存储器的结构、映像及CMD文件的编写

寄存器的C语言访问

在购买计算机的时候，硬盘空间通常是衡量计算机性能的指标之一，同样的，在选择嵌入式CPU之前，存储器也是必须考虑的指标之一。存储器就像是个仓库一样，堆放着各种程序代码和数据，CPU运行的时候就是在“仓库”里不断的搬入搬出各种代码和数据，作为“仓库”保管员的开发者，弄清楚“仓库”的结构及存放规则是必需的。F28335的内部具有总共34K×16位的SRAM和256K×16位的FLASH，本章将详细介绍F28335存储器的结构，映像，并讲解如何编写“仓库”的存放规则——CMD文件。

F28335的存储器·结构

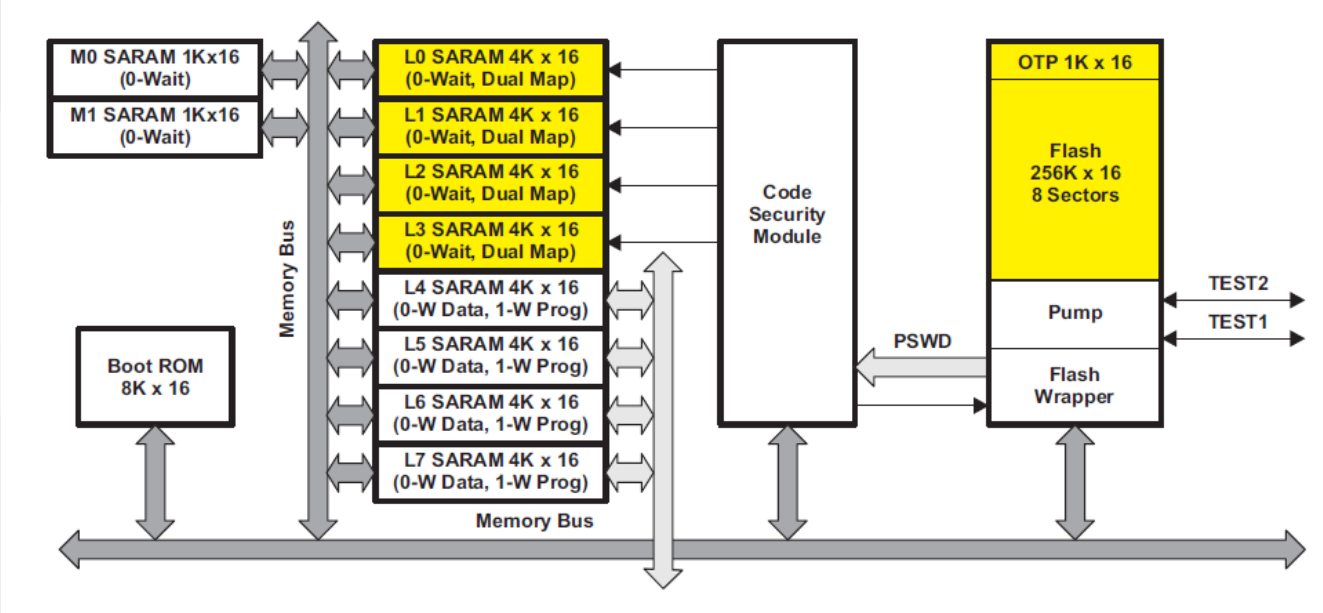

图4-1 F28335的存储器结构

TMS320F28335的CPU本身不含存储器，但它可以访问DSP片内其他地方的存储器或者片外扩展的存储器。F28335的存储器结构如图4-1所示。

F28335的存储器·结构

存储器名称	存储器容量
FLASH	256K*16位
M0(SRAM)	1K*16位
M1(SRAM)	1K*16位
L0(SRAM)	4K*16位
L1(SRAM)	4K*16位
L2(SRAM)	4K*16位
L3(SRAM)	4K*16位
L4(SRAM)	4K*16位
L5(SRAM)	4K*16位
L6(SRAM)	4K*16位
L7(SRAM)	4K*16位
Boot ROM	8K*16位
OTP(One Time Programmable ROM)	2K*16位

图4-1 F28335的存储器结构

F28335的存储器·映像

F28335具有32位的数据地址和22位的程序地址，总地址空间可以达到4M的数据空间和4M的程序空间。读到这句话的时候，不知道会不会产生这样的疑问，一个是32位的数据地址，一个是只有22位的程序地址，那么为什么其可寻址的空间却是一样大的呢？不妨来算一下，32位的数据地址，就是能访问2的32次，是4G，而22位的程序地址，就是能访问2的22次，是4M。也就是说，可寻址的数据空间应该是4G而不是4M，难道TI给出的文档有问题吗？其实，F28335可寻址的数据空间最大确实是4G，但是实际线性地址能达到的只有4M，原因是F28335的存储器分配采用的是分页机制，分页机制采用的是形如0xXXXXXXX的线性地址，所以数据空间能寻址的只有4M，不过也足够使用了。

F28335的存储器就像一个仓库，用来存放很多的货物，只不过存储器是用来存放指令和数据的。从表4-1可以看到，F28335内部有很多不同的存储器块，如何有效的去管理这些存储器块，如何高效的利用存储器空间，对于系统而言是非常重要的问题。

F28335的存储器·映像

先用一个通俗的例子来进行讲解。如图4-2所示，假设有一个物流公司，它有储藏货物的仓库若干个，每天来来往往有成千上万的货物要发送到全国各地，如果拿回来的货物乱七八糟的堆放的话，发货的时候麻烦就大了，发货人员不仅仅要一个仓库一个仓库去找，而且要一个货架一个货架的翻，这样效率肯定是极其低下的，匆忙之下也有可能将货物搞错。为了提高效率，老板肯定要想办法进行改进，首先把各个仓库分类，例如仓库1是发往江苏和上海的货物，仓库2是发往北京的货物，仓库3是发往深圳的货物，仓库4是发往西安的。其次，货物进来前要根据目的地贴上统一规格的标签，例如HD1000-HD2009的货物放在仓库1内。这样，发货的时候，只要根据标签就能方便的分辨出货物在哪个仓库的哪个货架，应该装上发往哪个地区的货车，一切井然有序。

F28335的存储器·映像

类似的，各个存储空间就像物流公司的仓库一样，有的是存放程序代码的，有的是用来存放数据的。F28335对各个存储单元进行了统一的编址，确定了各个存储单元在存储空间中的绝对位置，在放置代码或者数据的时候，根据它们的类型进行分配，决定究竟放在哪个区域，并记录下了它们的地址，这样需要用到的时候只要根据这些地址就能很方便的找到所需要的内容，而记录下如何分配存储空间内容的就是CMD文件。CMD文件的内容和如何编写的将在稍后会做详细的讲解。

F28335的存储器·映像

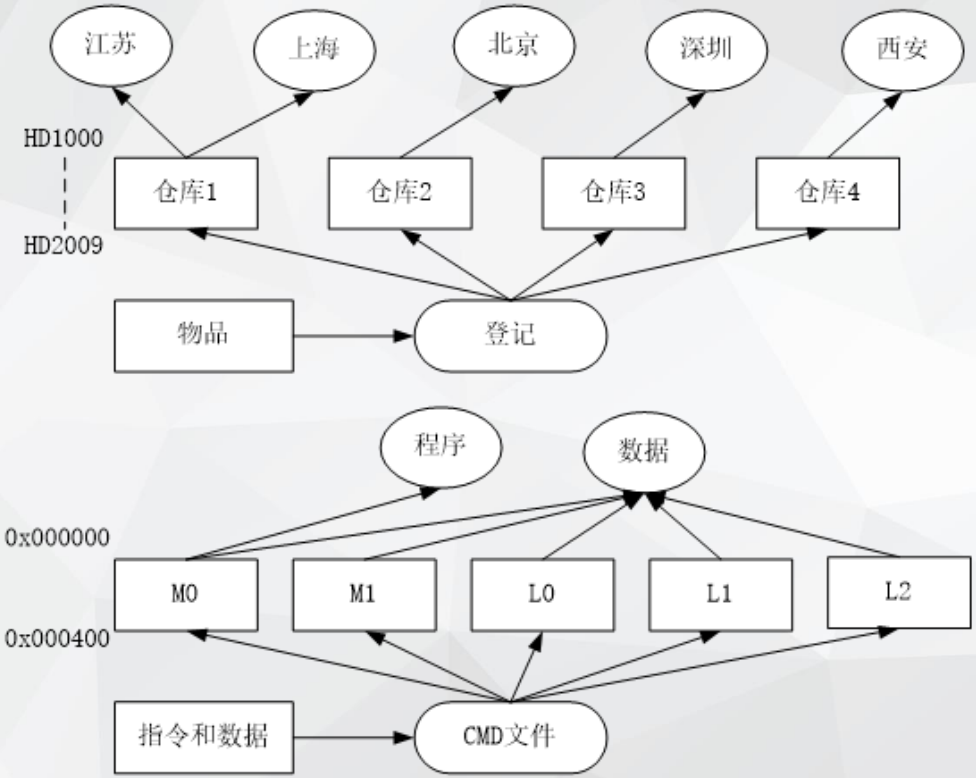

图4-2 映像和统一编址的理解

下面来解释一下什么是映像。“映像”用英文单词来表示是“Map”，“Map”在中文里又是“地图”的意思。在地图上，建筑物都有自己详细的地址，根据地图的指引，按照地址，就能找到相应的地方。类似的，当存储器单元的地址在设计时都确定下来后，就形成了存储器的“地图”，也就是存储器映像，根据存储单元的地址，就能找到相应的存储单元。

F28335的存储器·映像

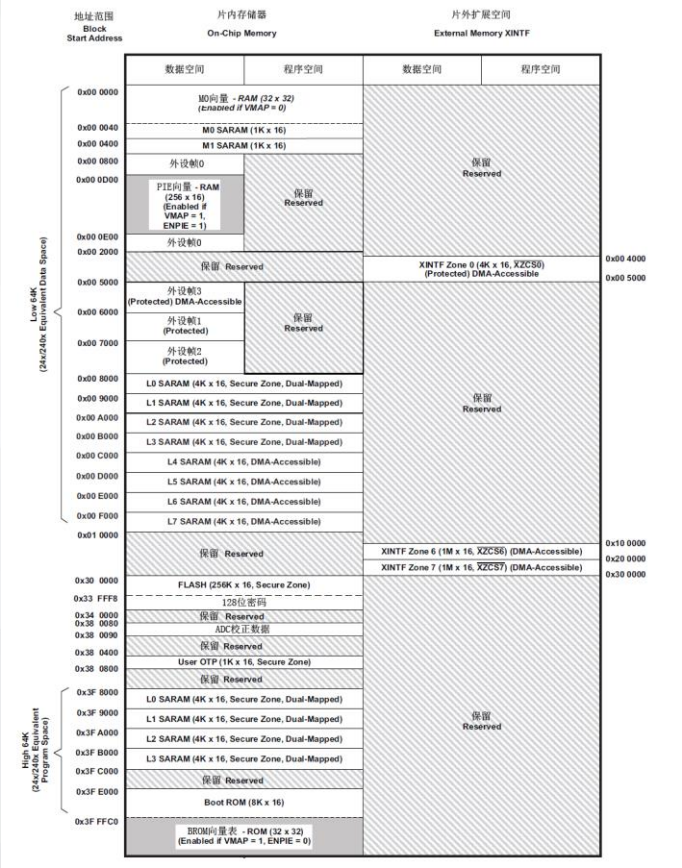

图4-3 TMS320F28335的存储器映像

（此处可右键选择“放大”功能查看图像）

F28335的存储器·映像

地址范围	存储器块名称
0x00 0000~0x00 003F	M0向量RAM（VMAP=0）
0x00 0040~0x00 03FF	M0 SARAM（1K×16）
0x00 0400~0x00 07FF	M1 SARAM（1K×16）
0x00 0800~0x00 0CFF	外设帧0（2K×16）
0x00 0D00~0x00 0DFF	PIE向量（VMAP=1，ENPIE=1，256×16）
0x00 0E00~0x00 1FFF	外设帧0
0x00 2000~0x00 3FFF	保留区域
0x00 4000~0x00 4FFF	外扩的XINTF Zone0（4K×16，受EALLOW保护）
0x00 5000~0x00 5FFF	外设帧3（4K×16，受EALLOW保护）
0x00 6000~0x00 6FFF	外设帧1（4K×16，受EALLOW保护）
0x00 7000~0x00 7FFF	外设帧2（4K×16，受EALLOW保护）
0x00 8000~0x00 8FFF	L0 SARAM（4K×16，受密码保护，双映射）
0x00 9000~0x00 9FFF	L1 SARAM（4K×16，受密码保护，双映射）
0x00 A000~0x00 AFFF	L2 SARAM（4K×16，受密码保护，双映射）
0x00 B000~0x00 BFFF	L3 SARAM（4K×16，受密码保护，双映射）
0x00 C000~0x00 CFFF	L4 SARAM（4K×16）
0x00 D000~0x00 DFFF	L5 SARAM（4K×16）
0x00 E000~0x00 EFFF	L6 SARAM（4K×16）
0x00 F000~0x00 FFFF	L7 SARAM（4K×16）
0x01 0000~0x0F FFFF	保留区域
0x10 0000~0x1F FFFF	外扩的XINTF Zone6（1M×16）
0x20 0000~0x2F FFFF	外扩的XINTF Zone7（1M×16）
0x30 0000~0x33 FFF7	FLASH（256K×16）
0x33 FFF8~0x33 FFFF	128位密码
0x34 0000~0x37 FFFF	保留区域
0x38 0000~0x38 03FF	TI OTP（1K×16）
0x38 0400~0x38 07FF	User OTP（1K×16）
0x38 0800~0x3F 7FFF	保留区域
0x3F 8000~0x3F 8FFF	L0 SARAM（4K×16，受密码保护，双映射）
0x3F 9000~0x3F 9FFF	L1 SARAM（4K×16，受密码保护，双映射）
0x3F A000~0x3F AFFF	L2 SARAM（4K×16，受密码保护，双映射）
0x3F B000~0x3F BFFF	L3 SARAM（4K×16，受密码保护，双映射）
0x3F C000~0x3F DFFF	保留空间
0x3F E000~0x3F FFBF	Boot ROM（8K*16）
0x3F FFC0~0x3F FFFF	BROM向量（VMAP=1，ENPIE=0）

对于图4-3所示的TMS320F28335的存储器映像，下面有几点需要特别注意的：

1.保留区是为将来的扩展而保留的，在实际应用时不应该去访问这些区域。

2.外设帧0、外设帧1、外设帧2和外设帧3的存储器只能映射到数据空间，用户程序不能在程序空间访问这些存储器。

3.图中标注Protected的存储器，即外设帧1、外设帧2、外设帧3和XINTF Zone0，受到EALLOW保护，以避免配置后的随意改写。

4.存储器L0、L1、L2、L3、FLASH以及User OTP均受密码CSM保护。

表4-2 F28335各个存储器块的地址范围

（此处可右键选择“放大”功能查看图像）

F28335的存储器·映像

地址范围	存储器块名称
0x00 0000~0x00 003F	M0向量RAM (VMAP=0)
0x00 0040~0x00 03FF	M0 SARAM (1K×16)
0x00 0400~0x00 07FF	M1 SARAM (1K×16)
0x00 0800~0x00 0CFF	外设帧0 (2K×16)
0x00 0D00~0x00 0DFF	PIE向量 (VMAP=1 , ENPIE=1 , 256×16)
0x00 0E00~0x00 1FFF	外设帧0
0x00 2000~0x00 3FFF	保留区域
0x00 4000~0x00 4FFF	外扩的XINTF Zone0 (4K×16 , 受EALLOW保护)
0x00 5000~0x00 5FFF	外设帧3 (4K×16 , 受EALLOW保护)
0x00 6000~0x00 6FFF	外设帧1 (4K×16 , 受EALLOW保护)
0x00 7000~0x00 7FFF	外设帧2 (4K×16 , 受EALLOW保护)
0x00 8000~0x00 8FFF	L0 SARAM (4K×16 , 受密码保护 , 双映射)
0x00 9000~0x00 9FFF	L1 SARAM (4K×16 , 受密码保护 , 双映射)
0x00 A000~0x00 AFFF	L2 SARAM (4K×16 , 受密码保护 , 双映射)
0x00 B000~0x00 BFFF	L3 SARAM (4K×16 , 受密码保护 , 双映射)
0x00 C000~0x00 CFFF	L4 SARAM (4K×16)
0x00 D000~0x00 DFFF	L5 SARAM (4K×16)
0x00 E000~0x00 EFFF	L6 SARAM (4K×16)
0x00 F000~0x00 FFFF	L7 SARAM (4K×16)
0x01 0000~0x0F FFFF	保留区域
0x10 0000~0x1F FFFF	外扩的XINTF Zone6 (1M×16)
0x20 0000~0x2F FFFF	外扩的XINTF Zone7 (1M×16)
0x30 0000~0x33 FFF7	FLASH (256K×16)
0x33 FFF8~0x33 FFFF	128位密码
0x34 0000~0x37 FFFF	保留区域
0x38 0000~0x38 03FF	TI OTP (1K×16)
0x38 0400~0x38 07FF	User OTP (1K×16)
0x38 0800~0x3F 7FFF	保留区域
0x3F 8000~0x3F 8FFF	L0 SARAM (4K×16 , 受密码保护 , 双映射)
0x3F 9000~0x3F 9FFF	L1 SARAM (4K×16 , 受密码保护 , 双映射)
0x3F A000~0x3F AFFF	L2 SARAM (4K×16 , 受密码保护 , 双映射)
0x3F B000~0x3F BFFF	L3 SARAM (4K×16 , 受密码保护 , 双映射)
0x3F C000~0x3F DFFF	保留空间
0x3F E000~0x3F FFBF	Boot ROM (8K*16)
0x3F FFC0~0x3F FFFF	BROM向量 (VMAP=1 , ENPIE=0)

5.TI OTP ROM是只读的，而且包含ADC校验程序，它不是可编程的。

6.在同一个时刻，M0向量、PIE向量、BROM向量中只能有一种向量被使能。

存储器的映像就是存储单元的“地图”，规定了各个存储单元在存储空间中的绝对地址。F28335对数据空间和程序空间进行了统一编址，有些地址空间既可以做数据空间用也可以做程序空间用，而有的地址空间只能做数据空间用，不能当程序空间用，具体的可仔细察看图4-3。

表4-2 F28335各个存储器块的地址范围

（此处可右键选择“放大”功能查看图像）

F28335的存储器·特点

前面介绍了F28335是由哪些存储器模块组成的，并了解了这些存储器模块在F28335存储空间中的地址分布情况，但对各个存储器模块具体的特点并不了解，接下来就详细介绍这些存储器模块，看看其各自有哪些特点。

F28335的存储器·特点

1.片内SARAM

SARAM是Single Access RAM的缩写，即为单口随机读/写存储器，后面就简称片内RAM。单口RAM是相对于双口RAM而言的，双口RAM是在一个RAM存储器上具有两套完全独立的数据线、地址线和读写控制线，并允许两个独立的系统同时对该存储器进行随机访问。片内RAM总共有34K*16位大小，由M0、M1、L0~L7十个存储块组成，每个存储块各自的大小见表4-2。这些存储器块都可以被单独的访问，并且均可以作为程序空间或者数据空间，用来存放指令代码或者存储数据。值得注意的是，L0、L1、L2和L3里面的内容受到CSM的保护，即需要密码才能从JTAG口读取的，其余存储器块都不受密码保护。

F28335的存储器·特点

2.片内OTP

片内OTP实质是ROM空间。OTP是One Time Programmable的缩写，即一次性可编程的ROM，其大小为2K*16位，其中1K*16位由TI公司保留作为系统测试使用，剩余1K*16位用户可以使用，这部分空间也均可以作为程序空间或者数据空间。OTP里面的内容受到CSM的保护。

F28335的存储器·特点

3.Boot ROM

Boot ROM，可以叫做引导ROM。该存储空间内有TI公司产品的版本号、发布的数据、校验求和信息、复位矢量、CPU矢量(仅为测试)及数学表等内容。Boot ROM的主要作用是实现DSP程序的引导功能（Bootloader）。芯片出厂时，在Boot ROM的0x3FFC00~0x3FFFBF存储器内装有厂家的引导装载程序，当，DSP被置位微计算机模式时，CPU在复位后将执行这段程序，从而完成Bootloader功能。

F28335的存储器·特点

4.片内FLASH

F28335具有256K×16的片内FLASH，这部分空间也是均可以作为程序空间或者数据空间的，其内容也是受到CSM的保护的。FLASH存储器由8个32K×16位的扇区组成，用户可以单独对其中任何一个扇区进行擦除、编程和校验，而其他扇区不变。但是，不能在其中一个扇区上执行程序来擦除和编程其他的扇区。具体的区段划分如表4-3所示。

地址范围	区段名称
0x30 0000~0x30 7FFF	段H（32K×16）
0x30 8000~0x30 FFFF	段G（32K×16）
0x31 0000~0x31 7FFF	段F（32K×16）
0x31 8000~0x31 FFFF	段E（32K×16）
0x32 0000~0x32 7FFF	段D（32K×16）
0x32 8000~0x32 FFFF	段C（32K×16）
0x33 0000~0x33 7FFF	段B（32K×16）
0x33 8000~0x33 FFFF	段A（32K×16）
0x33 FF80~0x33 FFF5	当采用密码保护时，编程为0x0000
0x33 FFF6~0x33 FFF7	FLASH启动入口地址（这里有程序分支指令）
0x33 FFF8~0x33 FFFF	128位密码

表4-3 F28335片内FLASH区段的划分

5.代码安全模块CSM

CSM是Code Security Module的缩写，即代码安全模块。在开发完程序，将代码烧写进芯片的存储器后，常常会担心别人通过JTAG口从存储器中将代码读出来，为了保护代码安全，F28335设计有代码安全模块CSM，其地址为0x33 FFF8~0x33 FFFF，共128位。受到CSM保护的模块有FLASH、OTP、L0、L1、L2和L3。密码保护的概念应该很好理解，FLASH、OTP、L0、L1、L2、L3这些模块就像是一个保险箱，把代码装载入存储单元之后，就给保险箱设一个密码，当需要再取这些存储单元中的内容时，需要凭密码来打开，只有当输入的密码和之前设置的密码相同时，才能打开保险箱，否则，则无法打开保险箱，即无法读取存储单元中的内容。

F28335的存储器·特点

图4-3是CCS6中FLASH烧写设置界面，其可以在工程属性中找到（右击工程，在所弹出的菜单中选择Properties选项）。从图4-3可以看到正如前面所介绍的，能对各个FLASH的段进行单独擦除。这里，主要看Code Security Password区域，CSM模块由8个16位的单元组成，默认的各位全是1，当128位全为1的时候，说明器件此时是不安全的，并未受密码保护。在烧写FLASH程序时，在此处可以设置好密码。值得注意的是，不能使用全0作为一个密码或者在FLASH存储器上执行一个清0程序后再复位该芯片，否则，该芯片会被锁死，不能调试或再编程。

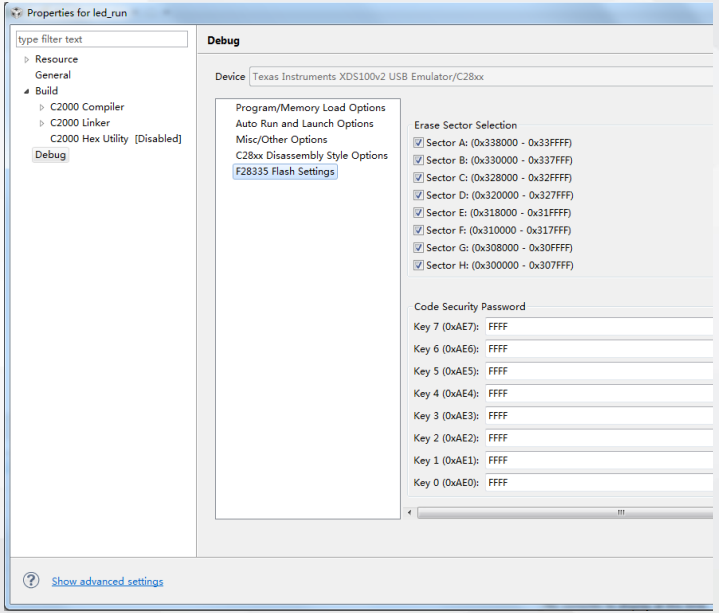

图4-3 F28335 Flash设置界面

F28335的存储器·特点

6.外设帧PF

F28335片内具有4个外设帧0、外设帧1、外设帧2和外设帧3，专门用于外设寄存器的映像空间。除了CPU寄存器之外，其他寄存器均放在了这些外设帧内，具体的分布情况如表4-4、4-5、4-6和4-7所示。

名称	地址范围	大小(×16)	访问类型
器件仿真寄存器	0x00 0880~0x00 09FF	384	EALLOW保护
FLASH寄存器	0x00 0A80~0x00 0ADF	96	EALLOW保护
CSM模块寄存器	0x00 0AE0~0x00 0AEF	16	EALLOW保护
ADC寄存器	0x00 0B00~0x00 0B0F	16	不受EALLOW保护
XINTF寄存器	0x00 0B20~0x00 0B3F	32	EALLOW保护
CPU定时器0/1/2寄存器	0x00 0C00~0x00 0C3F	64	不受EALLOW保护
PIE寄存器	0x00 0CE0~0x00 0CFF	32	不受EALLOW保护
PIE向量表	0x00 0D00~0x00 0DFF	256	EALLOW保护
DMA寄存器	0x00 1000~0x00 11FF	512	EALLOW保护

表4-4 外设帧0各寄存器的映像分布情况

F28335的存储器·特点

6.外设帧PF

F28335片内具有4个外设帧0、外设帧1、外设帧2和外设帧3，专门用于外设寄存器的映像空间。除了CPU寄存器之外，其他寄存器均放在了这些外设帧内，具体的分布情况如表4-4、4-5、4-6和4-7所示。

表4-5 外设帧1各寄存器的映像分布情况

名称	地址范围	大小(×16)
eCAN-A寄存器	0x00 6000~0x00 61FF	512
eCAN-B寄存器	0x00 6200~0x00 63FF	512
ePWM1+HRPWM1寄存器	0x00 6800~0x00 683F	64
ePWM2+HRPWM2寄存器	0x00 6840~0x00 687F	64
ePWM3+HRPWM3寄存器	0x00 6880~0x00 68BF	64
ePWM4+HRPWM4寄存器	0x00 68C0~0x00 68FF	64
ePWM5+HRPWM5寄存器	0x00 6900~0x00 693F	64
ePWM6+HRPWM6寄存器	0x00 6940~0x00 697F	64
eCAP1寄存器	0x00 6A00~0x00 6A1F	32
eCAP2寄存器	0x00 6A20~0x00 6A3F	32
eCAP3寄存器	0x00 6A40~0x00 6A5F	32
eCAP4寄存器	0x00 6A60~0x00 6A7F	32
eCAP5寄存器	0x00 6A80~0x00 6A9F	32
eCAP6寄存器	0x00 6AA0~0x00 6ABF	32
eQEP1寄存器	0x00 6B00~0x00 6B3F	64
eQEP2寄存器	0x00 6B40~0x00 6B7F	64
GPIO寄存器	0x00 6F80~0x00 6FFF	128

F28335的存储器·特点

6.外设帧PF

F28335片内具有4个外设帧0、外设帧1、外设帧2和外设帧3，专门用于外设寄存器的映像空间。除了CPU寄存器之外，其他寄存器均放在了这些外设帧内，具体的分布情况如表4-4、4-5、4-6和4-7所示。

名称	地址范围	大小(×16)
系统控制寄存器	0x00 7010~0x00 702F	32
SPI-A寄存器	0x00 7040~0x00 704F	16
SCI-A寄存器	0x00 7050~0x00 705F	16
外部中断寄存器	0x00 7070~0x00 707F	16
ADC寄存器	0x00 7100~0x00 711F	32
SCI-B寄存器	0x00 7750~0x00 775F	16
SCI-C寄存器	0x00 7770~0x00 777F	16
I2C-A寄存器	0x00 7900~0x00 793F	64

表4-6 外设帧2各寄存器的映像分布情况

6.外设帧PF

F28335片内具有4个外设帧0、外设帧1、外设帧2和外设帧3，专门用于外设寄存器的映像空间。除了CPU寄存器之外，其他寄存器均放在了这些外设帧内，具体的分布情况如表4-4、4-5、4-6和4-7所示。

名称	地址范围	大小(×16)
McBSP-A寄存器（DMA）	0x5000~0x503F	64
McBSP-B寄存器（DMA）	0x5040~0x507F	64
ePWM1+HRPWM1（DMA）	0x5800~0x583F	64
ePWM2+HRPWM2（DMA）	0x5840~0x587F	64
ePWM3+HRPWM3（DMA）	0x5880~0x58BF	64
ePWM4+HRPWM4（DMA）	0x58C0~0x58FF	64
ePWM5+HRPWM5（DMA）	0x5900~0x593F	64
ePWM6+HRPWM6（DMA）	0x5940~0x597F	64

表4-6 外设帧2各寄存器的映像分布情况

F28335的存储器·特点

从上述表格可以看到：

1.外设帧0中有的寄存器受EALLOW指令的保护，而有的却不受EALLOW指令的保护，外设帧1、外设帧2和外设帧3的寄存器都受EALLOW指令的保护，这也是为什么在写外设寄存器的相关程序时，有的在操作前需要加指令EALLOW，操作结束后使用指令EDIS，而有的寄存器却不需要，原因就在这里。使用EALLOW指令的保护可以防止一些偶然的代码或指针去破坏寄存器的内容。

2.外设帧0中的FLASH寄存器既受EALLOW保护，同时也受CSM模块的保护。

3.当使用DMA时，ePWM和HRPWM模块可以被映射到外设帧3，此时MAPCNF寄存器的第0位MAPEPWM必须被设为1。倘若MAPEPWM的值为0，ePWM和HRPWM模块被映射到外设帧1。

CMD文件

连接命令文件（Linker Command Files），以后缀.cmd结尾，简称为CMD文件。前面介绍过，CMD文件的作用就像仓库的货物摆放记录一样，为程序代码和数据分配存储空间。初学者往往会觉得CMD文件比较难懂，打开CMD文件研究时也是一头雾水，接下来，将从C语言语法的角度出发，由浅入深的揭秘CMD文件，也顺带简单介绍一下F28335所采用的通用目标文件格式COFF和段的概念。

CMD文件·COFF格式和段的概念

通用目标文件格式COFF（Common Object File Format），是一种很流行的二进制可执行文件格式。二进制可执行文件包括了库文件（以后缀.lib结尾），目标文件（以后缀.obj结尾），最终的可执行文件（以后缀.out结尾）等，平时烧写程序时使用的就是.out结尾的文件。

详细的COFF文件格式包括有段头、可执行代码、初始化数据、可重定位信息、行号入口、符号表、字符串表等等，当然这些属于编写操作系统和编译器人员关心的范畴。从应用的角度来讲，大家只需掌握两点就可以了，一是通过伪指令定义段（Section），二是给段分配空间，至于二进制文件到底如何组织分配，则交由编译器来完成。

CMD文件·COFF格式和段的概念

使用段的好处是鼓励模块化编程，提供更强大而又灵活的方法来管理代码和目标系统的存储空间。这里模块化编程的意思是指程序员可以自由决定愿意把哪些代码归属到哪些段，然后加以不同的处理。比如，把已经初始化的数据放到一个段里，未初始化的数据放到另一个段里，而不是混杂的放在一起。

编译器处理段的过程为：

1.把每个源文件都编译成独立的目标文件（以后缀.obj结尾），每个目标文件都含有自己的段。

2.连接器把这些目标文件中相同段名的部分连接在一起，生成最终的可执行文件（以后缀.out结尾）。

这里，正好可以重新提一下CCS软件中编写完程序需要编译时，使用Compile file和Build操作的区别。Compile file操作只是执行了上述过程的第1步，而Build操作执行了上述完整的第1步和第2步。

CMD文件·C语言生成的段

C语言生成的段可以分为两大类：已初始化的段和未初始化的段。已初始化的段含有真实的指令和数据，存放在程序存储空间。未初始化的段只是保留变量的地址空间，在DSP上电调用_c_int0初始化库前，未初始化的段并没有真实的内容。未初始化的段存放在数据存储空间。

CMD文件·C语言生成的段

1.已初始化的段

(1).text：编译C语言中的语句时，生成的汇编指令代码存放于此。

(2).cinit：存放用来对全局和静态变量初始化的常数。

(3).const：包含字符串常量和初始化的全局变量和静态变量（由const声明）的初始化和说明。

(4) .econst：包含字符串常量和初始化的全局变量和静态变量（由far const声明）的初始化和说明。

(5).pinit：全局构造器（C++）程序列表。

(6).switch：存放switch语句产生的常数表格。

CMD文件·C语言生成的段

这里需要详细说明的是，在C语言中，有以下3种情况会产生.const段：

(1)关键字const 由关键字const声明的全局变量的初始化值，例如“const int a=18;”。但是由const声明的局部变量的初始化值，不会产生.const段，局部变量都是运行时开辟在.bss段中的。

(2)字符串常数 字符串常数出现在表达式中，例如，“strcpy(s,”abc”)；”。字符串常数用来初始化指针变量，例如“char *p=”abc”；”。但是，当字符串常数用来初始化数组变量时，不论是全局变量还是局部变量，都不会产生.const段，此时字符串常数生成的是.cinit段。

(3)数组和结构体的初始值 数组和结构体是局部变量时，其初始化值会产生.const段，但当数组和结构体是全局变量时，其初始化值不会产生.const段，此时生成的是.cinit段。

CMD文件·C语言生成的段

2.未初始化的段

(1).bss：为全局变量和局部变量保留的空间，在程序上电时，.cinit空间中的数据复制出来并存储在.bss空间中。

(2).ebss：为使用大寄存器模式时的全局变量和静态变量预留的空间，在程序上电时，.cinit空间中的数据复制出来并存储在.ebss中。

(3).stack：为系统堆栈保留的空间，主要用于和函数传递变量或为局部变量分配空间。

(4).system：为动态存储分配保留的空间。如果有宏函数，此空间被宏函数占用，如果没有的话，此空间保留为0。

(5).esysmem:为动态存储分配保留的空间。如果有far函数，此空间被相应的占用，如果没有的化，此空间保留为0。

CMD文件·C语言生成的段

上面介绍的段都是C语言预先已经定义好的段，那作为开发人员是否可以自己定义段呢？答案肯定是可以的，在上一章中介绍寄存器文件的空间分配时，就讲到了使用“#pragma DATA_SECTION”命令来定义数据段，下面做更为详细的介绍。

CMD文件·C语言生成的段

#pragma是标准C语言中保留的预处理命令，在F28335中，大家可以通过#pragma来定义自己的段，这是预处理命令#pragma的主要用法。#pragma的语法格式如下：

```
#pragma CODE_SECTION(symbol,”section name”) ;
#pragma DATA_SECTION(symbol,”section name”) ;
```

需要说明的是：

(1)symbol是符号，可以是函数名也可以是全局变量名。Section name是用户自己定义的段名。

(2)CODE_SECTION用来定义代码段，而DATA_SECTION用来定义数据段。

(3)不能在函数体内声明#pragma。

(4)必须在符号被定义和使用前使用#pragma。

CMD文件·C语言生成的段

【例4-1】将全局数组变量s[100]单独编译成一个新的段，取名为”newsect”。

```
#pragma DATA_SECTION(s,”newsect”) ;
unsigned int s[100] ;
void main(void)
{
    ……
}
```

在实际应用时，如果没有用到某些段，例如很多人可能不会用到.system段，则可以不用在CMD文件中为其分配存储空间，当然保险起见，也可以无论用到与否，均为其分配存储空间。表4-8是前面所介绍的这些段的存储特性，也就是这些段应当放在什么样的存储器里，应当分配到程序空间还是数据空间。由于F28335的存储空间采用的是分页制，在CMD文件中，PAGE0代表程序空间，PAGE1代表数据空间。

CMD文件·C语言生成的段

段	存储器类型	分配的存储空间
.text	ROM OR RAM (FLASH)	PAGE0
.cinit	ROM OR RAM (FLASH)	PAGE0
.const	ROM OR RAM (FLASH)	PAGE1
.econst	ROM OR RAM (FLASH)	PAGE1
.pinit	ROM OR RAM (FLASH)	PAGE0
.switch	ROM OR RAM (FLASH)	PAGE0/PAGE1
.bss	RAM	PAGE1
.ebss	RAM	PAGE1
.stack	RAM	PAGE1
.system	RAM	PAGE1
.esystem	RAM	PAGE1
通过#pragma CODE_SECTION定义的段	ROM OR RAM (FLASH)	PAGE0
通过#pragma DATA_SECTION定义的段	RAM	PAGE1

表4-8 段的存储特性

CMD文件·CMD文件的编写

CMD文件支持C语言中的块注释符“/*”和“*/”，但不支持行注释符“//”。CMD文件会使用到为数不多的几个关键字，下面会根据需要来介绍一些常用的关键字。值得注意的是，虽然某些关键字既能大写也能小写，例如run，也可以写成RUN，fill也可以写成FILL，但有些关键字是必须区分大小写的，比如MEMORY、SECTIONS只能大写。

CMD文件的两大主要功能是指示存储空间和分配段到存储空间，CMD文件其实也就是由这两部分内容构成的，下面分别进行介绍。

1.通过MEMORY伪指令来指示存储空间

MEMORY伪指令语法如下：

```
MEMORY
{
    PAGE0 : name0[(attr)]:origin=constant , length=constant
  PAGEn : namen[(attr)]:origin=constant , length=constant
}
```

其中：

PAGE 用来标识存储空间的关键字。PAGEn的最大值为PAGE255。F28335用的是PAGE0、PAGE1，其中PAGE0为程序空间，PAGE1为数据空间。

name 代表某一属性或地址范围的存储空间名称。名称可以是1~8个字符，在同一个页内名称不能相同，不同页内名称能相同。

attr 用来规定存储空间的属性。共有4个属性，分别用4个字母来表示。只读R，只写W，该空间可包含可执行代码X，该空间可以被初始化I。实际使用时，为了简化起见，通常会忽略此选项，表示存储空间具有所有的属性。

origin 用来定义存储空间的起始地址。

length 用来定义存储空间的长度。

2.通过SECTIONS伪指令来分配到存储空间

SECTIONS伪指令语法如下：

```
SECTIONS
{
    name:[property , property , property , …]
   name:[property , property , property , …]
   ……
}
```

其中：name为输出段的名称；property 输出段的属性，常用的属性如下：

CMD文件·CMD文件的编写

① load：定义输出段将被装载到哪里的关键字，其语法如下：

```
load=allocation 或者allocation或者 >allocation
```

allocation可以是绝对地址，比如“load=0x000400”，当然，更多的时候，allocation是存储空间的名称，这也是最为通常的用法。

② run：定义输出段从哪里开始运行的关键字，其语法如下：

```
run=allocation或者run>allocation
```

CMD文件中规定，当只出现一个关键字load或者run时，表示load地址和run地址是重叠的。实际应用中，大部分的load地址和run地址都是重叠的，除了.const段。

③ 输入段。其语法如下：

```
{input_sections}
```

花括号“{}”中是输入段名。这里对输入段和输出段做一个区分，每一个C语言文件经过编译都会生成若干个段，多个汇编或C语言文件生成的段大都是同名的，常见的如前面已经介绍的段.cinit，.bss等等，这些都属于输入段。这些归属于不同文件的输入段，在CMD文件的指示下，会被连接器连接在一起生成输出段。

④PAGE：定义段分配到存储空间的类型。其语法如下：

```
PAGE=0 或PAGE=1
```

当PAGE=0，说明段分配到程序空间，而当PAGE=1，说明段分配到数据空间。

3.实际工程中的CMD文件

CMD文件的语法就是上面介绍的这些了，下面来看看在F28335的工程中，CMD文件是不是和上面介绍的一致。打开共享资料中任意一个完整的工程，首先需要来看一下DSP2833x_GlobalVariableDefs.c文件中的内容。

CMD文件·CMD文件的编写

DSP2833x_GlobalVariableDefs.c文件中，使用“#pragma DATA_SECTION”自定义了很多段，这些段都是F28335外设寄存器的结构体文件编译后生成的。这些自定义的段和系统预定义的段，例如.text，.cinit，.bss等一起在CMD文件里进行存储空间的分配，只是寄存器的段文件分配的地址是固定的，譬如段AdcRegsFile是外设ADC寄存器编译后产生的段文件，由于ADC寄存器的起始地址在0x000B00，长度为16，因此段AdcRegsFile必须分配到这个空间上去。共享文件中有两个F28335常用的CMD文件，分别是：DSP2833x_Headers_nonBIOS.cmd、F28335_RAM_lnk.cmd。

CMD文件·CMD文件的编写

第1部分就是MEMORY伪指令，在PAGE0和PAGE1内分别定义不同的存储空间，各个存储空间的名字是可以任意取的，譬如定义空间RAML0的时候，可以取名为RAML0，也可以叫其他，从名称上可以看出RAML0是使用了F28335片内L0的空间，起始地址是0x008000，长度为0x001000。

下面来强调一下在定义存储空间的时候，需要注意的几点：

①同一页内空间的名称不能相同，不同页内空间名称可以相同。

②如果将一个较大的存储器划分成若干个存储空间，则地址范围不能有重叠。分开的存储空间的总和不能超过这个存储器的容量。

③存储空间的地址需要根据F28335存储器映像来决定，定义的空间地址范围一定要满足F28335的存储器映像，否则也会出错，譬如，RAM空间L0的起始地址是0x008000，长度为0x1000，如果RAML0定义的起始地址为0x007FFF，就会出错，因为起始地址不符合存储器映像，0x007FFF这个地址已经在L0地址范围0x008000~0x008FFF的外面了。总之，存储空间在定义时，无论是RAM空间或者是外设帧的空间，一定要仔细参考F28335的存储器映像。

第2部分就是SECTIONS伪指令，将编译器编译后产生的各个段分配到前面定义好的存储空间去。随意拿出一条语句来分析一下：

SciaRegsFile　　: > SCIA,　　PAGE = 1

这句话的意思很明显，就是将段SciaRegsFile装载到名为SCIA的空间，这个空间为数据空间，并且运行时也是在空间SCIA。段SciaRegsFile的内容是外设SCI-A的寄存器，空间SCIA的起始地址为0x007050，长度为0x000010，即16，这和表4-6中外设帧2内SCI-A寄存器的地址范围是吻合的。

CMD文件·CMD文件的编写

根据SECTIONS中属性load的语法，将“>”改为“load=”也是可以的，也就是说上面的语句也可写成：

SciaRegsFile　　: load= SCIA,　PAGE = 1

在开发DSP时，平时都是在调试程序，是把程序下载到RAM空间内的，而当开发完成时，就需要将程序烧写到FLASH空间内，对不同的存储空间进行操作时，很显然，CMD文件是不一样的。对RAM空间进行下载时就需要符合RAM空间的CMD文件，对FLSAH空间进行烧写时就需要符合FLASH空间的CMD文件。在共享资料的编程素材文件夹内有DSP2833x_Headers_nonBIOS.cmd、F28335_RAM_lnk.cmd和F28335.cmd，这些是通用的CMD文件，通常可以不做修改便能拿来使用。