CMD 简介

三兄弟嵌入式

官方网站：

http://sxdembed.taobao.com/

为何要讲CMD

- 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。
- 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。

存储器的两大种类

- 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。

两大种类的优劣

- 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。

如何选择存储器

- 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。

TMS320F28335存储器类型

- TI在设计DSP芯片时，包含了FLASH、ROM、BROM、OTP ROM，SRAM、SARAM、DARAM、
- FIFO等。

CMD文件的工作原理

- 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”。

CMD的两方面内容

- 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。

CMD何时作用

- 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！

- CMD是用来分配ROM和RAM空间用的,告诉链接程序怎样计
- 算地址和分配空间。所以不同的芯片就有不同大小的ROM和RAM,
- 存放用户程序的地方也不尽相同。所以要根据芯片进行修改，分为
- MEMORY和SECTIONS两个部分。

MEMORY格式：

```
MEMORY
{
 PAGE 0:   /* Program Memory */

 PAGE 1:   /* Data Memory */

   DEV_EMU     : origin = 0x000880, length = 0x000180     /*
device emulation registers */
   FLASH_REGS  : origin = 0x000A80, length = 0x000060     /*
FLASH registers */
   CSM         : origin = 0x000AE0, length = 0x000010     /*
code security module registers */

   ADC_MIRROR  : origin = 0x000B00, length = 0x000010     /*
ADC Results register mirror */
}
```

注意：

- 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。

SECTIONS格式：

```
SECTIONS
{
   PieVectTableFile : > PIE_VECT,   PAGE = 1

/*** Peripheral Frame 0 Register Structures ***/
   DevEmuRegsFile    : > DEV_EMU,    PAGE = 1
   FlashRegsFile     : > FLASH_REGS, PAGE = 1
   CsmRegsFile       : > CSM,        PAGE = 1
   AdcMirrorFile     : > ADC_MIRROR, PAGE = 1
   XintfRegsFile     : > XINTF,      PAGE = 1
}
```

- 3、 SECTIONS伪指令
- SECTIONS指令的语法如下:
- SECTIONS
- {
- .text:{所有.text输入段名}load = 加载地址 run = 运行地址
- .data:{所有.data输入段名}load = 加载地址 run = 运行地址
- .bss:{所有.bss输入段名}load = 加载地址 run = 运行地址
- .other:{所有.other输入段名}load = 加载地址 run = 运行地址
- }

- 如下例所示:
- .const:{略}load = PROG run = 0x0800, 常量加载在程序存储区, 配
- 置为在RAM里调用。“load = 加载地址”的几种写法需要说明一下,
- 首先“load”关键字可以省略, “=”可以写成“>”,“加载地址”可以是:
- 地址值、存储区间的名字、PAGE关键词等,
- “run = 运行地址”中的“=”可以
- 用“>”, 其它的简化写法就没有了

多关键字

- 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“STM320F28335 Assembly Language Tools User's Guide.pdf”，汇编语言工具指南。

CMD要点注意：

- 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
- 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。
- 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！

- 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？

系统定义

- 存储模型：c程序的代码和数据如何定位
- 系统定义：
- .cinit 存放程序中的变量初值和常量
- .const 存放程序中的字符常量、浮点常量和用const声明的常量
- .switch 存放程序中switch语句的跳转地址表
- .text 存放程序代码
- .bss 为程序中的全局和静态变量保留存储空间

- .far 为程序中用far声明的全局和静态变量保留空间
- .stack 为程序系统堆栈保留存储空间，用于保存返回地址、函数间的
- 参数传递、存储局部变量和保存中间结果
- .sysmem 用于程序中的malloc 、calloc 、和realoc 函数动态分配存
- 储空间
-

C语言的段

- C语言的段
- 未初始化块（data）：
  - .bss 存放全局和静态变量
  - .ebss 长调用的.bss(超过了64K地址限制)
  - .stack 存放C语言的栈
  - .sysmem 存放C语言的堆
  - .esysmem 长调用的.sysmem(超过了64K地址限制)
- 初始化块：
  - .text 可执行代码和常数(program)
  - .switch switch语句产生的常数表格（program/低64K数据空间）
  - .pinit Tables for global constructors (C++)(program)
  - .cinit 用来存放对全局和静态变量的初始化常数值(program)
  - .const 全局和静态的const变量初始化值和字符串常数（data）
  - .econst 长.const（可定位到任何地方）（data）

自定义段（C语言）

- 3> 自定义段（C语言）
- #pragma DATA_SECTION(函数名或全局变量名,"用户自定义在数据
- 空间的段名"); #pragma CODE_SECTION(函数名或全局变量名,"用户
- 自定义在程序空间的段名"), 不能在函数体内声明，必须在定义和使
- 用前声明，#pragma可以阻止对未调用的函数的优化

ramfuncs

- ramfuncs : LOAD = FLASHD,
  RUN = RAML0,
- LOAD_START(_RamfuncsLoadStart),
- LOAD_END(_RamfuncsLoadEnd),
- RUN_START(_RamfuncsRunStart),
  PAGE = 0

EG

```
eg:
MEMORY
{
PAGE 0:VECS: origin = 00000h, length = 00040h
LOW: origin = 00040h, length = 03FC0h
SARAM: origin = 04000h, length = 00800h
B0: origin = 0FF00h, length = 00100h

PAGE 1: B0: origin = 00200h, length = 00100h
B1: origin = 00300h, length = 00100h
B2: origin = 00060h, length = 00020h
SARAM: origin = 08000h, length = 00800h
}

SECTIONS
{
.text : { } > LOW PAGE 0
.cinit : { } > LOW PAGE 0
.SWITCH : { } > LOW PAGE 0

.const : { } > SARAM PAGE 1
.data : { } > SARAM PAGE 1
.bss : { } > SARAM PAGE 1
.stack : { } > SARAM PAGE 1
.sysmem : { } > SARAM PAGE 1
}
```

- 结束：谢谢！