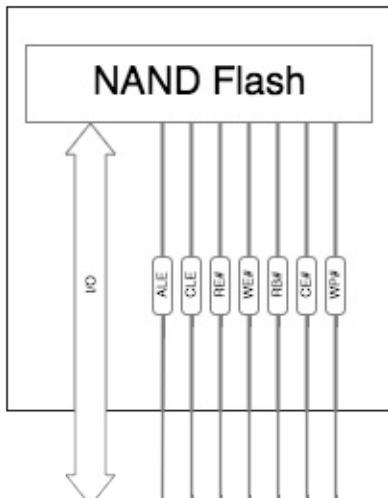
更多 eMMC 分区相关的细节，请参考 [eMMC 分区管理](#) 章节。

Flash Controller

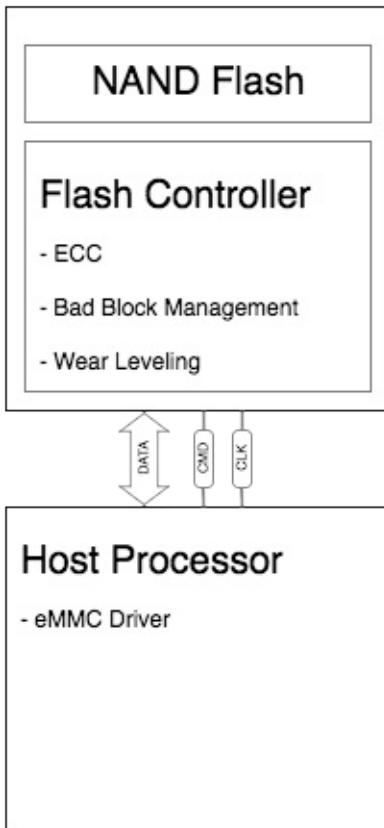
NAND Flash 直接接入 Host 时，Host 端通常需要有 NAND Flash Translation Layer，即 NFTL 或者 NAND Flash 文件系统来做坏块管理、ECC等的功能。

eMMC 则在其内部集成了 Flash Controller，用于完成擦写均衡、坏块管理、ECC校验等功能。相比于直接将 NAND Flash 接入到 Host 端，eMMC 屏蔽了 NAND Flash 的物理特性，可以减少 Host 端软件的复杂度，让 Host 端专注于上层业务，省去对 NAND Flash 进行特殊的处理。同时，eMMC 通过使用 Cache、Memory Array 等技术，在读写性能上也比 NAND Flash 要好很多。

NAND Flash



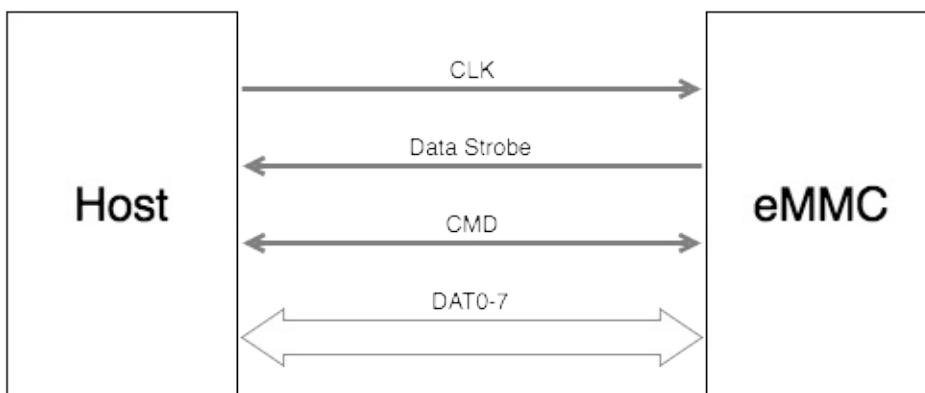
eMMC



图片：NAND Flash 与 eMMC

Host Interface

eMMC 与 Host 之间的连接如下图所示：



图片：eMMC Interface

各个信号的用途如下所示：

CLK

用于同步的时钟信号

Data Strobe

此信号是从 Device 端输出的时钟信号，频率和 CLK 信号相同，用于同步从 Device 端输出的数据。该信号在 eMMC 5.0 中引入。

CMD

此信号用于发送 Host 的 command 和 Device 的 response。

DAT0-7

用于传输数据的 8 bit 总线。

Host 与 eMMC 之间的通信都是 Host 以一个 Command 开始发起的。针对不同的 Command，Device 会做出不同的响应。
详细的通信协议相关内容，请参考 [eMMC 总线协议](#) 章节。

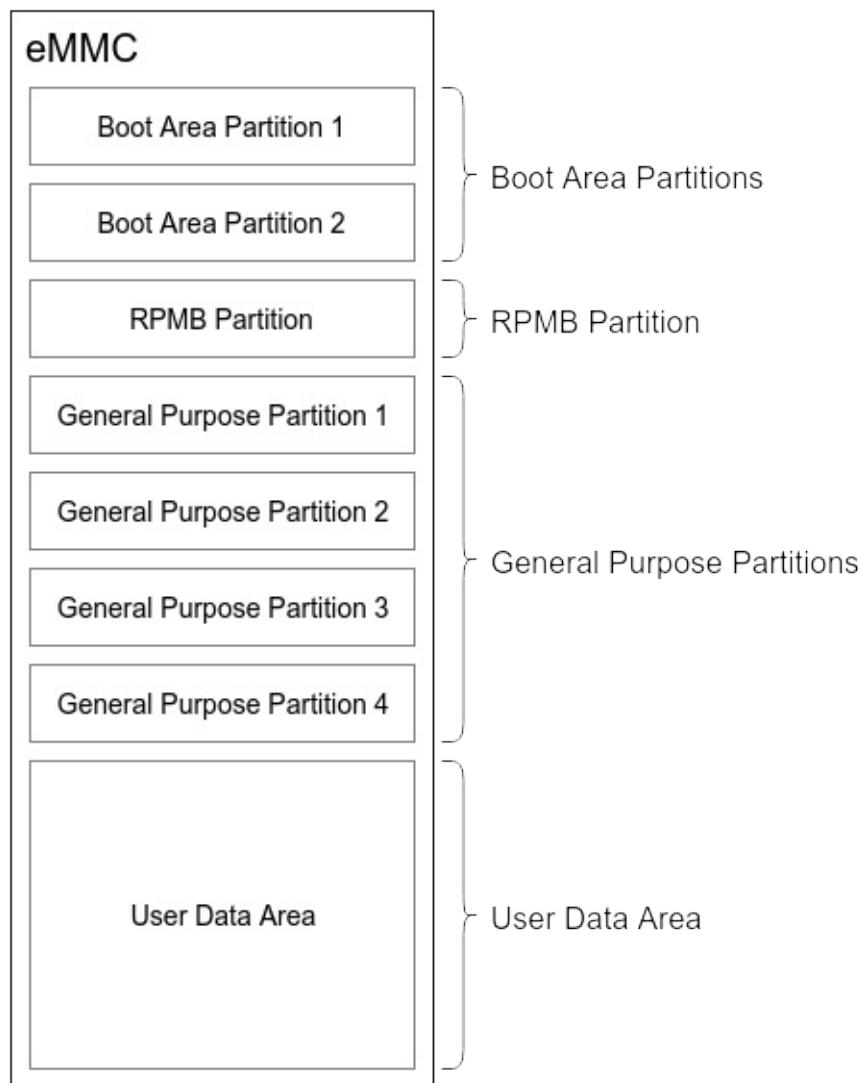
参考资料

1. [Embedded Multi-Media Card \(e•MMC\) Electrical Standard \(5.1\) \[PDF\]](#)

eMMC 分区管理

Partitions Overview

eMMC 标准中，将内部的 Flash Memory 划分为 4 类区域，最多可以支持 8 个硬件分区，如下图所示：



概述

一般情况下，Boot Area Partitions 和 RPMB Partition 的容量大小通常都为 4MB，部分芯片厂家也会提供配置的机会。General Purpose Partitions (GPP) 则在出厂时默认不被支持，即不存在这些分区，需要用户主动使能，并配置其所要使用的 GPP 的容量大小，GPP 的数量可以为 1 - 4 个，各个 GPP 的容量大小可以不一样。User Data Area (UDA) 的容量大小则为总容量大小减去其他分区所占用的容量。更多各个分区的细节将在后续小节中描述。

分区编址

eMMC 的每一个硬件分区的存储空间都是独立编址的，即访问地址为 0 - partition size。具体的数据读写操作实际访问哪一个硬件分区，是由 eMMC 的 Extended CSD register 的 PARTITION_CONFIG Field 中的 Bit[2:0]: PARTITION_ACCESS 决定的，用户可以通过配置 PARTITION_ACCESS 来切换硬件分区的访问。也就是说，用户在访问特定的分区前，需要先发送命令，配置 PARTITION_ACCESS，然后再发送相关的数据访问请求。更多数据读写相关的细节，请参考 [eMMC 总线协议](#) 章节。

eMMC 的各个硬件分区有其自身的功能特性，多分区的设计，为不同的应用场景提供了便利。

Boot Area Partitions

Boot Area 包含两个 Boot Area Partitions，主要用于存储 Bootloader，支持 SOC 从 eMMC 启动系统。

容量大小

两个 Boot Area Partitions 的大小是完全一致的，由 Extended CSD register 的 BOOT_SIZE_MULT Field 决定，大小的计算公式如下：

$$\text{Size} = 128\text{Kbytes} \times \text{BOOT_SIZE_MULT}$$

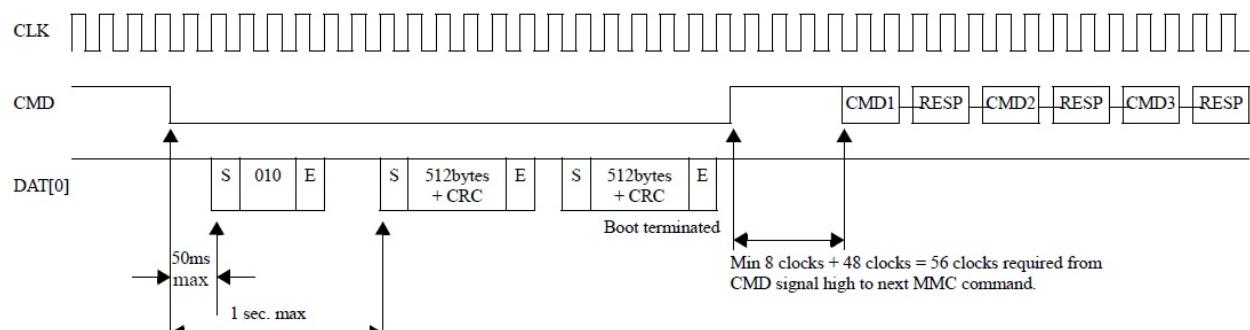
一般情况下，Boot Area Partition 的大小都为 4 MB，即 BOOT_SIZE_MULT 为 32，部分芯片厂家会提供改写 BOOT_SIZE_MULT 的功能来改变 Boot Area Partition 的容量大小。BOOT_SIZE_MULT 最大可以为 255，即 Boot Area Partition 的最大容量大小可以为 $255 \times 128\text{ KB} = 32640\text{ KB} = 31.875\text{ MB}$ 。

从 Boot Area 启动

eMMC 中定义了 Boot State，在 Power-up、HW reset 或者 SW reset 后，如果满足一定的条件，eMMC 就会进入该 State。进入 Boot State 的条件如下：

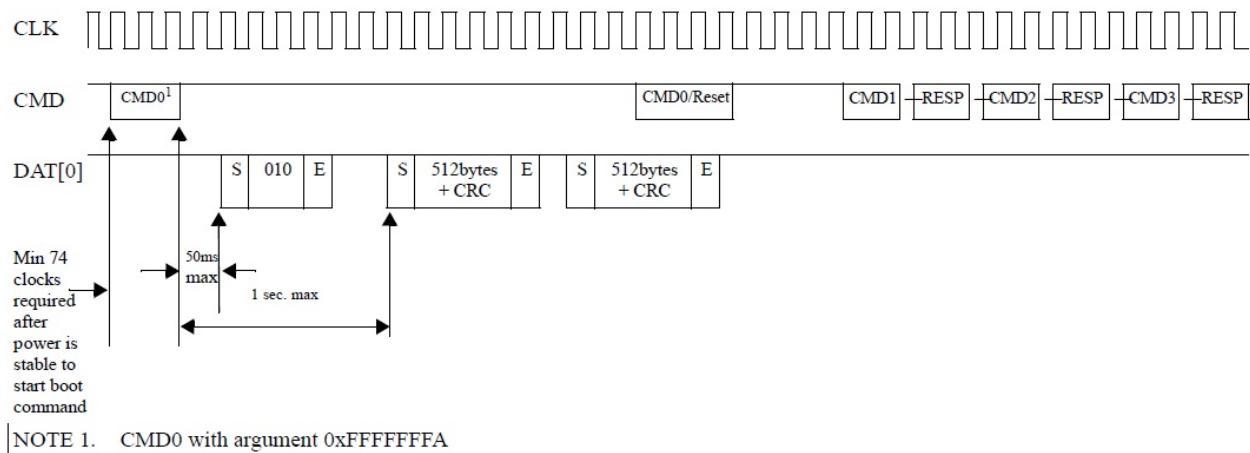
Original Boot Operation

CMD 信号保持低电平不少于 74 个时钟周期，会触发 Original Boot Operation，进入 Boot State。



Alternative Boot Operation

在 74 个时钟周期后，在 CMD 信号首次拉低或者 Host 发送参数为 0xFFFFFFF4 的 COM0 时，会触发 Alternative Boot Operation，进入 Boot State。



在 Boot State 下，如果有配置 BOOT_ACK，eMMC 会先发送“010”的 ACK 包，接着 eMMC 会将最大为 128Kbytes x BOOT_SIZE_MULT 的 Boot Data 发送给 Host。传输过程中，Host 可以通过拉高 CMD 信号 (Original Boot 中)，或者发送 Reset 命令 (Alternative Boot 中) 来中断 eMMC 的数据发送，完成 Boot Data 传输。

Boot Data 根据 Extended CSD register 的 PARTITION_CONFIG Field 的 Bit[5:3]:BOOT_PARTITION_ENABLE 的设定，可以从 Boot Area Partition 1、Boot Area Partition 2 或者 User Data Area 读出。

Boot Data 存储在 Boot Area 比在 User Data Area 中要更加的安全，可以减少意外修改导致系统无法启动，同时无法更新系统的情况出现。

(更多 Boot State 的细节，请参考 [eMMC 工作模式](#) 的 Boot Mode 章节)

写保护

通过设定 Extended CSD register 的 BOOT_WP Field，可以为两个 Boot Area Partition 独立配置写保护功能，以防止数据被意外改写或者擦出。

eMMC 中定义了两种 Boot Area 的写保护模式：

1. Power-on write protection，使能后，如果 eMMC 掉电，写保护功能失效，需要每次 Power on 后进行配置
2. Permanent write protection，使能后，即使掉电也不会失效，主动进行关闭才会失效

RPMB Partition

RPMB (Replay Protected Memory Block) Partition 是 eMMC 中的一个具有安全特性的分区。

eMMC 在写入数据到 RPMB 时，会校验数据的合法性，只有指定的 Host 才能够写入，同时在读数据时，也提供了签名机制，保证 Host 读取到的数据是 RPMB 内部数据，而不是攻击者伪造的数据。

RPMB 在实际应用中，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。RPMB 可以对写入操作进行鉴权，但是读取并不需要鉴权，任何人都可以进行读取的操作，因此存储到 RPMB 的数据通常会进行加密后再存储。

容量大小

两个 RPMB Partition 的大小是由 Extended CSD register 的 BOOT_SIZE_MULT Field 决定，大小的计算公式如下：

Size = 128Kbytes x BOOT_SIZE_MULT

一般情况下，Boot Area Partition 的大小为 4 MB，即 RPMB_SIZE_MULT 为 32，部分芯片厂家会提供改写 RPMB_SIZE_MULT 的功能来改变 RPMB Partition 的容量大小。RPMB_SIZE_MULT 最大可以为 128，即 Boot Area Partition 的最大容量大小可以为 $128 \times 128 \text{ KB} = 16384 \text{ KB} = 16 \text{ MB}$ 。

Replay Protect 原理

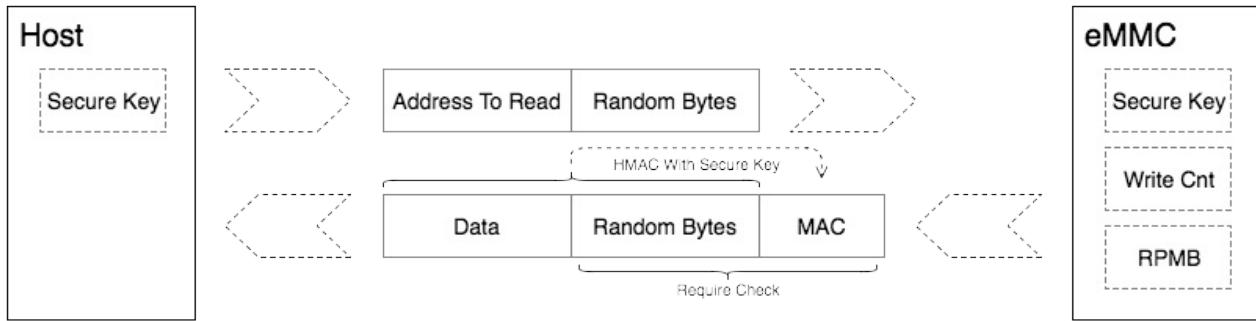
使用 eMMC 的产品，在产线生产时，会为每一个产品生产一个唯一的 256 bits 的 Secure Key，烧写到 eMMC 的 OTP 区域（只能烧写一次的区域），同时 Host 在安全区域中（例如：TEE）也会保留该 Secure Key。

在 eMMC 内部，还有一个 RPMB Write Counter。RPMB 每进行一次合法的写入操作时，Write Counter 就会自动加一。

通过 Secure Key 和 Write Counter 的应用，RPMB 可以实现数据读取和写入的 Replay Protect。

RPMB 数据读取

RPMB 数据读取的流程如下：

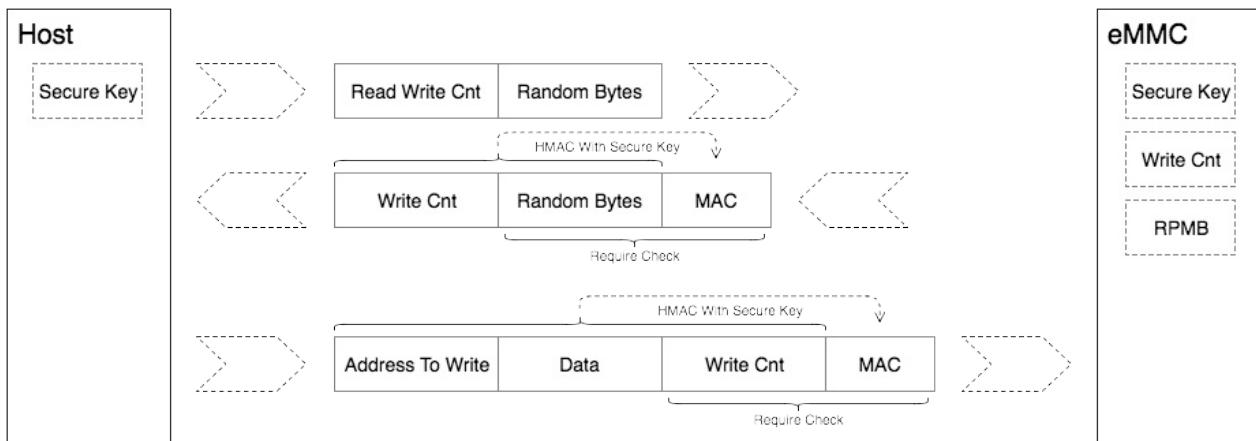


1. Host 向 eMMC 发起读 RPMB 的请求，同时生成一个 16 bytes 的随机数，发送给 eMMC。
2. eMMC 将请求的数据从 RPMB 中读出，并使用 Secure Key 通过 HMAC SHA-256 算法，计算读取到的数据和接收到的随机数拼接到一起后的签名。然后，eMMC 将读取到的数据、接收到的随机数、计算得到的签名一并发送给 Host。
3. Host 接收到 RPMB 的数据、随机数以及签名后，首先比较随机数是否与自己发送的一致，如果一致，再用同样的 Secure Key 通过 HMAC SHA-256 算法对数据和随机数组合到一起进行签名，如果签名与 eMMC 发送的签名是一致的，那么就可以确定该数据是从 RPMB 中读取到的正确数据，而不是攻击者伪造的数据。

通过上述的读取流程，可以保证 Host 正确的读取到 RPMB 的数据。

RPMB 数据写入

RPMB 数据写入的流程如下：



1. Host 按照上面的读数据流程，读取 RPMB 的 Write Counter。
2. Host 将需要写入的数据和 Write Counter 拼接到一起并计算签名，然后将数据、Write Counter 以及签名一并发给 eMMC。
3. eMMC 接收到数据后，先对比 Write Counter 是否与当前的值相同，如果相同那么再对数据和 Write Counter 的组合进行签名，然后和 Host 发送过来的签名进行比较，如果签名相同则鉴权通过，将数据写入到 RPMB 中。

通过上述的写入流程，可以保证 RPMB 不会被非法篡改。

更多 RPMB 相关的细节，可以参考 [eMMC RPMB 章节](#)。

General Purpose Partitions

eMMC 提供了 General Purpose Partitions (GPP)，主要用于存储系统和应用数据。在很多使用 eMMC 的产品中，GPP 都没有被启用，因为它在功能上与 UDA 类似，产品上直接使用 UDA 就可以满足需求。

容量大小

eMMC 最多可以支持 4 个 GPPs，每一个 GPP 的大小可以单独配置。用户可以通过设定 Extended CSD register 的以下三个 Field 来设 GPPx (x=1~4) 的容量大小：

- GP_SIZE_MULT_x_2
- GP_SIZE_MULT_x_1
- GP_SIZE_MULT_x_0

GPPx 的容量计算公式如下：

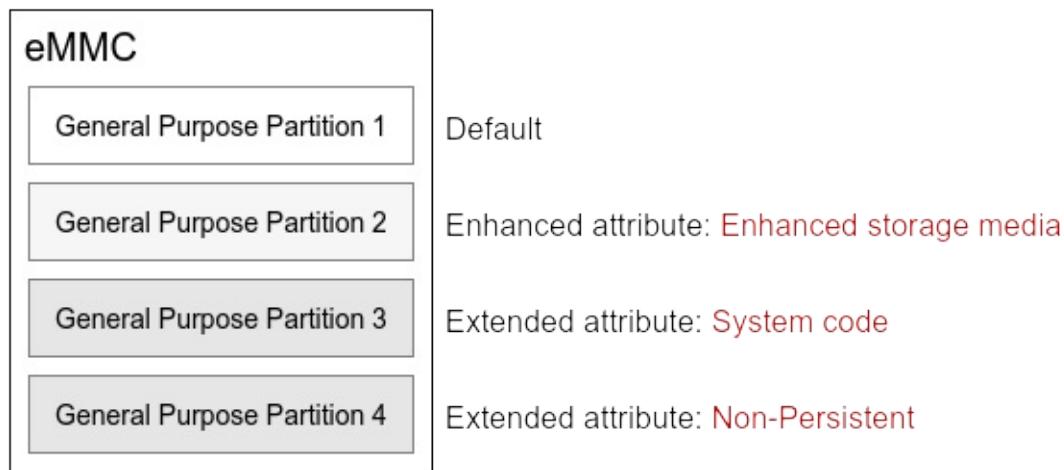
$\text{Size} = (\text{GP_SIZE_MULT_x_2} * 2^{16} + \text{GP_SIZE_MULT_x_1} * 2^8 + \text{GP_SIZE_MULT_x_0} * 2^0) * (\text{Write protect group size})$

$\text{Write protect group size} = 512\text{KB} * \text{HC_ERASE_GRP_SIZE} * \text{HC_WP_GRP_SIZE}$

- eMMC 中，擦除和写保护都是按块进行的，上述表达式中的 HC_WP_GRP_SIZE 为写保护的操作块大小， HC_ERASE_GRP_SIZE 则为擦除操作的快的大小。
- eMMC 芯片的 GPP 的配置通常是只能进行一次 (OTP)，一般会在产品量产阶段，在产线上进行。

分区属性

eMMC 标准中，为 GPP 定义了两类属性，Enhanced attribute 和 Extended attribute。每个 GPP 可以设定两类属性中的一种属性，不可以同时设定多个属性。



Enhanced attribute

- Default, 未设定 Enhanced attribute。
- Enhanced storage media, 设定 GPP 为 Enhanced storage media。

在 eMMC 标准中，实际上并未定义设定 Enhanced attribute 后对 eMMC 的影响。Enhanced attribute 的具体作用，由芯片制造商定义。

在实际的产品中，设定 Enhanced storage media 后，一般是把该分区的存储介质从 MLC 改变为 SLC，提高该分区的读写性能、寿命以及稳定性。由于 1 个存储单元下，MLC 的容量是 SLC 的两倍，所以在总的存储单元数量一定的情况下，如果把原本为 MLC 的分区改变为 SLC，会减少 eMMC 的容量，就是说，此时 eMMC 的实际总容量比标称的总容量会小一点。
(MLC 和 SLC 的细节可以参考 [Flash Memory](#) 章节内容)

Extended attribute

- Default, 未设定 Extended attribute。
- System code，设定 GPP 为 System code 属性，该属性主要用在存放操作系统类的、很少进行擦写更新的分区。
- Non-Persistent，设定 GPP 为 Non-Persistent 属性，该属性主要用于存储临时数据的分区，例如 tmp 目录所在分区、swap 分区等。

在 eMMC 标准中，同样也没有定义设定 Extended attribute 后对 eMMC 的影响。Extended attribute 的具体作用，由芯片制造商定义。Extended attribute 主要是跟分区的应用场景有关，厂商可以为不用应用场景的分区做不同的优化处理。

User Data Area

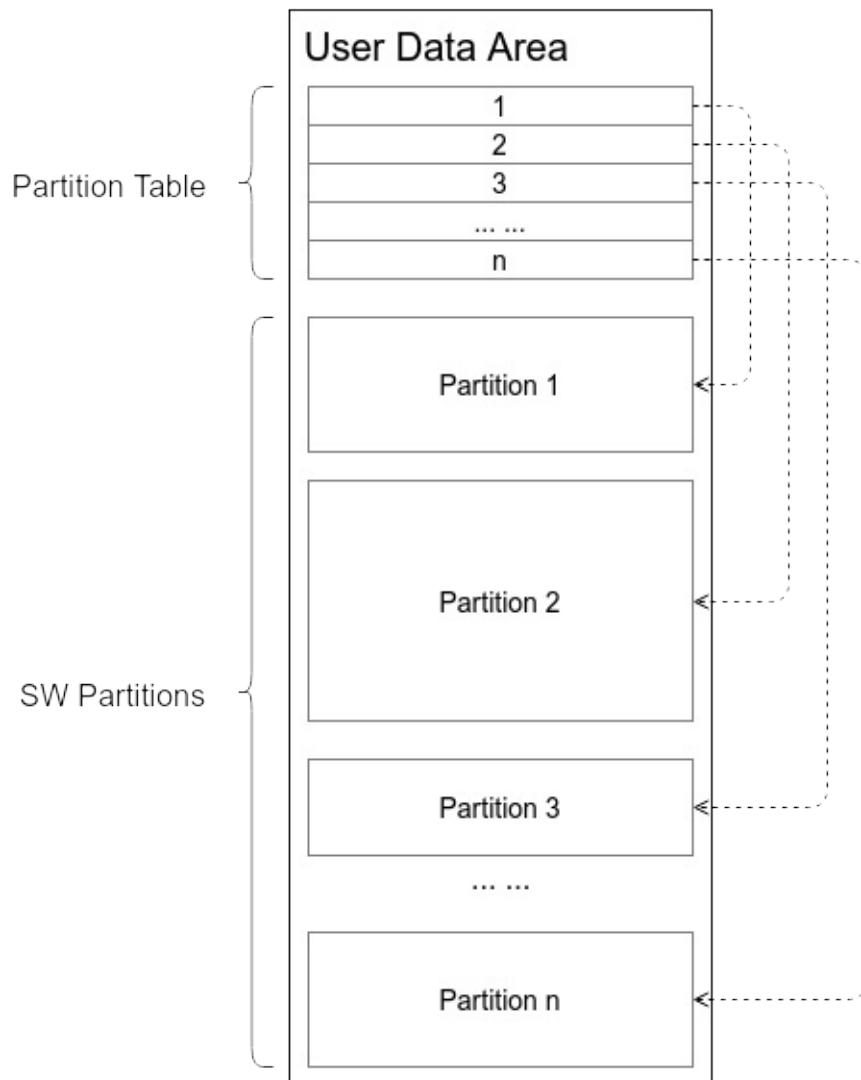
User Data Area (UDA) 通常是 eMMC 中最大的一个分区，是实际产品中，最主要的存储区域。

容量大小

UDA 的容量大小不需要设置，在配置完其他分区大小后，再扣除设置 Enhanced attribute 所损耗的容量，剩下的容量就是 UDA 的容量。

软件分区

为了更合理的管理数据，满足不同的应用需求，UDA 在实际产品中，会进行软件再分区。目前主流的软件分区技术有 MBR (Master Boot Record) 和 GPT (GUID Partition Table) 两种。这两种分区技术的基本原理类似，如下图所示：



软件分区技术一般是将存储介质划分为多个区域，既 SW Partitions，然后通过一个 Partition Table 来维护这些 SW Partitions。在 Partition Table 中，每一个条目都保存着一个 SW Partition 的起始地址、大小等的属性信息。软件系统在启动后，会去扫描 Partition Table，获取存储介质上的各个 SW Partitions 信息，然后根据这些信息，将各个 Partitions 加载到系统中，进行数据存取。

MBR 和 GPT 此处不展开详细介绍，更多的细节可以参考 wikipedia 上 [MBR](#) 和 [GPT](#) 相关介绍。

区域属性

eMMC 标准中，支持为 UDA 中一个特定大小的区域设定 Enhanced attribute。与 GPP 中的 Enhanced attribute 相同，eMMC 标准也没有定义该区域设定 Enhanced attribute 后对 eMMC 的影响。Enhanced attribute 的具体作用，由芯片制造商定义。

Enhanced attribute

- Default, 未设定 Enhanced attribute。
- Enhanced storage media，设定该区域为 Enhanced storage media。

在实际的产品中，UDA 区域设定为 Enhanced storage media 后，一般是把该区域的存储介质从 MLC 改变为 SLC。通常，产品中可以将某一个 SW Partition 设定为 Enhanced storage media，以获得更好的性能和健壮性。

eMMC 分区应用实例

在一个 Android 手机系统中，各个分区的呈现形式如下：

- mmcblk0 为 eMMC 的块设备；
- mmcblk0boot0 和 mmcblk0boot1 对应两个 Boot Area Partitions；
- mmcblk0rpmb 则为 RPMB Partition；
- mmcblk0px 为 UDA 划分出来的 SW Partitions；
- 如果存在 GPP，名称则为 mmcblk0gp1、mmcblk0gp2、mmcblk0gp3、mmcblk0gp4；

```
root@xxx:/ # ls /dev/block/mmcblk0*
/dev/block/mmcblk0
/dev/block/mmcblk0boot0
/dev/block/mmcblk0boot1
/dev/block/mmcblk0rpmb
/dev/block/mmcblk0p1
/dev/block/mmcblk0p2
/dev/block/mmcblk0p3
/dev/block/mmcblk0p4
/dev/block/mmcblk0p5
/dev/block/mmcblk0p6
/dev/block/mmcblk0p7
/dev/block/mmcblk0p8
/dev/block/mmcblk0p9
/dev/block/mmcblk0p10
/dev/block/mmcblk0p11
/dev/block/mmcblk0p12
/dev/block/mmcblk0p13
/dev/block/mmcblk0p14
/dev/block/mmcblk0p15
/dev/block/mmcblk0p16
/dev/block/mmcblk0p17
/dev/block/mmcblk0p18
/dev/block/mmcblk0p19
/dev/block/mmcblk0p20
/dev/block/mmcblk0p21
/dev/block/mmcblk0p22
/dev/block/mmcblk0p23
/dev/block/mmcblk0p24
/dev/block/mmcblk0p25
/dev/block/mmcblk0p26
/dev/block/mmcblk0p27
/dev/block/mmcblk0p28
/dev/block/mmcblk0p29
/dev/block/mmcblk0p30
/dev/block/mmcblk0p31
/dev/block/mmcblk0p32
```

每一个分区会根据实际的功能来设定名称。

```
root@xxx:/ # ls -l /dev/block/platform/mtk-msdc.0/11230000.msd0/by-name/
lrwxrwxrwx root root 2015-01-03 04:03 boot -> /dev/block/mmcblk0p22
lrwxrwxrwx root root 2015-01-03 04:03 cache -> /dev/block/mmcblk0p30
lrwxrwxrwx root root 2015-01-03 04:03 custom -> /dev/block/mmcblk0p3
lrwxrwxrwx root root 2015-01-03 04:03 devinfo -> /dev/block/mmcblk0p28
lrwxrwxrwx root root 2015-01-03 04:03 expdb -> /dev/block/mmcblk0p4
lrwxrwxrwx root root 2015-01-03 04:03 flashinfo -> /dev/block/mmcblk0p32
lrwxrwxrwx root root 2015-01-03 04:03 frp -> /dev/block/mmcblk0p5
lrwxrwxrwx root root 2015-01-03 04:03 keystore -> /dev/block/mmcblk0p27
lrwxrwxrwx root root 2015-01-03 04:03 lk -> /dev/block/mmcblk0p20
lrwxrwxrwx root root 2015-01-03 04:03 lk2 -> /dev/block/mmcblk0p21
lrwxrwxrwx root root 2015-01-03 04:03 logo -> /dev/block/mmcblk0p23
lrwxrwxrwx root root 2015-01-03 04:03 md1arm7 -> /dev/block/mmcblk0p17
lrwxrwxrwx root root 2015-01-03 04:03 md1dsp -> /dev/block/mmcblk0p16
lrwxrwxrwx root root 2015-01-03 04:03 md1img -> /dev/block/mmcblk0p15
lrwxrwxrwx root root 2015-01-03 04:03 md3img -> /dev/block/mmcblk0p18
lrwxrwxrwx root root 2015-01-03 04:03 metadata -> /dev/block/mmcblk0p8
lrwxrwxrwx root root 2015-01-03 04:03 nvdata -> /dev/block/mmcblk0p7
lrwxrwxrwx root root 2015-01-03 04:03 nvram -> /dev/block/mmcblk0p19
lrwxrwxrwx root root 2015-01-03 04:03 oemkeystore -> /dev/block/mmcblk0p12
lrwxrwxrwx root root 2015-01-03 04:03 para -> /dev/block/mmcblk0p2
lrwxrwxrwx root root 2015-01-03 04:03 ppl -> /dev/block/mmcblk0p6
lrwxrwxrwx root root 2015-01-03 04:03 proinfo -> /dev/block/mmcblk0p13
lrwxrwxrwx root root 2015-01-03 04:03 protect1 -> /dev/block/mmcblk0p9
lrwxrwxrwx root root 2015-01-03 04:03 protect2 -> /dev/block/mmcblk0p10
lrwxrwxrwx root root 2015-01-03 04:03 recovery -> /dev/block/mmcblk0p1
lrwxrwxrwx root root 2015-01-03 04:03 rstinfo -> /dev/block/mmcblk0p14
lrwxrwxrwx root root 2015-01-03 04:03 seccfg -> /dev/block/mmcblk0p11
lrwxrwxrwx root root 2015-01-03 04:03 secre -> /dev/block/mmcblk0p26
lrwxrwxrwx root root 2015-01-03 04:03 system -> /dev/block/mmcblk0p29
lrwxrwxrwx root root 2015-01-03 04:03 tee1 -> /dev/block/mmcblk0p24
lrwxrwxrwx root root 2015-01-03 04:03 tee2 -> /dev/block/mmcblk0p25
lrwxrwxrwx root root 2015-01-03 04:03 userdata -> /dev/block/mmcblk0p31
```

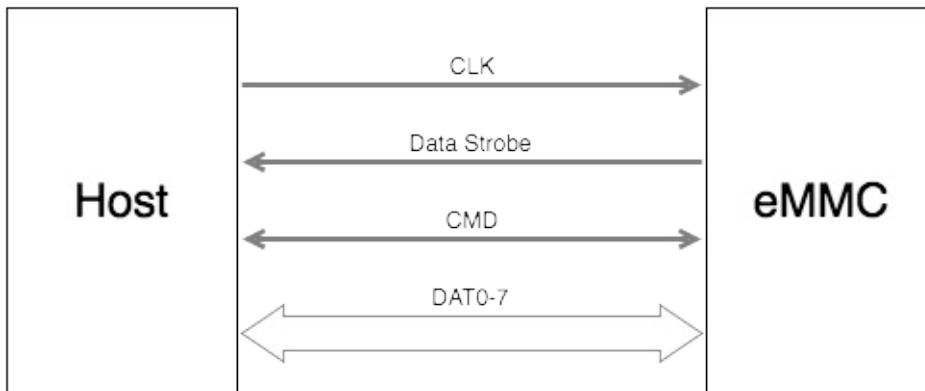
参考资料

1. [Embedded Multi-Media Card \(e•MMC\) Electrical Standard \(5.1\)](#) [PDF]
2. [Disk partitioning](#) [Web]
3. [Master Boot Record](#) [Web]
4. [GUID Partition Table](#) [Web]

eMMC 总线协议

eMMC 总线接口

eMMC 总线接口定义如下图所示：



各个信号的描述如下：

CLK

CLK 信号用于从 Host 端输出时钟信号，进行数据传输的同步和设备运作的驱动。

在一个时钟周期内，CMD 和 DAT0-7 信号上都可以支持传输 1 个比特，即 SDR (Single Data Rate) 模式。此外，DAT0-7 信号还支持配置为 DDR (Double Data Rate) 模式，在一个时钟周期内，可以传输 2 个比特。

Host 可以在通讯过程中动态调整时钟信号的频率（注，频率范围需要满足 Spec 的定义）。通过调整时钟频率，可以实现省电或者数据流控（避免 Over-run 或者 Under-run）功能。在一些场景中，Host 端还可以关闭时钟，例如 eMMC 处于 Busy 状态时，或者接收完数据，进入 Programming State 时。

CMD

CMD 信号主要用于 Host 向 eMMC 发送 Command 和 eMMC 向 Host 发送对应的 Response。Command 和 Response 的细节会在后续章节中介绍。

DAT0-7

DAT0-7 信号主要用于 Host 和 eMMC 之间的数据传输。在 eMMC 上电或者软复位后，只有 DAT0 可以进行数据传输，完成初始化后，可配置 DAT0-3 或者 DAT0-7 进行数据传输，即数据总线可以配置为 4 bits 或者 8 bits 模式。

Data Strobe

Data Strobe 时钟信号由 eMMC 发送给 Host，频率与 CLK 信号相同，用于 Host 端进行数据接收的同步。Data Strobe 信号只能在 HS400 模式下配置启用，启用后可以提高数据传输的稳定性，省去总线 tuning 过程。

NOTE:

Extended CSD byte[183] BUS_WIDTH 寄存器用于配置总线宽度和 Data Strobe

eMMC 总线模型

eMMC 总线中，可以有一个 Host，多个 eMMC Devices。总线上的所有通讯都由 Host 端以一个 Command 开发发起，Host 一次只能与一个 eMMC Device 通讯。

系统在上电启动后，Host 会为所有 eMMC Device 逐个分配地址（RCA，Relative device Address）。当 Host 需要和某一个 eMMC Device 通讯时，会先根据 RCA 选中该 eMMC Device，只有被选中的 eMMC Device 才会响应 Host 的 Command。

NOTE:

更详细的工作原理请参考 [eMMC 工作模式](#) 章节。

速率模式

随着 eMMC 协议的版本迭代，eMMC 总线的速率越来越高。为了兼容旧版本的 eMMC Device，所有 Devices 在上电启动或者 Reset 后，都会先进入兼容速率模式（Backward Compatible Mode）。在完成 eMMC Devices 的初始化后，Host 可以通过特定的流程，让 Device 进入其他高速率模式，目前支持以下的几种速率模式。

Mode	Data Rate	Bus Width	Frequency	Max Data Transfer (x8)
Backward Compatible	Single	x1, x4, x8	0-26 MHz	26 MB/s
High Speed SDR	Single	x1, x4, x8	0-52 MHz	52 MB/s
High Speed DDR	Dual	x4, x8	0-52 MHz	104 MB/s
HS200	Single	x4, x8	0-200 MHz	200 MB/s
HS400	Dual	x8	0-200 MHz	400 MB/s

NOTE:

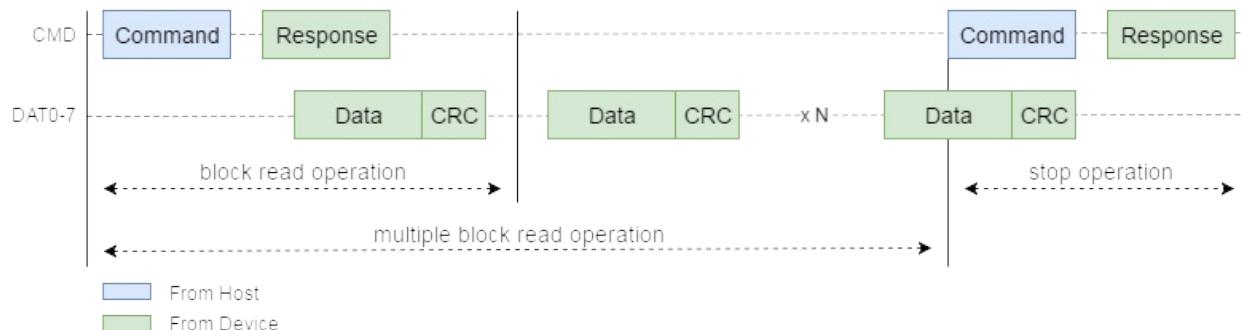
Extended CSD byte[185] HS_TIMING 寄存器可以配置总线速率模式

Extended CSD byte[183] BUS_WIDTH 寄存器用于配置总线宽度和 Data Strobe

通信模型

Host 与 eMMC Device 之间的通信都是由 Host 以一个 Command 开始发起的，eMMC Device 在完成 Command 所指定的任务后，则返回一个 Response。

Read Data



Host 从 eMMC Device 读取数据的流程如上图所示。

如果 Host 发送的是 Single Block Read 的 Command，那么 eMMC Device 只会发送一个 Block 的数据。

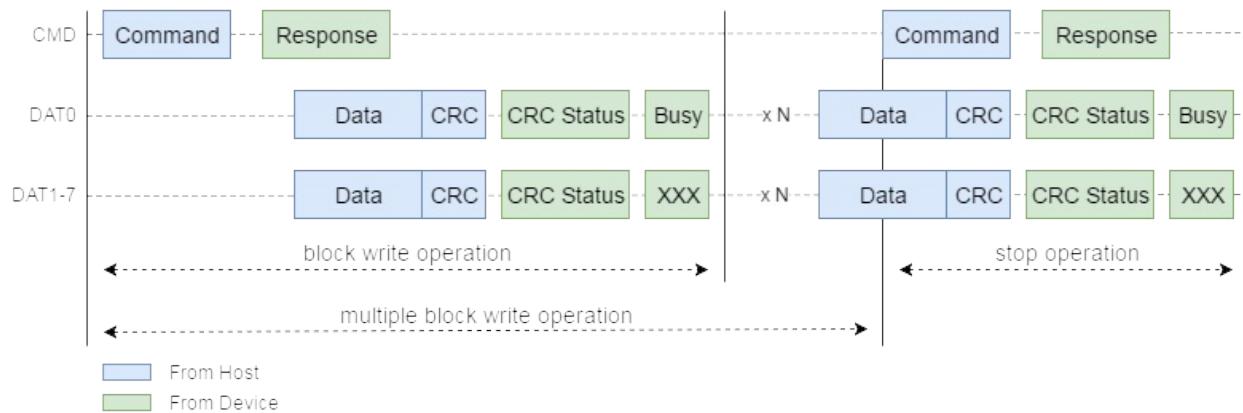
如果 Host 在发送 Multiple Block Read 的 Command 前，先发送一个设定需要读取的 Block Count 的 Command。eMMC Device 在完成指定 Block Count 的数据发送后，就自动结束数据传输，不需要 Host 主动发送 Stop Command。

如果 Host 没有发送设定需要读取的 Block Count 的 Command，发送 Multiple Block Read 的 Command 后，eMMC Device 会持续发送数据，直到 Host 发送 Stop Command 停止数据传输。

NOTE:

从 eMMC Device 读数据都是按 Block 读取的。Block 大小可以由 Host 设定，或者固定为 512 Bytes，不同的速率模式下有所不同。

Write Data



Host 向 eMMC Device 写入数据的流程如上图所示。

如果 Host 发送的是 Single Block Write Command，那么 eMMC Device 只会将后续第一个 Block 的数据写入的存储器中。

如果 Host 在发送 Multiple Block Write 的 Command 前，先发送一个设定需要读取的 Block Count 的 Command。eMMC Device 在接收到指定 Block Count 的数据后，就自动结束数据接收，不需要 Host 主动发送 Stop Command。

如果 Host 没有发送设定需要读取的 Block Count 的 Command，发送 Multiple Block Write 的 Command 后，eMMC Device 会持续接收数据，直到 Host 发送 Stop Command 停止数据传输。

eMMC Device 在接收到一个 Block 的数据后，会进行 CRC 校验，然后将校验结果通过 CRC Token 发送给 Host。

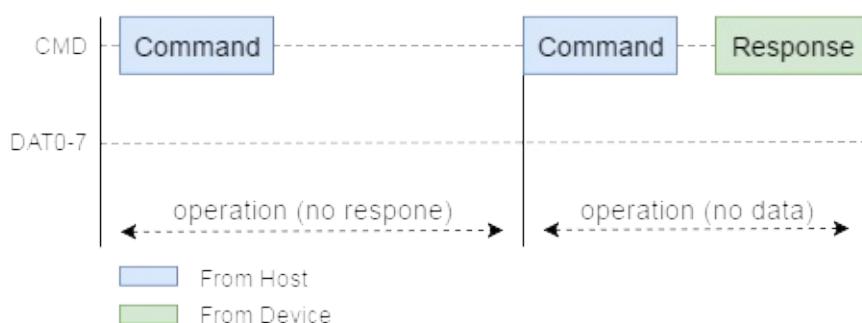
发送完 CRC Token 后，如果 CRC 校验成功，eMMC Device 会将数据写入到内部存储器时，此时 DAT0 信号会拉低，作为 Busy 信号。Host 会持续检测 DAT0 信号，直到为高电平时，才会接着发送下一个 Block 的数据。如果 CRC 校验失败，那么 eMMC Device 不会进行数据写入，此次传输后续的数据都会被忽略。

NOTE:

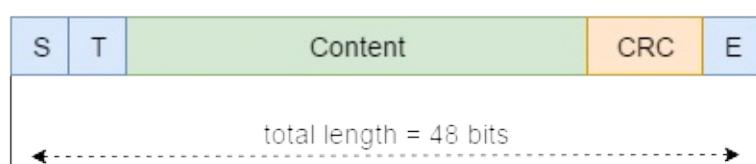
向 eMMC Device 写数据都是按 Block 写入的。Block 大小可以由 Host 设定，或者固定为 512 Bytes，不同的速率模式下有所不同。

No Data

在 Host 与 eMMC Device 的通信中，有部分交互是不需要进行数据传输的，还有部分交互甚至不需要 eMMC Device 的回复 Response。



Command



如上图所示，eMMC Command 由 48 Bits 组成，各个 Bits 的解析如下所示：

Description	Start Bit	Transmission Bit	Command Index	Argument	CRC7	End Bit
Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	"0"	"1"	x	x	x	"1"

Start Bit 固定为 "0"，在没有数据传输的情况下，CMD 信号保持高电平，当 Host 将 Start Bit 发送到总线上时，eMMC Device 可以很方便检测到该信号，并开始接收 Command。

Transmission Bit 固定为 "1"，指示了该数据包的传输方向为 Host 发送到 eMMC Device。

Command Index 和 Argument 为 Command 的具体内容，不同的 Command 有不同的 Index，不同的 Command 也有各自的 Argument。更多的细节，请参考 [eMMC Commands](#) 章节。

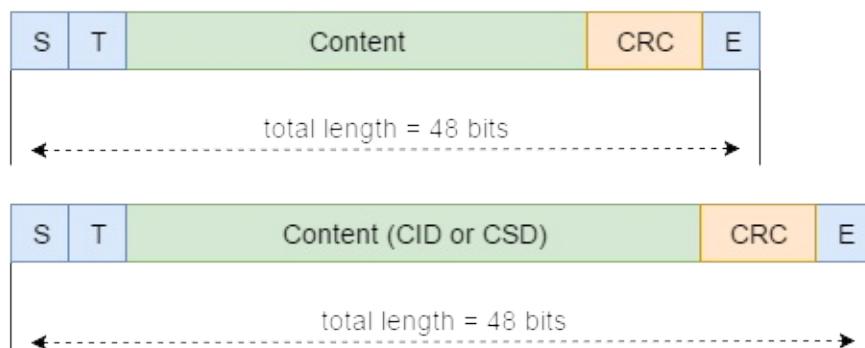
CRC7 是包含 Start Bit、Transmission Bit、Command Index 和 Argument 内容的 CRC 校验值。

End Bit 为结束标志位，固定为"1"。

NOTE:

CRC 校验简单来说，是发送方将需要传输的数据“除于”（模2除）一个约定的数，并将得到的余数附在数据上一并发送出去。接收方收到数据后，再做同样的“除法”，然后校验得到余数是否与接收的余数相同。如果不相同，那么意味着数据在传输过程中发生了改变。更多的细节不在本文展开描述，感兴趣的读者可以参考 [CRC wiki](#) 中的介绍。

Response



eMMC Response 有两种长度的数据包，分别为 48 Bits 和 136 Bits。

Start Bit 与 Command 一样，固定为 "0"，在没有数据传输的情况下，CMD 信号保持高电平，当 eMMC Device 将 Start Bit 发送到总线上时，Host 可以很方便检测到该信号，并开始接收 Response。

Transmission Bit 固定为 "0"，指示了该数据包的传输方向为 eMMC Device 发送到 Host。

Content 为 Response 的具体内容，不同的 Command 会有不同的 Content。更多的细节，请参考 [eMMC Responses](#) 章节。

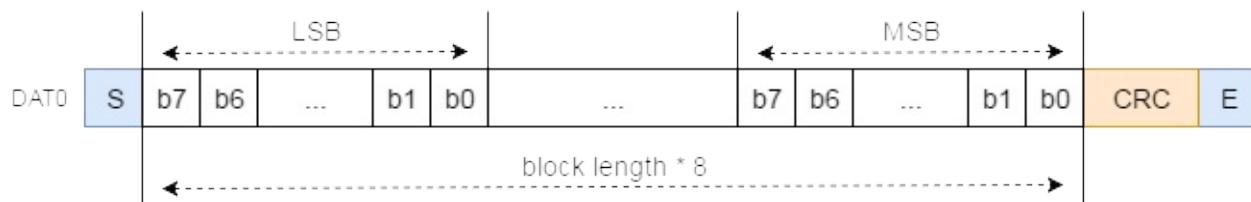
CRC7 是包含 Start Bit、Transmission Bit 和 Content 内容的 CRC 校验值。

End Bit 为结束标志位，固定为"1"。

Data Block

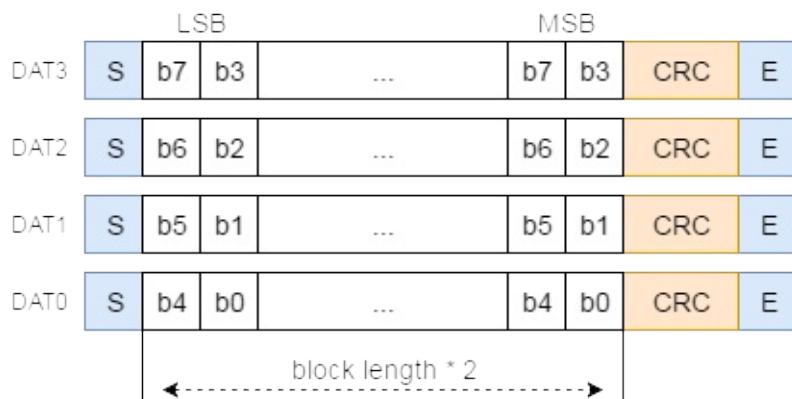
Data Block 由 Start Bit、Data、CRC16 和 End Bit 组成。以下是不同总线宽度和 Data Rate 下，Data Block 详细格式。

1 Bit Bus SDR



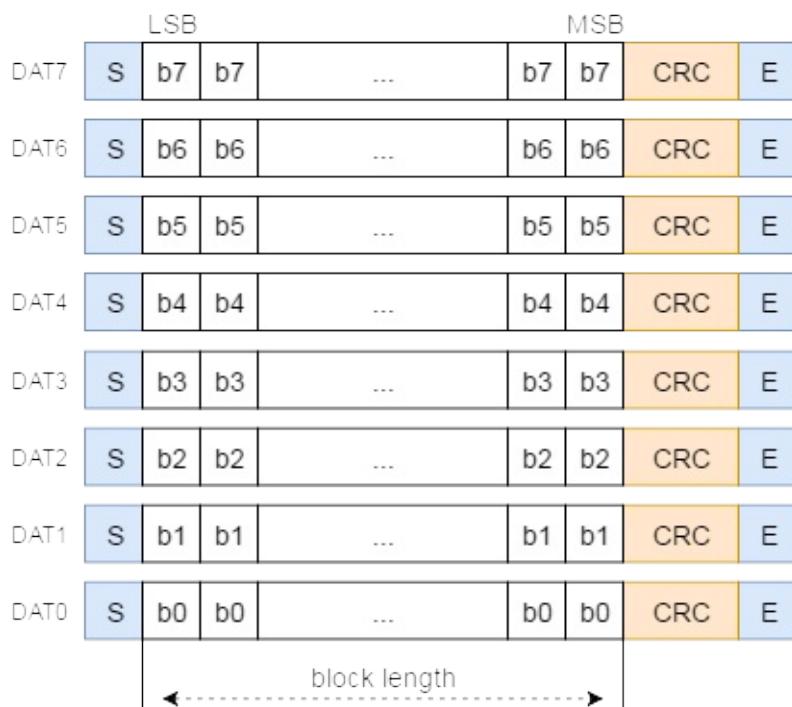
CRC 为 Data 的 16 bit CRC 校验值，不包含 Start Bit。

4 Bits Bus SDR



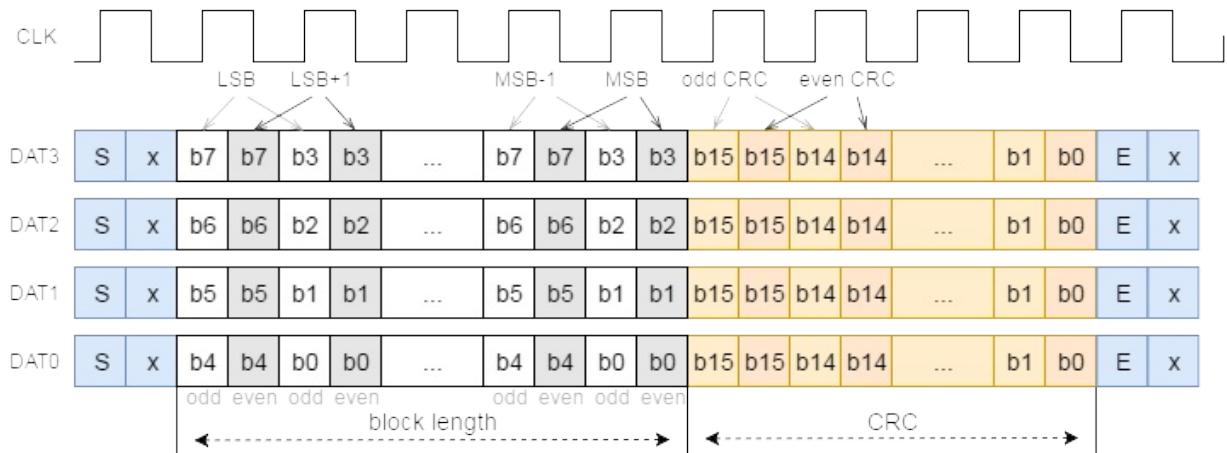
各个 Data Line 上的 CRC 为对应 Data Line 的 Data 的 16 bit CRC 校验值。

8 Bits Bus SDR

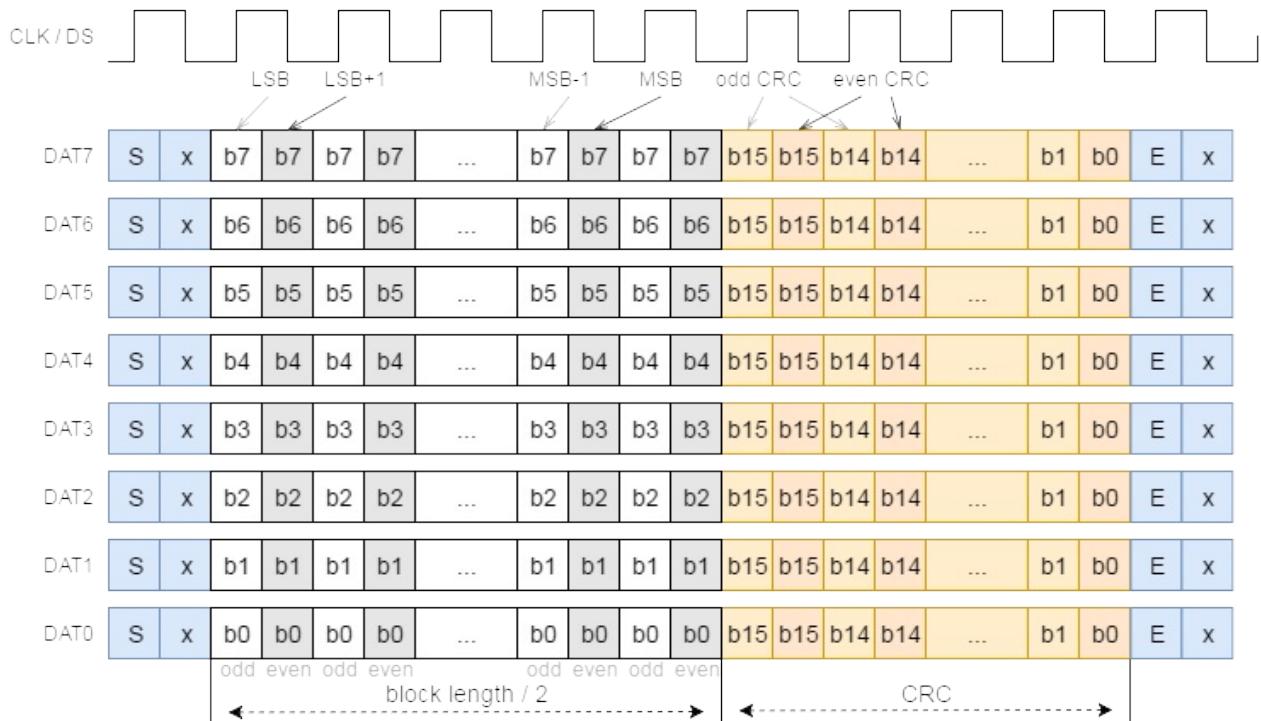


各个 Data Line 上的 CRC 为对应 Data Line 的 Data 的 16 bit CRC 校验值。

4 Bits Bus DDR



8 Bits Bus DDR



在 DDR 模式下，Data Line 在时钟的上升沿和下降沿都会传输数据，其中上升沿传输数据的奇数字节（Byte 1,3,5 ...），下降沿则传输数据的偶数字节（Byte 2,4,6 ...）。

此外，在 DDR 模式下，1 个 Data Line 上有两个相互交织的 CRC16，上升沿的 CRC 比特组成 odd CRC16，下降沿的 CRC 比特组成 even CRC16。odd CRC16 用于校验该 Data Line 上所有上升沿比特组成的数据，even CRC16 则用于校验该 Data Line 上所有下降沿比特组成的数据。

NOTE:

DDR 模式下使用两个 CRC16 作为校验，可能是为了更可靠的校验，选用 CRC16 而非 CRC32 则可能是出于兼容性设计的考虑。

CRC Status Token

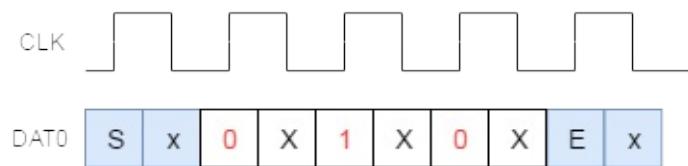
在写数据传输中，eMMC Device 接收到 Host 发送的一个 Data Block 后，会进行 CRC 校验，如果校验成功，eMMC 会在对应的 Data Line 上向 Host 发回一个 Positive CRC status token (010)，如果校验失败，则会在对应的 Data Line 上发送一个 Negative CRC status token (101)。

NOTE:

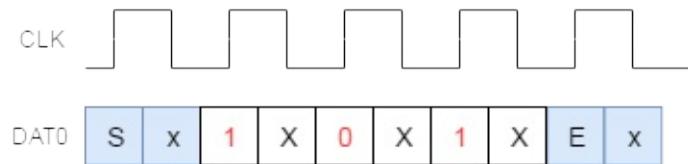
读数据时，Host 接收到 eMMC Device 发送的 Data Block 后，也会进行 CRC 校验，但是不管校验成功或者失败，都不会向 eMMC Device 发送 CRC Status Token。

详细格式如下图所示：

Positive CRC status token



Negative CRC status token



eMMC 总线测试过程

当 eMMC Device 处于 SDR 模式时，Host 可以发送 CMD19 命令，触发总线测试过程（Bus testing procedure），测试总线硬件上的连通性。如果 eMMC Device 支持总线测试，那么 eMMC Device 在接收到 CMD19 后，会发回对应的 Response，接着 eMMC Device 会发送一组固定的测试数据给 Host。Host 接收到数据后，检查数据正确与否，即可得知总线是否正确连通。

NOTE: 如果 eMMC Device 不支持总线测试，那么接收到 CMD19 时，不会发回 Response。

总线测试不支持在 DDR 模式下进行。

测试数据如下所示：

	LSB								MSB	
DAT7	S	0	1	0	0	0	0	0	CRC	E
DAT6	S	1	0	0	0	0	0	0	CRC	E
DAT5	S	0	1	0	0	0	0	0	CRC	E
DAT4	S	1	0	0	0	0	0	0	CRC	E
DAT3	S	0	1	0	0	0	0	0	CRC	E
DAT2	S	1	0	0	0	0	0	0	CRC	E
DAT1	S	0	1	0	0	0	0	0	CRC	E
DAT0	S	1	0	0	0	0	0	0	CRC	E
	0x55 0xAA 0x00 0x00 0x00 0x00 0x00 0x00								0x00 0x00	

NOTE: 总线宽度为 1 时，只发送 DAT0 上的数据，总线宽度为 4 时，则只发送 DAT0-3 上的数据

eMMC 总线 Sampling Tuning

由于芯片制造工艺、PCB 走线、电压、温度等因素的影响，数据信号从 eMMC Device 到达 Host 端的时间是存在差异的，Host 接收数据时采样的时间点也需要相应的进行调整。而 Host 端最佳采样时间点，则是通过 Sampling Tuning 流程得到。

NOTE:

不同 eMMC Device 最佳的采样点可能不同，同一 eMMC Device 在不同的环境下运作时的最佳采样点也可能不同。在 eMMC 标准中，定义了在 HS200 模式下可以进行 Sampling Tuning。

Sampling Tuning 流程

Sampling Tuning 是用于计算 Host 最佳采样时间点的流程，大致的流程如下：

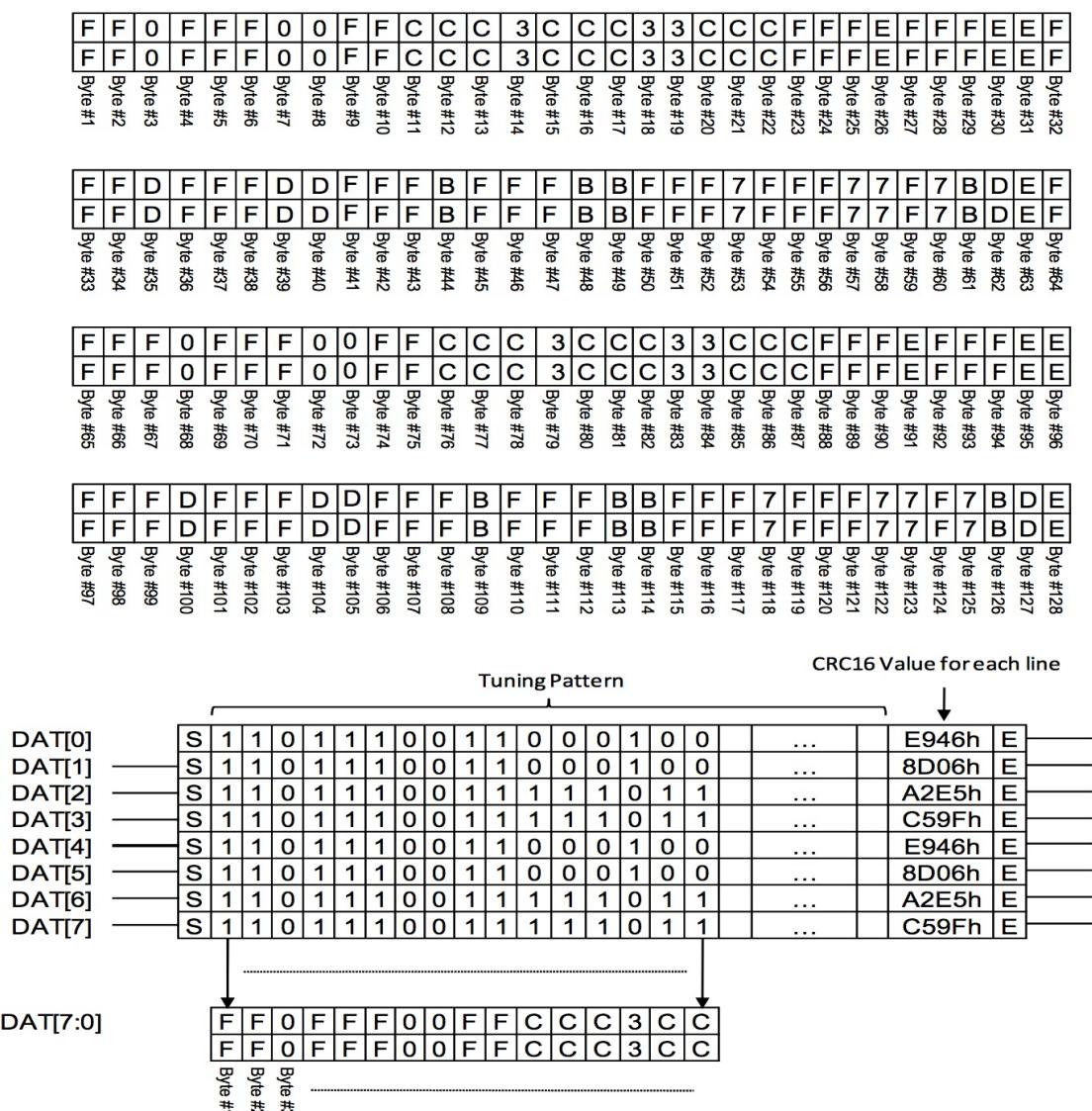
1. Host 将采样时间点重置为默认值
 2. Host 向 eMMC Device 发送 Send Tuning Block 命令
 3. eMMC Device 向 Host 发送固定的 Tuning Block 数据
 4. Host 接收到 Tuning Block 并进行校验
 5. Host 修改采样时点，重新从第 2 步开始执行，直到 Host 获取到一个有效采样时间点区间
 6. Host 取有效采样时间点区间的中间值作为采样时间点，并推出 Tuning 流程

NOTE:

上述流程仅仅是一个示例。Tuning 流程执行的时机、频率和具体的步骤是由 Host 端的 eMMC Controller 具体实现而定的。

Tuning Block 数据

Tuning Block 是专门为了 Tuning 而设计的一组特殊数据。相对于普通的数据，这组特殊数据在传输过程中，会更高概率的出现 high SSO noise、deterministic jitter、ISI、timing errors 等问题。这组数据的具体内容如下所示：



NOTE: 总线宽度为 1 时，只发送 DAT0 上的数据，总线宽度为 4 时，则只发送 DAT0-3 上的数据

参考资料

1. [Embedded Multi-Media Card \(e•MMC\) Electrical Standard \(5.1\)](#) [PDF]
2. [SD/MMC Controller, Hard Processor System \(HPS\) Technical Reference Manual \(TRM\)](#) [PDF]
3. [CRC wiki](#) [WEB]

eMMC 工作模式

Overview

TODO : Add Pic

eMMC Device 在 Power On、HW Reset 或者 SW Reset 时，Host 可以触发 eMMC Boot，让 eMMC 进入 Boot Mode。在此模式下，eMMC Device 会将 Boot Data 发送给 Host，这部分内容通常为系统的启动代码，如 BootLoader。

如果 Host 没有触发 Boot 流程或者 Boot 流程完成后，eMMC Device 会进入 Device Identification Mode。在此模式下，eMMC Device 将进行初始化，Host 会为 eMMC Device 设定工作电压、协商寻址模式以及分配 RCA 设备地址。

Device Identification Mode 结束后，就会进入 Data Transfer Mode。在此模式下，Host 可以发起数据读写流程。

进入 Data Transfer Mode 后，Host 可以发起命令，让 eMMC Device 进入 Interrupt Mode。在此模式下，eMMC Device 会等待内部的中断事件，例如，写数据完成等。eMMC Device 在收到内部中断事件时，会向 Host 发送 Response，然后切换到 Data Transfer Mode，等待 Host 后续的数据读写命令。

Boot Operation Mode

Boot From eMMC Device

TODO : Add Boot States

在 Power On、HW Reset 或者 SW Reset 后，如果 eMMC Device 有使能 Boot Mode（即，寄存器位 BOOT_PARTITION_ENABLE (EXT_CSD byte [179]) 指定了启动分区），那么 Host 有两种方式可以让 eMMC Device 进入 Boot Mode，分别定义为 Original Boot 和 Alternative Boot，如下：

1. Original Boot : 拉低 CMD 信号并保持不少于 74 个时钟周期
2. Alternative Boot : 保持 CMD 信号为高电平，74 个时钟周期后，发送参数为 0xFFFFFFFFA 的 **CMD0** 命令

进入 Boot Mode 后，eMMC Device 会根据寄存器位 BOOT_PARTITION_ENABLE 的设定，从两个 Boot partitions 和 UDA 中选择一个分区读取大小为 $128KB \times \text{BOOT_SIZE_MULT}$ (EXT_CSD byte [226]) 的 Boot Data 通过 Data Lines 发送给 Host。

在 Boot Data 数据传输过程中，Host 可以打断数据传输，提前结束 Boot Mode，方法如下：

1. Original Boot : 传输过程中，拉高 CMD 信号
2. Alternative Boot : 传输过程中，发送参数为 0xF0F0F0F0 的 **CMD0** 命令

NOTE:

Host 发送参数为 0xF0F0F0F0 的 **CMD0** 命令，可以让 eMMC Device 进行 SW Reset

Host 拉高 RST_n 信号可以触发 eMMC Device 进行 HW Reset

Boot Acknowledge

如果寄存器位 BOOT_ACK (EXT_CSD byte [179]) 被设定为 1，eMMC Device 会在 Host 触发 Boot Mode 的 50 ms 内，在 DAT0 上发送一个 "010" Boot ACK 给 Host。

包含 Boot ACK 的时序图如下所示：

TODO : Add Original Boot Pic

TODO : Add Alternative Boot Pic

Boot Bus 配置

EXT_CSD byte [177] BOOT_BUS_CONDITIONS 寄存器用于配置在 Boot Mode 时，数据传输的总线状态。

通过 BOOT_BUS_CONDITIONS 寄存器配置，在 Boot Mode 时，总线可以支持以下几种模式：

Mode	Data Rate	Bus Width	Frequency	Max Data Transfer (x8)
Backward Compatible	Single	x1, x4, x8	0-26 MHz	26 MB/s
High Speed SDR	Single	x1, x4, x8	0-52 MHz	52 MB/s
High Speed DDR	Dual	x4, x8	0-52 MHz	104 MB/s

BOOT_BUS_CONDITIONS 寄存器还可以配置退出 Boot Mode 后，是复位还是保留当前总线配置。如果配置为复位，那么退出 Boot Mode 后，总线会被复位为 Backward Compatible SDR x1 模式，如果配置为保留，那么退出 Boot Mode 后，总线会保留 Boot Mode 时的总线模式。

NOTE:

BOOT_BUS_CONDITIONS 寄存器为 nonvolatile 属性，配置内容掉电不会丢失。

如果 eMMC Device 没有经过 Boot Mode，BOOT_BUS_CONDITIONS 寄存器不会改变总线模式。

退出 Boot Mode 后，还可以通过 HS_TIMING 和 BUS_WIDTH 寄存器配置总线模式。

Boot Data 更新

eMMC Device 在从厂商出货时，没有存储内容，也没有使能 Boot Mode。使用 eMMC Devcie 产品需要先通过其他的方式（例如，通过 USB、UART 等）启动一个下载系统，将 Boot Data 以及其他的数据写入到 eMMC 中，同时使能 Boot Mode 并设定 Boot Bus 模式。而后，产品才能从 eMMC Device 上启动软件系统。

Boot Data 的更新与其他数据的写入类似，更多的数据写入细节，请参考 Data Transfer Mode 小节。

Device Identification Mode

如果 Host 没有触发 Boot 流程或者 Boot 流程完成后，eMMC Device 会进入 Device Identification Mode。

TODO : Add States

eMMC Device 在退出 Boot Mode 后或者没使能 Boot Mode 时 Power On、HW Reset 或者 SW Reset 后，会进入 Device Identification Mode 的 Idle State。

在 **Idle State** 下，eMMC Device 会进行内部初始化，Host 需要持续发送 **CMD1** 命令，查询 eMMC Device 是否已经完成初始化，同时进行工作电压和寻址模式协商。

Host 发送的 **CMD1** 命令的参数中，包含了 Host 所支持的工作电压和寻址模式信息，eMMC Device 在接收到这些信息后，会进行匹配。如果 eMMC Devcie 和 Host 所支持的工作电压和寻址模式不匹配，那么 eMMC Device 会进入 **Inactive State**。

eMMC Device 在接收到 **CMD1** 命令后，会将 **OCR register** 的内容作通过 Response 返回给 Host，其中包含了 eMMC Device 是否完成初始化的标志位、设备工作电压范围 Voltage Range 和存储访问模式 Memory Access Mode 信息。

eMMC Device 完成初始化后，就会进入 **Ready State**。在该 State 下，Host 会发送 **CMD2** 命令，获取 eMMC Device 的 CID。

CID，即 Device identification number，用于标识一个 eMMC Device。它包含了 eMMC Device 的制造商、OEM、设备名称、设备序列号、生产年份等信息，每一个 eMMC Device 的 CID 都是唯一的，不会与其他的 eMMC Device 完全相同。

eMMC Device 接收到 **CMD2** 后，会将 127 Bits 的 **CID register** 的内容通过 Response 返回给 Host。

发送完 CID 后，eMMC Device 接着就会进入 **Identification State**。而后，Host 会发送参数包含 16 Bits RCA 的 **CMD3** 命令，为 eMMC Device 分配 RCA。

设定完 RCA 后，eMMC Devcie 就完成了 Devcie Identification，进入 Data Transfer Mode。

NOTE:

本节只描述了单个 eMMC Device 的 Devcie Identification 过程，多个 eMMC 的 Device Identification 过程与此类似，更多的细节可以参考 eMMC Spec。

Voltage Range

eMMC Device 支持 3.3v 和 1.8v 两种工作电压模式。在 1.8v 模式下，eMMC Device 会更加的省电。

TODO：描述如何设定或者切换不同的工作电压模式

Memory Access Mode

Memory Access Mode 决定了 eMMC Device 在响应 Host 的数据读写请求时，是如何访问内部存储器的。在 eMMC 标准中存在两种 Memory Access Mode：Byte Access Mode 和 Sector Access Mode。

在数据读写的 Command 中，Host 会将读写的地址 A 作为 Command 的参数发送给 eMMC Device，在 Byte Access Mode 下，eMMC Device 将从第 A 个 Byte 开始进行读写操作，而在 Sector Access Mode 下，eMMC Device 将会从第 A 个 Sector 开始进行读写操作，一个 Sector 的大小为 512 Bytes 或者 4 KBytes，更大的 Sector 支持更大容量的存储器访问。

使用 Byte Access Mode 更加的灵活高效，但是由于寻址位数的限制，不能访问超过 2GB 的存储内容。Sector Access Mode 则支持大容量存储的访问，其中 512 Bytes Sector 可以支持最大 256 GB 容量的存储访问，更大容量的需求则可以使用 4 KBytes Sector。

RCA - Relative device Address

RCA 是在 Devcie Identification 过程中，由 Host 分配的 16 Bits 的设备地址，主要用于 Data Transfer Mode 下进行通信时，选定具体要进行操作的 eMMC Devcie。

Host 分配的 RCA 通常从 1 开始递增，0 地址作为广播地址。eMMC Devcie 的 [RCA register](#) 保存了 Host 分配的 RCA。

TODO：确认掉电重启后，RCA register 的值是否丢失。

Data Transfer Mode

eMMC Device 完成 Device Identification 后，就会进入到 Data Transfer Mode 的 Standby State。

在 Standby State 时，Host 可以通过发送 [CMD5](#) 命令，让 eMMC Devcie 进入 低功耗的 Sleep State，而后再发送 [CMD5](#) 命令则可以让 eMMC Device 退出 Sleep State。

在 Standby State 时，Host 可以通过发送 [CMD7](#) 命令，让 eMMC Devcie 进入 Transfer State，而后再发送 [CMD7](#) 命令则可以让 eMMC Device 退出 Transfer State。

Read Data

在 Transfer State 时，Host 可以发送以下的命令，触发数据读取流程：

命令	描述
CMD8	读取 EXT_CSD 寄存器数据
CMD17	从指定的地址开始，读取一个 Block 的数据
CMD18	从指定的地址开始，读取多个 Block 的数据
CMD21	读取 Tuning Block 的数据

eMMC Device 在接收到上述几个 CMD 时，就会进入 Sending-data State。在此 State 下，eMMC Device 会持续将数据发送给 Host，直到指定数量的数据 Block 传输完成或者接收到 Host 发送的 [CMD12](#) 传输停止命令。eMMC Device 在停止发送数据后，会返回到 Transfer State。

如果 Host 在发送 **CMD18** 前，先发送一个设定需要读取的 Block Count 的 **CMD23**。eMMC Device 在完成指定 Block Count 的数据发送后，就自动结束数据传输，不需要 Host 主动发送停止命令 **CMD12**。

如果 Host 没有发送设定需要读取的 Block Count 的 Command，发送 Multiple Block Read 的 Command 后，eMMC Device 会持续发送数据，直到 Host 发送 Stop Command 停止数据传输。

NOTE: 如果在发送 **CMD18** 前，先发送 **CMD23** 设定需要读取的 Block Count，那么 eMMC Device 会在发送完指定数量的 Block 后，自动停止发送数据。

Write Data

在 Transfer State 时，Host 可以发送以下的命令，触发数据写入流程：

命令	描述
CMD24	写入一个 Block 的数据
CMD25	写入多个 Block 的数据
CMD26	写入 CID 寄存器值
CMD27	写入 CSD 寄存器值

NOTE:

CID 寄存器值通常是只能写一次，由厂家在生产时确定并写入 **CSD** 寄存器值的部分位则可以多次改写。

eMMC Device 在接收到上述几个 CMD 时，就会进入 Receive-data State，在此 State 下，eMMC Devcie 会持续从 Host 接收数据，并存储到内部的 Buffer 或者寄存器中。

如果 Host 在发送 **CMD25** 前，先发送一个设定需要写入的 Block Count 的 **CMD23**。eMMC Device 在完成指定 Block Count 的数据接收后，就自动结束数据传输，不需要 Host 主动发送停止命令 **CMD12**。

如果 Host 没有发送设定需要写入的 Block Count 的 Command，发送 Multiple Block Write 的 Command 后，eMMC Device 会持续接收数据，直到 Host 发送 Stop Command 停止数据传输。

eMMC Device 在开始进行写入操作时，会先将接收到的数据存储在内部 Buffer 中，然后在后台将 Buffer 中的数据写入到 Flash 中。通常情况下，Host 发送数据的速度会比 eMMC 写入 Flash 的速度快，所以内部的 Buffer 会出现写满的状态，此时 eMMC Devcie 会将 DAT0 信号线拉低作为 Busy 信号。Host 收到 Busy 信号后，就会暂停发送数据，等到 eMMC Device 将 Buffer 中的数据处完一部分并解除 Busy 信号后，再重新发送数据。

当 eMMC Device 完成数据接收后，就会进入到 Programming State，将内部 Buffer 中剩余未写入的数据写入到 Flash 中。在该 State 下，eMMC Device 会持续将 DAT0 拉低，作为 Busy 信号。如果在完成写入前，有收到新的写入命令，那么 eMMC Device 会立刻退回到 Receive-data State，进行数据接收；如果在完成写入前，没有收到新的写入命令，则会在完成写入后，退回到 Transfer State。

如果 eMMC Devcie 在 Programming State 时，还没有完成写入操作，就收到参数不等于自身 RCA 的 **CMD7** 命令，那么 eMMC Device 会进入到 Disconnect State。在该 State 下，eMMC Device 会继续进行写入操作，写入完成后则进入到 Stand-by State。

如果 eMMC Device 在 Disconnect State 时，还没有完成写入操作，就收到参数等于自身 RCA 的 **CMD7** 命令，那么 eMMC Devcie 会从新回到 Programming State。

Packed Commands - Packed Write and Packed Read

在实际应用场景中，通常会对 eMMC Device 有很多随机数据读取和写入操作，这些随机读写的目标地址往往都不是连续的，每一个随机读写都需要通过一个独立的读写流程来实现。

在 eMMC 4.5 及以后的标准中，引入了 Packed Commands 机制，将多个地址不连续的数据写入请求封装到一个 Multiple Block Write 流程中，同时将多个地址不连续的数据读取请求封装到一个 Multiple Block Read 流程中，以此减少读写请求数量，提高数据读写的效率。

TODO : Add Packed Read and Packed Write pic

Packed Write

发起 Packed Write 流程时，首先 Host 端会需要发送 packed flag 置 1 的 [CMD23 SET_BLOCK_COUNT](#) 命令。其中，CMD23 中的 Block Count 参数为 Packed Command Header 和实际写入的数据所占 Block 的总数。

然后 Host 再发送 [CMD25](#) 命令给 eMMC Device，开始进行多个 Block 的数据写入。其中第 1 个（或者前 8 个）Block 数据为 Packed Command Header，它包含了各个写请求写入数据的起始地址和长度等信息。

eMMC Devcie 在接收到数据后，会根据 Packed Command Header 的信息，将数据写入到指定的位置。

Packed Read

发起 Packed Read 流程时，首先 Host 端会需要发送 packed flag 置 1 的 [CMD23 SET_BLOCK_COUNT](#) 命令。其中，CMD23 中的 Block Count 参数为 Packed Command Header 所占 Block 的数量。

然后 Host 再发送 [CMD25](#) 命令给 eMMC Device，开始进行 1 个（或者 8 个）Block 的 Packed Command Header 数据发送。Packed Command Header 包含了各个读请求读取数据的起始地址和长度等信息。

发送完 Packed Command Header 后，Host 会再发送一个 packed flag 置 1 的 [CMD23 SET_BLOCK_COUNT](#) 命令。其中，CMD23 中的 Block Count 参数为待读取数据的 Block 的数量。

紧接着，Host 再发送 [CMD18](#) 命令，开始进行多个 Block 的数据读取。eMMC Devcie 会解析接收到的 Packed Command Header，然后将指定的数据发送给 Host 端。

NOTE:

Host 也可以不发送第二个 CMD23 命令，在这种情况下，Host 需要主动发送 Stop 命令，通知 eMMC Device 停止数据发送。

Packed Command Header

TODO : add packed command header pic

Packed Command Header 的格式如上图所示，其中 CMD23_ARG_x 指示了各个请求数据读取或者写入的 Block 数，CMDxx_ARG_x（CMD18 或者 CMD25）则指示了各个请求数据读取或者写入的起始位置。

当 DATA_SECTOR_SIZE[61] = 0x00 时，即 Data Sector Size 为 512 Bytes 时，Packed Command Header 占 1 个 Block 大小，当 DATA_SECTOR_SIZE[61] = 0x01 时，即 Data Sector Size 为 4 KBytes 时，Packed Command Header 占 8 个 Block 大小。

NOTE:

Packed Command 的错误处理流程，请参考 eMMC Spec 文档中的描述，此处不再详细介绍。

数据擦除

eMMC 标准提供了几种主动擦除数据的方法，以满足不同的场景需求。

擦除方式	擦除单位	描述
Erase	Erase Group	按 Erase Group 擦除数据，完成后重新读取会返回全为 0 或者 1 的数据，但在物理存储介质上，可能还保留着原始数据
TRIM	Write Block	按 Write Block 擦除数据，完成后重新读取会返回全为 0 或者 1 的数据，但在物理存储介质上，可能还保留着原始数据
Discard	Write Block	按 Write Block 擦除数据，完成后重新读取可能会返回擦除前的数据
Sanitize	-	将标记擦除的 Block 的数据在物理介质上清除

Erase

Erase 操作以 Erase Group 为单位进行一个或者多个 Group 的数据擦除，一个 Erase Group 由一个或者多个 Write Block 组成。

eMMC Device 在执行 Erase 操作时，通常并不会进行实际物理数据的擦除，只是将待擦除的 Erase Group 中的 Block 从地址空间中 unmap，然后从后台的空闲 Block 中选择已经完成物理擦除的 Block，重新 map 到该地址空间中，然后告知 Host 端已完成 Erase 操作。实际物理擦除操作则在后台选择合适的时机进行。

这样的逻辑可以减少 Host 执行 Erase 操作的等待时间，提高 eMMC Devcie 的响应速度。

发起 Erase 流程时，首先 Host 会发送参数为待擦除 Erase Group 起始地址的 [CMD23 SET_BLOCK_COUNT](#) 命令

TRIM

Discard

Sanitize

Interrupt Mode

TODO

参考资料

1. [Embedded Multi-Media Card \(e•MMC\) Electrical Standard \(5.1\)](#) [PDF]

TODO

eMMC RPMB

TODO

eMMC 设备寄存器

OCR register

OCR，即 Operation Conditions Register，此寄存器包含 eMMC Device 支持的电压模式、数据寻址模式（按 Byte 寻址 or 按 Sector 寻址）以及 Busy 标志位。

OCR bit	VCCQ voltage window ²	High Voltage MultimediaCard	Dual voltage MultimediaCard and e•MMC			
[6:0]	Reserved	000 0000b	000 0000b			
[7]	1.70 – 1.95V	0b	1b			
[14:8]	2.0 – 2.6V	000 0000b	000 0000b			
[23:15]	2.7 – 3.6V	1 1111 1111b	1 1111 1111b			
[28:24]	Reserved	0 0000b	0 0000b			
[30:29]	Access Mode	00b (byte mode) 10b (sector mode)	00b (byte mode) 10b (sector mode)			
[31]	(Card power up status bit (busy)) ¹					
NOTE 1 This bit is set to LOW if the Device has not finished the power up routine.						
NOTE 2 VDD voltage window in case of High Voltage MultimediaCard.						

此寄存器的值，会在 Device Identification Mode 中，作为 CMD1 的响应内容返回给 Host。

NOTE:

Voltage Window 指明 eMMC Device 支持 1.70v - 1.95v 和 2.7v - 3.6v 两个工作电压范围。容量小于等于 2 GB 的 eMMC Devcie 的 Access Mode 为 Byte Mode，容量大于 2 GB 的 eMMC Devcie 的 Access Mode 为 Sector Mode。

CID register

Name	Field	Width	CID-Slice
Manufacturer ID	MID	8	[127:120]
Reserved		6	[119:114]
Device/BGA	CBX	2	[113:112]
OEM/Application ID	OID	8	[111:104]
Product name	PNM	48	[103:56]
Product revision	PRV	8	[55:48]
Product serial number	PSN	32	[47:16]
Manufacturing date	MDT	8	[15:8]
CRC7 checksum	CRC	7	[7:1]
not used, always “1”	-	1	[0:0]

MID [127:120]

MID is an 8 bit binary number that identifies the device manufacturer. The MID number is controlled, defined and allocated to an e•MMC manufacturer by JEDEC. This procedure is established to ensure uniqueness of the CID register.

CBX [113:112]

CBX indicates the device type.

[113:112]	Type
00	Device (removable)
01	BGA (Discrete embedded)
10	POP
11	Reserved

OID [111:104]

OID is an 8-bit binary number that identifies the Device OEM and/or the Device contents (when used as a distribution media either on ROM or FLASH Devices). The OID number is controlled, defined and allocated to an eMMC manufacturer by JEDEC. This procedure is established to ensure uniqueness of the CID register

PNM [103:56]

The product name, PNM, is a string, 6 ASCII characters long.

PRV [55:48]

The product revision, PRV, is composed of two Binary Coded Decimal (BCD) digits, four bits each, representing an “n.m” revision number. The “n” is the most significant nibble and “m” is the least significant nibble. As an example, the PRV binary value field for product revision “6.2” will be: 0110 0010.

PSN [47:16]

PSN is a 32-bit unsigned binary integer.

MDT [15:8]

The manufacturing date, MDT, is composed of two hexadecimal digits, four bits each, representing a two digits date code m/y; The “m” field, most significant nibble, is the month code. 1 = January. The “y” field, least significant nibble, is the year code. 0 = 1997. As an example, the binary value of the MDT field for production date “April 2000” will be: 0100 0011

CRC [7:1]

The CRC7 checksum (7 bits). This is the checksum of the CID contents computed according to 0.

CSD register

The Device-Specific Data (CSD) register provides information on how to access the Device contents. The CSD defines the data format, error correction type, maximum data access time, data transfer speed, whether the DSR register can be used etc. The programmable part of the register (entries marked by W or E below) can be changed by CMD27. The type of the CSD Registry entries below is coded as follows:

R: Read only.

W: One time programmable and not readable.

R/W: One time programmable and readable.

W/E: Multiple writable with value kept after power failure, H/W reset assertion and any CMD0 reset and not readable.

R/W/E: Multiple writable with value kept after power failure, H/W reset assertion and any CMD0 reset and readable.

R/W/C_P: Writable after value cleared by power failure and HW/rest assertion (the value not cleared by CMD0 reset) and readable.

R/W/E_P: Multiple writable with value reset after power failure, H/W reset assertion and any CMD0 reset and readable.

W/E_P: Multiple writable with value reset after power failure, H/W reset assertion and any CMD0 reset and not readable.

Name	Field	Width	Cell Type	CSD-slice
CSD structure	CSD_STRUCTURE	2	R	[127:126]
System specification version	SPEC_VERS	4	R	[125:122]
Reserved	-	2	R	[121:120]
Data read access-time 1	TAAC	8	R	[119:112]
Data read access-time 2 in CLK cycles (NSAC*100)	NSAC	8	R	[111:104]
Max. bus clock frequency	TRAN_SPEED	8	R	[103:96]
Device command classes	CCC	12	R	[95:84]
Max. read data block length	READ_BL_LEN	4	R	[83:80]
Partial blocks for read allowed	READ_BL_PARTIAL	1	R	[79:79]
Write block misalignment	WRITE_BLK_MISALIGN	1	R	[78:78]
Read block misalignment	READ_BLK_MISALIGN	1	R	[77:77]
DSR implemented	DSR_IMP	1	R	[76:76]
Reserved	-	2	R	[75:74]
Device size	C_SIZE	12	R	[73:62]
Max. read current @ VDD min	VDD_R_CURR_MIN	3	R	[61:59]
Max. read current @ VDD max	VDD_R_CURR_MAX	3	R	[58:56]
Max. write current @ VDD min	VDD_W_CURR_MIN	3	R	[55:53]
Max. write current @ VDD max	VDD_W_CURR_MAX	3	R	[52:50]
Device size multiplier	C_SIZE_MULT	3	R	[49:47]
Erase group size	ERASE_GRP_SIZE	5	R	[46:42]
Erase group size multiplier	ERASE_GRP_MULT	5	R	[41:37]
Write protect group size	WP_GRP_SIZE	5	R	[36:32]
Write protect group enable	WP_GRP_ENABLE	1	R	[31:31]
Manufacturer default ECC	DEFAULT_ECC	2	R	[30:29]
Write speed factor	R2W_FACTOR	3	R	[28:26]
Max. write data block length	WRITE_BL_LEN	4	R	[25:22]
Partial blocks for write allowed	WRITE_BL_PARTIAL	1	R	[21:21]
Reserved	-	4	R	[20:17]
Content protection application	CONTENT_PROT_APP	1	R	[16:16]
File format group	FILE_FORMAT_GRP	1	R/W	[15:15]
Copy flag (OTP)	COPY	1	R/W	[14:14]
Permanent write protection	PERM_WRITE_PROTECT	1	R/W	[13:13]
Temporary write protection	TMP_WRITE_PROTECT	1	R/W/E	[12:12]
File format	FILE_FORMAT	2	R/W	[11:10]
ECC code	ECC	2	R/W/E	[9:8]
CRC	CRC	7	R/W/E	[7:1]
Not used, always'1'	-	1	—	[0:0]

NOTE:

更多 CSD register 的细节请参考 eMMC Spec.

RCA register

The writable 16-bit relative Device address (RCA) register carries the Device address assigned by the host during the Device identification. This address is used for the addressed host-Device communication after the Device identification procedure. The default value of the RCA register is 0x0001. The value 0x0000 is reserved to set all Devices into the Stand-by State with CMD7.

Extended CSD register

The Extended CSD register defines the Device properties and selected modes. It is 512 bytes long. The most significant 320 bytes are the Properties segment, that defines the Device capabilities and cannot be modified by the host. The lower 192 bytes are the Modes segment, that defines the configuration the Device is working in. These modes can be changed by the host by means of the SWITCH command.

eMMC Commands

Abbr.	Type	Argument	Resp	Description

Basic commands (class 0 and class 1)

CMD0

Abbr.	Type	Argument	Resp	Description
GO_IDLE_STATE	bc	[31:0] 00000000	None	Resets the Device to idle state
GO_PRE_IDLE_STATE	bc	[31:0] F0F0F0F0	None	Resets the Device to pre-idle state
BOOT_INITIATION	-	[31:0]FFFFFFFA	None	Initiate alternative boot operation

CMD1

Abbr.	Type	Argument	Resp	Description
SEND_OP_COND	bcr	[31:0] OCR without busy	R3	Asks Device, in idle state, to send its Operating Conditions Register contents in the response on the CMD line.

CMD2

Abbr.	Type	Argument	Resp	Description
ALL_SEND_CID	bcr	[31:0] stuff bits	R2	Asks Device to send its CID number on the CMD line

CMD3

Abbr.	Type	Argument	Resp	Description
SET_RELATIVE_ADDR	ac	[31:16] RCA, [15:0] stuff bits	R1	Assigns relative address to the Device

CMD4

Abbr.	Type	Argument	Resp	Description
SET_DSR	bc	[31:16] DSR, [15:0] stuff bits	-	Programs the DSR of the Device

CMD5

Abbr.	Type	Argument	Resp	Description
SLEEP_AWAKE	ac	[31:16] RCA, [15]Sleep/Awake, [14:0] stuff bits	R1b	Toggles the Device between Sleep state and Standby state.

CMD6

Abbr.	Type	Argument	Resp	Description
SWITCH	ac	[31:26] Set to 0, [25:24] Access, [23:16] Index, [15:8] Value, [7:3] Set to 0, [2:0] Cmd Set	R1b	Switches the mode of operation of the selected Device or modifies the EXT_CSD registers.

CMD7

Abbr.	Type	Argument	Resp	Description
SELECT/DESELECT_CARD	ac	[31:16] RCA, [15:0] stuff bits	R1/R1b	Command toggles a device between the standby and transfer states or between the programming and disconnect states.

CMD8

Abbr.	Type	Argument	Resp	Description
SEND_EXT_CSD	adtc	[31:0] stuff bits	R1	Device sends its EXT_CSD register as a block of data.

CMD9

Abbr.	Type	Argument	Resp	Description
SEND_CSD	ac	[31:16] RCA, [15:0] stuff bits	R2	Addressed Device sends its Device-specific data (CSD) on the CMD line.

CMD10

Abbr.	Type	Argument	Resp	Description
SEND_CID	ac	[31:16] RCA, [15:0] stuff bits	R2	Addressed Device sends its Device identification (CID) on CMD the line.

CMD12

Abbr.	Type	Argument	Resp	Description
STOP_TRANSMISSION	ac	[31:16] RCA, [15:1] stuff bits, [0] HPI	R1/R1b	Forces the Device to stop transmission. If HPI flag is set the device shall interrupt its internal operations in a well-defined timing.

CMD13

Abbr.	Type	Argument	Resp	Description
SEND_STATUS	ac	[31:16] RCA, [15] SQS, [14:1] stuff bits, [0] HPI	R1	In case SQS bit = 0: Addressed Device sends its status register. If HPI flag is set the device shall interrupt its internal operations in a well-defined timing. In case SQS bit = 1: indicate that this is a QSR query. In response device shall send the QSR (Queue Status Register). In this case HPI must be set to '0'

CMD14

Abbr.	Type	Argument	Resp	Description
BUSTEST_R	adtc	[31:0] stuff bits	R1	A host reads the reversed bus testing data pattern from a Device.

CMD15

Abbr.	Type	Argument	Resp	Description
GO_INACTIVE_STATE	ac	[31:16] RCA, [15:0] stuff bits	-	Sets the Device to inactive state

CMD19

Abbr.	Type	Argument	Resp	Description
BUSTEST_W	adtc	[31:0] stuff bits	R1	A host sends the bus test data pattern to a Device

NOTE 1 - CMD7, R1 while selecting from Stand-By State to Transfer State; R1b while selecting from Disconnected State to Programming State.

NOTE 2 - CMD12, RCA in CMD12 is used only if HPI bit is set. The argument does not imply any RCA check on the device side.

NOTE 3 - CMD12, R1 for read cases and R1b for write cases.

Block-oriented read commands (class 2)

CMD16

Abbr.	Type	Argument	Resp	Description
SET_BLOCKLEN	ac	[31:0] block length	R1	Sets the block length (in bytes) for all following block commands (read and write). Default block length is specified in the CSD

CMD17

Abbr.	Type	Argument	Resp	Description
READ_SINGLE_BLOCK	adtc	[31:0] data address	R1	Reads a block of the size selected by the SET_BLOCKLEN command

CMD18

Abbr.	Type	Argument	Resp	Description
READ_MULTIPLE_BLOCK	adtc	[31:0] data address	R1	Continuously transfers data blocks from Device to host until interrupted by a stop command, or the requested number of data blocks is transmitted. If sent as part of a packed read command, the argument shall contain the first read data address in the pack (address of first individual read command inside the pack).

CMD21

Abbr.	Type	Argument	Resp	Description
SEND_TUNING_BLOCK	adtc	[31:0] stuff bits	R1	128 clocks of tuning pattern (64 byte in 4bit mode or 128 byte in 8 bit mode) is sent for HS200 optimal sampling point detection.

NOTE 1 - CMD17 & CMD18, Data address for media =<2 32="" gb="" is="" a="" bit="" byte="" address="" and="" data="" for="" media=""> 2GB is a 32 bit sector (512 B) address.

NOTE 2 - CMD17, The transferred data must not cross a physical block boundary, unless READ_BLK_MISALIGN is set in the CSD register.

Block-oriented write commands (class 4)

CMD23 (default)

Abbr.	Type	Argument	Resp	Description
SET_BLOCK_COUNT	ac	[31] Reliable Write Request, [30] '0' non-packed, [29] tag request,[28:25] context ID, [24]: forced programming, [23:16] set to 0, [15:0] number of blocks	R1	Defines the number of blocks (read/write) and the reliable writer parameter (write) for a block read or write command.

CMD23 (packed)

Abbr.	Type	Argument	Resp	Description
SET_BLOCK_COUNT	ac	[31] set to 0, [30] '1' packed, [29:16] set to 0, [15:0] number of blocks	R1	Defines the number of blocks (read/write) for the following packed write command or for the header of the following packed read command.

CMD24

Abbr.	Type	Argument	Resp	Description
WRITE_BLOCK	adtc	[31:0] data address	R1	Writes a block of the size selected by the SET_BLOCKLEN command.

CMD25

Abbr.	Type	Argument	Resp	Description
WRITE_MULTIPLE_BLOCK	adtc	[31:0] data address	R1	Continuously writes blocks of data until a STOP_TRANSMISSION follows or the requested number of block received. If sent as a packed command (either packed write, or the header of packed read) the argument shall contain

the first read/write data address in the pack (address of first individual command inside the pack). |

CMD26

Abbr.	Type	Argument	Resp	Description
PROGRAM_CID	adtc	[31:0] stuff bits	R1	Programming of the Device identification register. This command shall be issued only once. The Device contains hardware to prevent this operation after the first programming. Normally this command is reserved for the manufacturer.

CMD27

Abbr.	Type	Argument	Resp	Description
PROGRAM_CSD	adtc	[31:0] stuff bits	R1	Programming of the programmable bits of the CSD.

CMD49

Abbr.	Type	Argument	Resp	Description
SET_TIME	adtc	[31:0] stuff bits	R1	Sets the real time clock according to the RTC information in the 512 B data block.

NOTE 1 - CMD24 & CMD25, Data address for media =<2 32="" gb="" is="" a="" bit="" byte="" address="" and="" data="" for="" media=""> 2GB is a 32 bit sector (512 B) address.

NOTE 2 - CMD24, The transferred data must not cross a physical block boundary, unless READ_BLK_MISALIGN is set in the CSD register.

Block-oriented write protection commands (class 6)

Erase commands (class 5)

I/O mode commands (class 9)

Lock Device commands (class 7)

Application-specific commands (class 8)

Security Protocols (class 10)

Command Queue (Class 11)

SD Card

UFS

NAND Flash

TODO

NOR Flash

TODO

Linux SD/MMC Framework

TODO

Memory

SDRAM

SRAM

DRAM

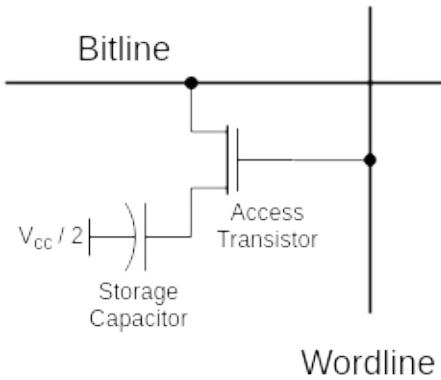
本章节尝试从介绍一个 DRAM Storage Cell 开始，逐步构建 Memory Array、Memory Banks、控制逻辑等模块，最终完成一个 DRAM Device 的介绍。

DRAM Storage Cell

Storage Capacitor

DRAM Storage Cell 使用 Storage Capacitor 来存储 Bit 信息。

从原理层面上看，一个最简单的，存储一个 Bit 信息的 DRAM Storage Cell 的结构如下图所示：



由以下 4 个部分组成：

- Storage Capacitor，即存储电容，它通过存储在其中的电荷的多和少，或者说电容两端电压差的高和低，来表示逻辑上的 1 和 0。
- Access Transistor，即访问晶体管，它的导通和截止，决定了允许或禁止对 Storage Capacitor 所存储的信息的读取和改写。
- Wordline，即字线，它决定了 Access Transistor 的导通或者截止。
- Bitline，即位线，它是外界访问 Storage Capacitor 的唯一通道，当 Access Transistor 导通后，外界可以通过 Bitline 对 Storage Capacitor 进行读取或者写入操作。

Storage Capacitor 的 Common 端接在 $V_{cc}/2$ 。

当 Storage Capacitor 存储的信息为 1 时，另一端电压为 V_{cc} ，此时其所存储的电荷 $Q = +V_{cc}/2 / C$

当 Storage Capacitor 存储的信息为 0 时，另一端电压为 0，此时其所存储的电荷 $Q = -V_{cc}/2 / C$

数据读写原理

从上面的结构图上分析，我们可以很容易的推测出 DRAM Storage Cell 的数据读写流程：

1. 读数据时，Wordline 设为逻辑高电平，打开 Access Transistor，然后读取 Bitline 上的状态
2. 写数据时，先把要写入的电平状态设定到 Bitline 上，然后打开 Access Transistor，通过 Bitline 改变 Storage Capacitor 内部的状态。

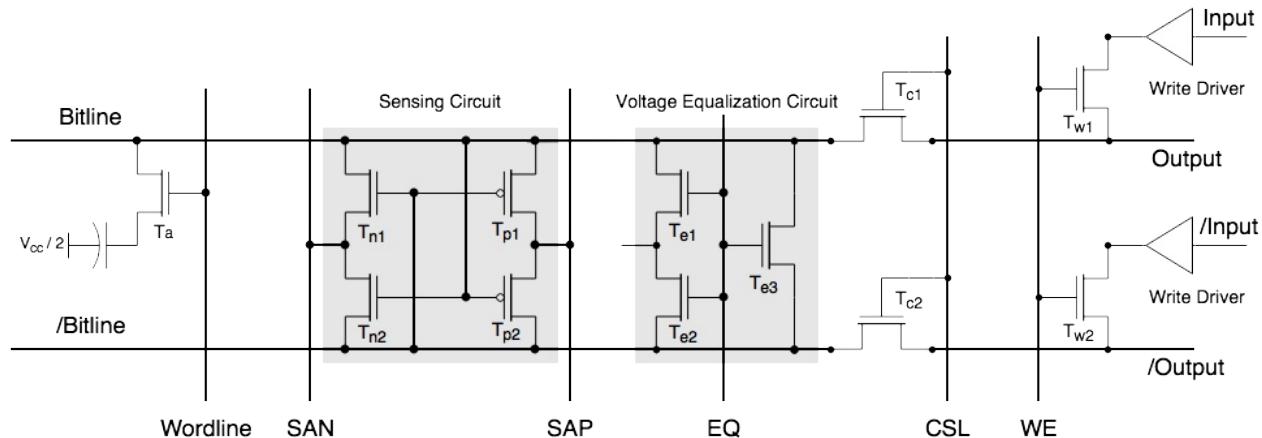
然而，在具体实现上，如果按照上面的流程对 DRAM Storage Cell 进行读写，会遇到以下的问题：

1. 外界的逻辑电平与 Storage Capacitor 的电平不匹配
由于 Bitline 的电容值比 Storage Capacitor 要大的多（通常为 10 倍以上），当 Access Transistor 导通后，如果 Storage Capacitor 存储的信息为 1 时，Bitline 电压变化非常小。外界电路无法直接通过 Bitline 来读取 Storage Capacitor 所存储的信息。
2. 进行一次读取操作后，Storage Capacitor 存储的电荷会变化
在进行一次读取操作的过程中，Access Transistor 导通后，由于 Bitline 和 Storage Capacitor 端的电压不一致，会导致 Storage Capacitor 中存储的电荷量被改变。最终可能会导致在下一次读取操作过程中，无法正确的判断 Storage Capacitor 内存储的信息。
3. 由于 Capacitor 的物理特性，即使不进行读写操作，其所存储的电荷都会慢慢变少
这个特性要求 DRAM 在没有读写操作时，也要主动对 Storage Capacitor 进行电荷恢复的操作。

为解决上述的问题，DRAM 在设计上，引入了 Differential Sense Amplifier。

Differential Sense Amplifier

Differential Sense Amplifier 包含 Sensing Circuit 和 Voltage Equalization Circuit 两个主要部分。它主要的功能就是将 Storage Capacitor 存储的信息转换为逻辑 1 或者 0 所对应的电压，并且呈现到 Bitline 上。同时，在完成一次读取操作后，通过 Bitline 将 Storage Capacitor 中的电荷恢复到读取之前的状态。



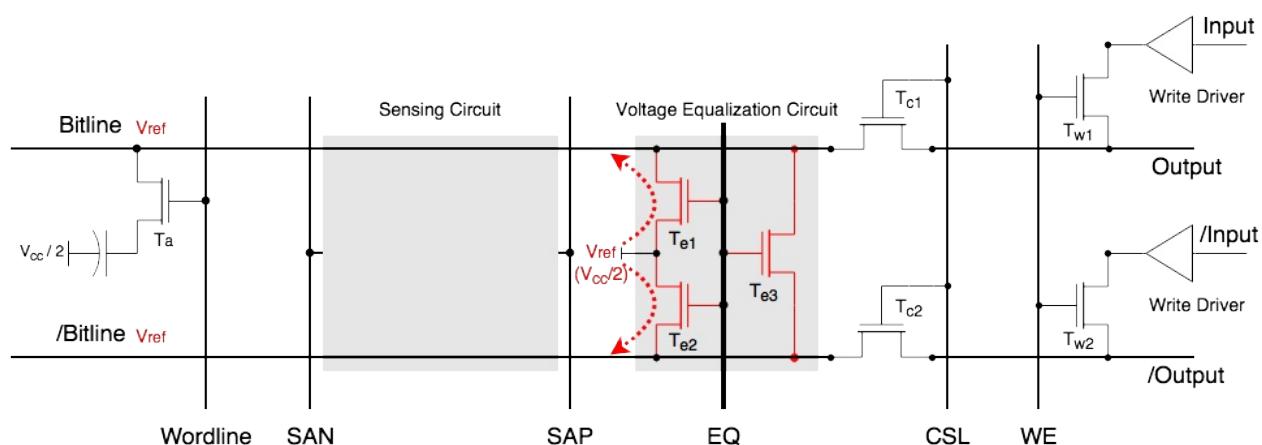
在后面的小节中，我们通过完整的数据读取和写入过程，来了解 Differential Sense Amplifier 工作原理。

Read Operation

一个完整的 Read Operation 包含了，Precharge、Access、Sense、Restore 四个阶段。后续的小节中，将描述从 Storage Capacitor 读取 Bit 1 的完整过程。

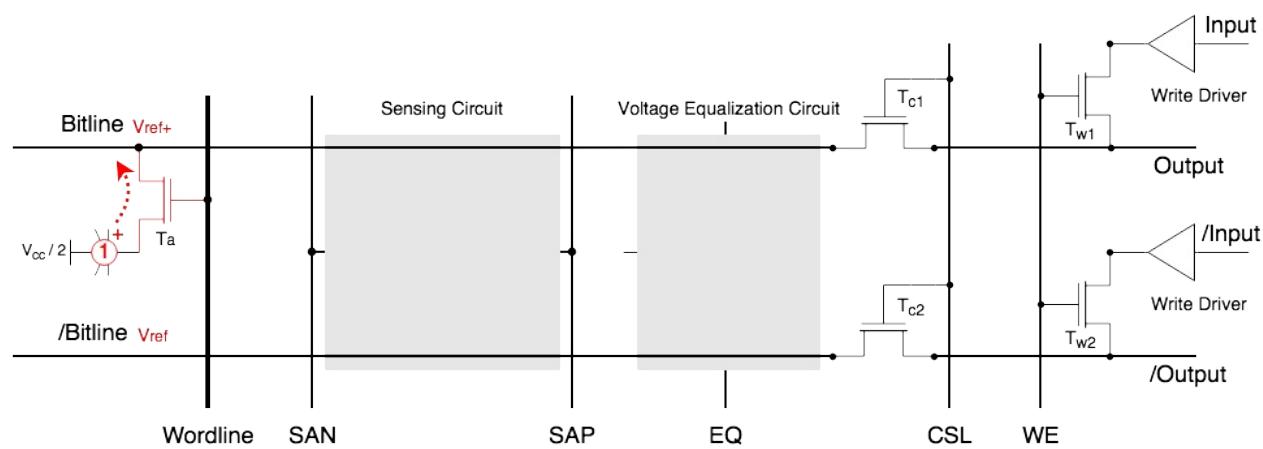
Precharge

在这个阶段，首先会通过控制 EQ 信号，让 T_{e1} 、 T_{e2} 、 T_{e3} 晶体管处于导通状态，将 Bitline 和 /Bitline 线上的电压稳定在 V_{ref} 上， $V_{ref} = V_{CC}/2$ 。然后进入到下一个阶段。



Access

经过 Precharge 阶段，Bitline 和 /Bitline 线上的电压已经稳定在 V_{ref} 上了，此时，通过控制 Wordline 信号，将 T_a 晶体管导通。Storage Capacitor 中存储正电荷会流向 Bitline，继而将 Bitline 的电压拉升到 V_{ref+} 。然后进入到下一个阶段。

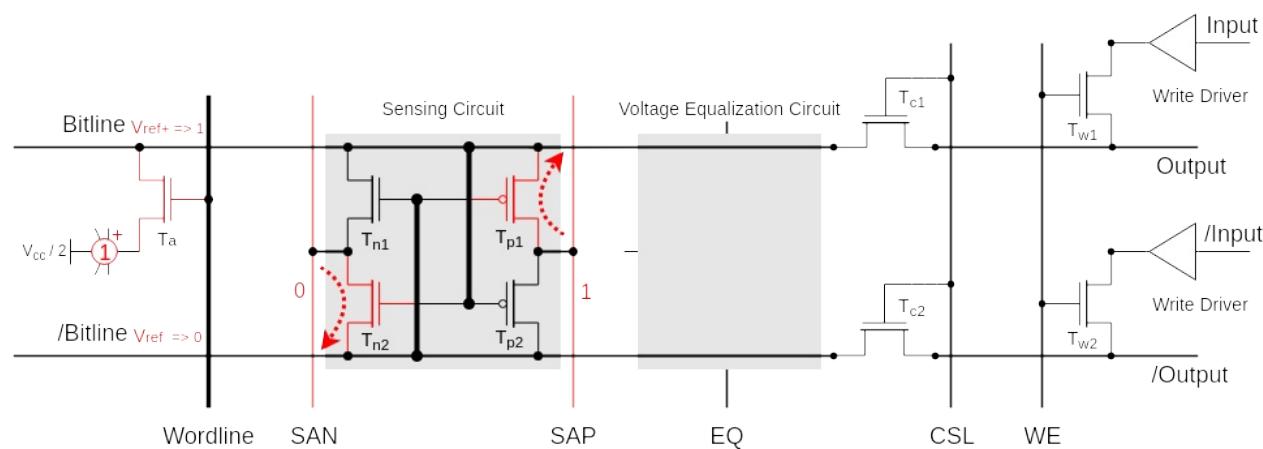


Sense

由于在 Access 阶段，Bitline 的电压被拉升到 V_{ref+} ， T_{n2} 会比 T_{n1} 更具导通性， T_{p1} 则会比 T_{p2} 更具导通性。

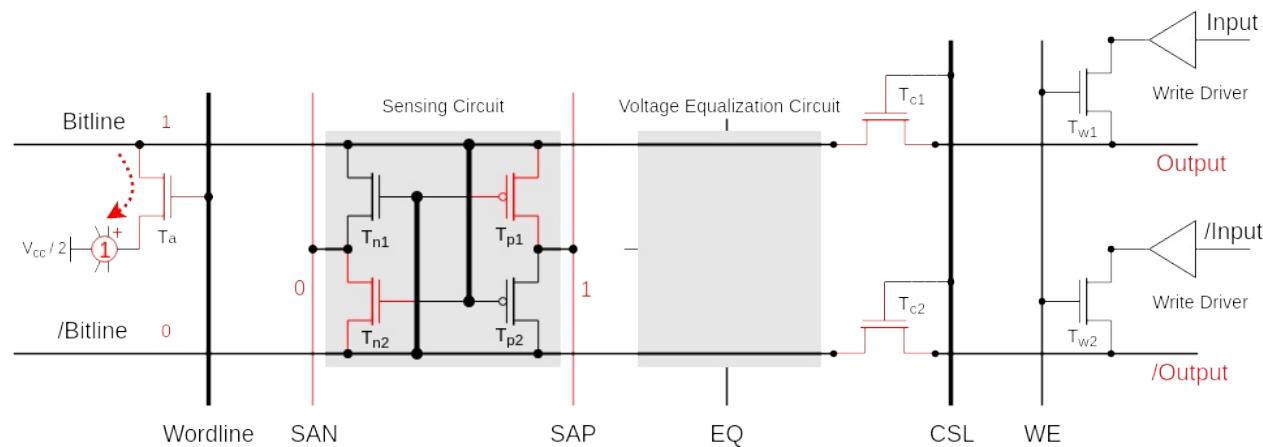
此时，SAN (Sense-Amplifier N-Fet Control) 会被设定为逻辑 0 的电压，SAP (Sense-Amplifier P-Fet Control) 则会被设定为逻辑 1 的电压，即 V_{CC} 。由于 T_{n2} 会比 T_{n1} 更具导通性，/Bitline 上的电压会更快被 SAN 拉到逻辑 0 电压，同理，Bitline 上的电压也会更快被 SAP 拉到逻辑 1 电压。接着 T_{p1} 和 T_{n2} 进入导通状态， T_{p2} 和 T_{n1} 进入截止状态。

最后，Bitline 和 /Bitline 的电压都进入稳定状态，正确的呈现了 Storage Capacitor 所存储的信息 Bit。



Restore

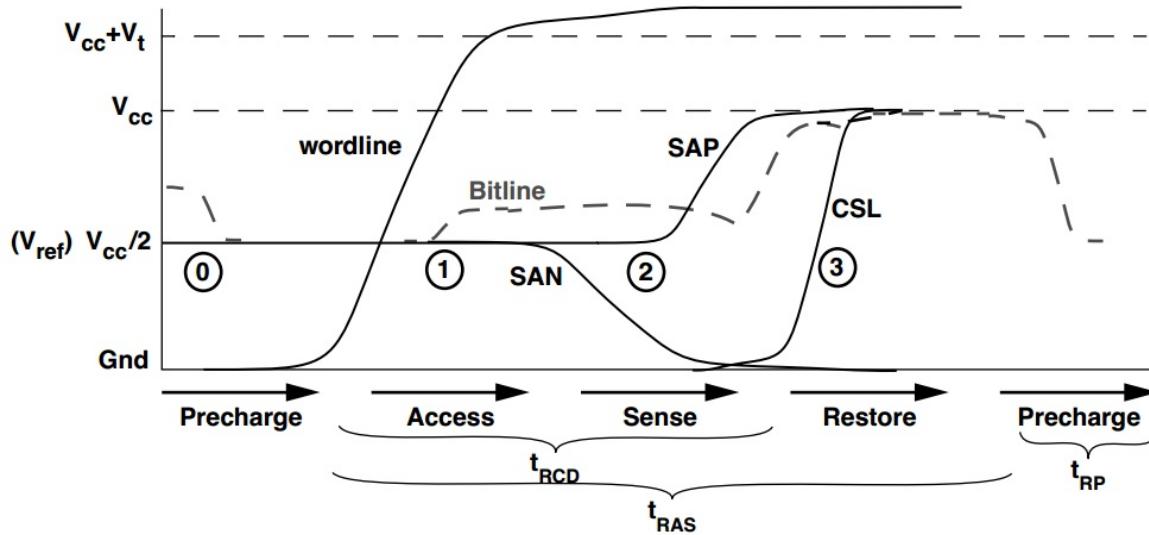
在完成 Sense 阶段的操作后，Bitline 线处于稳定的逻辑 1 电压 V_{CC} ，此时 Bitline 会对 Storage Capacitor 进行充电。经过特定的时间后，Storage Capacitor 的电荷就可以恢复到读取操作前的状态。



最后，通过 CSL 信号，让 T_{c1} 和 T_{c2} 进入导通状态，外界就可以从 Bitline 上读取到具体的信息。

Timing

整个 Read Operation 的时序如下图所示，其中的 V_{cc} 即为逻辑 1 所对应的电压，Gnd 为逻辑 0。



Write Operation

Write Operation 的前期流程和 Read Operation 是一样的，执行 Precharge、Access、Sense 和 Restore 操作。差异在于，在 Restore 阶段后，还会进行 Write Recovery 操作。

Write Recovery

在 Write Recovery 阶段时，通过控制 WE (Write Enable) 信号，让 T_{w1} 和 T_{w2} 进入导通状态。此时，Bitline 会被 input 拉到逻辑 0 电平，/Bitline 则会被 /input 拉到逻辑 1 电平。

经过特定的时间后，当 Storage Capacitor 的电荷被 Discharge 到 0 状态时，就可以通过控制 Wordline，将 Storage Capacitor 的 Access Transistor 截止，写入 0 的操作就完成了。

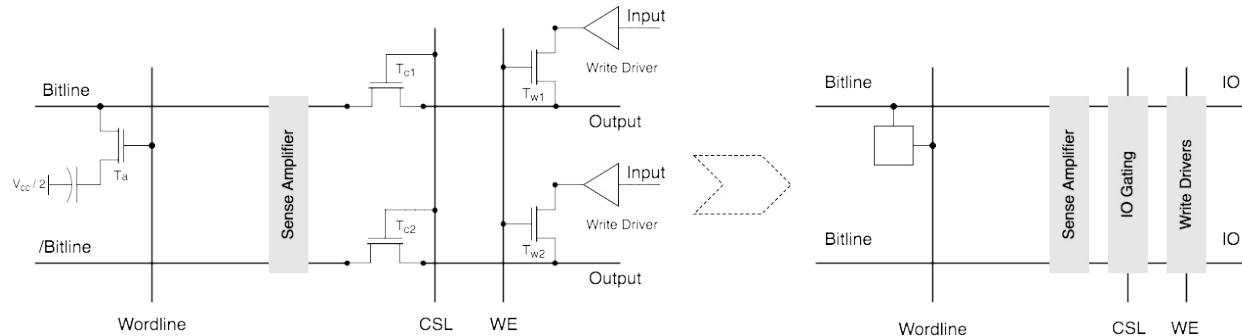
参考资料

1. Memory Systems - Cache Dram and Disk

DRAM Memory Organization

在 [DRAM Storage Cell](#) 章节中，介绍了单个 Cell 的结构。在本章节中，将介绍 DRAM 中 Cells 的组织方式。

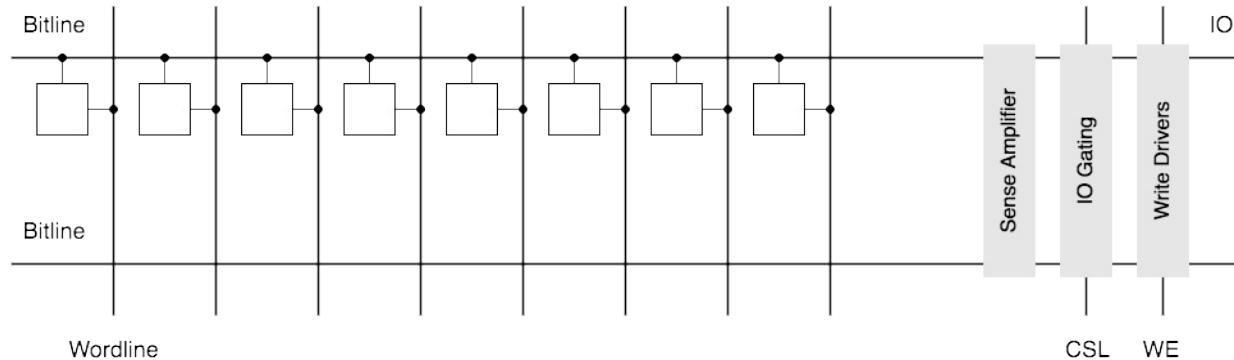
为了更清晰的描述 Cells 的组织方式，我们先对上一章节中的 DRAM Storage Cell 进行抽象，最后得到新的结构图，如下：



Memory Array

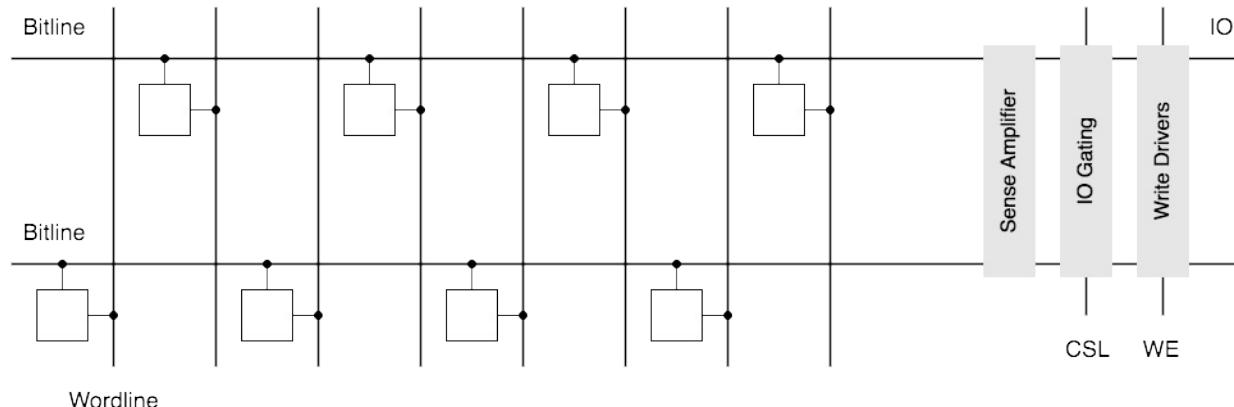
DRAM 在设计上，将所有的 Cells 以特定的方式组成一个 Memory Array。本小节将介绍 DRAM 中是如何将 Cells 以特定形式的 Memory Array 组织起来的。

首先，我们在不考虑形式的情况下，最简单的组织方式，就是在一个 Bitline 上，挂接更多的 Cells，如下图所示：

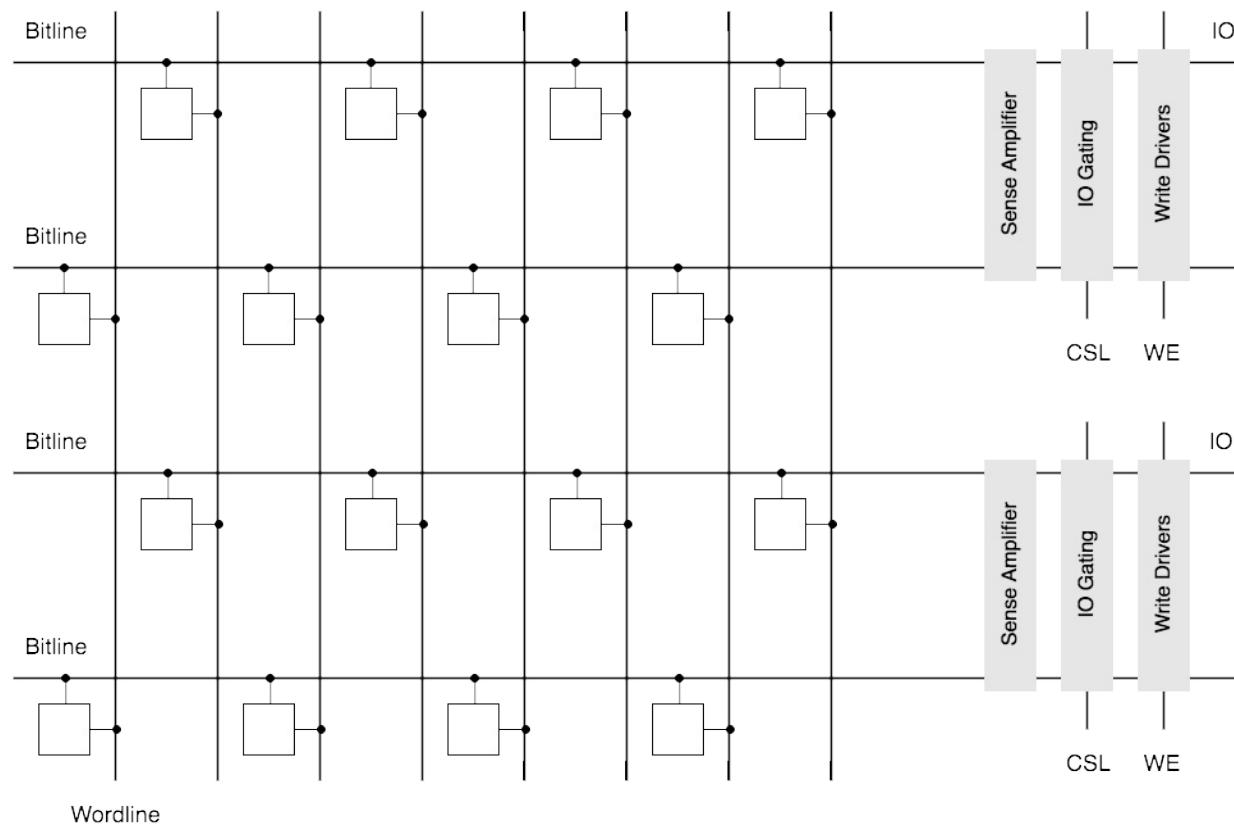


然而，在实际制造过程中，我们并不会无限制的在 Bitline 上挂接 Cells。因为 Bitline 挂接越多的 Cells，Bitline 的长度就会越长，也就意味着 Bitline 的电容值会更大，这会导致 Bitline 的信号边沿速率下降（电平从高变低或者从低变高的速率），最终导致性能的下降。为此，我们需要限制一条 Bitline 上挂接的 Cells 的总数，将更多的 Cells 挂接到其他的 Bitline 上去。

从 Cell 的结构图中，我们可以发现，在一个 Cell 的结构中，有两条 Bitline，它们在功能上是完全等价的，因此，我们可以把 Cells 分摊到不同的 Bitline 上，以减小 Bitline 的长度。然后，Cells 的组织方式就变成了如下的形式：



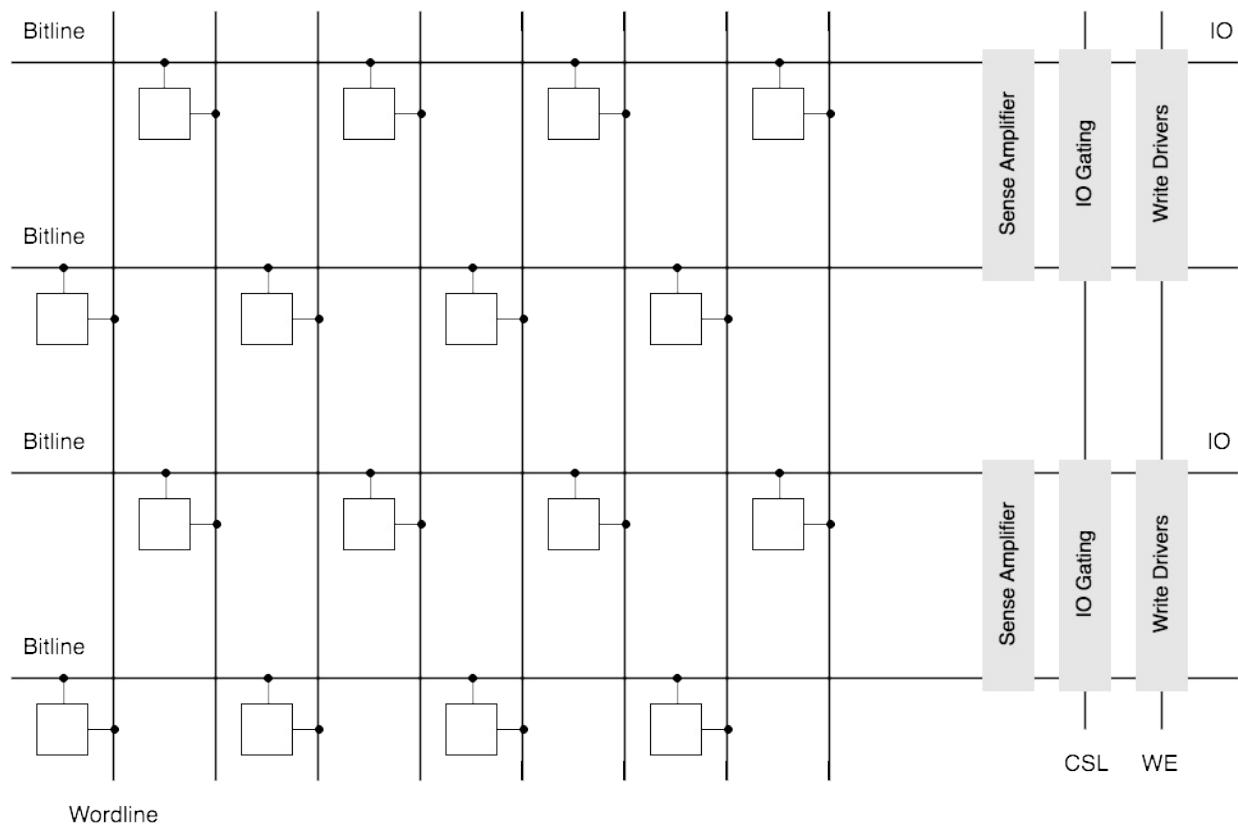
当两条 Bitline 都挂接了足够多的 Cells 后，如果还需要继续拓展，那么就只能增加 Bitline 了，增加后的结构图如下：



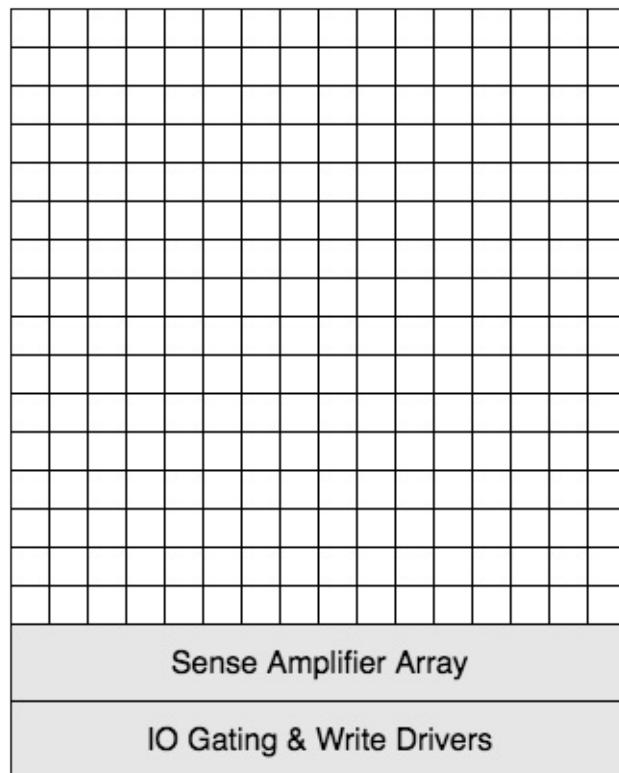
从图中我们可以看到，增加 Bitline 后，Sense Amplifier、Read Latch 和 Write Driver 的数量也相应的增加了，这意味着成本、功耗、芯片体积都会随着增加。由于这个原因，在实际的设计中，会优先考虑增加 Bitline 上挂接的 Cells 的数量，避免增加 Bitline 的数量，这也意味着，一般情况下 Wordline 的数量会比 Bitline 多很多。

上图中，呈现了一个由 16 个 Cells 组成的 Memory Array。其中的控制信号有 8 个 Wordline、2 个 CSL、2 个 WE，一次进行 1 个 Bit 的读写操，也就是可以理解为一个 $8 \times 2 \times 1$ 的 Memory Array。

如果把 2 个 CSL 和 2 个 WE 合并成 1 个 CSL 和 1 个 WE，如下图所示。此时，这个 Memory Array 就有 8 Wordline、1 个 CSL、1 个 WE，一次可以进行 2 个 Bit 的读写操作，也就是成为了 $8 \times 1 \times 2$ 的 Memory Array。



按照上述的过程，不断的增加 Cells 的数量，最终可以得到一个 $m \times n \times w$ 的 Memory Array，如下图所示



其中， m 为 Wordline 的数量、 n 为 CSL 和 WE 控制信号的数量、 w 则为一次可以进行读写操作的 Bits。

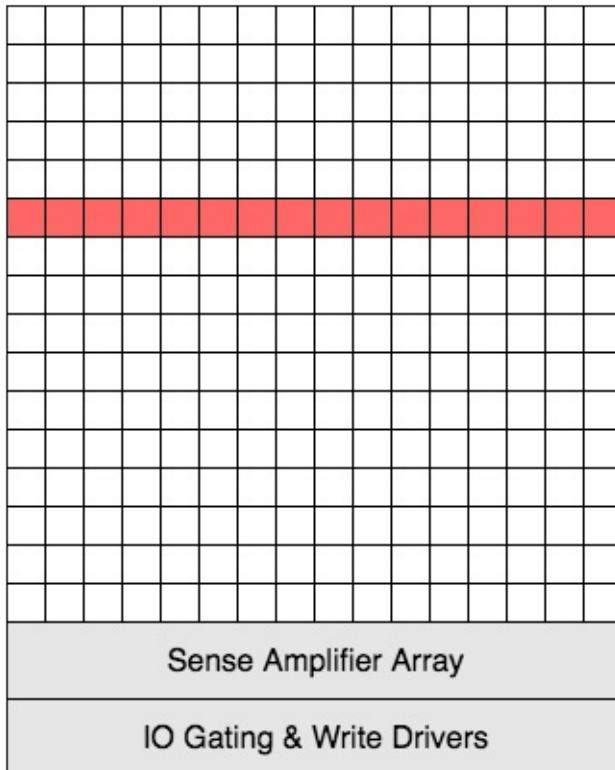
在实际的应用中，我们通常以 Rows x Columns x Data Width 来描述一个 Memory Array。后续的小节中，将对这几个定义进行介绍。

Data Width

Memory Array 的 Data Width 是指对该 Array 进行一次读写操作所访问的 Bit 位数。这个位数与 CSL 和 WE 控制线的组织方式有关。

Rows

DRAM Memory 中的 Row 与 Wordline 是一一对应的，一个 Row 本质上就是所有接在同一根 Wordline 上的 Cells，如下图所示。



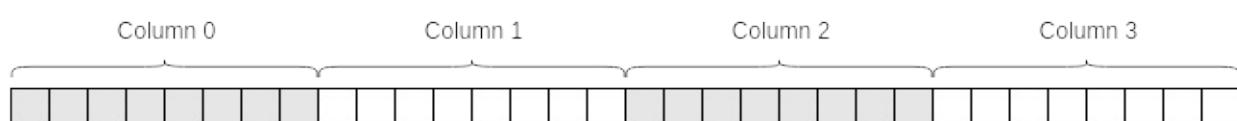
DRAM 在进行数据读写时，选中某一 Row，实质上就是控制该 Row 所对应的 Wordline，打开 Cells，并将 Cells 上的数据缓存到 Sense Amplifiers 上。

Row Size

一个 Row 的 Size 即为一个 Row 上面的 Cells 的数量。其中一个 Cell 存储 1 个 Bit 的信息，也就是说，Row Size 即为一个 Row 所存储的 Bit 位数。

Columns

Column 是 Memory Array 中可寻址的最小单元。一个 Row 中有 n 个 Column，其中 $n = \text{Row Size} / \text{Data Width}$ 。下图是 Row Size 为 32，Data Width 为 8 时，Column 的示例。



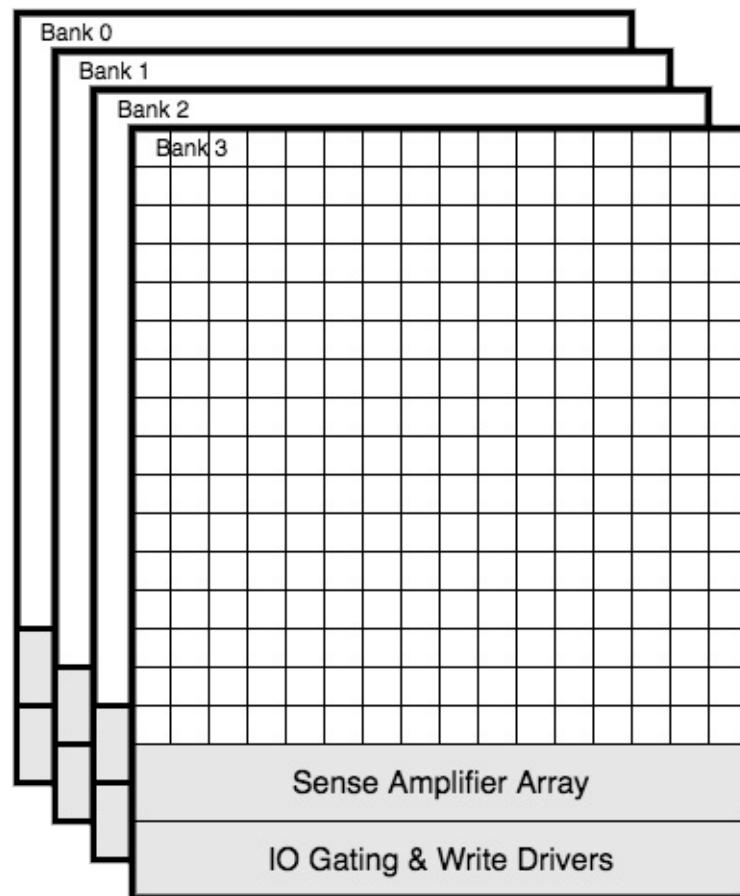
Column Size

一个 Column 的 Size 即为该 Column 上所包含的 Cells 的数量，与 Data Width 相同。Column Size 和 Data Width 在本质上是一样的，也是与 CSL 和 WE 控制线的组织方式有关（参考 [Memory Array](#) 小节中关于 CSL 的描述）。

Memory Bank

随着 Bitline 数量的不断增加，Wordline 上面挂接的 Cells 也会越来越多，Wordline 会越来越长，继而也会导致电容变大，边沿速率变慢，性能变差。因此，一个 Memory Array 也不能无限制的扩大。

为了在不减损性能的基础上进一步增加容量，DRAM 在设计上将多个 Memory Array 堆叠到一起，如下图所示：



其中的每一个 Memory Array 称为一个 Bank，每一个 Bank 的 Rows、Columns、Data Width 都是一样的。在 DRAM 的数据访问时，只有一个 Bank 会被激活，进行数据的读写操作。

以下是一个 DRAM Memory Organization 的例子：

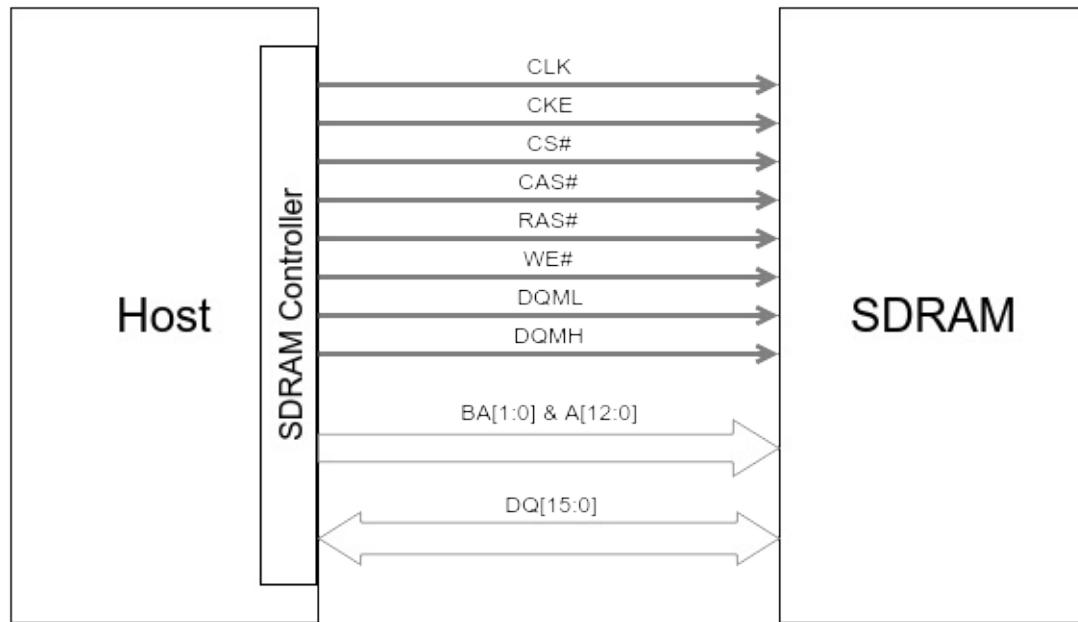
Banks	4
Rows / Bank	16K
Columns / Row	1024
Column Size	16 Bits

DRAM Device

在前面的章节中，介绍了 DRAM Cell 和 Memory Array。在此章节中，将以 SDR SDRAM 为例，描述 DRAM Device 与 Host 端的接口，以及其内部的其他模块，包括 Control Logic、IO、Row & Column Decoder 等。

SDRAM Interface

SDR SDRAM 是 DRAM 的一种，它与 Host 端的硬件接口如下图所示：



总线上各个信号的描述如下表所示：

Symbol	Type	Description
CLK	Input	从 Host 端输出的同步时钟信号
CKE	Input	用于指示 CLK 信号是否有效，SDRAM 会根据此信号进入或者退出 Power down、Self-refresh 等模式
CS#	Input	Chip Select 信号
CAS#	Input	Column Address Strobe，列地址选通信号
RAS#	Input	Row Address Strobe，行地址选通信号
WE#	Input	Write Enable，写使能信号
DQML	Input	当进行写数据时，如果该 DQML 为高，那么 DQ[7:0] 的数据会被忽略，不写入到 DRAM
DQMH	Input	当进行写数据时，如果该 DQMH 为高，那么 DQ[15:8] 的数据会被忽略，不写入到 DRAM
BA[1:0]	Input	Bank Address，用于选择操作的 Memory Bank
A[12:0]	Input	Address 总线，用于传输行列地址
DQ[15:0]	I/O	Data 总线，用于传输读写的数据内容

SDRAM Operations

Host 与 SDRAM 之间的交互都是由 Host 以 Command 的形式发起的。一个 Command 由多个信号组合而成，下面表格中描述了主要的 Command。

Command	CS#	RAS#	CAS#	WE#	DQM	BA[1:0] & A[12:0]	DQ[15:0]
Active	L	L	H	H	X	Bank & Row	X
Read	L	H	L	H	L/H	Bank & Col	X
Write	L	H	L	L	L/H	Bank & Col	Valid
Precharge	L	L	H	L	X	Code	X
Auto-refresh	L	L	L	H	X	X	X
Self-refresh	L	L	L	H	X	X	X
Load Mode Register	L	L	L	L	X	REG Value	X

Active

Active Command 会通过 BA[1:0] 和 A[12:0] 信号，选中指定 Bank 中的一个 Row，并打开该 Row 的 wordline。在进行 Read 或者 Write 前，都需要先执行 Active Command。

Read

Read Command 将通过 A[9:0] 信号，发送需要读取的 Column 的地址给 SDRAM。然后 SDRAM 再将 Active Command 所选中的 Row 中，将对应 Column 的数据通过 DQ[15:0] 发送给 Host。如果 A10 地址线为 1，那么在 Read Command 结束后，DRAM 会自动执行一次 Precharge 操作，即 Auto-Precharge。

Host 端发送 Read Command，到 SDRAM 将数据发送到总线上的需要的时钟周期个数定义为 CL。

Write

Write Command 将通过 A[9:0] 信号，发送需要写入的 Column 的地址给 SDRAM，同时通过 DQ[15:0] 将待写入的数据发送给 SDRAM。然后 SDRAM 将数据写入到 Activated Row 的指定 Column 中。如果 A10 地址线为 1，那么在 Read Command 结束后，DRAM 会自动执行一次 Precharge 操作，即 Auto-Precharge。

SDRAM 接收到最后一个数据到完成数据写入到 Memory 的时间定义为 tWR (Write Recovery)。

Precharge

在进行下一次的 Read 或者 Write 操作前，必须要先执行 Precharge 操作。（具体的细节可以参考 [DRAM Storage Cell](#) 章节）

Precharge 操作是以 Bank 为单位进行的，可以单独对某一个 Bank 进行，也可以一次对所有 Bank 进行。如果 A10 为高，那么 SDRAM 进行 All Bank Precharge 操作，如果 A10 为低，那么 SDRAM 根据 BA[1:0] 的值，对指定的 Bank 进行 Precharge 操作。

SDRAM 完成 Precharge 操作需要的时间定义为 tRP。

Auto-Refresh

DRAM 的 Storage Cell 中的电荷会随着时间慢慢减少，为了保证其存储的信息不丢失，需要周期性的对其进行刷新操作。

SDRAM 的刷新是按 Row 进行，标准中定义了在一个刷新周期内（常温下 64ms，高温下 32ms）需要完成一次所有 Row 的刷新操作。

为了简化 SDRAM Controller 的设计，SDRAM 标准定义了 Auto-Refresh 机制，该机制要求 SDRAM Controller 在一个刷新周期内，发送 8192 个 Auto-Refresh Command，即 AR，给 SDRAM。

SDRAM 每收到一个 AR，就进行 n 个 Row 的刷新操作，其中，n = 总的 Row 数量 / 8192。

此外，SDRAM 内部维护一个刷新计数器，每完成一次刷新操作，就将计数器更新为下一次需要进行刷新操作的 Row。

刷新操作通常是在 SDRAM 的所有 bank 上同时进行的，例如 SDRAM 有 4 个 Bank 时，执行一次 AR 操作时，每个 Bank 上同时进行 $n / 4$ 个 Row 的刷新操作。

一般情况下，SDRAM Controller 会周期性的发送 AR，每两个 AR 直接的时间间隔定义为 $tREFI = 64ms / 8192 = 7.8 \mu s$ 。

SDRAM 完成一次刷新操作所需要的时间定义为 tRFC，这个时间会随着 SDRAM Row 的数量的增加而变大。

由于 AR 会占用总线，阻塞正常的数据请求，同时 SDRAM 在执行 refresh 操作是很费电，所以在 SDRAM 的标准中，还提供了一些优化的措施，例如 DRAM Controller 可以最多延时 8 个 tREFI 后，再一起把 8 个 AR 同时发出。

更多相关的优化可以参考《大容量 DRAM 的刷新开销问题及优化技术综述》文中的描述。

Self-Refresh

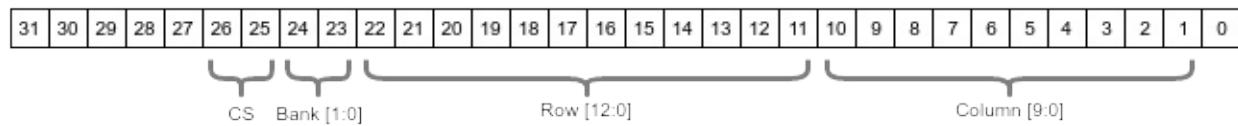
Host 还可以让 SDRAM 进入 Self-Refresh 模式，降低功耗。在该模式下，Host 不能对 SDRAM 进行读写操作，SDRAM 内部自行进行刷新操作保证数据的完整。通常在设备进入待机状态时，Host 会让 SDRAM 进入 Self-Refresh 模式，以节省功耗。

更多各个 Command 相关的细节，可以参考后续的 [DRAM Timing](#) 章节。

Address Mapping

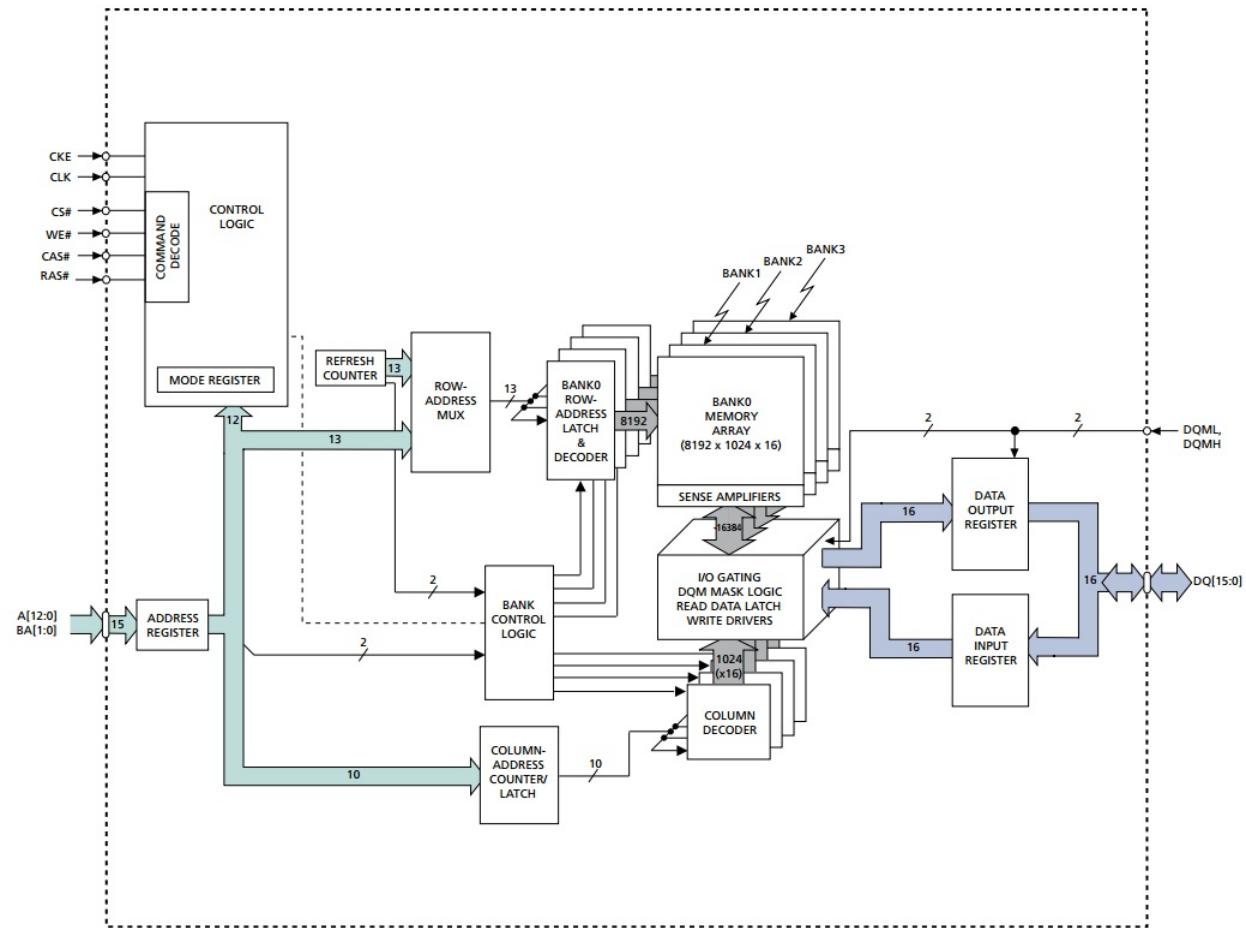
SDRAM Controller 的主要功能之一是将 CPU 对指定物理地址的内存访问操作，转换为 SDRAM 读写时序，完成数据的传输。

在实际的产品中，通常需要考虑 CPU 中的物理地址到 SDRAM 的 CS、Bank、Row 和 Column 地址映射。下图是一个 32 位物理地址映射的一个例子：



SDRAM 内部结构

如图所示，DRAM Device 内部主要有 Control Logic、Memory Array、Decoders、Reflash Counter 等模块。在后续的小节中，将逐一介绍各个模块的主要功能。



Control Logic

Control Logic 的主要功能是解析 SDRAM Controller 发出的 Command，然后根据具体的 Command 做具体内部模块的控制，例如：选中指定的 Bank、触发 refresh 等的操作。

Control Logic 包含了 1 个或者多个 Mode Register。该 Register 中包含了时序、数据模式等的配置，更多的细节会在 [DRAM Timing](#) 章节进行描述。

Row & Column Decoder

Row Decoder 的主要功能是将 Active Command 所带的 Row Address 映射到具体的 wordline，最终打开指定的 Row。同样 Column Decoder 则是把 Column Address 映射到具体的 csl，最终选中特定的 Column。

Memory Array

Memory Array 是存储信息的主要模块，具体细节可以参考 [DRAM Memory Organization](#) 章节的描述。

IO

IO 电路主要是用于处理数据的缓存、输入和输出。其中 Data Latch 和 Data Register 用于缓存数据，DQM Mask Logic 和 IO Gating 等则用于输入输出的控制。

Refresh Counter

Refresh Counter 用于记录下次需要进行 refresh 操作的 Row。在接收到 AR 或者在 Self-Refresh 模式下，完成一次 refresh 后，Refresh Counter 会进行更新。

不同类型的 SDRAM

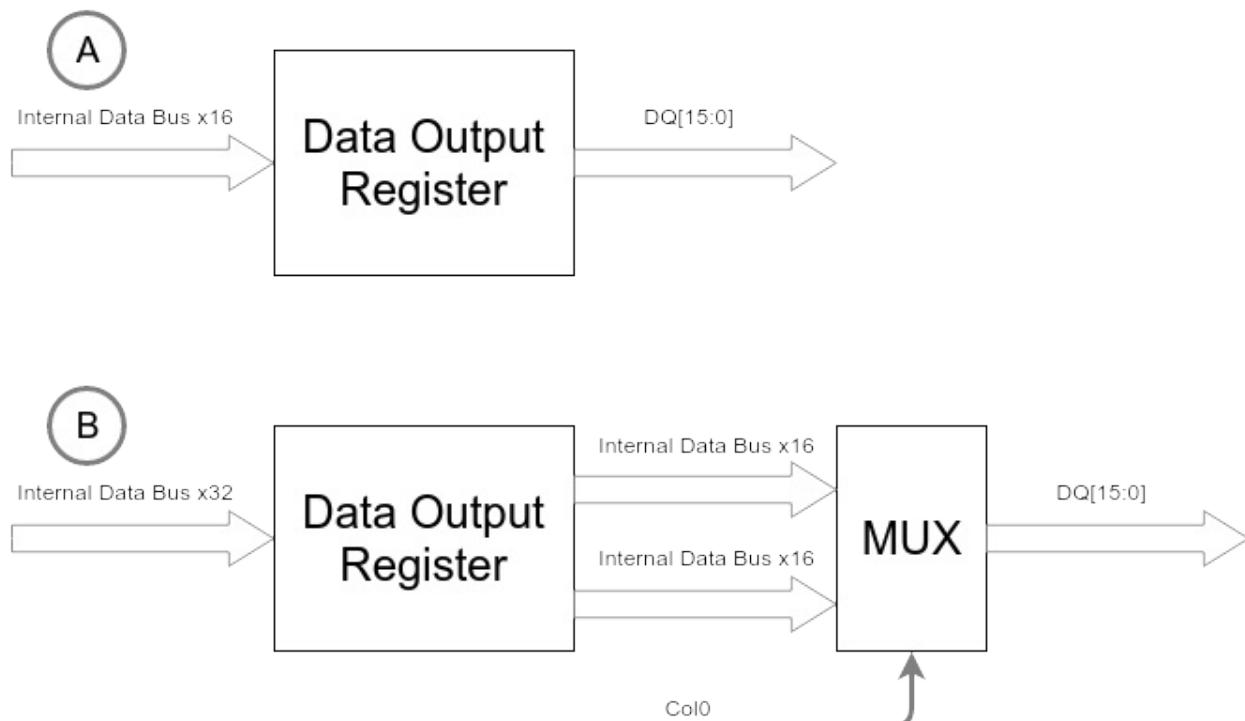
目前市面上在使用的 DRAM 主要有 SDR、DDR、LPDDR、GDDR 这几类，后续小节中，将对各种类型的 DRAM 进行简单的介绍。

SDR 和 DDR

SDR (Single Data Rate) SDRAM 是第一个引入 Clock 信号的 DRAM 产品，SDR 在 Clock 的上升沿进行总线信号的处理，一个时钟周期内可以传输一组数据。

DDR (Double Data Rate) SDRAM 是在 SDR 基础上的一个更新。DDR 内部采用 2n-Prefetch 架构，相对于 SDR，在一个读写周期内可以完成 2 倍宽度数据的预取，然后在 Clock 的上升沿和下降沿都进行数据传输，最终达到在相同时钟频率下 2 倍于 SDR 的数据传输速率。（更多 2n-Prefetch 相关的细节可以参考《Micron Technical Note - General DDR SDRAM Functionality》文中的介绍）

Prefetch 的基本原理如下图所示。在示例 B 中，内部总线宽度是 A 的 2 倍，在一次操作周期内，可以将两倍于 A 的数据传输到 Output Register 中，接着外部 IO 电路再以 2 倍于 A 的频率将数据呈现到总线上，最终实现 2 倍 A 的传输速率。



DDR 后续还有 DDR2、DDR3、DDR4 的更新，基本上每一代都通过更多的 Prefetch 和更高的时钟频率，达到 2 倍于上一代的数据传输速率。

DDR SDRAM Standard	Bus clock (MHz)	Internal rate (MHz)	Prefetch (min burst)	Transfer Rate (MT/s)	Voltage
DDR	100–200	100–200	2n	200–400	2.5/2.6
DDR2	200–533.33	100–266.67	4n	400–1066.67	1.8
DDR3	400–1066.67	100–266.67	8n	800–2133.33	1.5
DDR4	1066.67–2133.33	133.33–266.67	8n	2133.33–4266.67	1.05/1.2

Transfer Rate (MT/s) 为每秒发生的 Transfer 的数量，一般为 Bus Clock 的 2 倍（一个 Clock 周期内，上升沿和下降沿各有一个 Transfer）

Internal rate (MHz) 则是内部 Memory Array 读写的频率。由于 SDRAM 采用电容作为存储介质，由于工艺和物理特性的限制，电容充放电的时间难以进一步的缩短，所以内部 Memory Array 的读写频率也受到了限制，目前最高能到 266.67 MHz，这也是 SDR 到 DDR 采用 Prefetch 架构的主要原因。

Memory Array 读写频率受到限制，那就只能在读写宽度上做优化，通过增加单次读写周期内操作的数据宽度，结合总线和 IO 频率的增加来提高整体传输速率。

LPDDRx

LPDDR，即 Low Power DDR SDRAM，主要是用在移动设备上，例如手机、平板等。相对于 DDR，LPDDR 采用了更低的工作电压、Partial Array Self-Refresh 等机制，降低整体的功耗，以满足移动设备的低功耗需求。

GDDRx

GDDR，即 Graphic DDR，主要用在显卡设备上。相对于 DDR，GDDR 具有更高的性能、更低的功耗、更少的发热，以满足显卡设备的计算需求。

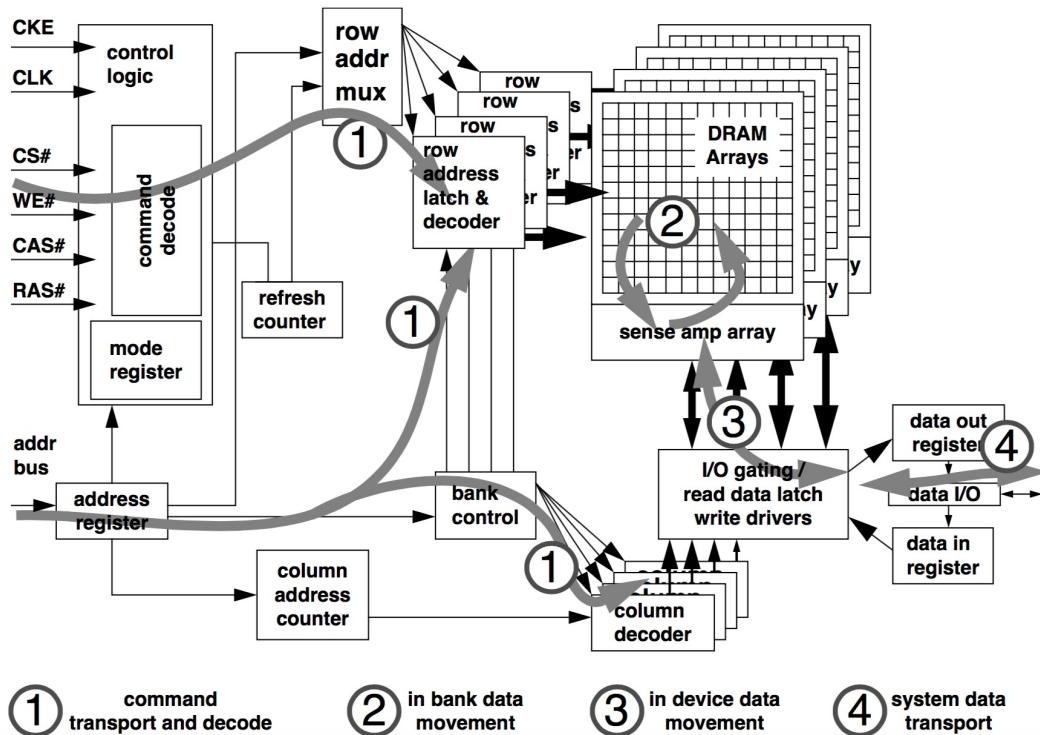
参考资料

1. Memory Systems - Cache Dram and Disk
2. 大容量 DRAM 的刷新开销问题及优化技术综述 [PDF]
3. Micron Technical Note - General DDR SDRAM Functionality [PDF]
4. [Everything You Need To Know About DDR, DDR2 and DDR3 Memories](#) [WEB]
5. [記憶體10年技術演進史](#) [WEB]

DRAM Timing

在 DRAM Device 章节中，我们简单介绍了 SDRAM 的 Active、Read、Write 等的操作，在本章节中，我们将详细的介绍各个操作的时序。

Overview



如上图所示，SDRAM 的相关操作在内部大概可以分为以下几个阶段：

1. Command transport and decode

在这个阶段，Host 端会通过 Command Bus 和 Address Bus 将具体的 Command 以及相应参数传递给 SDRAM。SDRAM 接收并解析 Command，接着驱动内部模块进行相应的操作。

2. In bank data movement

在这个阶段，SDRAM 主要是将 Memory Array 中的数据从 DRAM Cells 中读出到 Sense Amplifiers，或者将数据从 Sense Amplifiers 写入到 DRAM Cells。

3. In device data movement

这个阶段中，数据将通过 IO 电路缓存到 Read Latches 或者通过 IO 电路和 Write Drivers 更新到 Sense Amplifiers。

4. System data transport

在这个阶段，进行读数据操作时，SDRAM 会将数据输出到数据总线上，进行写数据操作时，则是 Host 端的 Controller 将数据输出到总线上。

在上述的四个阶段中，每个阶段都会有一定的耗时，例如数据从 DRAM Cells 搬运到 Read Latches 的操作需要一定的时间，因此在一个具体的操作需要按照一定时序进行。

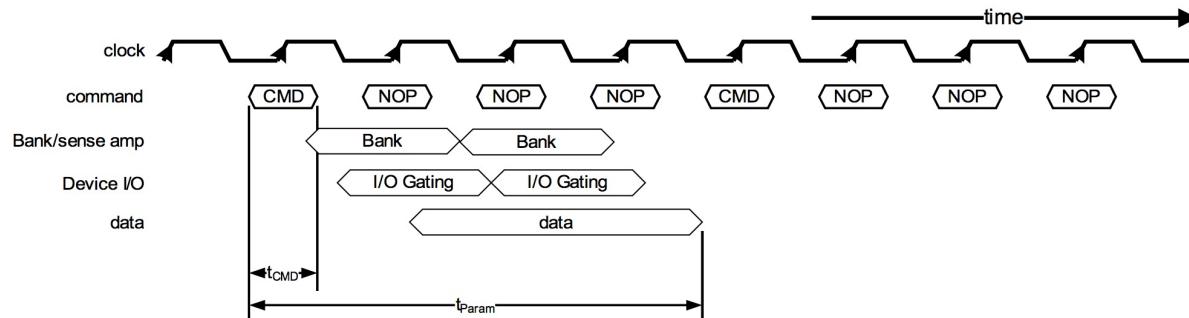
同时，由于内部的一些部件可能会被多个操作使用，例如读数据和写数据都需要用到部分 IO 电路，因此多个不同的操作通常不能同时进行，也需要遵守一定的时序。

此外，某些操作会消耗很大的电流，为了满足 SDRAM 设计上的功耗指标，可能会限制某些操作的执行频率。

基于上面的几点限制，SDRAM Controller 在发出 Command 时，需要遵守一定的时序和规则，这些时序和规则由相应的 SDRAM 标准定义。在后续的小节中，我们将对各个 Command 的时序进行详细的介绍。

时序图例

后续的小节中，我们将通过下图类似的时序图，来描述各个 Command 的详细时序。



上图中，Clock 信号是由 SDRAM Controller 发出的，用于和 DRAM 之间的同步。在 DDRx 中，Clock 信号是一组差分信号，在本文中为了简化描述，将只画出其中的 Positive Clock。

Controller 与 DRAM 之间的交互，都是以 Controller 发起一个 Command 开始的。从 Controller 发出一个 Command 到 DRAM 接收并解析该 Command 所需要的时间定义为 t_{CMD} ，不同类型的 Command 的 t_{CMD} 都是相同的。

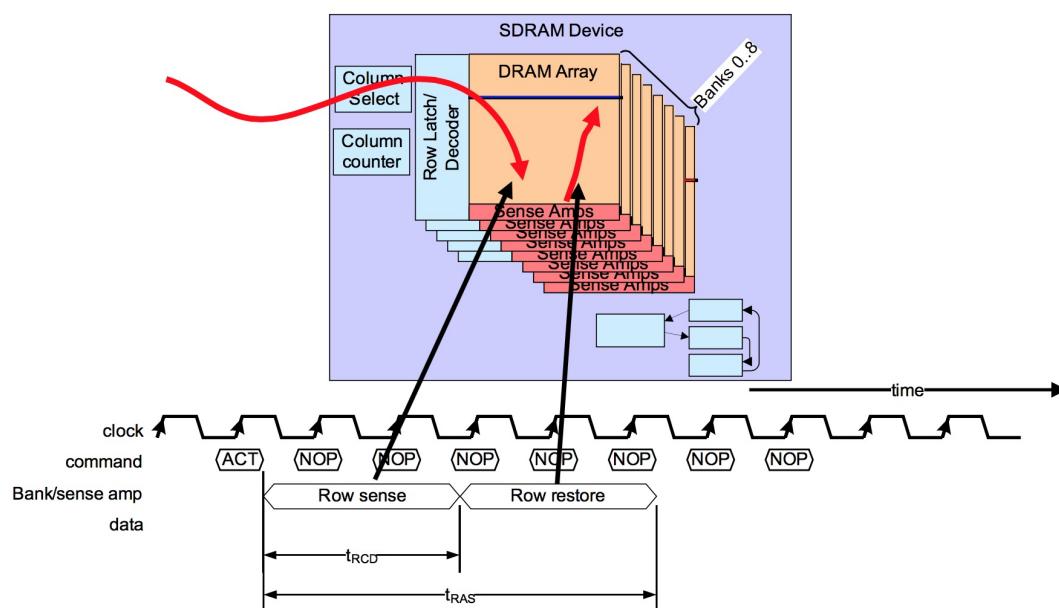
DRAM 在成功解析 Command 后，就会根据 Command 在内部进行相应的操作。从 Controller 发出 Command 到 DRAM 执行完 Command 所对应的操作所需要的时间定义为 t_{Param} 。不同类型的 Command 的 t_{Param} 可能不一样，相同 Command 的 t_{Param} 由于 Command 参数的不同也可能会不一样。

NOTE:

各种 Command 的定义和内部操作细节可以参考前面的几个章节，本章节中将主要关注时序方面的细节。

Row Active Command

在进行数据的读写前，Controller 需要先发送 Row Active Command，打开 DRAM Memory Array 中的指定的 Row。Row Active Command 的时序如下图所示：



Row Active Command 可以分为两个阶段：

Row Sense

Row Active Command 通过地址总线指明需要打开某一个 Bank 的某一个 Row。

DRAM 在接收到该 Command 后，会打开该 Row 的 Wordline，将其存储的数据读取到 Sense Amplifiers 中，这一时间定义为 tRCD (RCD for Row Address to Column Address Delay)。

DRAM 在完成 Row Sense 阶段后，Controller 就可以发送 Read 或 Write Command 进行数据的读写了。这也意味着，Controller 在发送 Row Active Command 后，需要等待 tRCD 时间才能接着发送 Read 或者 Write Command 进行数据的读写。

Row Restore

由于 DRAM 的特性，Row 中的数据在被读取到 Sense Amplifiers 后，需要进行 Restore 的操作（细节请参考 [DRAM Storage Cell 章节](#)）。Restore 操作可以和数据的读取同时进行，即在这个阶段，Controller 可能发送了 Read Command 进行数据读取。

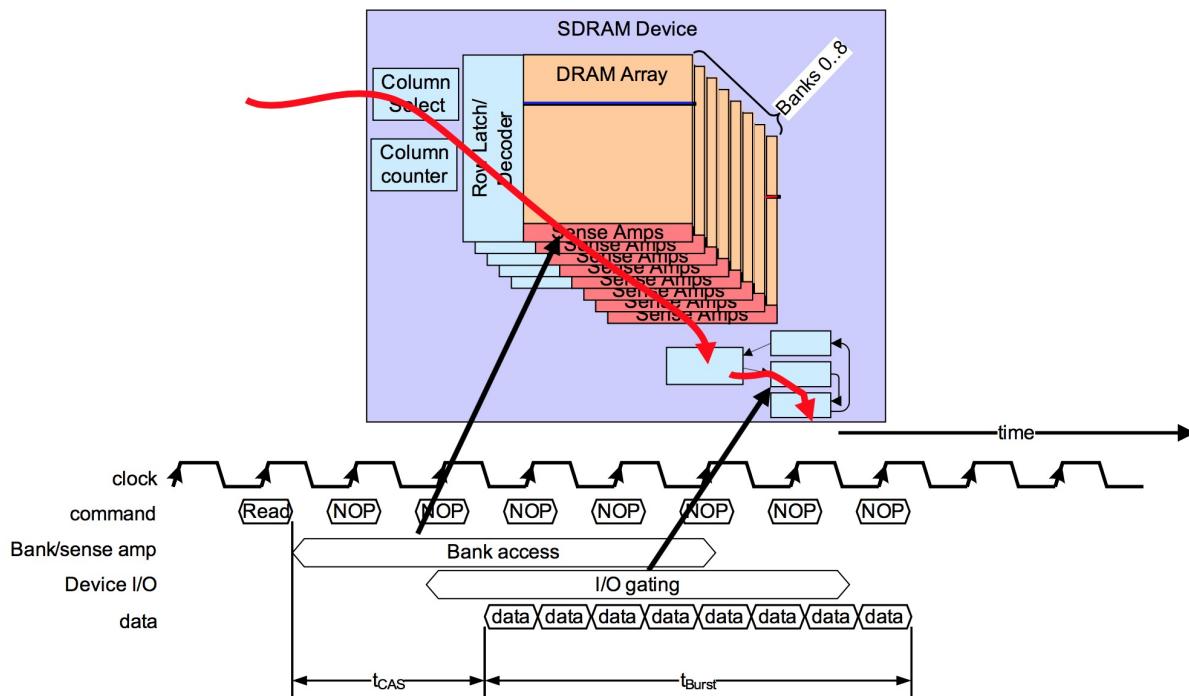
DRAM 接收到 Row Active Command 到完成 Row Restore 操作所需要的时间定义为 tRAS (RAS for Row Address Strobe)。

Controller 在发出一个 Row Active Command 后，必须要等待 tRAS 时间后，才可以发起另一次的 Precharge 和 Row Access。

Column Read Command

Controller 发送 Row Active Command 并等待 tRCD 时间后，再发送 Column Read Command 进行数据读取。

数据 Burst Length 为 8 时的 Column Read Command 时序如下图所示：



Column Read Command 通过地址总线 A[0:9] 指明需要读取的 Column 的起始地址。DRAM 在接收到该 Command 后，会将数据从 Sense Amplifiers 中通过 IO 电路搬运到数据总线上。

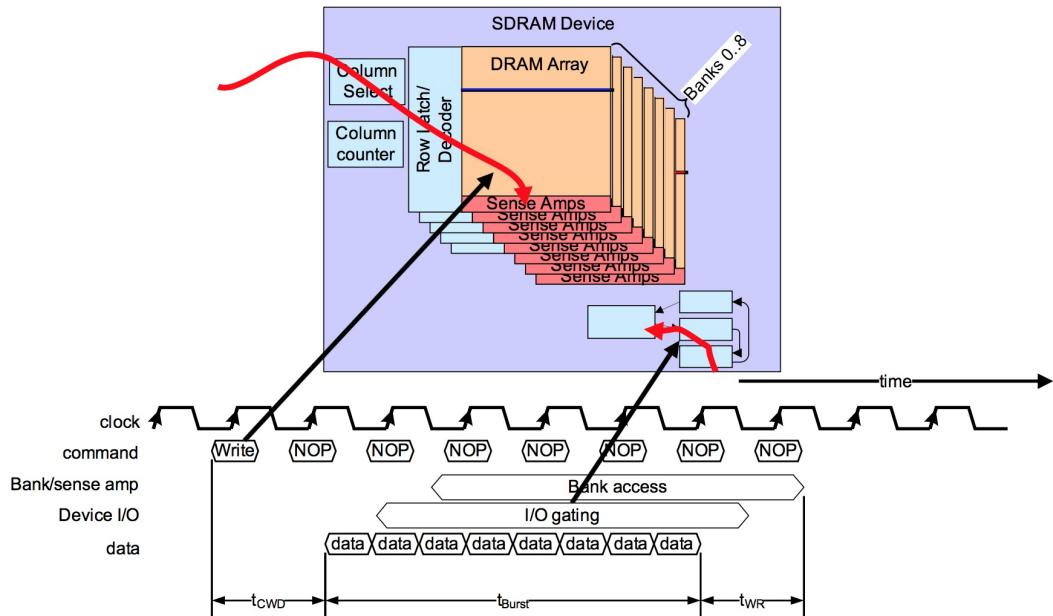
DRAM 从接收到 Command 到第一组数据从数据总线上输出的时间称为 tCAS (CAS for Column Address Strobe)，也称为 tCL (CL for CAS Latency)，这一时间可以通过 mode register 进行配置，通常为 3~5 个时钟周期。

DRAM 在接收到 Column Read Command 的 tCAS 时间后，会通过数据总线，将 n 个 Column 的数据逐个发送给 Controller，其中 n 由 mode register 中的 burst length 决定，通常可以将 burst length 设定为 2、4 或者 8。

开始发送第一个 Column 数据，到最后一个 Column 数据的时间定义为 tBurst。

Column Write Command

Controller 发送 Row Active Command 并等待 tRCD 时间后，再发送 Column Write Command 进行数据写入。数据 Burst Length 为 8 时的 Column Write Command 时序如下图所示：



Column Write Command 通过地址总线 A[0:9] 指明需要写入数据的 Column 的起始地址。Controller 在发送完 Write Command 后，需要等待 tCWD (CWD for Column Write Delay) 时间后，才可以发送待写入的数据。tCWD 在一些描述中也称为 tCWL (CWL for Column Write Latency)

tCWD 在不同类型的 SDRAM 标准有所不同：

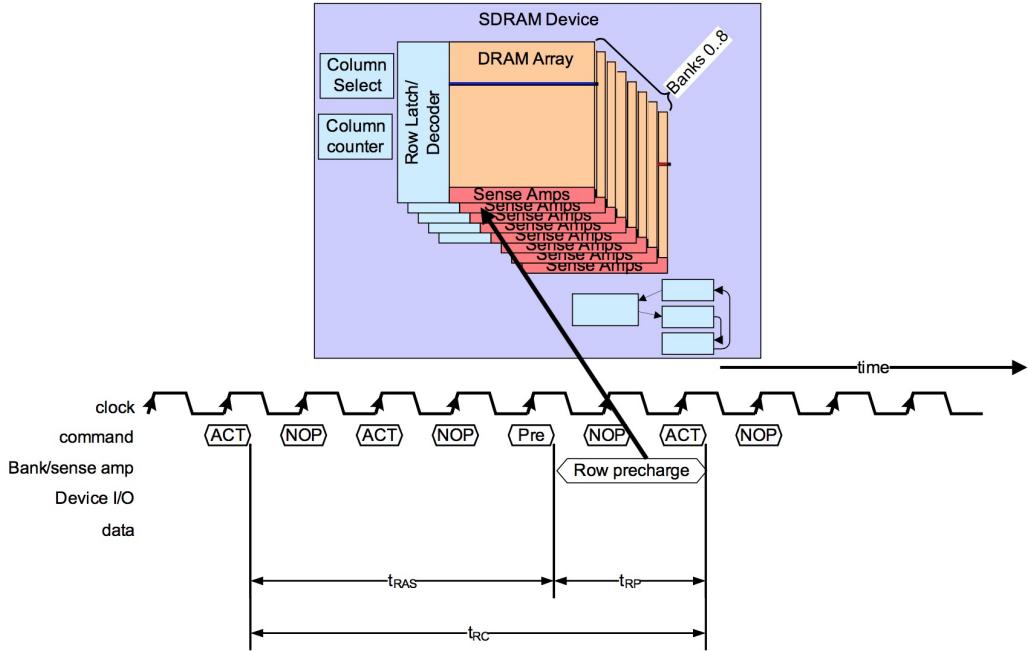
Memory Type	tCWD
SDRAM	0 cycles
DDR SDRAM	1 cycle
DDR2 SDRAM	tCAS - 1 cycle
DDR3 SDRAM	programmable

DRAM 接收完数据后，需要一定的时间将数据写入到 DRAM Cells 中，这个时间定义为 tWR (WR for Write Recovery)。

Precharge Command

在 [DRAM Storage Cell](#) 章节中，我们了解到，要访问 DRAM Cell 中的数据，需要先进行 Precharge 操作。相应地，在 Controller 发送 Row Active Command 访问一个具体的 Row 前，Controller 需要发送 Precharge Command 对该 Row 所在的 Bank 进行 Precharge 操作。

下面的时序图描述了 Controller 访问一个 Row 后，执行 Precharge，然后再访问另一个 Row 的流程。



DRAM 执行 Precharge Command 所需要的时间定义为 tRP (RP for Row Precharge)。Controller 在发送一个 Row Active Command 后，需要等待 tRC (RC for Row Cycle) 时间后，才能发送第二个 Row Active Command 进行另一个 Row 的访问。

从时序图上我们可以看到， $t_{RC} = t_{RAS} + t_{RP}$ ， t_{RC} 时间决定了访问 DRAM 不同 Row 的性能。在实际的产品中，通常会通过降低 t_{RC} 耗时或者在一个 Row Cycle 执行尽可能多数据读写等方式来优化性能。

NOTE :

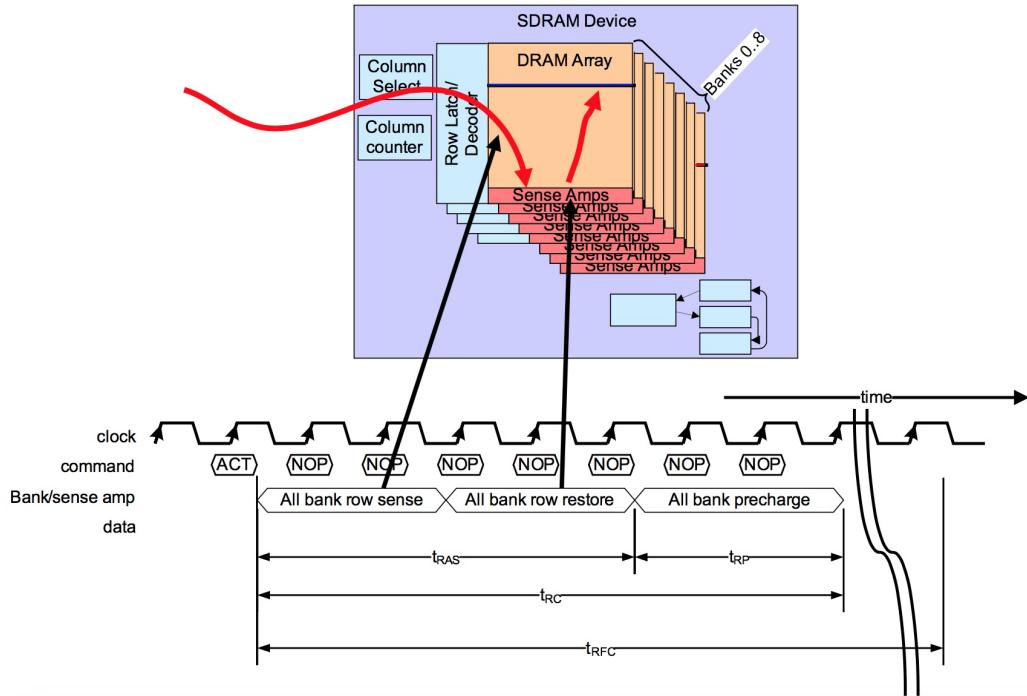
在一个 Row Cycle 中，发送 Row Active Command 打开一个 Row 后，Controller 可以发起多个 Read 或者 Write Command 进行一个 Row 内的数据访问。这种情况下，由于不用进行 Row 切换，数据访问的性能会比需要切换 Row 的情况好。

在一些产品上，DRAM Controller 会利用这一特性，对 CPU 发起的内存访问进行调度，在不影响数据有效性的情况下，将同一个 Row 上的数据访问汇聚到一起执行，以提供整体访问性能。

Row Refresh Command

一般情况下，为了保证 DRAM 数据的有效性，Controller 每隔 tREFI (REFI for Refresh Interval) 时间就需要发送一个 Row Refresh Command 给 DRAM，进行 Row 刷新操作。DRAM 在接收到 Row Refresh Command 后，会根据内部 Refresh Counter 的值，对所有 Bank 的一个或者多个 Row 进行刷新操作。

DRAM 刷新的操作与 Active + Precharge Command 组合类似，差别在于 Refresh Command 是对 DRAM 所有 Bank 同时进行操作的。下图为 DRAM Row Refresh Command 的时序图：



DRAM 完成刷新操作所需的时间定义为 tRFC (RFC for Refresh Cycle)。

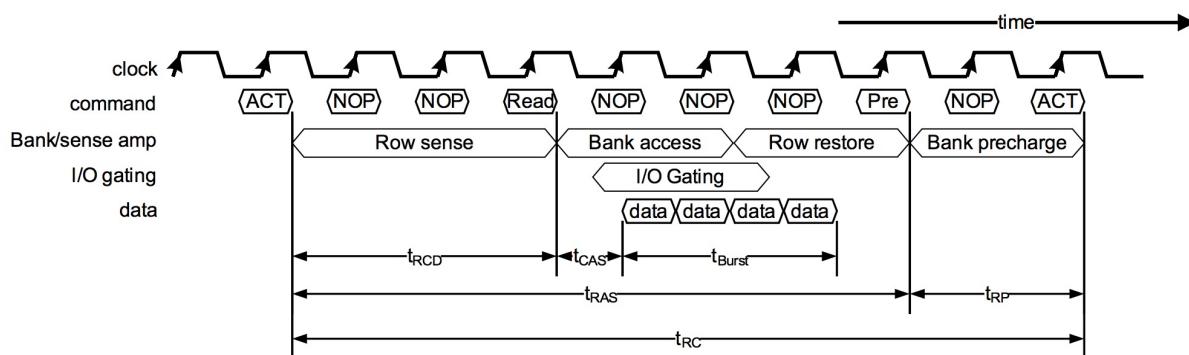
tRFC 包含两个部分的时间，一是完成刷新操作所需要的时间，由于 DRAM Refresh 是同时对所有 Bank 进行的，刷新操作会比单个 Row 的 Active + Precharge 操作需要更长的时间；tRFC 的另一部分时间则是为了降低平均功耗而引入的延时，DRAM Refresh 操作所消耗的电流会比单个 Row 的 Active + Precharge 操作要大的多，tRFC 中引入额外的时延可以限制 Refresh 操作的频率。

NOTE:

在 DDR3 SDRAM 上，tRFC 最小的值大概为 110ns，tRC 则为 52.5ns。

Read Cycle

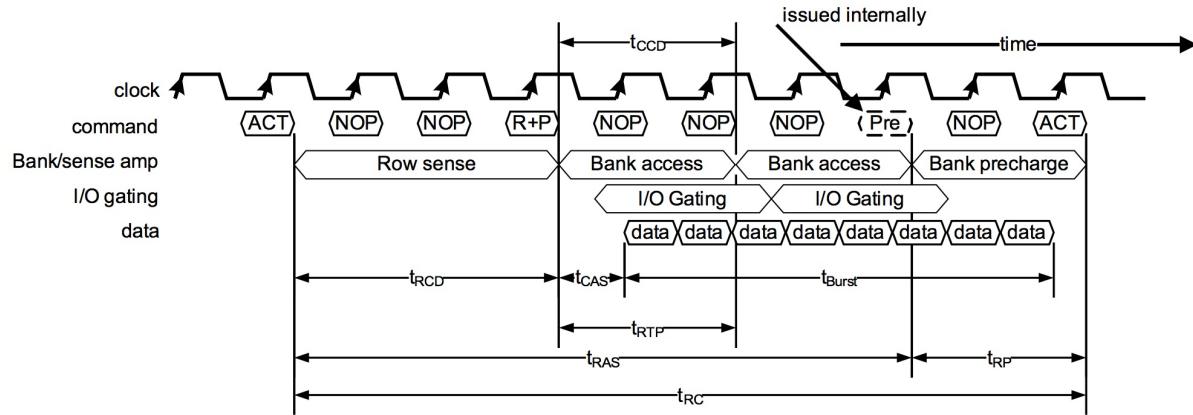
一个完整的 Burst Length 为 4 的 Read Cycle 如下图所示：



Read Command With Auto Precharge

DRAM 还可以支持 Auto Precharge 机制。在 Read Command 中的地址线 A10 设为 1 时，就可以触发 Auto Precharge。此时 DRAM 会在完成 Read Command 后的合适的时机，在内部自动执行 Precharge 操作。

Read Command With Auto Precharge 的时序如下图所示：



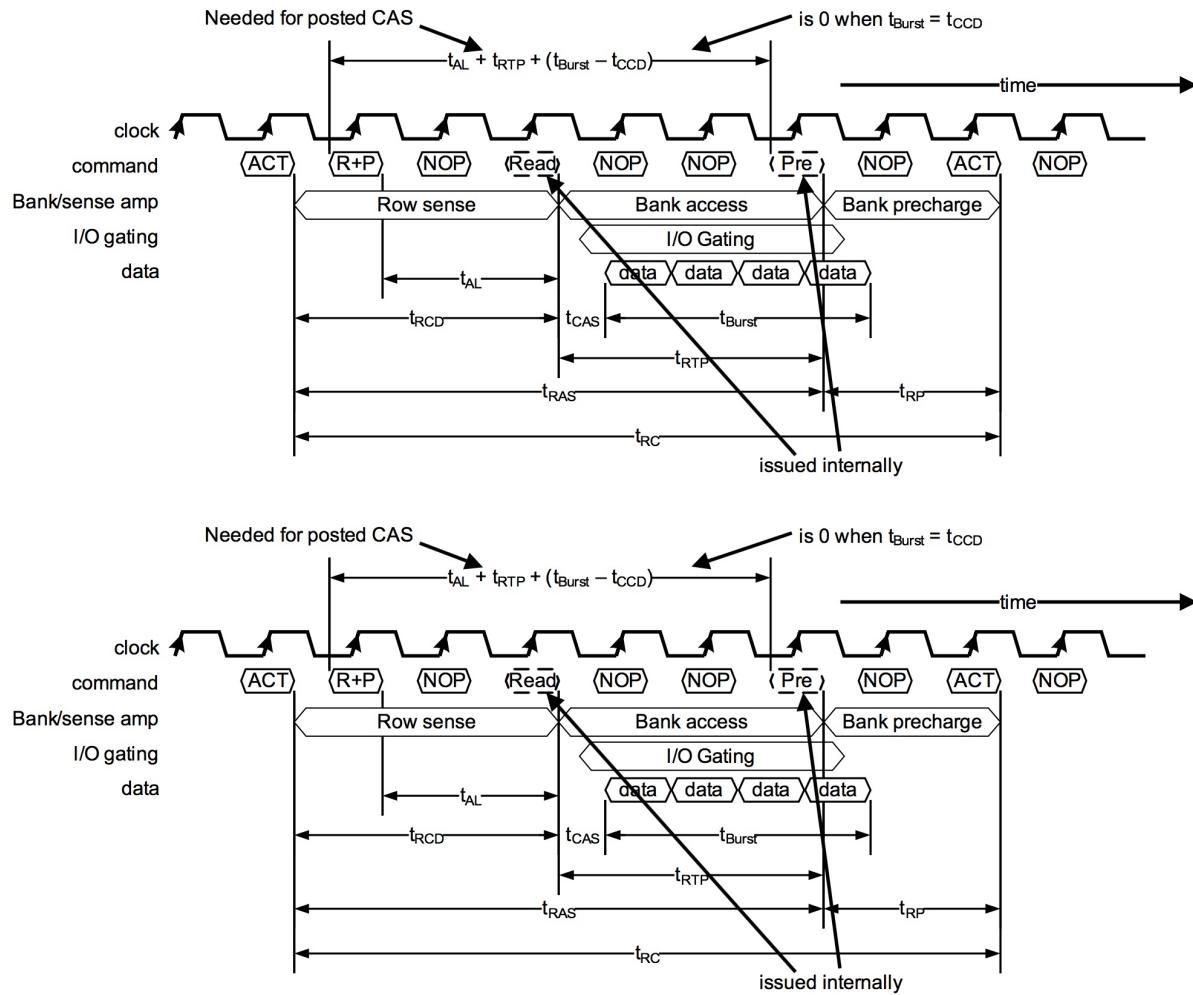
Auto Precharge 机制的引入，可以降低 Controller 实现的复杂度，进而在功耗和性能上带来改善。

NOTE:

Write Command 也支持 Auto Precharge 机制，参考下一小节的时序图。

Additive Latency

在 DDR2 中，又引入了 Additive Latency 机制，即 AL。通过 AL 机制，Controller 可以在发送完 Active Command 后紧接着就发送 Read 或者 Write Command，而后 DRAM 会在合适的时机（延时 t_{AL} 时间）执行 Read 或者 Write Command。时序如下图所示：



Additive Latency 机制同样是降低了 Controller 实现的复杂度，在功耗和性能上带来改善。

DRAM Timing 设定

上述的 DRAM Timing 中的一部分参数可以编程设定，例如 tCAS、tAL、Burst Length 等。这些参数通常是在 Host 初始化时，通过 Controller 发起 Load Mode Register Command 写入到 DRAM 的 Mode Register 中。DRAM 完成初始化后，就会按照设定的参数运行。

NOTE:

初始化和参数设定过程不在本文中详细描述，感兴趣的同学可以参考具体 CPU 和 DRAM 芯片的 Datasheet。

参考资料

1. Memory Systems - Cache Dram and Disk
2. High Performance Dram System Design Constraints and Considerations

DRAM Devices Organization

随着系统对内存容量、带宽、性能等方面的需求提高，系统会接入多个 DRAM Devices。而多个 DRAM Devices 不同的组织方式，会带来不同的效果。本小节将对不同的组织方式及其效果进行简单介绍。

Single Channel DRAM Controller 组织方式

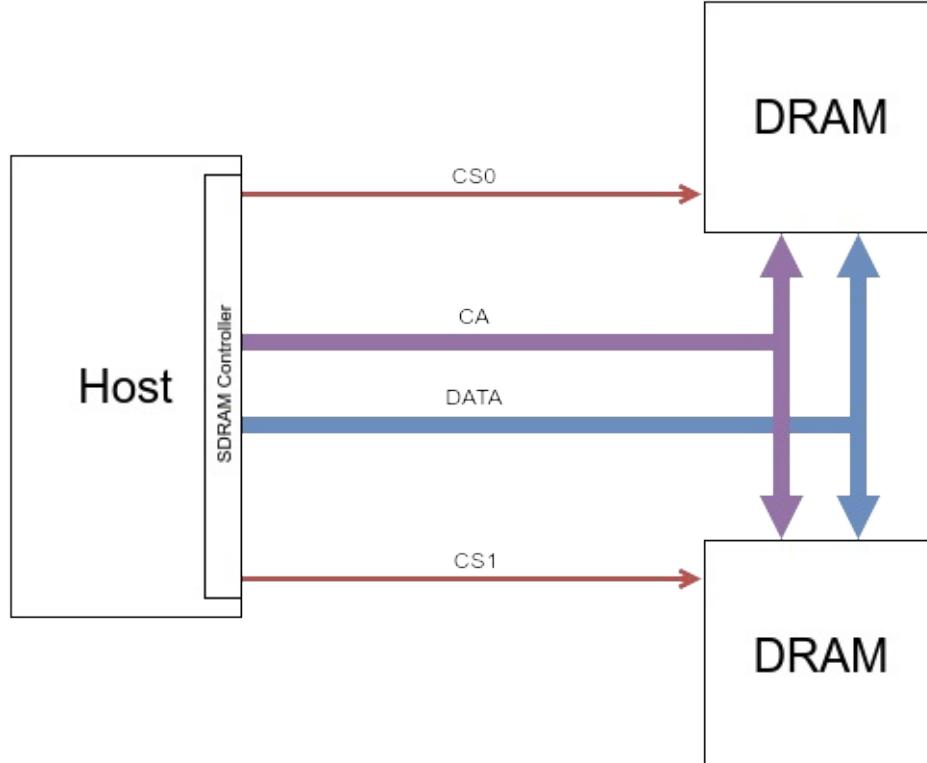
Single Channel 指 DRAM Controller 只有一组控制和数据总线。在这种场景下，DRAM Controller 与单个或者多个 DRAM Devices 的连接方式如下所示：

连接单个 DRAM Device



Single Channel 连接单个 DRAM Device 是最常见的一种组织方式。由于成本、工艺等方面的因素，单个 DRAM Device 在总线宽度、容量上有所限制，在需要大带宽、大容量的产品中，通常接入多个 DRAM Devices。

连接多个 DRAM Devices

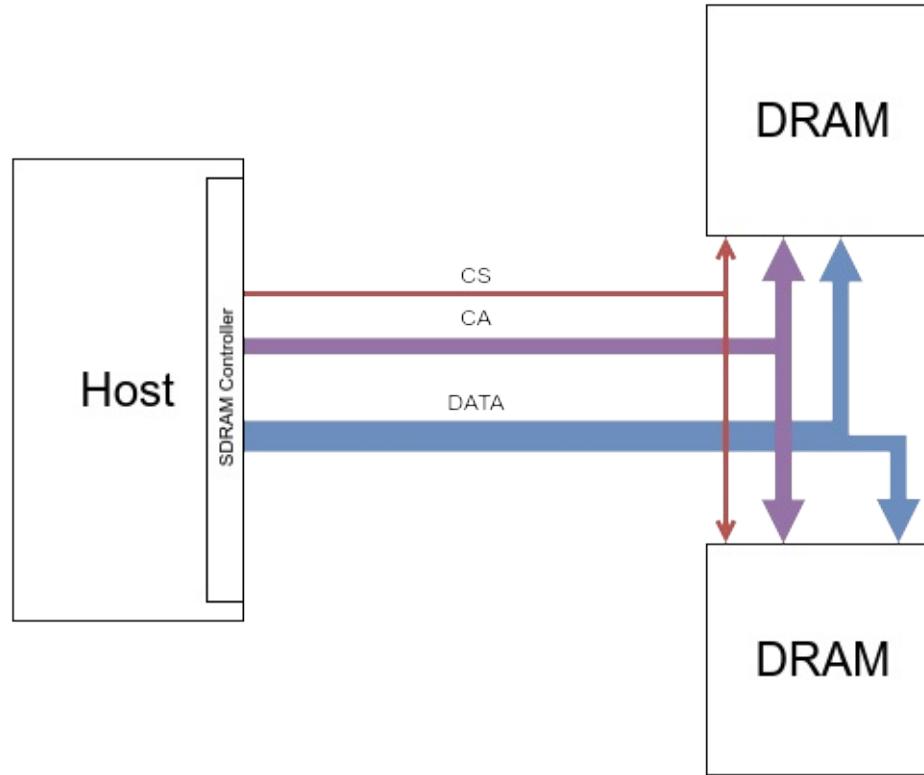


上图中，多个 DRAM Devices 共享控制和数据总线，DRAM Controller 通过 Chip Select 分时单独访问各个 DRAM Devices。此外，在其中一个 Device 进入刷新周期时，DRAM Controller 可以按照一定的调度算法，优先执行其他 Device 上的访问请求，提高系统整体内存访问性能。

NOTE :

CS0 和 CS1 在同一时刻，只有一个可以处于使能状态，即同一时刻，只有一个 Device 可以被访问。

上述的这种组织方式只增加总体容量，不增加带宽。下图中描述的组织方式则可以既增加总体容量，也增加带宽。

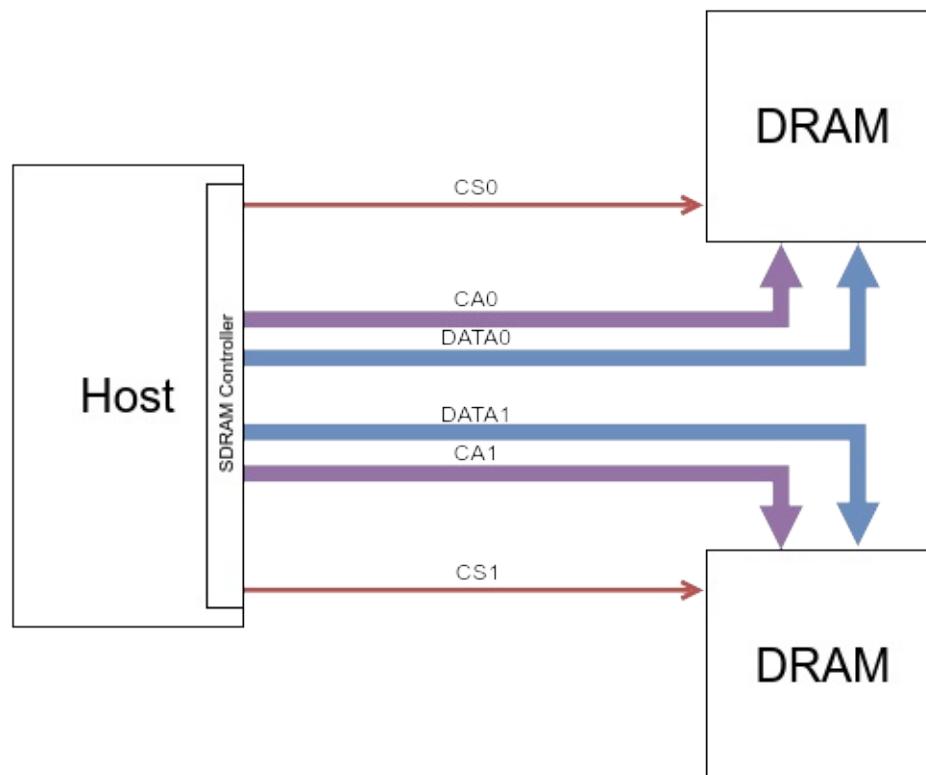


上图中，多个 DRAM Devices 共享控制总线和 Chip Select 信号，DRAM Controller 同时访问每个 DRAM Devices，各个 Devices 的数据合并到一起，例如 Device 1 的数据输出到数据总线的 DATA[0:7] 信号上，Device 2 的数据输出到数据总线的 DATA[8:15] 上。这样的组织方式下，访问 16 bits 的数据就只需要一个访问周期就可以完成，而不需要分解为两个 8 bits 的访问周期。

Multi Channel DRAM Controller 组织方式

Multi Channel 指 DRAM Controller 只有多组控制和数据总线，每一组总线可以独立访问 DRAM Devices。在这种场景下，DRAM Controller 与 DRAM Devices 的连接方式如下所示：

连接 Single Channel DRAM Devices

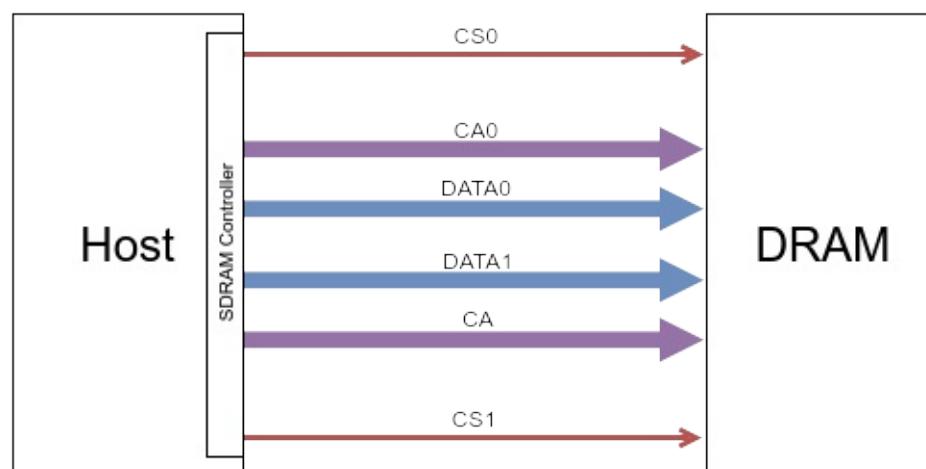


这种组织方式的优势在于多个 Devices 可以同时工作，DRAM Controller 可以对不同 Channel 上的 Devices 同时发起读写请求，提高了读写请求的吞吐率。

NOTE :

CS0 和 CS1 在同一时刻，可以同时处于使能状态，即同一时刻，两个 Devices 可以同时被访问。

连接 Multi Channel DRAM Device



在一些 DRAM 产品中，例如 LPDDR3、LPDDR4 等，引入了 Multi Channel 的设计，即一个 DRAM Devices 中包括多个 Channel。这样就可以在单个 Device 上达成 Multi Channel 同时访问的效果，最终带来读写请求吞吐率的提升。

Introduction of caching snoop protocols

Attaching a cache to each CPU increases performance in many ways. Bringing memory closer to the CPU reduces the average memory access time and at the same time reducing the bandwidth load on the memory bus. The challenge with adding cache to each CPU in a shared memory architecture is that it allows multiple copies of a memory block to exist. This is called the cache-coherency problem. To solve this, caching snoop protocols were invented attempting to create a model that provided the correct data while not trying to eat up all the bandwidth on the bus. The most popular protocol, write invalidate, erases all other copies of data before writing the local cache. Any subsequent read of this data by other processors will detect a cache miss in their local cache and will be serviced from the cache of another CPU containing the most recently modified data. This model saved a lot of bus bandwidth and allowed for Uniform Memory Access systems to emerge in the early 1990s. Modern cache coherency protocols are covered in more detail by part 3.

SMP is an acronym for “Symmetric Multi-Processor”. It describes a design in which two or more identical CPU cores share access to main memory.

The cache behavior becomes relevant to this discussion when each CPU core has its own private cache. In a simple model, the caches have no way to interact with each other directly. The values held by core #1's cache are not shared with or visible to core #2's cache except as loads or stores from main memory. The long latencies on memory accesses would make inter-thread interactions sluggish, so it's useful to define a way for the caches to share data. This sharing is called cache coherency, and the coherency rules are defined by the CPU architecture's cache consistency model.

Observability

Before going further, it's useful to define in a more rigorous fashion what is meant by “observing” a load or store. Suppose core 1 executes “ $A = 1$ ”. The store is initiated when the CPU executes the instruction. At some point later, possibly through cache coherence activity, the store is observed by core 2. In a write-through cache it doesn't really complete until the store arrives in main memory, but the memory consistency model doesn't dictate when something completes, just when it can be observed.

Microarchitecture

<https://en.wikipedia.org/wiki/Microarchitecture> https://en.wikibooks.org/wiki/Microprocessor_Design

Execution units are also essential to microarchitecture. Execution units include arithmetic logic units (ALU), floating point units (FPU), load/store units, branch prediction, and SIMD.

https://en.wikipedia.org/wiki/Out-of-order_execution

The way the instructions are ordered in the original computer code is known as program order, in the processor they are handled in data order, the order in which the data, operands, become available in the processor's registers.

Re-order buffer A re-order buffer (ROB) is used in a Tomasulo algorithm for out-of-order instruction execution. It allows instructions to be committed in-order.

Why multiprocessing

Computer architects have become stymied by the growing mismatch in CPU operating frequencies and DRAM access times. None of the techniques that exploited instruction-level parallelism (ILP) within one program could make up for the long stalls that occurred when data had to be fetched from main memory. Additionally, the large transistor counts and high operating frequencies needed for the more advanced ILP techniques required power dissipation levels that could no longer be cheaply cooled. For these reasons, newer generations of computers have started to exploit higher levels of parallelism that exist outside of a single program or program thread.

Conceptually, multithreading is equivalent to a context switch at the operating system level. The difference is that a multithreaded CPU can do a thread switch in one CPU cycle instead of the hundreds or thousands of CPU cycles a context switch normally requires. This is achieved by replicating the state hardware (such as the register file and program counter) for each active thread.

In processor design, there are two ways to increase on-chip parallelism with fewer resource requirements: one is superscalar technique which tries to exploit instruction level parallelism (ILP); the other is multithreading approach exploiting thread level parallelism (TLP).

Interleaved multithreading Simultaneous multithreading (SMT) Chip-level multiprocessing (CMP or multicore) Any combination of multithreaded/SMT/CMP The key factor to distinguish them is to look at how many instructions the processor can issue in one cycle and how many threads from which the instructions come.

To avoid false operand dependencies, which would decrease the frequency when instructions could be issued out of order, a technique called register renaming is used. In this scheme, there are more physical registers than defined by the architecture. The physical registers are tagged so that multiple versions of the same architectural register can exist at the same time.

The traditional means to improve performance in processor architectures include dividing instructions into substeps so the instructions can be executed partly at the same time (termed pipelining), dispatching individual instructions to be executed independently, in different parts of the processor (superscalar architectures), and even executing instructions in an order different from the program (out-of-order execution). These methods all complicate hardware (larger circuits, higher cost and energy use) because the processor must make all of the decisions internally for these methods to work. In contrast, the VLIW method depends on the programs providing all the decisions regarding which instructions to execute simultaneously and how to resolve conflicts. As a practical matter, this means that the compiler (software used to create the final programs) becomes far more complex, but the hardware is simpler than in many other means of parallelism.

https://en.wikipedia.org/wiki/Very_long_instruction_word