CMD 文件的原理 玄德（网名）于 2009 年 3 月

一、前言

开发 TI 公司的 DSP 芯片，肯定要编写或者修改 CMD 文件，这是在单片机开发中没有碰到过的新事物，也是学习 DSP 的难点。面对里面种类繁多、名称各异、来历不明、作用不清、功能千差万别的存储器、区域和变量、寄存器，初学者往往都会一头雾水。甚至很多人已经把项目成功地完成了，对 CMD 文件仍然是一知半解。

笔者也经历了极度困惑的过程，曾经大量地看书，下载资料，分析所能搜集到的 CMD 源文件。可惜的是，无论是 TI 公司的原始文档，还是网上的资料，或者 BBS 的帖子，都没有透彻地说明 CMD 文件的原理和使用，只说“然”，要靠自己去体会“所以然”，去“悟”。

终于有一天，我悟到了，也许只是“一些”。现在，我把自己的“一些”写下来。我将细致而通俗地说明 CMD 文件的原理，给您“鱼”，更给您“渔”，一步步地引导象我当初一样的初学者。我将以 TI 的 2407 为对象展开说明，对于 TI 公司其他型号、其他系列的 DSP，道理是完全相同的。 用时下学术界最最最流行的语式， 叫做“基于 2407”——这个词起源于英文的“based on”，或“something based”，被我们大量地引用，以至于令人反胃了——我们美妙、绚烂的语言，现在只剩下“基于”了。

笔者水平有限，但保证会用心去写，您会看到很多别处没有的思路和信息，相信会基本打通初学者的任督二脉。本文适用于那些有单片机的开发基础、刚开始学习 DSP 的初学者。如果你还不知道程序空间，数据空间这些名词，可能就比较困难了。

二、CMD 文件的起源

在 DSP 系统中，存在大量的、各式各样的存储器，CMD 文件所描述的，就是开发工程师对物理存储器的管理、分配和使用情况。

有必要先复习一下存储器的知识。目前的物理存储器，种类繁多，原理、功能、参数、速度各不相同，有 PROM、EPROM、EEPROM、FLASH、NAND FLASH、NOR FLASH 等（ROM 类），还有 SRAM、DRAM、SDRAM、DDR、DDR2、FIFO 等（RAM 类）。无论多么复杂，从断电后保存数据的能力来看，只有两类：断电后仍然能够保存数据的叫做非易失性存储器（non-volatile，本文称为 ROM 类），数据丢失的叫做易失性存储器（本文称为 RAM 类）；ROM 类的芯片都是非易失性的，而 RAM 类都是易失性的。即使同为 ROM

类或同为 RAM 类存储器，仍然存在速度、读写方法、功耗、成本等诸多方面的差别。比如 SRAM 的读写速度，从过去的 15ns、12ns，提高到现在的 8ns、10ns，FLASH 的读取速度从 120ns、75ns，到现在的 40ns、30ns。

有没有人这样想过：使用存储器的人，希望存在这样的区别吗？

或者说，理想的存储器，应当是什么样的？

……

我们使用存储器时，如果没有人为地改变它，就希望里面的数据永远不要变，即使断了电也要完好地保存；如果里面的内容是我不需要的或者不能用的，我自然就会给它写入有用的内容，比如初始化。理想的存储器就应当永远保存数据，无论掉电与否，而且，希望读写速度为每秒无穷多字节，是 0ns，而不是什么 8ns，10ns。——不是吗？

然而，人类实现存储器芯片的技术，还没有达到理想情况，所以才会有这么多类别。“非易失”和“速度”就是一对典型的矛盾。非易失的 ROM 类存储器，可以“永远”地保存数据，但读写速度却很低，比如 30ns；RAM 的速度（8ns）一般都比 ROM（30ns）快得多，但却不能掉电保存。这是很无奈的现实。假如有那么一天，ROM 类的读写速度和 RAM 一样快，或者 RAM 也可以掉电保存数据，就不存在易失和非易失的区别了，那将是革命性的进步。那时，智能芯片和智能系统的设计将会有很大的变化，编写 CMD 文件就会很简单，甚至不需要了。

已经有芯片厂家做了一些这方面的工作，比如把电池和 RAM 结合起来，就是一个能掉电保存的 RAM。它既可以作为传统的 ROM 使用，又可以当 RAM 使用。但这显然只是一个暂时、折中的方法，其原理、成本、体积、容量还不如人意，不能算是“革命性”的进步。

我们平时在用到存储器的时候，要考虑哪些因素呢？

首先必须确认，在你的使用场合，是要永久保存数据，还是暂时保存？这关系到选择非易失性，还是易失性存储器的大问题，是首要的问题。在某些场合，如果必须永远地保存数据，即使希望速度快一些，也只能选择非易失的 ROM 类存储器，而把速度问题放在其次，或者另外想办法解决；另外一些场合，却要把速度放在第一位，只要在通电期间能够始终保存数据，就够了，当然就要选择 RAM 类的存储器了。

这两种情况我们都会遇到：程序代码一般都要存储在 ROM 类存储器中，否则，从设备生产开始，储存、运输，一直到用户手里，要必备不间断电源，还要保证不发生断电的意外；程序运行的时候，为了提高速度，就必须在 RAM 中运行，试想想，如果你的 MP4 放电影一停一顿的，谁还会用它看电影呢？所以 ROM 和 RAM 都是必不可少的，各有各

的用途，而且，出于功能、参数、速度、读写方法、功耗、工艺、成本等方面的考虑，往往要同时使用不止一种存储器。

事实上，TI 在设计 DSP 芯片时，也遇到同样的问题，TI 考虑的情况要比我们更多，更复杂。要知道，设计芯片的人是最牛 X 的，开发工程师只是跟在人家后面，在人家规定的框框里亦步亦趋。翻开 DSP 的 PDF 文档，找到 memory map 就会看到，芯片上集成了形形色色的存储器： FLASH、ROM、BROM、OTP ROM，SRAM、SARAM、DARAM、FIFO 等。就 2407 和 2812 而言，如果是做个流水灯之类的小东东，DSP 芯片加晶体加电源就可以了，片上集成的 ROM 和 RAM，在仿真状态下已经足够用了，烧写并脱离仿真器运行也足够。所以，它们的最小系统不需要外扩任何存储器。但也只能做简单的东东，往往还需要外扩一些 ROM 和/或 RAM 存储器，才能委以大用。（顺便说一句，DSP 的最小系统，要比 8951 芯片的最小系统大得多。）

千万不要被这些存储器的名称所迷惑！翻来覆去，其实就是两大类：非易失和易失。初学者往往忽略了这一点。

两大类！记住这一点，CMD 文件就是以这两类存储器为主轴，然后展开的。

DSP 芯片的片内存储器，只要没有被 TI 占用，用户都可以全权支配。TI 设计了“CMD 文件”这种与用户的接口形式，用户通过编写 CMD 文件，来管理、分配系统中的所有物理存储器和地址空间。CMD 文件其实就是用户的“声明”，包括两方面的内容：

1、用户声明的整个系统里的存储器资源。无论是 DSP 芯片自带的，还是用户外扩的，凡是可以使用的、需要用到的存储器和空间，用户都要一一声明出来：有哪些存储器，它们的位置和大小。如果有些资源根本用不到，可以视为不存在，不必列出来；列出来也无所谓。

2、用户如何分配这些存储器资源，即关于资源分配情况的声明。用户根据自己的需要，结合芯片的要求，把各种数据分配到适当种类、适当特点、适当长度的存储器区域，这是编写 CMD 文件的重点。

用户编写完自己的程序以后，要经过开发环境（编译器）的安排和解释（即编译），转换为芯片可以识别的机器码，最后下载到芯片中运行。CMD 文件就是在编译源程序、生成机器码的过程中，发挥作用的，它作为用户的命令或要求，交给开发环境（编译器）去执行：就这么分配！

下面将从这两个方面入手，详细说明如何编写 CMD 文件。

三、编写 CMD 文件之——资源清单

如上文所述，CMD 文件包含两大内容，首先就是存储器的资源清单，或者说，系统中（电路板上）可用的存储器资源。

TI 规定，CMD 文件的资源清单用关键字“MEMORY”作为标识，具体内容写在后面的大括号 { } 里面。如下面的形式：

```
MEMORY
{
PAGE 0:
    xxx     :   org = 0x1234 ,      length = 0x5678         /*This is my house.*/
PAGE 1:
    aaa     :   org = 0x1357 ,      length = 0x2468         /*My home here.*/
}
```

其中，MEMORY，PAGE n，org，length，包括冒号、等于号、花括号，都是关键字符，必不可少。

PAGE n 表示把可用的资源空间再划分成几个大块，最多允许分 256 块，从 PAGE 0 到 PAGE 255。如果把 MEMORY 比作图书馆，PAGE n 就是其中的“社科类”、“工程类”、“外文类”等。大家都习惯于把 PAGE 0 作为程序空间，把 PAGE 1 作为数据空间。如果你很好奇，也可以试试别的数字。凡智能芯片，都离不开这两种“空间”，大名鼎鼎的冯•诺依曼结构和哈佛结构，都是建立在程序空间和数据空间两种结构的基础上，我们面对的 DSP 也是如此。只要学习过单片机，就很容易理解。如果你构思出第三种结构，恭喜您，您将与这二位齐名了。

CMD 文件中还可以写上注释，用“/*”和“*/”包围起来，但不允许用“//”，这一点和 C 语言不同。

上面的例子，仅仅就是个“例子”，不针对任何特定的芯片。带注释的语句有两行，每一行都是一项声明，表示在程序空间或数据空间下，再细分更小的块，好比是“社科类”又分了几个书架。比如

```
xxx :    org = 0x1234 ,  length = 0x5678
```

表示在程序空间 PAGE 0 里面，划分出一个命名为 xxx 的小块空间，起始地址从存储单元 0x1234 开始，总长度为 0x5678 个存储单元，地址和长度通常都以十六进制数表示。所以，xxx 空间的实际地址范围从 0x1234 开始，到 0x1234 + 0x5678 － 1 = 0x68AB 结束（起始地址加长度再减一），这一段连续的存储区域，就属于 xxx 小块了。上面的例子中，PAGE 0 和 PGAE 1 各包含了只有一个“小块”，用户可以根据自己的情况，按照同样的格式任意增加。在支持多个 CMD 文件的开发环境里，某个或某几个 CMD 文件中，“小块”的数量可以为 0，也就是说，关键字 PAGE 0 或 PAGE 1 下面，可以是空白的。但不允许所有的 CMD 文件的同一空间都是空白。另外，没有资料提到过“小块”数量上限的限制，需要

去查阅文档或咨询 TI 公司。

很多关键字，还允许有别的写法，比如“org”可以写为“o”，“length”可以写为“len”。这些规定和其他细节，可以去查阅 TI 的 pdf 文档，一般叫做“xxxxx Assembly Language Tools User's Guide.pdf”，汇编语言工具指南，xxxxx 是芯片的型号或系列。但这个文档不适合初学者。

实践证明，至少对于 C2000 系列的 2407 和 2812 而言，存储单元的单位是“字 word”，即 16bit。但 TI 的文档却说是“字节 byte”，应当是 TI 写错了。

要特别注意以下几点：

1、必须在 DSP 芯片的空间分配的架构体系以内，分配所有的存储器。这里举两个例子：

a、对于 2407，程序空间和数据空间都是从地址 0x0000 到 0xFFFF，最大数值是四个 F，共 64K 字范围。所以，2407 的 CMD 文件中不能出现五位数的地址，也不允许任何一个小块空间的地址范围覆盖到 64K 以外的区域，因为 2407 根本就无法控制这些区域，或者说不能访问、无法寻址。要注意，起始地址和长度不要算错了。2812 也有同样的问题。

b、2407 的数据空间里，0x0100~0x01FF 和其他几块区域，是 TI 声明的保留空间（Reserved 或 illegal），也是芯片无法访问的，分配资源的时候不能涉及到这些区域。同样地，2812 的程序空间和数据空间，都有大片的保留区域，不能使用。

2、每个小块的空间，必须是一片连续的区域。因为，编译器在使用这块区域的时候，默认它是连续的，而且每个存储单元都是可用的。

3、同一空间下面，任何两个小块之间，不能有任何的相互覆盖和重叠。

在外扩存储器时，要保证片外的存储空间之间，特别是片外与片内的存储空间之间，不要发生冲突。有些空间，已经被 DSP 芯片的内部存储器占用了，用户是不可更改的，或只能通过模式配置，在一定范围内改动，用户自行扩展存储器时，要避开这些地方。

4、用户所声明的空间划分情况，必须与用户电路板的实际情况相符合！

对于用户自制的电路板，这是很容易出错的地方，通常会出现两种错误：

a、在设计硬件电路的时候，通常用 CPLD 作为片外存储器的选通信号，用 verilog 或者 VHDL 进行编程；也有用 74 或 4000 系列芯片来搭建的，已经很少了。如果 CPLD 逻辑出错，或者逻辑并没有真正写入 CPLD 芯片里面，即使 CMD 文件是正确的，即使编译已经通过，在仿真下载或者烧写的时候，PC 机都会报错而无法继续操作。

b、电路板有虚焊的地方，主要发生在 DSP 芯片的管脚、电平转换芯片的管脚，

及片外存储器的管脚上。这种情况，效果等同于上面所说的 CPLD 逻辑错误。更要命的是，补焊一次、两次甚至几次，虚焊仍然存在，这最容易把人搞糊涂了。笔者就经常遇到这样的事情。

出现这些硬件错误时，初学者往往不能正确地对故障作出定位，一会儿认为 CMD 文件有问题，一会儿觉得硬件电路有问题，反复地折腾，最后陷入迷茫。这时，一定要保持清醒的头脑：先检查原理设计；再检查硬件电路板，保证逻辑正确，焊接可靠；最后再去检查 CMD 文件。

5、一般地，初学者会找一些现成的 CMD 文件来用，一点改动都不敢。其实，胆子可以大一些，改一改，试一试，没什么大不了的。想学会游泳，必须要下水。DSP 芯片上的存储器，只要没有被 TI 用作专门的用途，用户都可以全权支配。空间的划分，是由用户决定的，可以根据需要，甚至个人的喜好来划分，名称也可以随意起，和 C 语言的变量名一样。

这里应当举一个 CMD 文件资源声明的例子，但为时过早。资源声明常常与资源分配是密切相关的，笔者把例子放在下一节，与资源分配一起详细说明，效果会好一些。

四、编写 CMD 文件之——资源分配

系统资源已经声明完了，现在就要说明，用户是如何分配这些存储器资源的，即向编译器声明资源的分配情况。

要合理地分配存储器资源，首先要搞清一个问题：资源要分配给谁？有哪些东东需要占用存储器？

我们来看下面这段不严格的 C 程序：

```
main（）
{
unsigned   int       i；
i ++；
}
```

这“段”程序只是笔者建立的一个模型，用它来代表几乎所有的程序：哪怕变量（包括数组）有一千个、一万个，都用一个“i”来代表；哪怕程序主体包含了各种搬移、运算、逻辑等动作，哪怕有一万行那么长，都用一句“i++”来表示。

让我们站在 TI 公司和编译器的角度，来考虑下面的问题：程序经过编译以后，会产生哪些对存储器资源有要求的“状况”？

有单片机开发经验的人都知道，至少要产生两种情况：

1、指令码，即二进制形式的指令，需要占用芯片的“程序空间”。这些数据，完全

等价于或等同于用户编写的程序，只是转换成了另一种形式而已。这种“数据”有两个特点：a、只要用户程序编写完成，这些“数据”就已经是可知的、可预期的，是由用户编写的程序代码和编译器共同决定的。b、在系统运行过程中，这些数据的内容不会发生任何变化，只会被读取，不会被修改。

2、在运行过程中，动态变化的“量”，需要占用“数据空间”。上面例子程序中的变量 i，就属于这种情况。这些数据，在设计师编写程序的时候，有时会预先写入具体的数值，即初始化，有时甚至根本不需要进行初始化。在运行过程中，既要被读取，又会被改写，经常在变化。设计师自己也很难确切知道，在某一时刻，这些数据的具体的数值是什么，最多只知道它们的位数、最大和最小值的范围。

那么，什么样的物理存储器适合于数据空间使用，什么样的存储器适合于程序空间使用呢？

对于数据空间，其最基本、最首要的要求是速度快，并不要求掉电保存数据的能力，显然应当由 RAM 类存储器来承担，所以，RAM 一般都必不可少。但是，并不是说数据空间只能连接 RAM 芯片，只要你能够接受比较慢的速度，并且安排好芯片的控制时序，你完全可以在数据空间扩展 ROM 类存储器。

程序空间的代码数据，一般都要求掉电保存，只能由 ROM 来承担，所以 ROM 必不可少。那么，ROM 的读取速度慢的问题，怎么解决呢？对于有些低速的智能芯片，ROM 的速度慢一点，是完全可以接受的，可以直接从 ROM 中读取代码指令，然后译码、执行；我们熟悉的 MCS51、PIC 系列单片机，都是这么做的（以下信息笔者不能保证正确性：2407 脱离仿真器运行时，似乎也是直接从 ROM 中读取程序代码）。另外有一些低端的智能芯片，生产商通过特殊的技术手段，在一定范围内等效地提高内部程序 ROM 的读取速度，比如 NXP 公司的 ARM 芯片 LPC213x，虽然 ARM 内核的数据接口只有 32 位，但 LPC213x 的片内 FLASH 程序存储器，与内核之间的接口居然是 128 位宽度，通过所谓“加速器”相连接。对于高速的智能芯片，从 ROM 直接读取代码并执行，已经不能满足速度的要求了，通常的解决方法是，把程序代码储存在 ROM 中，在每次上电运行时，通过“引导程序”把用户代码读出并保存在 RAM 中，然后从 RAM 中运行，这样做既解决了 ROM 速度慢的问题，又解决了 RAM 掉电丢失数据的问题。

实际操作中，并不是只有指令码和变量 i 这么简单，除这两项以外，还会出现很多小“状况”；而且，当芯片型号不同，甚至用户源程序不同时，出现的细节也是变化的。恰恰就是这些变化，导致 CMD 文件变得复杂。

但是，任何大“状况”、小“状况”，都归属于对程序空间和数据空间的操作，不存

在第三种空间。（有些 DSP 的所谓“IO 空间”，实质上是数据空间的一个变种，但又脱离了数据空间，不属于 CMD 文件考虑的范围。）

编写 CMD 文件，就是要搞清楚以下情况，并对编译器做出声明：

1、你的系统都有哪些存储器资源？

2、哪些存储器安排在程序空间，哪些在数据空间？

3、你的系统会产生哪些大“状况”和小“状况”？

4、哪些状况属于程序空间，哪些属于数据空间？

5、程序空间的“状况”如何安排在程序空间的资源里，数据空间的“状况”如何安排在数据空间的资源里？

笔者想从事情的起源入手，逐步引导初学者自己去发现“资源要分配给谁？有哪些东东需要占用存储器？”这个问题的答案，所以使用了一些不正规的术语，比如“状况”这个词。

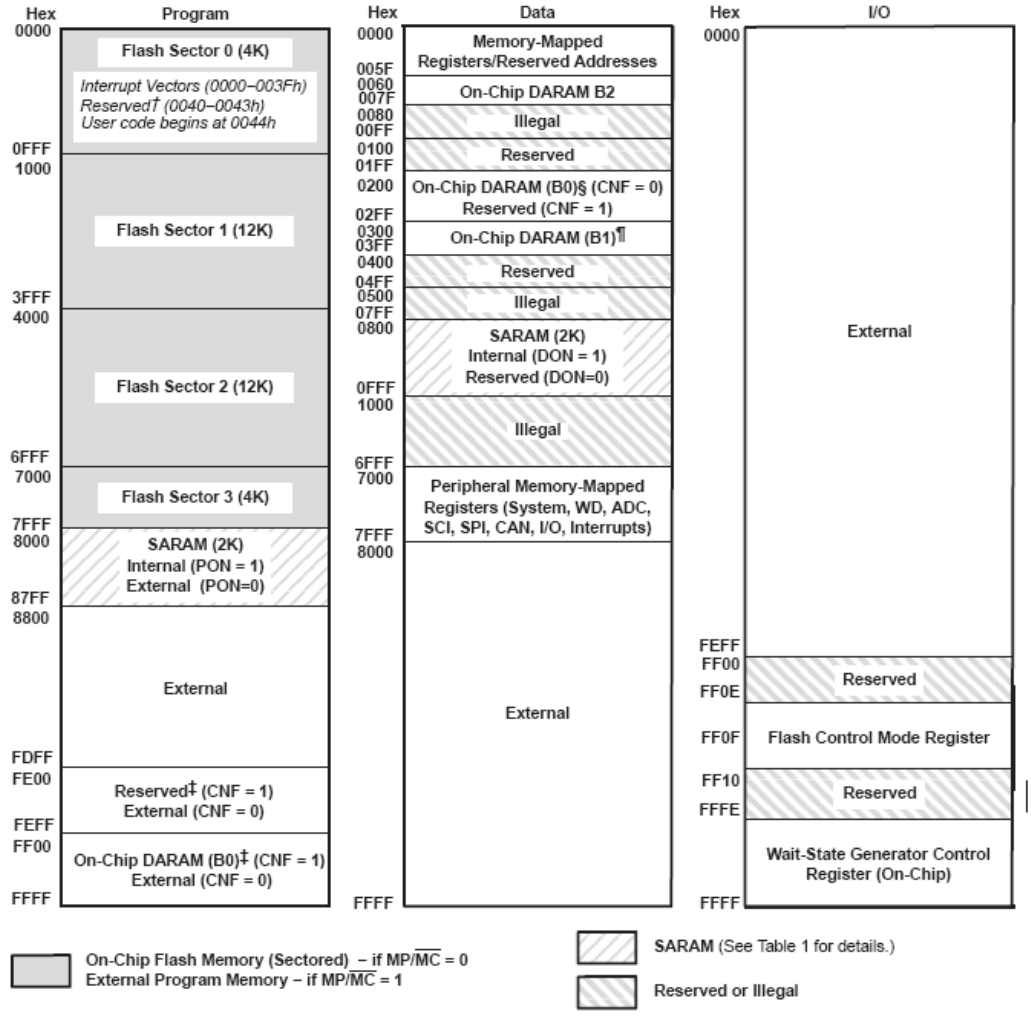
让我们从一个实际使用过的 2407 芯片的 CMD 文件来展开说明，其他 DSP 芯片的 CMD 文件与此大同小异：

```
/*********************************************************************************************/
-stack 200h                                                          /* #1   */
/*********************************************************************************************/
MEMORY                                                               /* #2   */
{
PAGE 0 :
 VECS       : origin = 0000h , length = 0040h      /*  中断向量          */   /* #3   */
 PROG       : origin = 0100h , length = 7F00h      /*  片上 FLASH        */   /* #4   */

PAGE 1 :
 B2         : origin = 0060h , length = 0020h      /* DARAM B2 块        */   /* #5   */
 B0B1       : origin = 0200h , length = 0200h      /* DARAM B0 块        */   /* #6   */
 SARAM      : origin = 0800h , length = 0800h      /* SARAM  块          */   /* #7   */
 ExtSRAM    : origin = 8000h , length = 8000h      /*  外部存储器        */   /* #8   */
}
/*********************************************************************************************/
SECTIONS                                                             /* #9   */
{
 .vectors   :  >   VECS      PAGE 0          /*  中断向量表        */   /* #10 */
 .text      :  >   PROG      PAGE 0          /*  代码              */   /* #11 */
 .cinit     :  >   PROG      PAGE 0                                     /* #12 */

 .bss       :  >   SARAM     PAGE 1                                     /* #13 */
 .stack     :  >   B0B1      PAGE 1                                     /* #14 */

 .extdata   :  >   ExtSRAM   PAGE 1                                     /* #15 */
```

```
}
/*****************************************************************************************************/
```

下图是 2407 芯片的空间分配情况（Memory Map），是从 2407 的数据手册直接复制过来的：

memory maps



NOTE A: Boot ROM: If the boot ROM is enabled, then addresses 0000–00FF in the program space will be occupied by boot ROM.
† Addresses 0040h–0043h in on-chip program memory are reserved for code security passwords.
‡ When CNF = 1, addresses FE00h–FEFFh and FF00h–FFFFh are mapped to the same physical block (B0) in program-memory space. For example, a write to FE00h has the same effect as a write to FF00h. For simplicity, addresses FE00h–FEFFh are referred to as reserved when CNF = 1.
§ When CNF = 0, addresses 0100h–01FFh and 0200h–02FFh are mapped to the same physical block (B0) in data-memory space. For example, a write to 0100h has the same effect as a write to 0200h. For simplicity, addresses 0100h–01FFh are referred to as reserved.
¶ Addresses 0300h–03FFh and 0400h–04FFh are mapped to the same physical block (B1) in data-memory space. For example, a write to 0400h has the same effect as a write to 0300h. For simplicity, addresses 0400h–04FFh are referred to as reserved.

Figure 1. TMS320LF2407A Memory Map

#2 行至 #8 行，MEMORY {……} 部分，就是上一节我们已经说明的，系统可用资源的声明，包括程序空间 PAGE 0 和数据空间 PAGE 1 两部分。

程序空间 PAGE 0，又分为 VECS 区域和 PROG 区域。

#4 行所声明的 PROG 区域，是为用户指令码分配的存储空间，这部分空间一般都

很大（比如 0x7E00h）。

相当于 PROG 用户指令码区域，#3 行声明的 VECS 区域是一个特殊的“小状况”，TI 在设计 2407 的硬件电路时，用这块区域来保存各种中断服务程序的入口地址，即中断向量，与硬件电路挂钩，不能与一般的程序代码相混杂，所以要单独声明。按照芯片手册的说法，0x0000 至 0x003F 共 0x40 个存储器单元是中断向量，0x0040 至 0x0043 四个单元是保留位置。在上面的例子中，由于 0x0040 ~ 0x0043 四个单元暂时无用（***reserved***），所以，VECS 区域只覆盖了 0x0000 ~ 0x003F；如果把 0x0040 ~ 0x0043 也覆盖进来，估计也没有问题，因为存放中断服务程序入口地址，是编译器根据用户的声明填充的，它会把有用的地址数据安排到对应的单元里，至于没用的空间，无论保存了什么样的地址，对于用户都无所谓。另外，按手册的说法，用户代码似乎应当从 0x0044 单元开始（***User code begins at 0044h***），实际上可以这么做，也可以不这么做，只要在芯片的程序空间里，与其他空间不发生冲突，从哪个单元开始都可以，编译器自然会安排，上面的例子就是从 0x0100 单元开始存储程序代码。长度也是用户确定的，不一定要象例子那样，在 0x7FFF 单元结束。

笔者自行扩展了一块 SRAM 存储芯片，型号为 IS61LV6416，是 ISSI 公司的产品，总容量 64K 字（word），通过 CPLD 逻辑电路，把一半的容量安排在程序空间的 0x0000 至 0x7FFF，覆盖了 PROG 和 VECS 两块区域。所谓“安排”，就是常说的“映射”。仔细看一下 2407 的 Memory Map，程序空间从 0x0000 至 0x7FFF，已经全部被片内 FLASH 存储器“占用”，怎么能分配给其他芯片呢？再说，程序代码保存到 SRAM 里面，掉电岂不丢失？…… TI 在设计 2407 硬件电路的时候，给用户提供了一个 MP/MC 管脚，该管脚接 0 电平时，程序空间通向外部存储器的接口（External Memory Interface）被切断，只对片内的 FLASH 存储器进行寻址，程序空间全部被 FLASH 占用；该管脚接 1 电平时，片内 FLASH 被隔离，只对外接的存储器进行访问。在开发阶段，程序代码写入 SRAM，断电当然就丢失了，但这只麻烦开发人员一个人，每次都要重新往 SRAM 里写一遍，开发的时候，程序本来就在变，就必须重写；开发成功了，再写入 FLASH 里，交付用户。那么，TI 这么做，是否多此一举，直接在 FLASH 里开发，不行吗？笔者不好妄下结论，估计是出于以下考虑：a、烧写 FLASH，需要特殊的算法即时序，在仿真状态下进行烧写可能有困难，或存在其他问题；b、在 FLASH 中运行程序时，难以同时进行仿真；c、FLASH 存储器的烧写寿命有限。各位可以结合自己的经验，考虑一下这个问题。……总之，TI 设计了这种方式，在仿真开发阶段，使用外扩的 SRAM 存储器，工程师把 VECS 数据和 PROG 数据，通过仿真器和 CCS 环境的“load program”指令，下载到 SRAM 芯片里运行；开

发成功以后，再通过 TI 提供的专用烧写插件，把代码烧到 FLASH 存储器的对应空间里，交付用户使用。所以，开发成功以后，程序空间外扩的 SRAM 芯片也就不需要了，完全可以删除，说不定还能节省一些产品成本呢。

顺便说一下，对于 2407，无论是仿真开发还是脱离仿真，最好不要使用 0x8000 ~ 0xFFFF 的高 32K 程序空间，原因有三：a、仿真阶段和脱离仿真器运行时，无法使用同一个 CMD 文件；b、会出现中断不正常的问题，在网上的论坛里，经常有人提问；c、最重要的原因，是笔者的经历，曾经搞一个项目，代码量超出了 32K，需要在高 32K 空间扩展程序存储器，咨询 TI 公司后得知，必须由 TI 提供特殊的 CCS 文件，而且 TI 不能保证结果的正确性！ 后来笔者只好缩减代码。在 CMD 文件中，有意把片内 FLASH 的地址和片外的 SRAM 地址相重合，只需要用跳线改变 MP/MC 管脚的电平， 就能同时避开 a 和 b 两个问题，何乐而不为呢 ？！**在仿真阶段和脱离仿真阶段，完全可以使用同一个 CMD 文件**。

IS61LV6416 的另一半，安排在**数据**空间，下文会进一步说明。至于把 IS61LV6416 的低 32K 安排在**程序**空间、高 32K 安排在**数据**空间，还是正好相反，都无所谓，也都是可以实现的，仅仅 CPLD 的逻辑不同而已，很多人会在这里糊涂半天。

PAGE 1 是数据空间。#5、#6、#7 三行所声明的 B2、B0B1、SARAM 三块存储区，是 2407 芯片内部集成的存储器，彼此的地址都不连续，所以要分别声明。B0B1 块，是由 B0 块和 B1 块合并组成，但二者是有区别的：B1 块地址始终固定在数据空间的 0x0**3**00h ~ 0x0**3**FF 区域，B0 块在芯片复位后的默认地址是数据空间的 0x0**2**00h ~ 0x0**2**FF 区域，但用户可以通过软件设置 CNF 位，把 B0 块转移到程序空间里。如果用户需要这样的转移，就不能把 B0、B1 合并起来；如果用户不做这样的转移，就可以象这个例子一样，B0B1 合并起来整体使用，占用地址范围为 0x0**200**h ~ 0x0**3**FF。

#8 行所声明的区域，就是上文所说的 IS61LV6416 芯片的“另一半”。之所以安排在数据空间的 0x8000 至 0xFFFF 区域，原因很简单，因为这里是 TI 指定的外扩数据存储器的位置，数据空间的其他位置，基本上都被片内集成的存储器所占用，或者被禁用。起始位置和长度是由用户自己决定的，你可以把这片区域分成几个小块来使用，只要相互不重叠，不超出 0x8000 至 0xFFFF 区域，并且修改和增加对应的声明，就可以。声明语句的格式都是一样的。另一方面，如果片内的 2K 多的存储器已经够用，就不必外扩了，也就不再需要#8 这一行的声明。所以，如果只是用 2407 做个流水灯之类的小东东，就不需要外扩任何存储器，片上的 ROM 和 RAM 资源，在仿真状态下已经足够用了，脱离仿真器运行也足够。

#4~#8 五行所声明的空间，都可以进一步拆分，如我在前面所说的，是由用户决定的，可以根据需要，甚至个人的喜好来划分。但#3 行的 VECS 区域，因为与硬件挂钩的缘故，一般都不再细分。#3~#8 共六行资源声明里，VECS、PROG、B2、B0B1、SARAM、ExtSRAM 这些名称，都允许用户自己来起名，和 C 语言的变量名一样；但在后面 #10 ~ #15 的几行里引用的时候，必须使用同样的名称。

片上的存储器，B2、B0、B1 三块是 DARAM，全称是 dual-access RAM，根据手册的说明，它能够在同一个“循环”内（***in the same cycle***），同时完成读出和写入；另一块是 SARAM， 全称 single-access RAM，不要与“SRAM”相混淆。手册上和 TI 网站的其他材料上，没有进一步的介绍。我们可以推断，DARAM 的速度要比 SARAM 快，SARAM 比 SRAM 快，但是电路结构的复杂性、实现的成本也与速度成正比关系，所以，2407 的片内 DARAM 只有 544 个字，SARAM 却有 2K 字之大。好了，我们不必知道它们的细节，总线接口、读取方式、写入方式、刷新方式、指标参数，这些是 TI 更关心的事，我们只要记住它们的特点：它们都是 RAM 类存储器，掉电要丢失数据的； DARAM 的读写速度最快，SARAM 次之。我们分配资源的时候要考虑这些特点，量才适用。

#9 行至#15 行，SECTIONS {……} 部分，就是所谓资源的分配。

首先，SECTIONS，PAGE，包括花括号、冒号，都是关键字符。注意：SECTIONS 字符是复数形式。在花括号内，每一行最左侧的“.vectors”、“.text”、“.cinit”、“.bss”、“.stack”这些名称，包括小数点，都是 TI 默认的关键字符，只有“.extdata”是用户自己定义的名称。另外，“VECS”、“PROG”、“SARAM”、“B0B1”、“ExtSRAM”必须是在 MEMORY 里声明过的资源名称。除此以外，有些字符也允许有别的写法，参见“Assembly Language Tools User's Guide.pdf”，汇编语言工具指南。

这些东西，就是前文所说，对存储器资源有要求的“状况”！前面声明的存储器资源，就是要分配给这些“状况”使用的！

初次接触这些名称，一定会一头雾水：这些都是什么东西？从哪里冒出来的？

在 TI 的《汇编语言工具指南》里，这些名词统称为“directives”，“指令”的意思，实际上是针对编译器的“伪指令”，在芯片的指令集里是找不到这些指令的，不要把二者相混淆。“.vectors”、“.text”、“.cinit”、“.bss”、“.stack”，这些包括小数点的单词，都是 TI 规定的关键字（这些定义应当隐含在 TI 提供的某个文件中国，比如 .lib 库文件），在用户自己的源程序中，一般不能也不需要对这些关键字做定义或声明，只是去引用它们；但“.extdata”与这些关键字不同，是由用户自己定义的，下文会进一步说明。每条伪指令，要求编译器在程序空间或数据空间里，保留指定数量的存储单元，这些存储单元叫做

“sections”，一般翻译为“段”，与笔者所说的“状况”相对应。大家在提到这些伪指令名词的时候，有时来代表这些指令，更多的时候是代表它们所对应的段。如果使用汇编语言开发 DSP，一定要用到这些伪指令，也会对它们有较深刻的理解。

段分为两类：已初始化段（**Initialized Sections**）和未初始化段（**Uninitialized Sections**）。所谓“已初始化”，具有两个特点：a、只要用户程序编写完成，这些“数据”就已经是可知的、可预期的，是由用户编写的程序代码和编译器共同决定的；b、在系统运行过程中，这些数据的内容不会发生任何变化，只会被读取，不会被修改，下次再通电，这些数据依然存在。显然，指令码就属于“已初始化的”，但“已初始化”并非只包含指令码，指令码只是“段”的一种而已，一般还会有其他的“段”。所谓“未初始化”，就是上文所说的，“设计师自己也很难确切知道，在某一时刻，这些数据的具体的数值是什么”的意思，上面提到的变量 **i** 就属于“未初始化的”，同样地，“未初始化”还包括其他段。这里的“初始化”，是从编译器的角度来考虑的，是“可预知、可预期”的意思，并不是我们通常说的，给某个变量赋予初始值的那个“初始化”。

当芯片型号不同，甚至用户源程序不同时，编译产生的“段”也是不同的；反之，产生哪些段，是由芯片型号和用户的源程序共同决定的。

那么，我们怎么知道，我的工程项目会产生哪些“段”呢？工程项目在编译之后，会在项目文件夹内产生一个 .map 文件，用随便一个文本编辑器就可以打开，内容也很容易理解。初学者可以先找一个现成的 CMD 文件，稍作修改或者不修改，加入项目中进行编译，如果编译失败（failure 或 error），则根据提示进行修改，如果只是告警（warning）则不必理会。成功编译之后，查看 .map 文件中“output section”那一列，那些长度（length）非 0 的段，就是你的项目真正会产生的段；那些长度为 0 的段，基本都可以从 CMD 文件中删除。有时也存在这样的情况：某些长度为 0 的段，即使开发人员并没有在 CMD 文件中作出声明，仍然会在 .map 文件里出现，这对我们的开发并没有影响。

我们仍然通过前面的 CMD 例子，来看这些段都是什么意思，为了方便读者理解，我把说明的顺序调整了一下。另外，开发过 MCS51 或其他单片机的人，应当边看边想，想想单片机的程序，去体会与这些段相对应的东西。

“.text”，就是编译后生成的二进制指令代码段。我们甚至可以用手工把 C 程序或汇编程序，翻译成二进制指令代码，所以，它显然属于“已初始化的”段。我们编写的 main 主函数，“子”函数或子程序，中断服务函数或程序，它们都会产生指令代码，也都属于这个段。通过#11 行的声明

```
.text   :   > PROG        PAGE 0
```

编译器就知道设计师的意图了，是要把所有的二进制代码，按顺序串行地汇集起来，一起编入 PROG 区域，即#4 行已经声明的，程序空间 PAGE 0 的 **0x0100h ~ 0x7FFF** 地址范围内。每个函数的代码块的首地址，长度等信息，都记录在 .map 文件中。至于这些代码最终写入哪个物理存储器，是片内的 FLASH，还是片外扩展的程序 SRAM，是由 MP/MC 管脚决定的（对于 2407 芯片），已经不是 CMD 文件的责任了。

“.vectors”，表示“中断向量段”，也就是中断服务程序的入口地址段。很显然，这个段要求物理存储器必须能够掉电保存数据。我们在编写用户程序的时候，普通的函数完全按照标准 C 语言的语法，比如 *void main(void) { …… }* ，但所有的硬件中断服务函数，必须在前面加一个关键字“interrupt”，比如某个服务函数 *abc( )*是这样写的：*interrupt void abc( ) { ……}* 。对于 2407，这些还不够，源程序还必须包含一个 vectors.asm 文件，其中的一句声明，把中断服务函数 *abc ( )* 与具体的硬件中断对应起来：

int1: b _abc

有了这些声明，编译器就会把函数 *abc ( )* 的代码块的首地址，编排到与 int1 中断对应的向量中，写入#3 行定义的 **0x0000h ~ 0x0040** 处，中断向量的地址空间里。这个首地址，编译器当然是知道的，所以“.vectors”也属于“已初始化的”段。用户如果想知道中断服务程序的入口地址，只能去查看 .map 文件。对于 2812，情况基本相同，不同之处是用 C 语句代替 vectors.asm 文件：*PieVectTable.TINT0 = &abc;* 或者 *PieVectTable.XINT1 = &xint1_int;* 之类。

“.cinit”段，定义比较模糊，有文章解释为“对全局变量和静态变量初始化的常数”。按笔者理解，我们经常用到的数表，比如七段显示器的代码表、液晶的显示字符代码表、正弦数表等，都属于这个段。那么，还包括哪些内容呢？笔者也不是很确定。但有一点是肯定的：它属于“已初始化的”段，必须作为代码，存储在程序空间里，而且必须能够掉电保存。

“.stack”，就是我们常说的堆栈，我们根本不可能知道堆栈内部数据的变化情况，所以，它属于“未初始化的”段，定位在数据空间。在调用函数、保存现场时，一定要用到这个段，但笔者怀疑，还会有其他的用途，比如用堆栈来批量交换数据。显然，堆栈内部的数据，没有掉电保存的必要。在上面作为例子的 CMD 文件中，#1 行是用户对堆栈空间的大小所作的声明，是按照 TI 公司规定的语法，200h 表示 512 个单元；如果用户没有作出#1 行的声明，编译器将按照默认的数量来分配空间。一般来说，如果你无法确定程序运行究竟需要多大的堆栈，就尽量设置大一点。例子中把整个 B0B1 存储器块，都作为堆栈使用。

“.bss”，定义同样比较模糊，但很容易意会。有文章介绍为“保存全局变量和静态变量”。很显然，它属于“未初始化的”段，要定位在数据空间。前面 C 程序例子中，变量 i 就定位在这里。这个段，同样不要求掉电保存数据。

“.extdata”，是笔者自行定义的段，属于“未初始化的”，专门用来保存一个 20000 个元素的数组 *xyz[20000]*，数据定位在外扩的 SRAM 中，不需要掉电保存。这是 TI 为用户设计的，把某些内容从 bss 段里分出来，进行特殊定位的方法。具体作法是：

a、在 CMD 文件中作出 #15 行的声明

```
.extdata    :  >    ExtSRAM   PAGE 1
```

“extdata”是用户自行定义的名称。

b、在源程序中包含如下的语句：

```
unsigned   int    xyz[20000];
#pragma    DATA_SECTION    (xyz,".extdata")
```

其中，*#pragma*、*DATA_SECTION* 和小括号，都是规定的关键字符。

这样，用户的声明就完整了，编译器将把数组 *xyz[20000]* 从 bss 段里分出来，单独定位。所以，这个指令的优先级要高于 bss。

我们再来看几个未初始化段的物理定位：

```
.bss        :  >    SARAM     PAGE 1          /* #13 */
.stack      :  >    B0B1      PAGE 1          /* #14 */
.extdata    :  >    ExtSRAM   PAGE 1          /* #15 */
```

这三个段的数据，都不需要掉电保存。例子中，三个段都定位在数据空间的 RAM 类存储器中，但分别属于三个不同的物理存储器。实际上，你完全可以按照自己的意愿甚至个人喜好，随意地定位，比如把任意一个段定位在任意一个物理存储器里，或者把某两个段定位在同一个物理存储器里，或者三个段都挤在一个物理存储器里，都可以，实际运行起来都没有任何问题！反正三个存储器都属于数据空间，反正谁都可以闲着谁都可以忙起来，反正爱谁闲着就闲着爱谁忙就忙，不需要顾及它们的情绪。当然，前提是物理空间够用。但是，想把 stack 分成几块，分别定位在几个存储器，可能是行不通的。bss 也无法分成两块，因为无论你怎么分，剩下的还叫做 bss，除非你用前面介绍的方法，把所有的变量都人工定位，然后是什么情况，笔者也不知道了。那么，分来配去，总要追求点什么意义才好吧？！前面说过，这些存储器的区别主要是速度，能够最大程度提高系统运行速度，才是应当追求的目标。具体该怎么分配，就靠读者自己去体味吧。笔者讲一个例子，供读者参考：前面的数组 *xyz[20000]* ，如果没有对它单独定位，它就会被编译器并入 bss 段，bss 段的长度将超过 20000 了，由于片内存储器的容量都没有这么大，只能把 bss 定位在

外扩的 SRAM 中。但是，笔者又不愿意浪费片内的 SARAM，它的访问速度毕竟要比 SRAM 要快一些，闲着实在可惜，所以才把数组从 bss 中分出来，以便给 SARAM 和 SRAM 分别派上适当的用场。

以上介绍的都是 2407 芯片常见的段，还有几个段，比如 “.switch”，“.const”，稍微复杂一些，这里不再详述。相信读者掌握了本文介绍的思想以后，能够很快地理解和掌握 CMD 文件。

五、DSP 系统硬件电路板的故障排查

自己焊接装配的电路板，刚装配完成的时候，一般都会存在各种故障。查找故障的部位，是电子工程师的基本功。

第一步，一个 DSP 电路板刚装配完，必须先排除电路板的硬故障，比如电流异常，芯片发热，DSP 始终处于复位状态，振荡器没有起振等。这些故障没有“放之四海而皆准”的方法，每个人遇到的情况都会不一样。一般来说，把所有的引脚重新焊一遍，可以解决大部分故障。

第二步，看仿真器能否与目标板连接。把 PC 机与仿真器连接（要保证仿真器已经正确驱动），仿真器与目标电路板正确连接，目标板通电。这些硬件操作完成后，再启动 CCS（要保证 CCS 已经按照目标板的芯片型号进行了设置）。几秒种后，如果已经正确连接，在 CCS 界面的左下角会出现“目标板已经连接”的提示。当然还有其他的提示方式，比如弹出汇编语言窗口等。如果仿真器无法成功与目标板连接，就说明目标板上有故障。据笔者的经验，一般是 JTAG 接口的几条线上有短路或断路，数据线、地址线上有短路或断路，READY 信号错误。

第三步，这时就可以调试我们的程序了。当我们把程序编译完成，通过 CCS 开发平台的“load program”命令，把代码调入外扩 SRAM 的时候，经常会遇到错误告警，说在某地址处出现“校验错误”。这大多是因为地址总线、数据总线，或者控制线的虚焊、短路引起的，主要发生在 DSP 芯片的总线管脚、电平转换芯片的管脚，及片外存储器的管脚上；或者片选逻辑电路有错误。CCS 把代码写入 SRAM 之后，会逐个单元读出，与原始数据进行比较，如果二者不相同，就会提示这样的错误。

遇到这个问题怎么办呢？这里介绍一个用软件去查找硬件故障的方法，非常有效。前面说过，DSP 的最小系统不需要外扩任何存储器，其自身的存储器足够运行在仿真器状态，或脱离仿真器的状态。我们可以利用这个特点，把外扩的 SRAM 或其他外设，都作为外设来操作，用软件生成一些有规律的波形，帮助我们定位故障的部位。具体操作如下：

a、编写一段程序，向外扩 SRAM 的存储单元里，按地址顺序，依次写入连续变化的数据，象下面这段程序：

```
unsigned int      i,*R;
for  (   i = 0x8000;   i <= 0xFFFF;     i++   )
    {
    R = ( unsigned int *) i;
    *R = i;
    }
```

这里的 0x8000 ~ 0xFFFF 是外扩的数据存储器的地址。无条件地、反复地运行这段程序，就会在地址线、数据线上产生连续的方波，最低位的 A0 和 D0 的周期最短、频率最高，A1 和 D1 的周期分别是 A0、D0 的两倍，A2、D2 是 A1、D1 的两倍，依次类推。

b、在 CMD 文件中，把这段程序定位在片内程序空间的 RAM 类存储器中，而不是外扩的存储器，就是前面说的思路：利用 DSP 的最小系统，把外扩 RAM 作为外设来操作。

c、在仿真状态下，全速运行，用示波器检查所有数据总线、地址总线、控制总线和片选信号的波形，顺藤摸瓜，就能查到故障位置。

六、结语

CMD 文件是开发 DSP 芯片的一个难点。如果站在 TI 公司或者编译器的角度，去考虑问题，对于理解 CMD 文件会有很大的帮助，从而更快地掌握 CMD 文件的编写。本文正是按这个思路展开的，如果能够对初学者有一些帮助，将是笔者莫大的荣幸。

错误和不足之处，欢迎批评指正：2001LHL@21CN.COM。