应用报告
ZHCA550L – 2013 年 1 月

从 TMS320F28xxx 数字信号处理器（DSP）上的内部闪存存储器上运行一个应用

David M.Alter 嵌入式处理器和微控制器 – 半导体产品组

摘要

在 TMS320F28xxx 上的片载闪存存储器上运行一个应用有特殊要求。 这些要求不会在代码开发期间自我显示，这是因为 Code Composer Studio™ (CCS) 调试程序能够屏蔽与初始化部分和它们在存储器中的关联部分相关的问题。 这份应用报告涵盖了从片载闪存存储器执行所需的应用软件修改。 介绍了 DSP/BIOS™ 和非 DSP/BIOS 项目的要求。 同时还讨论了某些性能注意事项和技巧。

示例 CCS v5 项目被提供用于 F2812，F2808，F28335，F28027，F28055 和 F28069（即，每个 F28xxx 子系列的扩展集器件）。 这些示例可从 http://www-s.ti.com/sc/techlit/spra958.zip 内下载，并且可提供一个与本应用报告无关的代码开发起始点。

请注意，本应用报告所讨论的问题适用于这些 TMS320F28xxx 器件系列的当前产品成员，特别是：

F281x： F2810，F2811，F2812
F280x/2801x/28044： F2801，F2802，F2806，F2808，F2809，F28015，F28016，F28044
F2823x/2833x： F28232，F28234，F28235，F28332，F28334，F28335
F2802x：F28020，F28021，F28022，F28023，F28026，F28027，F280200
F2803x：F28030，F28031，F28032，F28033，F28034，F28035
F2805x：F28050，F28051，F28052，F28053，F28054，F28055
F2806x：F28062，F28063，F28064，F28065，F28066，F28067，F28068，F28069

虽然相类似，但不保证对其它 F28xxx 器件的适用性。 此外，给出了应用于所采用的开发工具版本的代码和方法，特别是：

CCS v5.3.0，代码生成工具 v6.1.1，DSP/BIOS v5.42.0.07

请注意，虽然将保持全部可能的向后兼容性，这样，这里讨论的技术应该仍可工作，但是未来工具版本也许会有所不同 还请注意的是，针对 SYS/BIOS (BIOS v6) 的操作和设置不同与 DSP/BIOS (BIOS v5) 的操作和设置。 这份应用报告只适用于 DSP/BIOS。

最后，这份应用报告不提供编写和构建用于 F28xxx 的教程。 假定读者已经至少拥有从 RAM 运行他们的应用的主框架。 这份报告只识别将应用程序移动至片载闪存存储器上时所必须考虑的特定项。

内容

1
1 简介 4
2 创建一个用户连接程序命令文件 4
2.1 非 DSP/BIOS 项目 4
2.2 DSP/BIOS 项目 5
3 在何处连接段 6
3.1 非 DSP/BIOS 项目 7
3.2 DSP/BIOS 项目 8
4 将段从闪存复制到 RAM。 10
4.1 复制中断矢量（只适用于非 DSP/BIOS 项目） 10
4.2 复制 .hwi_vec 段（只适用于 DSP/BIOS 项目） 11
4.3 复制 .trcdata 段（只适用于 DSP/BIOS 项目） 12
4.4 初始化闪存控制寄存器（DCP/BIOS 和非 DSP/BIOS 项目） 14
4.5 通过执行 RAM 中的时间关键函数来大大提升性能。 17
4.6 通过将关键全局常量连接到 RAM 来大大提升性能 17
4.6.1 方法 1： 从 RAM 运行全部常量。 18
4.6.2 方法 2： 从 RAM 中运行一个特定常量数组。 21
5 设定代码安全模块密码 22
5.1 单区域安全器件（DSP/BIOS 和非 DSP/BIOS 项目） 23
5.2 双区域安全器件（DSP/BIOS 和非 DSP/BIOS 项目） 26
6 DSP 复位后，从闪存执行您的代码 34
7 C 语言环境引导期间禁用安全装置定时器 36
8 C 语言代码示例 39
8.1 概述 39
8.2 目录结构 41
8.3 其他信息 42
参考书目 46
修订历史记录 47

图表

Figure 1. 在 DSP/BIOS 配置工具中指定用户初始化函数 13
Figure 2. 在 Code Composer Studio v5 中指定连接顺序。 20
Figure 3. 针对 CSM 密码位置的 DSP/BIOS MEM 属性 25
Figure 4. 针对 CSM 保留位置的 DSP/BIOS MEM 属性 26
Figure 5. DCSM 区域选择块 27
Figure 6. DCSM 安全区域配置表 OTP 内存 28
Figure 7. 针对 DCSM_OTP_Z2_P0 内存的 DSP/BIOS MEM 属性 32
Figure 8. 针对 DCSM_ZSEL_Z2 内存的 DSP/BIOS MEM 属性 32
Figure 9. 针对 DCSM_OTP_Z1_P0 内存的 DSP/BIOS MEM 属性 33
Figure 10. 针对 DCSM_ZSEL_Z1_P0 内存的 DSP/BIOS MEM 属性 33

Figure 11. 针对跳转至闪存进入点的 DSP/BIOS MEM 属性 36

表格

Table 1. 针对非 DSP/BIOS 项目的段连接（较大内存模型）..... 7
Table 2. 针对 DSP/BIOS 项目的段连接（较大内存模型）..... 8
Table 3. CCS 示例代码目录说明 41

1 简介

TMS320F28xxx DSP 系列已经被设计用于嵌入式控制器应用的独立运行。 片载闪存通常不需要外部非易失性存储器以及一个引导加载主机处理器。 如果用户遵照几个简单的步骤进行操作时，将一个应用配置成从闪存存储器运行是相对简单的。 这份报告涵盖了将应用软件配置成从从内部闪存存储器中执行的适当配置。 提供了对 DSP/BIOS 和非 DSP/BIOS 项目的要求。 同时还介绍某些性能注意事项和技巧。

提供了针对 F2812，F2808，F28335，F28027，F28035，F28055 和 F28069 的示例 CCS v5 项目（即，通常是每个 F28xxx 子系列的扩展集器件）。 这些示例可从 http://www-s.ti.com/sc/techlit/spra958.zip 内下载，并且可提供一个独立于这份应用报告的代码开发起始点。

请注意，本应用报告内讨论的问题专门适用于 TMS320F28xxx 器件系列的这些当前产品成员：

请注意，本应用报告内讨论的问题专门适用于 TMS320F28xxx 器件系列的这些当前产品成员：
F2810，F2811，F2812
F280x/2801x/28044： F2801，F2802，F2806，F2808，F2809，F28015，F28016，F28044
F2823x/2833x： F28232，F28234，F28235，F28332，F28334，F28335
F2802x：F28020，F28021，F28022，F28023，F28026，F28027，F280200
F2803x：F28030，F28031，F28032，F28033，F28034，F28035
F2805x：F28050，F28051，F28052，F28053，F28054，F28055
F2806x：F28062，F28063，F28064，F28065，F28066，F28067，F28068，F28069

虽然可能，但是不保证对于其它 F28xxx 器件的适用性。 此外，提供的代码和方法专门适用于采用的开发工具版本。

CCS v5.3.0，代码生成工具，6.1.1，DSP/BIOS v5.42.0.07

请注意，虽然将保持全部可能的向后兼容性，这样，这里讨论的技术应该仍可工作，但是未来工具版本也许会有所不同。 还请注意的是，针对 SYS/BIOS (BIOS v6) 的操作和设置不同与 DSP/BIOS (BIOS v5) 的操作和设置。 这份应用报告只适用于 DSP/BIOS。

最后，这份应用报告不提供为 F28xxx 编写和构建代码的教程。

假定读者已经至少拥有从 RAM 运行他们的应用的主框架。 这份报告只识别将应用程序移动至片载闪存存储器上时所必须考虑的特定项。

2 创建一个用户连接程序命令文件

2.1 非 DSP/BIOS 项目

在非 DSP/BIOS 应用，用户连接程序命令文件将位于大多数内存被定义的位置，以及大多数程序段的连接被指定的位置。 为了使您能够从 RAM 运行您的应用，这个文件的格式与您当前使用的连接程序命令文件的格式一样。 唯一不同的是您连接程序段的位置（将在部分 3 中进行讨论）。与连接程序命令文件相关的更多信息可在参考 [11] 中找到。 此外，与这份应用报告一同提供的非 DSP/BIOS 代码项目包含可被用作参考的连接程序命令文件。

外设头文件（请见参考 [25-32]）包含以下名称的命令文件

DSP281x_Headers_nonBIOS.cmd	F2802x_Headers_nonBIOS.cmd
DSP280x_Headers_nonBIOS.cmd	DSP2803x_Headers_nonBIOS.cmd
DSP2804x_Headers_nonBIOS.cmd	F2805x_Headers_nonBIOS.cmd
DSP2833x_Headers_nonBIOS.cmd	F2806x_Headers_nonBIOS.cmd

这些文件包含连接外设寄存器结构所需的连接程序内存和程序段声明。 由于 CCS 在一个项目中支持多于一个的连接程序命令文件，所以除了您的用户连接程序命令文件外，用户需要做的是其中一个适当的连接程序命令文件添加到您的用户代码项目中。

总的来说，连接器命令文件的顺序无关紧要，这是因为在项目构建期间，CCS 在评估任一连接器命令文件的程序段部分前，评估每个连接程序命令文件的内存段部分。 这样确保了所有内存在将任一程序段连接至内存前被定义。 然而，高级用户也许需要在某些不常出现的情况下手工控制连接器命令文件的顺序。 这可以在 Project→Properties（项目属性）菜单上的 CCS v5 内，选择 CCS Build category（构建目录）的 Link Order 连接顺序标签页内指定。

2.2 DSP/BIOS 项目

DSP/BIOS 配置工具生成一个连接器命令文件，此文件规定如何连接所有 DSP/BIOS 生成的程序段，并且在缺省情况下，连接至所有 C 语言编译程序生成的程序段。 当从 RAM 运行您的应用时，这个连接器命令文件也许是唯一一个在用文件。 然而，当从闪存存储器中执行时，很有可能需要生成且连接一个或多个用户定义的程序段。 特别是，任何配置片载闪存控制寄存器（例如，闪存等待状态）的代码不能从闪存执行。 此外，用户也许需要从 RAM（而非闪存）中运行特定时间关键函数来大幅提升性能。 必须创建一个用户连接器命令文件来处理这些用户定义的程序段。

除了用户和 DSP/BIOS 生成的连接器命令文件，外设头文件（请见参考 [25 - 32]）包含以下名称的连接器命令文件

DSP281x_Headers_BIOS.cmd	F2802x_Headers_BIOS.cmd
DSP280x_Headers_BIOS.cmd	DSP2803x_Headers_BIOS.cmd
DSP2804x_Headers_BIOS.cmd	F2805x_Headers_BIOS.cmd
DSP2833x_Headers_BIOS.cmd	F2806x_Headers_BIOS.cmd

这些部分包含用于连接外设寄存器结构的连接器内存和数据段声明。 由于 CCS 支持一个项目中有多于一个连接器命令文件，所以用户需要做的是将全部三个连接器命令文件添加到它们的项目中。

总的来说，连接程序命令文件的顺序无关紧要，这是因为在项目构建期间，CCS 在评估任一连接器命令文件的数据段部分前，评估每个连接器命令文件的内存段部分。 这样确保了所有内存在将任一数据段连接至内存前被定义。 然而，高级用户也许需要在某些不常出现的情况下（例如，优先占用和使 一个数据段的 DSP/BIOS 连接无效）手工控制连接器命令文件评估的顺序。 这可以在 Project →Properties（项目属性）菜单上的 CCS v5 内，选择 CCS Build category（构建目录）的 Link Order 连接顺序标签页内指定。

3 在何处连接段

存在有两个基本的段类型：已初始化和未初始化。 已初始化段必须在器件加电时包含有效数据。例如，可在已初始化段中发现代码和常量。 当设计一个具有 F28xxx DSP（例如，没有在用的仿真器或调试程序，没有执行引导加载的主机处理器）的独立嵌入式系统时，所有已初始化段必须被连接至非易失性内存（例如，片载闪存）。 一个未经初始化的段在器件加点时不包含有效值。例如，在未经初始化的段中会发现变量。 代码执行时，代码将把值写入到变量位置。 因此，未经初始化的段必须被连接至易失性内存（例如，RAM）。

建议调用 -w 连接器选项（对于所有新建的 CCS 项目，此选项为缺省选中）。 -w 选项将在连接器发现您项目中的任一段还没有针对连接一个连接器命令文件而被明确指定时产生一个警告。当连接器遇到未指定的段时，它使用一个缺省的分配算法来将段连接至内存（它将把段连接至具有走狗可用空闲空间的第一个被定义的内存）。 这几乎始终存在风险，并且会导致不可靠且无法预计的代码运行。 -w 选项将识别任一未指定的段（例如，那些被用户偶尔忘记的段），这样，用户可以在需要时添加合适的连接器命令文件。 在 CCS v5 中，-w 选项复选框可在 Project → Properties 菜单，然后选择 Build →C2000 Linker →Diagnostic Category（构建 C2000 连接器诊断目录）找到。

注意：

C 语言编译程序使用较大内存模型十分重要（相对于较小内存模型）。 小型内存模型要求特定的已初始化段被连接至下层 64Kw 可寻址空间内的非易失性内存。 然而，任何 F28xxx 器件上的这个区域内没有闪存存储器，并且未来的 F28xxx 器件也可能是这样。 因此，应该使用较大内存模型。 对于 CCS v5 项目来说，较大内存模型复选框可在 Project→ Properties 菜单，然后选择 Build→ C2000 Compiler →Basic Options（构建 C2000 编译程序基本选项）目录中找到。 对于所有新创建的 CCS 项目，它被缺省选中。

对于非 DSP/BIOS 项目，用户应该将较大内存模型 C 语言编译程序运行支持库包含在他们的代码项目中。 对于定点器件，为库 *rts2800_ml.lib*（相对于用于较小内存模型的 *rts2800.lib*）。. 对于浮点器件，用于标准 C 语言代码的为 *rts2800_fpu32.lib*，或者用于 C++ 代码的 rts2800_fpu32_eh.lib（没有针对浮点器件的较小内存模型库）。在 CCS v5 中，有一个针对库的“自动”设置，此设置可在根据您的项目设置（例如，浮点支持和内存模型选择），让 CCS 为您选择正确的库时使用。

对于 DSP/BIOS 项目，DSP/BIOS 将负责将所需的库包括在内。 用户不应在一个 DSP/BIOS 项目中包含任何运行支持库。

3.1 非 DSP/BIOS 项目

此编译程序使用一定数量的专用段。 不论您是从 RAM 运行还是从闪存运行，这些段都是一样的。然而，当从闪存运行一个程序时，所有已初始化段必须被连接至非易失性内存，而所有未经初始化段必须被连接至易失性内存。 Table 1 显示了 F28xxx DSP 上每个编译程序生成的段的连接位置。 与每个段的功能有关的信息可在参考 [12] 中找到。 任何用户创建的经初始化段应该被连接至闪存（例如，那些被创建使用 CODE_SECTION 编译程序 pragma 指令的段），而任何用户创建的未经初始化的段应该被连接至 RAM（例如，那些被创建使用 DATA_SECTION 编译程序 pragma 指令的段）。

Table 1. 针对非 DSP/BIOS 项目的段连接（较大内存模型）

段名称	连接位置
.cinit	Flash
.cio	RAM
.const	Flash
.econst	Flash
.pinit	Flash
.switch	Flash
.text	Flash
.bss	RAM
.ebss	RAM
.stack	下层 64Kw RAM
.sysmem	RAM
.esysmem	RAM
.reset	RAM

Table 1 注释：

[1] *.reset* 段包含指向运行支持库（-c_int00 例程）中的 C 语言编译程序引导函数的 32 位以下的中断矢量。 它几乎从未使用。 此外，用户通常创建他们自己的分支指令来指向代码的起始点（请见部分 6 和 0）。 当未被使用时，通过在连接器命令文件中使用一个 DSET 修饰符，*.reset* 段应该被代码构建所省略。 例如：

```
/*****************************************************************
* 用户连接器命令文件
*****************************************************************/

SECTIONS
{
      .reset       : > FLASH,     PAGE = 0, TYPE = DSECT
}
```

3.2 DSP/BIOS 项目

DSP/BIOS 配置工具中的内存段管理器使得用户能够指定在何处连接不同的 DSP/BIOS 和 C 语言编译程序生成的段。 Table 2 表示内存段管理器上的每个标签页上显示的段应该被连接至何处（也就是说，是 RAM 还是闪存）。 请注意，这个信息已经针对这份报告中使用的 DSP/BIOS 版本而被专门地表格化了（请见部分 1）。 DSP/BIOS 的晚些版本，虽然与很相似，但还是有一些差异。 读者应该检查他们正在使用的版本，并在进行下一步操作前只需注意可能存在的不同。在 CCS v5 中，DSP/BIOS 版本被接至每个单独的项目。 前往 Project→Properties 菜单，然后选择 General（普通）目录。

Table 2. 针对 DSP/BIOS 项目的段连接（较大内存模型）

内存段管理器 TAB	段名称	连接位置
总体说明	用于 DSP/BIOS 项目的段	RAM
	针对 malloc()/free() 的段	RAM
BIOS 数据	自变量缓冲器选择 (.args)	RAM
	堆栈段 (.stack)	下层 64Kw RAM
	DSP/BIOS 初始值表 (.gblinit)	Flash
	TRC 初始值 (.trcdata)	RAM[1]
	DSP/BIOS 内核状态 (.sysdata)	RAM
	DSP/BIOS 配置段 (*.obj)	RAM
BIOS 代码	BIOS 代码段 (.bios)	Flash
	启动代码段 (.sysinit)	Flash
	函数插桩内存 (.hwi)	Flash
	中断处理表内存 (.hwi_vec)	PIEVECT RAM[2]
	RTDX 文本段 (.rtdx_text)	Flash

编译程序段	文本段 (.text)	Flash
	开关语句跳转表 (.switch)	Flash
	C 语言变量段 (.bss)	RAM
	C 变量段 (.ebss)	RAM
	数据初始化段 (.cinit)	Flash
	C 语言函数初始化表 (.pinit)	Flash
编译程序段（继续）	常数段 (.econst)	Flash
	常数段 (.const)	Flash
	数据段 (.data)	Flash
	数据段 (.cio)	RAM
载入地址	载入地址 - BIOS 代码选择 (.bios)	Flash[3]
	载入地址 - 启动代码选择 (.sysinit)	Flash[3]
	载入地址 - DSP/BIOS 初始值表 (.gblinit)	Flash[3]
	载入地址 - TRC 初始值表 (.trcdata)	Flash[1]
	载入地址 - 文本段 (.text)	Flash[3]
	载入地址 - 开关语句跳转表 (.switch)	Flash[3]
	载入地址 - 数据初始化段 (.cinit)	Flash[3]
	载入地址 - C 语言函数初始化表 (.pinit)	Flash[3]
	载入地址 - 常数段 (.econst)	Flash[3]
	载入地址 - 常数段 (.const)	Flash[3]
	载入地址 - 数据段 (.data)	Flash[3]
	载入地址 - 函数插桩内存 (.hwi)	Flash[3]
	载入地址 - 中断处理表内存 (.hwi_vec)	Flash[2]
	载入地址 - RTDX 文本段 (.rtdx_text)	Flash[3]

Table 2 注释：

[1] *.trcdata* 段必须在运行时由用户从它的载入地址（在 Load_Address 标签页上指定）复制到它的运行地址（在 BIOS_Data 标签页上指定）。 执行这个复制操作的更多细节请见 4.3。

[2] PIEVECT RAM 与外设中断扩展 (PIE) 外设相关的 RAM 的特定块。 在本报告所涉及的 F28xxx 器件上，PIE RAM 为数据空间内起始地址为 0x000D00 的 256x16 大小的块。 对于其它器件，请在器件数据表中确认起始地址。 DSP/BIOS 配置工具中的内存段管理器应该已经具有一个名为 PIEVECT 的预定义内存。 *.hwi_vec* 段必须在运行时由用户从它的载入地址（在内存段管理器 Load_Address 标签页上指定）复制到它的运行地址（在内存段管理器 BIOS_Code 标签页上指定）。 执行这个复制操作的细节请参见部分 4.2。

[3] 被选中为这个段载入地址的特定闪存存储器应该与之前被选中为段运行地址的闪存存储器一样（例如，在 BIOS 数据，BIOS 代码或编译程序段标签页上）。

4 将段从闪存复制到 RAM。

4.1 复制中断矢量（只适用于非 DSP/BIOS 项目）

外设中断扩展 (PIE) 模块管理 F28xxx 器件上的中断请求。 加电时，所有中断必须位于非易失性内存内（即闪存），但是被作为您代码中的器件初始化过程的一部分被复制到 PIEVECT RAM 中。PIEVECT RAM 是 F28xxx 器件上 RAM 的一个专用块，本报告中所涉及的为数据空间内起始地址为 0x000D00 的大小为 256x16 的块。

有几个方法可将中断矢量连接至闪存，然后可在运行时将它们复制到 PIEVECT RAM 中。 一个方法是创建一个包含全部 128 个 32 位矢量的恒定 C 语言结构函数指针。 如果使用外设头文件结构（请见参考 [25-32]，这样一个被称为 *PieVectTableInit* 的结构已经在相应的文件 *DSP28xxx_PieVect.c* 中被创建。 由于这个结构使用恒定类型限定符声明，它将被编译程序放置在 *.econst* 段中。 一个方法是创建一个包含全部 128 个 32 位矢量的恒定 C 语言结构函数指针。 C 语言编译程序运行支持库包含一个名为 *memcpy()* 内存复制函数，此函数可被用来执行下面的复制任务：

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

/*********************************************************************
* 注释： 这个函数假设使用外设头文件
* 结构（请见参考 [25 - 32]）。
*********************************************************************/

#include <string.h>

#include <string.h>
{
/*** 初始化 PIE_RAM ***/
      PieCtrlRegs.PIECTRL.bit.ENPIE = 0;  // Disable the PIE（禁用 PIE）
      asm(" EALLOW");                     // Enable EALLOW protected register access（启用 EALLOW 受保护
寄存器存取）
      memcpy((void *)0x000D00, &PieVectTableInit, 256);
      asm(" EDIS");                       // Disable EALLOW protected register access（禁用 EALLOW 受保
护寄存器存取）
}
```

请注意，由于 *memcpy()*复制 16 位字，所以复制长度为 256。

上面的示例使用了一个固化在硬件中的地址用于 PIE RAM 的起始地址，特定为 0x000D00。 如果这不太适用（由于固化在硬件中的地址不是良好的编程做法），用户可以使用 DATA_SECTIN pragma 指令来创建一个未经初始化的虚变量，并将此变量连接至 PIE RAM。 然后虚变量的地址可被用来代替固化在硬件中的地址。 例如，当使用任一 ‘28xxx 器件外设结构时，一个名为 *PieVectTable* 的未经初始化结构被创建并被连接在 PIEVECT RAM 上。 之前示例中的 *memcpy()* 函数可被替换为：

```
memcpy(&PieVectTable, &PieVectTableInit, 256);
```

最后，在某些器件上，特别是 ‘Piccolo’ 器件上 (F2802x，F2803x，F2805x，F2806x)，头三个 32 位 PIE 矢量位置在调试程序被使用时被用于引导模式选择。 因此，应该对代码进行修改以避免写覆盖下面这些位置：

```
memcpy((void *)0x000D06, (Uint16 *)&PieVectTableInit+6, 256-6);
```

或者

```
memcpy((Uint16 *)&PieVectTable+6, (Uint16 *)&PieVectTableInit+6, 256-6);
```

4.2 复制 .hwi_vec 段（只适用于 DSP/BIOS 项目）

DSP/BIOS.*hwi_vec* 段包含中断矢量，并且必须被载入到闪存中，但是从 RAM 中运行。 用户负责将这个段从其载入地址复制到它的运行地址。 这通常在 *main()*中完成。 DSP/BIOS 配置工具生成可由代码访问的全局符号，以确定载入地址、运行地址和.*hwi_vec* 段的长度。 这些符号名称为：

hwi_vec_loadstart　　hwi_vec_loadend
hwi_vec_loadsize　　hwi_vec_runstart

每个符号的名称应该可以表示此符号的用途。 请注意，符号不是指针，而是段的相应位置（即起始或末尾）上找到的符号表示的 16 位数据值。 C 语言编译程序运行支持库包含一个名为 *memcpy()*内存复制函数，此函数可被用来执行复制任务： 下面显示了一个如何使用这个函数来执行段复制的代码示例。

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

#include <string.h>

extern unsigned int hwi_vec_loadstart;
extern unsigned int hwi_vec_loadsize;
extern unsigned int hwi_vec_runstart;

void main(void)
{
/*** 初始化 .hwi_vec 段 ***/
      asm(" EALLOW");                    /* 启用 EALLOW 受保护寄存器访问 */
      memcpy(&hwi_vec_runstart, &hwi_vec_loadstart, (Uint32)&hwi_vec_loadsize);
      asm(" EDIS");                      /* 禁用 EALLOW 受保护寄存器访问 */
}
```

最后，在某些器件上，特别是 ‘Piccolo’ 器件上 (F2802x，F2803x，F2805x，F2806x)，头三个 32 位 PIE 矢量位置在调试程序被使用时被用于引导模式选择。 因此，应该对代码进行修改以避免写覆盖下面这些位置：

```
memcpy(&hwi_vec_runstart+6, &hwi_vec_loadstart+6,
                            (Uint32)(&hwi_vec_loadsize-(Uint16*)6));
```

4.3 复制 .trcdata 段（只适用于 DSP/BIOS 项目）

DSP/BIOS *.trcdata* 段必须被载入到闪存，但是从 RAM 运行。 用户负责将这个段从其载入地址复制到它的运行地址。 然而，与 *.hwi_vec* 段不同，*.trcdata* 的复制必须在 *main()*之前执行。 这是因为 DSP/BIOS 在 DSP/BIOS 初始化期间（也发生在 *main()*之前）修改了 *.trcdata* 的内容。

DSP/BIOS 配置工具提供了一个用户初始化函数，此函数可被用来在 main() 和 DSP/BIOS 初始化之前执行 *.trcdata* 段。 这可在系统 →Global Settings Properties（全局设置属性）下的 CCS v5 项目配置文件中找到，如 Figure 1 中所示。

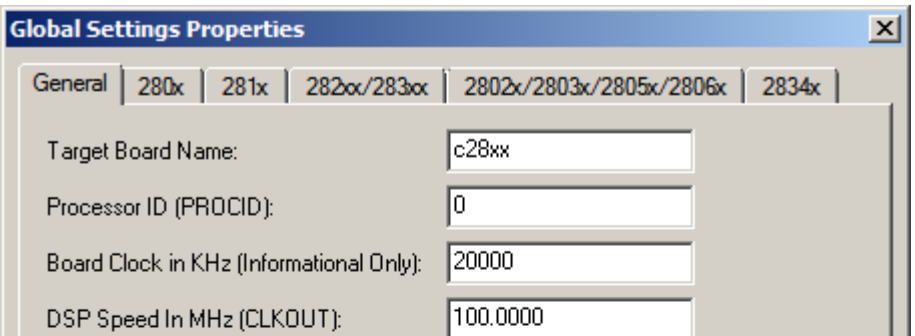

Figure 1. 在 DSP/BIOS 配置工具中指定用户初始化函数

剩下的就是创建用户初始化函数。 DSP/BIOS 配置工具生成可由代码访问的全局符号，以确定载入地址、运行地址和段的长度。 这些符号名为：

trcdata_loadstart　　trcdata_loadend
trcdata_loadsize　　trcdata_runstart

每个符号的名称应该可以表示此符号的用途。 请注意，符号不是指针，而是段的相应位置（即起始或末尾）上找到的用符号表示的 16 位数据值。 C 语言编译程序运行支持库包含一个名为 *memcpy()*内存复制函数，此函数可被用来执行复制任务。 一个执行下面.*trcdata* 段复制的用户初始化函数的 C 语言代码示例。

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

#include <string.h>

extern unsigned int trcdata_loadstart;
extern unsigned int trcdata_loadsize;
extern unsigned int trcdata_runstart;

void UserInit(void)
{
/*** 在 main() 之前初始化 .trcdata 段***/
      memcpy(&trcdata_runstart, &trcdata_loadstart, (Uint32)&trcdata_loadsize);
}
```

4.4 初始化闪存控制寄存器（DCP/BIOS 和非 DSP/BIOS 项目）

不能从闪存存储器中执行针对闪存控制寄存器的初始化代码，否则会发生不可预计的结果。 因此，闪存控制寄存器的初始化函数必须在运行时从闪存（它的载入地址）复制到 RAM（它的运行地址）。

注意：

根据器件的不同，闪存控制寄存器受到代码安全模块 (CSM) 或者双区域代码安全模块 (DCSM) 的保护。

在具有 CSM 的器件上，如果 CSM 受到保护，您必须从 CSM 安全 RAM 上运行闪存寄存器初始化代码，否则的话，初始化代码将不能访问闪存寄存器（请参见器件数据表的内存映射部分来识别安全内存）。 请注意，虽然 ROM 引导加载程序在您使用 0xFFFF 的假密码时会将其解锁，CSM 始终在器件复位时被锁定。

在具有 DCSM 的器件上，闪存控制寄存器的保护由 FLSM 寄存器的 SEM 字段控制。复位时，SEM 为缺省值，这样的话，闪存控制寄存器可被从任一内存运行的代码写入。通常情况下，软件将在改变 SEM 字段前配置闪存控制寄存器，因此，用户只需将配置代码复制到任一 RAM 并从那里执行。 然而，如果软件首先改变了 SEM 字段，闪存配置代码将需要从一个 RAM 块中执行，此 RAM 块具有到闪存配置寄存器的安全访问权限。这个访问由 SEM 字段值决定。 与 FLSM 寄存器的 SEM 字段相关的信息请参考 [18]。

C 语言编译程序的 CODE_SECTION pragma 指令可被用来创建一个用于闪存初始化函数的单独可连接段。 例如，假定闪存寄存器配置将由 C 语言函数 *InitFlash()*执行，并且需要将这个函数放置在一个名为 *secureRamFuncs* 的可连接段的内部。 以下 C 语言示例代码显示了 CODE_SECTION pragma 指令的适当使用，以及一个闪存寄存器的示例配置：

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

/*********************************************************************
* 注释： 这里显示的 InitFlash() 函数只是一个
* 闪存控制寄存器的初始化的示例。  查阅器件
* 数据表以获得生产等待状态和任何其他相关
* 的信息。  这里显示的等待状态特指目前的 F280x
* 器件，其运行频率为 100MHz。
* 注释： 这个函数假定使用外设头文件
* 结构（请参考 [25 - 32]）。
*********************************************************************/

#pragma CODE_SECTION(InitFlash, "secureRamFuncs")
void InitFlash(void)
{
      asm(" EALLOW");                          // Enable EALLOW protected register access（启用 EALLOW 受
保护寄存器访问）
      FlashRegs.FPWR.bit.PWR = 3;                     // Flash set to active mode（闪存被设定为激活模式）
      FlashRegs.FSTATUS.bit.V3STAT = 1;               // Clear the 3VSTAT bit（清零 3VSTAT 位）
      FlashRegs.FSTDBYWAIT.bit.STDBYWAIT = 0x01FF;    // Sleep to standby cycles（睡眠至待机周期）
      FlashRegs.FACTIVEWAIT.bit.ACTIVEWAIT = 0x01FF; // Standby to active cycles（睡眠至激活周期）
      FlashRegs.FBANKWAIT.bit.RANDWAIT = 3;           // F280x Random access wait states（F280x 随机访问
等待状态）
      FlashRegs.FBANKWAIT.bit.PAGEWAIT = 3;           // F280x Paged access wait states（F280x 页式访问等
待状态）
      FlashRegs.FOTPWAIT.bit.OTPWAIT = 5;             // F280x OTP wait states（F280x OTP 等待状态）
      FlashRegs.FOPT.bit.ENPIPE = 1;                  // Enable the flash pipeline（启用闪存管道模式）
      asm(" EDIS");                           // Disable EALLOW protected register access（禁用 EALLOW 受
保护寄存器访问）

/*** 强制一个完整的管道冲洗以确保在
     返回前写入最后一个被配置的寄存器。  最安全的做法是等待 8 个完整周期。  ***/

      asm(" RPT #6 || NOP");               // Takes 8 cycles to execute（用 8 个周期执行）

} //end of InitFlash() (InitFlash() 的末尾)
```

然后，段 secureRamFuncs 可使用连接器命令文件连接。 这个段将要求单独的载入和运行地址。此外，连接器需要生成某些全局符号，这些全局符号可被用来确定载入地址、运行地址和段的长度。 从段载入地址复制到它的运行地址需要这个信息。 用户连接器命令文件将显示如下：

```
/*************************************************************************
* 用户连接器命令文件
*************************************************************************/

SECTIONS
{
/*** 用户定义的） ***/
secureRamFuncs:    LOAD = FLASH,       PAGE = 0
                   RUN = SECURE_RAM,   PAGE = 0
                   LOAD_START(_secureRamFuncs_loadstart),
                   LOAD_SIZE(_secureRamFuncs_loadsize),
                   RUN_START(_secureRamFuncs_runstart)
}
```

在这个示例中，内存 FLASH（闪存） 和 SECURE_RAM 被假定已经在用户连接器命令文件的 MEMORY（内存）段（针对非 DSP/BIOS 项目）或者 DSP/BIOS 配置工具的内存段管理器（针对 DSP/BIOS 项目）内被定义。 针对这些内存的 PAGE 指定应该与内存定义的 PAGE 指定相匹配。上面这个示例假定两个内存已经在 PAGE 0（程序内存空间）内被声明。 LOAD_START，LOAD_SIZE 和 RUN_START 指令将生成全局符号，这些符号具有针对相应地址的指定名称。 请注意全局符号定义上的前置下划线的使用（例如，*_secureRamFuncs_runstart*）。

最后，必须在运行时将这个段从闪存复制到 RAM 中。 如部分 4.1 – 4.3 中显示的那样，可使用编译程序运行支持库内的函数 *memcpy()*。

```
/*************************************************************************
* 用户 C 语言源文件
*************************************************************************/

#include <string.h>

extern unsigned int secureRamFuncs_loadstart;
extern unsigned int secureRamFuncs_loadsize;
extern unsigned int secureRamFuncs_runstart;

void main(void)
{
/* 复制 secureRamFuncs 段 */
      memcpy(&secureRamFuncs_runstart,
             &secureRamFuncs_loadstart,
             (Uint32)&secureRamFuncs_loadsize);

/* 初始化片载闪存寄存器 */
      InitFlash();
}
```

4.5 通过执行 RAM 中的时间关键函数来大大提升性能。

（DSP/BIOS 和非 DSP/BIOS 项目）

本报告中所涉及的 F28xxx 器件的片载 RAM 内存提供 MIPS（每秒百万条指令）级的代码执行，这个执行速度与器件以 MHz 为单位的时钟频率相等（150MHz 时的 150 MIPS，100MHz 时的 100 MIPS 等）。 然而，根据器件频率和应用的不同，片载闪存存储提供略低的有效代码执行性能。本报告中所涵盖的针对器件的大致闪存执行算为[1]：

150MHz 时为 90 -95 MIPS
100MHz 时为 80-85 MIPS
80MHz 时为 65-70MHz
60MHz 时为 50-55 MIPS
40MHz 时为 37-39MHz

因此，也许用户想要从片载 RAM 上运行特定时间关键或对计算有严格要求的例程，特别是对于运行频率为 15MHz 的器件。 然而，在一个独立嵌入式系统中，所有代码在最初时必须驻留在非易失性内存上。 为了从 RAM 上运行这些函数，必须设置单独的载入和运行地址，并且必须在运行时执行一个复制操作，将它们从片载闪存移动到 RAM。 为了达到这一目的，请采用之前在 4.4 中描述的同样步骤。

通过使用 CODE_SECTION pragma 指令，用户可以将多个函数添加到同一个可连接段中。 然后，整个段被指定用来从一个特定的 RAM 块中运行，并且用户可以一次将整个段复制到 RAM 中，正如 4.4 中描述的那样。 如果需要更精细的连接间隔，可为每个函数创建一个单独的段名称。

4.6 通过将关键全局常量连接到 RAM 来大大提升性能

（DSP/BIOS 和非 DSP/BIOS 项目）

常量是使用 C 语言 *const* 类型修饰符声明的数据结构。 编译程序将所有常量放在.*econst* 段中（假定采用的是大内存模型）。 虽然本报告中涉及的 F28xxx 器件上的特殊管线装置加速了代码执行时的有效闪存性能，但是这样的管线装置不存在于对片载闪存内数据常量的存取。 每个闪存数据存取会花费多个周期。 通常情况下，闪存等待状态在 150MHz 器件上为 5 个周期，在 100MHz 或 90MHz 器件上为 3 个周期，在 80 或 60 MHz 器件上为 2 个周期，在小于等于 50MHz 器件上为 1 个周期[1]。

注意事项：
不同 F28xxx 器件间的闪存时序会有所不同。 检查在用器件的专用数据表很重要。

因此，有必要将经常访问的常量和常量表保存在片载 RAM 中。 然而，一个独立嵌入式系统所有经初始化的数据（例如，常量）在最初时驻留在非易失性内存中。 因此，对于那些您希望在 RAM 中访问的常量，必须设置单独的载入和运行地址，并且必须执行复制操作以便在运行时将它们从片载闪存复制到 RAM 中。 将介绍两个不同的实现方法。

[1] 这些估算值只适用于本报告所涉及的器件（请见部分 1），这些器件采用同样的 180nm CMOS 制造工艺。 最新的 F28xxx 器件（例如，F28M35x，F28M36x）采用 65nm 的制造工艺，并且同样具有一个宽范围的闪存预取缓冲器。它们比 180nm 器件具有高出很多的有效闪存执行性能。

4.6.1 方法 1： 从 RAM 运行全部常量。

这个方法针对整个.*econst* 段调用指定的单独载入和运行地址。 这个方法的优势是易于使用，而缺点是过多的 RAM 用量（也许只有少数几个常量要求高速访问，但是在使用这个方法时，所有常量将被移入 RAM 中）。

4.6.1.1 非 DSP/BIOS 项目

可使用部分 4.4 中讨论的同样的方法。 仅需在用户连接器命令文件中指定针对.*econst* 段的单独载入和运行地址，然后将代码添加到您的项目中来在运行时将整个.*econst* 段复制到 RAM 中。例如：

```
/*************************************************************************
* 用户连接器命令文件
*************************************************************************/

SECTIONS
{
.econst:            LOAD = FLASH,   PAGE = 0
                    RUN = RAM,      PAGE = 1
                    LOAD_START(_econst_loadstart),
                    LOAD_SIZE(_econst_loadsize),
                    RUN_START(_econst_runstart)
}
```

```
/*************************************************************************
* 用户 C 语言源文件
*************************************************************************/

#include <string.h>

extern unsigned int econst_loadstart;
extern unsigned int econst_loadsize;
extern unsigned int econst_runstart;

void main(void)
{
/* 复制 .econst 段 */
      memcpy(&econst_runstart, &econst_loadstart, (Uint32)&econst_loadsize);
}
```

4.6.1.2 DSP/BIOS 项目

虽然 DSP/BIOS 配置工具可针对.*econst* 段指定不同的载入和运行地址，它将不会生成执行内存复制操作所需的代码可访问标签。 因此，用户必须在评估 DSP/BIOS 生成的连接器命令文件前，优先连接用户连接器命令文件内的.*econst* 段。 用户连接器命令文件将显示如下：

```
/*********************************************************************
* 用户连接器命令文件（DSP/BIOS 项目）
*********************************************************************/

SECTIONS
{
/*** 优先连接 .econst 段 ***/
/* 必须在 DSP/BIOS 连接器命令文件被评估前执行 */

.econst:            LOAD = FLASH,  PAGE = 0
                    RUN = RAM,     PAGE = 1
                    LOAD_START(_econst_loadstart),
                    LOAD_SIZE(_econst_loadsize),
                    RUN_START(_econst_runstart)
}
```

为了在项目构建期间保证用户连接器命令文件在 DSP/BIOS 生成连接器命令文件前被评估，用户必须在 CCS 中指定连接顺序。 要在 CCS v5 中进行此操作，前往 Project→Properties，然后选择 Build 目录，连接顺序标签页。 然后，您可以通过单击“Add…”按钮来为有疑问的连接器命令文件指定合适的顺序。 请注意，DSP/BIOS 生成的连接器命令文件将被在文件选择列表中明确列出。 相反地，您应该选择 “$(GEN_CMDS_QUOTED)”，意思是 DSP/BIOS 生成的 .cmd 文件。 Figure 2 显示了此操作的一个示例，在这里，*F2808_BIOS_flash.cmd* 是用户连接器命令文件，而 “$(GEN_CMDS_QUOTED)” 是指 DSP/BIOS 生成的 *F2808_example_BIOS_flashcfg.cmd* 文件。

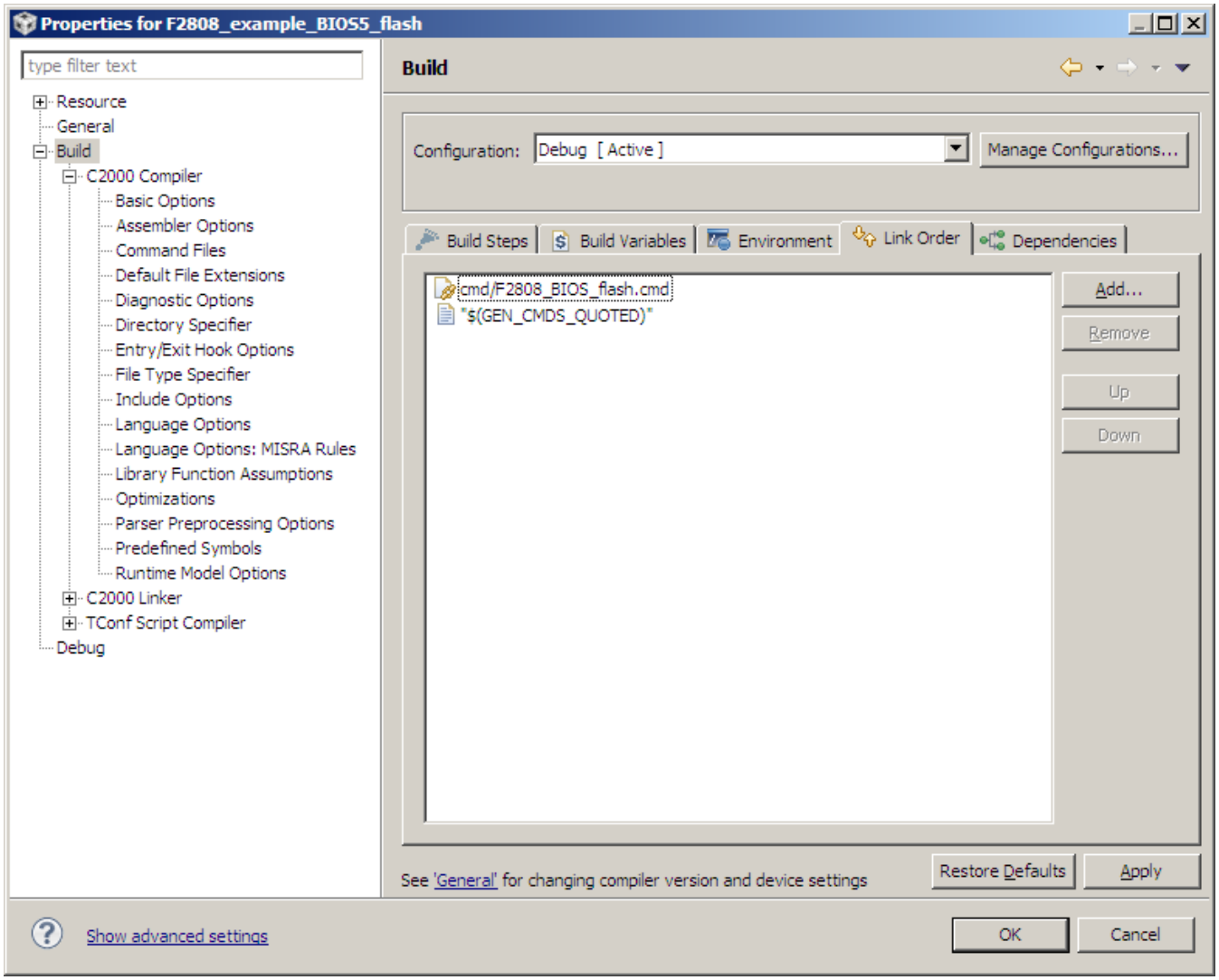

Figure 2. 在 Code Composer Studio v5 中指定连接顺序。

然后，如下所示，可将.*econst* 段从其载入地址复制到其运行地址：

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

#include <string.h>

extern unsigned int econst_loadstart;
extern unsigned int econst_loadsize;
extern unsigned int econst_loadsize;

void main(void)
{
/* 复制 .econst 段 */
     memcpy(&econst_runstart, &econst_loadstart, (Uint32)&econst_loadsize);
}
```

4.6.2 方法 2： 从 RAM 中运行一个特定常量数组。

（DSP/BIOS 和非 DSP/BIOS 项目）

这个方法涉及在运行时有选择性的将常量从闪存复制到 RAM 中。 完成这一操作的步骤与方法 1 类似，除非只将选中的常量放置在一个已命名的段中，并被复制到 RAM 中（而不是将所有常量复制到 RAM 中。）

对于此示例，假设用户希望创建一个运行自 RAM 的名为 *table[ ]*的 5 字常量数组。 一个 DATA_SECTION pragma 指令被用来将 *table[ ]*仿真在一个名为 *ramconsts* 的用户定义段中。 C 语言源文件将显示如下：

```
/*********************************************************************
* 用户的 C 语言源文件
*********************************************************************/

#pragma DATA_SECTION(table, "ramconsts")
const int table[5] = {1,2,3,4,5};

void main(void)
{

}
```

为了使用用户连接器命令文件载入到闪存，但是运行自使用用户连接器命令文件，要连接 *ramconsts* 段，并且生成全局符号来便利内才复制。 用户连接器命令文件将显示如下：

```
/*********************************************************************
* 用户连接器命令文件
*********************************************************************/

SECTIONS
{
/*** 用户定义的段 ***/
ramconsts:          LOAD = FLASH, PAGE = 0
                    RUN = RAM,    PAGE = 1
                    LOAD_START(_ramconsts_loadstart),
                    LOAD_SIZE(_ramconsts_loadsize),
                    RUN_START(_ramconsts_runstart)
}
```

最后，*table[ ]*必须在运行时从其载入地址复制到其运行地址：

```
/*********************************************************************
* 用户 C 语言源文件
*********************************************************************/

#include <string.h>

extern unsigned int ramconsts_loadstart;
extern unsigned int ramconsts_loadsize;
extern unsigned int ramconsts_runstart;

void main(void)
{
/* 初始化 ramconsts 段 */
      memcpy(&ramconsts_runstart, &ramconsts_loadstart, (Uint32)&ramconsts_loadsize);
}
```

5 设定代码安全模块密码

F28xxx 器件上的代码安全模块防止您的软件有害复制以及对您软件的篡改。 本报告所涉及的器件具有一个单区代码安全模块 (CSM)，或有一个双区域代码安全模块 (DCSM)。 CSM 使用一个单个 128 位密码，此密码限制对所有安全内存的访问。 DCSM 提供两个 128 位密码，每个密码限制对两个安全区域中其中一个的访问。 某些安全内存资源被固定在它们分配的两个区域中的一个，而其它安全内存资源由软件分配至任一区域。

单区域代码安全模块器件：

F281x： F2810，F2811，F2812 **F280x/2801x/28044：** F2801，F2802，F2806，F2808，F2809，F28015，F28016，F28044 **F2823x/2833x：** F28232，F28234，F28235，F28332，F28334，F28335 **F2802x：**F28020，F28021，F28022，F28023，F28026，F28027，F280200 **F2803x：**F28030，F28031，F28032，F28033，F28034，F28035 **F2806x：**F28062，F28063，F28064，F28065，F28066，F28067，F28068，F28069

双区域代码安全模块器件：

F2805x： F28050，F28051，F28052，F28053，F28054，F28055

下面的子部分将解释如何针对单区域和双区域安全器件，将代码安全密码合并到您的代码项目中。详细解释安全模块的操作已超出本报告的范围。 对于这些信息，读者请参考 [13 - 19]。

5.1 单区域安全器件（DSP/BIOS 和非 DSP/BIOS 项目）

在单区域器件上，CSM 保护整个闪存，OTP 内存，以及某些 ‘L’ SARAM 块（器件专用信息请见器件数据表）。 闪存配置寄存器也同样受到保护。 当被锁定时，只有从安全内存中执行的代码才能访问其它安全内存中的数据（读或写）。 从未受保护的内存中执行的代码可分支至（或调用）安全内存中的代码，但是不能访问安全内存的任何数据。

CSM 使用一个由 8 个独立的 16 位字组成的 128 位密码。 对于本报告所涉及的 CSM 器件，这些密码被存储在闪存内的最上部 8 个字内（例如，F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件上的 0x3F7FFF 直到 0x3F7FF8，F2823x 和 F2833x 器件上的 0x33FFF8 直到 0x33FFFF）。 开发期间，建议使用 0xFFFF 的假密码。 当使用假密码时，只需虚读取密码位置即可解锁 CSM。 可轻松将虚密码放置在密码位置中，这是因为 0xFFFF 将是闪存在闪存编程期间被擦除后，这些位置的状态。 用户只需避免将任一段连接至代码项目内的密码地址，而密码将保留在 0xFFFF 内。

开发后，用户也许想使用真实的密码。 此外，为了正确锁定本报告中所涉及的器件的 CSM 模块，值 0x0000 必须被写入 118 个闪存地址，这些地址从 CSM 密码起始地址前的 120 个字的位置开始，例如，F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件上的 0x3F7F80 直到 0x3F7FF5，以及 F2823x 和 F2833x 器件上的 0x33FF80 直到 0x33FFF5（请见参考 [1 -6 和 8]）。 完成这两项任务的简便方法是使用少量简单汇编语言编程。 以下的示例汇编代码文件指定所需的代码值并将它们放置在一个名为 *passwords* 已初始化段中。 它还创建一个名为 *csm_rsvd* 的已初始化段，它包含 0x0000 值，并且具有适合于上面提到的 118 字地址寄存器的合适长度。 与使用的汇编语言指令相关的更多信息请参考 [11]。

```
;******************************************************************************
; 文件：  Passwords.asm
;******************************************************************************

;******************************************************************************
; 显示了 0xFFFF 的虚密码。   用户可将这些虚密码更改为
; 需要的值。
;
; 注意事项：  不要将 0x0000 用于所有 8 个密码，否则 CSM 模块会
; 被永久锁定。   请参考 [13-17 和 19] 以获得更多
; 信息。
;******************************************************************************
     .sect "passwords"
     .int   0xFFFF                      ;PWL0 (128 位密码的 LSW)
     .int   0xFFFF                      ;PWL1
     .int   0xFFFF                      ;PWL2
     .int   0xFFFF                      ;PWL3
     .int   0xFFFF                      ;PWL4
     .int   0xFFFF                      ;PWL5
     .int   0xFFFF                      ;PWL6
     .int   0xFFFF                      ;PWL7 (128 位密码的 MSW)
;------------------------------------------------------------------------------
     .sect "csm_rsvd"
     .loop (3F7FF5h - 3F7F80h + 1)
     .int   0x0000
     .endloop
;------------------------------------------------------------------------------

     .end                               ; 文件 passwords.asm 的末尾
```

请注意这个示例显示了 0xFFFF 的虚地址值。 用您需要的密码替代这些值。

注意事项：

不要将 0x0000 用于所有 8 个密码。 这样做的话会永久锁定 CSM 模块！ 更多信息请参考 [13-17 和 19]。

passwords 和 *csm_rsvd* 段应该被放置在具有用户连接器命令文件的内存中。

对于非 DSP/BIOS 项目，用户应该在用户连接器命令文件的 MEMORY 部分的 PAGE 0 内定义名为（例如）*PASSWORDS* 和 *CSM_RSVD*。 然后段 *passwords* 和 *csm_rsvd* 可被连接至这些内存。 下面的示例应用于 F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件。 对于其它器件，查阅器件数据表来确认密码和 CSM 保留位置的地址。

```
/*************************************************************************
* 用户连接器命令文件（非 DSP/BIOS 项目）
*************************************************************************/

内存
{
PAGE0:    /* 程序存储器 */
   CSM_RSVD     : origin = 0x3F7F80, length = 0x000076
   PASSWORDS    : origin = 0x3F7FF8, length = 0x000008
}


SECTIONS
{
/*** 代码安全密码位置 ***/
   passwords:      > PASSWORDS,     PAGE = 0
   csm_rsvd:       > CSM_RSVD,      PAGE = 0
}
```

对于 DSP/BIOS 项目，用户应该使用 DSP/BIOS 配置工具的内存段管理器来定义名为（例如）*PASSWORDS* 和 *CSM_RSVD* 内存。 下面的两个图表显示了针对这些 F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件上的内存的 DSP/BIOS 内存段管理器属性。 始终查阅器件数据表以确认正确的地址和长度，或者针对其它 F28xx 器件的内存映射信息。

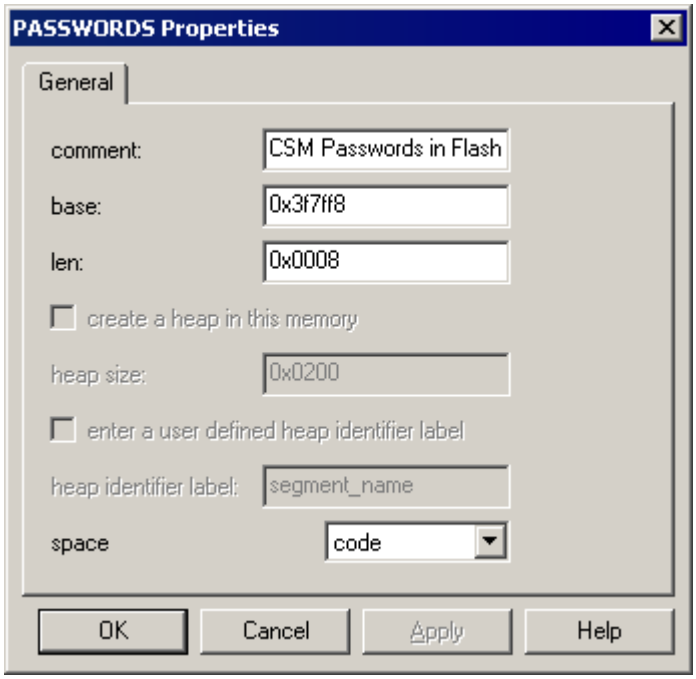

Figure 3. 针对 CSM 密码位置的 DSP/BIOS MEM 属性

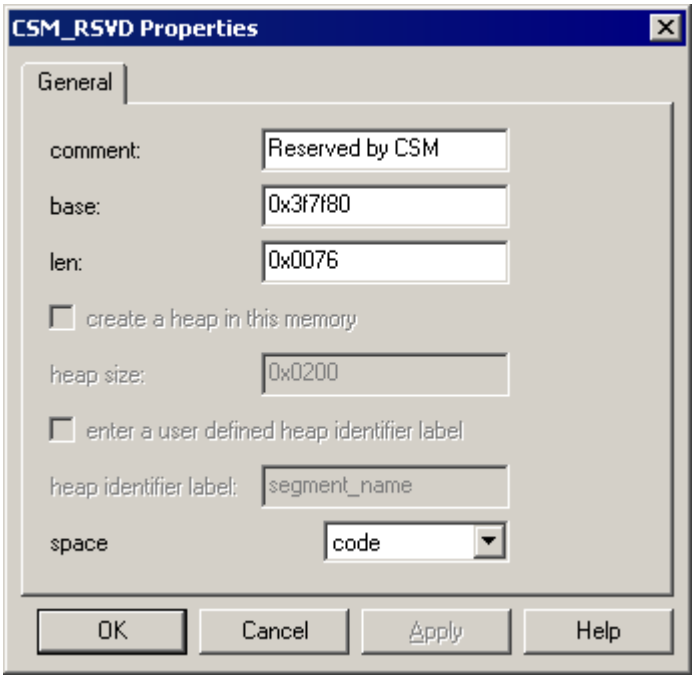

Figure 4. 针对 CSM 保留位置的 DSP/BIOS MEM 属性

然后可在用户连接器命令文件中将段 *passwords* 和 *csm_rsvd* 连接至这些内存。 对于 DSP/BIOS 项目，用户连接器命令文件将显示为：

```
/*************************************************************************
* 用户连接器命令文件（DSP/BIOS 项目）
*************************************************************************/

SECTIONS
{
/*** 代码安全密码位置 ***/
   passwords:     > PASSWORDS,     PAGE = 0
   csm_rsvd:      > CSM_RSVD,      PAGE = 0
}
```

5.2 双区域安全器件（DSP/BIOS 和非 DSP/BIOS 项目）

除了三个主要差异外，双区域安全操作与之前部分中描述的单区域安全的操作相似：

1. 从一个受保护的区域执行的代码只能访问位于那个区域，或者位于未受保护内存中的数据。一个受保护区域内的代码不能访问位于其它受保护的区域，虽然此代码可分支或调用驻留在任何内存中的代码。 或者，一个内存块可被配置成只执行，这样，没有代码能够访问那个内存中的数据（其中包括从同一内存中执行的代码）。

2. 每个区域使用 OTP 内存中的一个 16x16 位‘区域选择块’来控制到那个区域的安全访问。此区域选择块包含用于那个区域的 128 位安全密码，以及针对可保护内存的区域分配和仅执行指定。 Figure 5 显示了一个区域选择块的条目和排列。

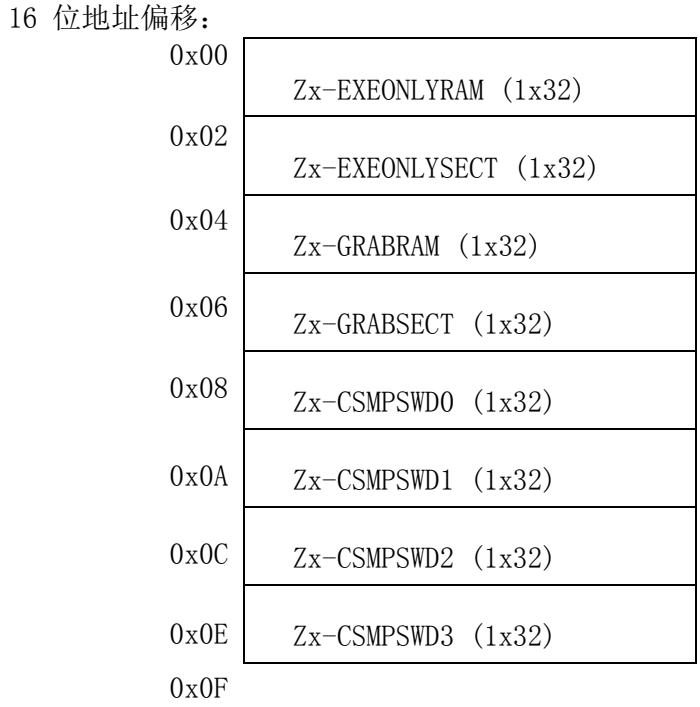

Figure 5. DCSM 区域选择块

3. 由于安全配置表位于 OTP 内存中，一个区域选择块内的每个条目只能被写入一次（更准确的说，一个 1 位可随时翻转为一个 0 位，但是位 0 不能被改变）。 为了在产品的使用周期内提供少数几次改变安全配置的灵活性，并辅助开发，多个区域选择块被用作 OTP 内总 512x16 位区域配置表的一部分。 配置表中的第一个条目是一个指定表中的哪一个区域选择块被激活的连接值。 此连接值可通过随后的 1 位到 0 位的变换（从位位置 0 开始）来更新。 在这个方法中，区域选择块可被“更新”多达 30 次。 Figure 6 显示一个区域配置表的内存中的条目和排列。 为了明确在 OTP 内存中有两个配置表，每个配置表用于两个安全区域的其中一个。

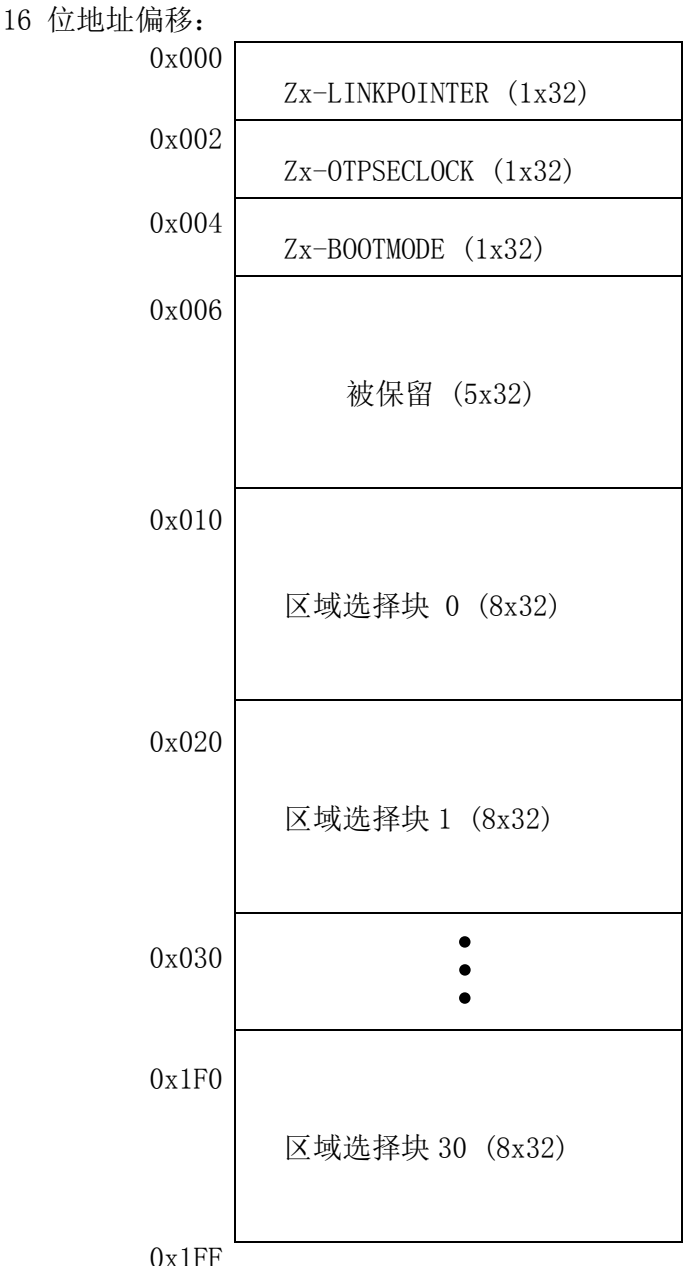

Figure 6. DCSM 安全区域配置表 OTP 内存

对于一个指定器件，应该查阅数据表来确定每个 DCSM 安全配置表的基地址。 例如，在 F2805x 器件上，区域 1 安全配置表的起始地址为 0x3D7A00，而区域 2 的起始地址在 0x3D7800。

如之前所述，应该参考用于特定器件的文档以获得全部 DCSM 操作和安全配置表的详细说明。主要目的是显示如何将安全配置表放置在内存中。

一个创建区域配置表的简单方法是使用一个少量的汇编语言编码。 对于这两个配置表中的每个表，需要两个数据段（也就是说总共四个数据段）。 第一个段为表基地址段，并且包含表中的头三个条目： Zx-LINKPOINTER，Zx-OTPSECLOCK 和 Zx-BOOTMODE。 它始终被连接至配置表的基地址。 包含区域选择块的第二段图示于 Figure 5 中。 它应该被连接至偏移自表基地址的一个地址上。 这个偏移由 Zx-LINKPOINTER 值指定。 请见针对 Zx-LINKPOINTER 值的器件文档。它不是一个简单地‘等值偏移’关系。 总的来说，用户将按顺序使用区域选择块，从偏移地址 0x010 上的区域选择块开始。 要更改为一个新的（下一个）区域选择块，Zx-LINKPOINTER 内的最低有效 1 位被设定为一个 0，并且新的区域选择块将被设定为配置中的下一个偏移地址。

以下两个示例汇编代码文件创建了每个安全区域所需的数据元素。 基地址段被命名为 *dcsm_otp_z1* 和 *dcsm_otp_z2*，而区域选择块段被命名为 *dcsm_zsel_z1* 和 *dcsm_zsel_z2*。 与所使用的汇编语言命令相关的更多信息请参考 [11]。

```
;*************************************************************************
; 文件： Passwords_zone1.asm
;*************************************************************************

;*************************************************************************
; 对于所有条目显示 0xFFFFFFFF 的虚值。 用户可以
; 将这些值改为需要的值。
;
; 注意事项： 不要将 0x00000000 用于全部 4 个密码，否则 DCSM 区域
不要将 0x00000000 用于全部 4 个，密码，否则 DCSM 区域 请参考 18 以获得更多信息。
;*************************************************************************

    .sect "dcsm_otp_z1"
    .long  0xFFFFFFFF                ;Z1-LINKPOINTER
    .long  0xFFFFFFFF                ;Z1-OTPSECLOCK
    .long  0xFFFFFFFF                ;Z1-BOOTMODE

    .sect "dcsm_zsel_z1"
    .long  0xFFFFFFFF                ;Z1-EXEONLYRAM
    .long  0xFFFFFFFF                ;Z1-EXEONLYSECT
    .long  0xFFFFFFFF                ;Z1-GRABRAM
    .long  0xFFFFFFFF                ;Z1-GRABSECT
    .long  0xFFFFFFFF                ;Z1-CSMPSWD0 (128 位密码 的 LSW)
    .long  0xFFFFFFFF                ;Z1-CSMPSWD1
    .long  0xFFFFFFFF                ;Z1-CSMPSWD2
    .long  0xFFFFFFFF                ;Z1-CSMPSWD3 (128 位密码 的 MSW)

    .end                             ; Passwords_zone1.asm
```

```
*************************************************************************
* 文件： Passwords_zone2.asm
*************************************************************************

*************************************************************************
* 对于全部条目，显示 0xFFFFFFFF 的虚值。  用户可以
* 将这些值改为需要的值。
*
* 注意事项： 不要将 0x00000000 用于全部 4 个，密码，否则 DCSM 区域
* 可被永久锁定。  更多信息请参考 [18]。
*************************************************************************

    .sect "dcsm_otp_z2"
    .long  0xFFFFFFFF                  ;Z2-LINKPOINTER
    .long  0xFFFFFFFF                  ;Z2-OTPSECLOCK
    .long  0xFFFFFFFF                  ;Z2-BOOTMODE

    .sect "dcsm_zsel_z2"
    .long  0xFFFFFFFF                  ;Z2-EXEONLYRAM
    .long  0xFFFFFFFF                  ;Z2-EXEONLYSECT
    .long  0xFFFFFFFF                  ;Z2-GRABRAM
    .long  0xFFFFFFFF                  ;Z2-GRABSECT
    .long  0xFFFFFFFF                  ;Z2-CSMPSWD0 (128 位密码的 LSW)
    .long  0xFFFFFFFF                  ;Z2-CSMPSWD1
    .long  0xFFFFFFFF                  ;Z2-CSMPSWD2
    .long  0xFFFFFFFF                  ;Z2-CSMPSWD3 (128 位密码的 LSW)

    .end                               ; 文件 Passwords_zone2.asm 的末尾
```

请注意，对于所有配置值，这个示例显示 0xFFFFFFFF 的虚值。 当您为保护器件做好准备时，这些值将被替换为您需要的值。

注意事项：

请不要将 0x00000000 用于区域选择块内的全部 4 个密码。 这样做的话会永久锁定 DCSM 模块中的那个区域。 更多信息请参考 18。

现在 *dcsm_otp_z1*，*dcsm_otp_z2*，*dcsm_zsel_z1* 和 *dcsm_zsel_z2* 需要被连接至内存。

对于非 DSP/BIOS 项目，用户应该定义四个内存以在用户连接器命令文件中保存这些段。 例如，定义 *DCSM_OTP_Z1_P0* 和 *DCSM_OTP_Z2_P0* 来保存两个表基地址项目段，定义 *DCSM_ZSEL_Z1_P0* *DCSM_ZSEL_Z2_P0* 来保存两个区域选择块段。 然后，段 d*csm_otp_z1*，*dcsm_otp_z2*，*dcsm_zsel_z1* 和 *dcsm_zsel_z2* 段可如以下示例中显示的那样被连接至这些内存。

```
/*************************************************************************
* 用户连接器命令文件（非 DSP/BIOS 项目）
*************************************************************************/

内存
{
PAGE0:    /* 程序存储器 */
   DCSM_OTP_Z2_P0   : origin = 0x3D7800, length = 0x000006    /* Z2 DCSM 表基 */
   DCSM_ZSEL_Z2_P0  : origin = 0x3D7810, length = 0x000010    /* Z2 选择块 */
   DCSM_OTP_Z1_P0   : origin = 0x3D7A00, length = 0x000006    /* Z1 DCSM 表基 */
   DCSM_ZSEL_Z1_P0  : origin = 0x3D7A10, length = 0x000010    /* Z1 选择块 */

PAGE1:    /* 数据存储器 */
}


SECTIONS
{
   dcsm_otp_z1:     > DCSM_OTP_Z1_P0,     PAGE = 0
   dcsm_zsel_z1:    > DCSM_ZSEL_Z1_P0,    PAGE = 0
   dcsm_otp_z2:     > DCSM_OTP_Z2_P0,     PAGE = 0
   dcsm_zsel_z2:    > DCSM_ZSEL_Z2_P0,    PAGE = 0
}
```

对于 DSP/BIOS 项目，使用 DSP/BIOS 配置工具的内存段管理器来定义这四个内存。 下面的四个图表显示了针对这些 F2805x 器件上内存的 DSP/BIOS 内存段管理器属性。 请始终查阅器件数据表来确认正确地址和长度，或者其他具有 DCSM 模块的 F28xx 器件的内存映射信息。

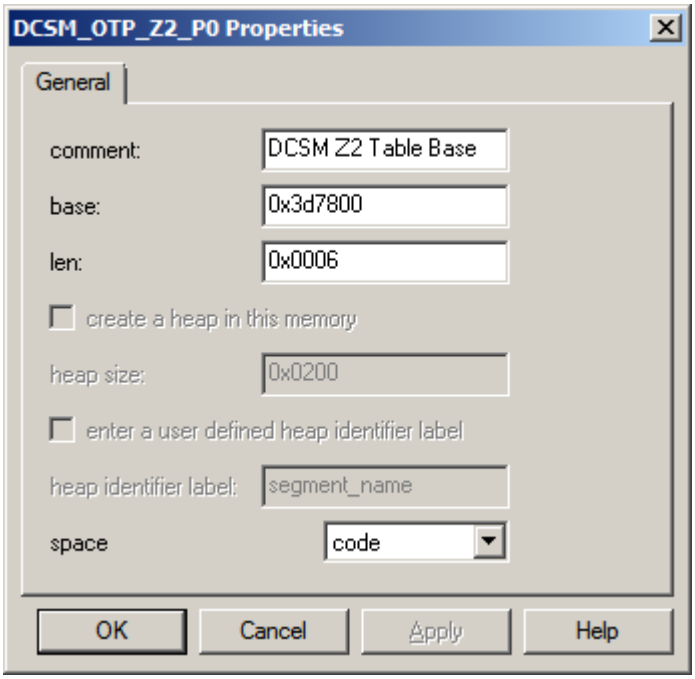

Figure 7. 针对 DCSM_OTP_Z2_P0 内存的 DSP/BIOS MEM 属性

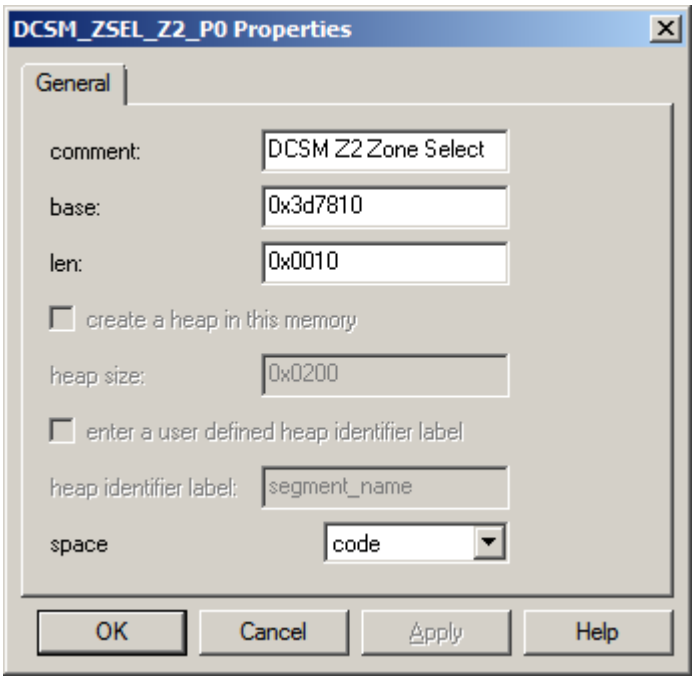

Figure 8. 针对 DCSM_ZSEL_Z2 内存的 DSP/BIOS MEM 属性

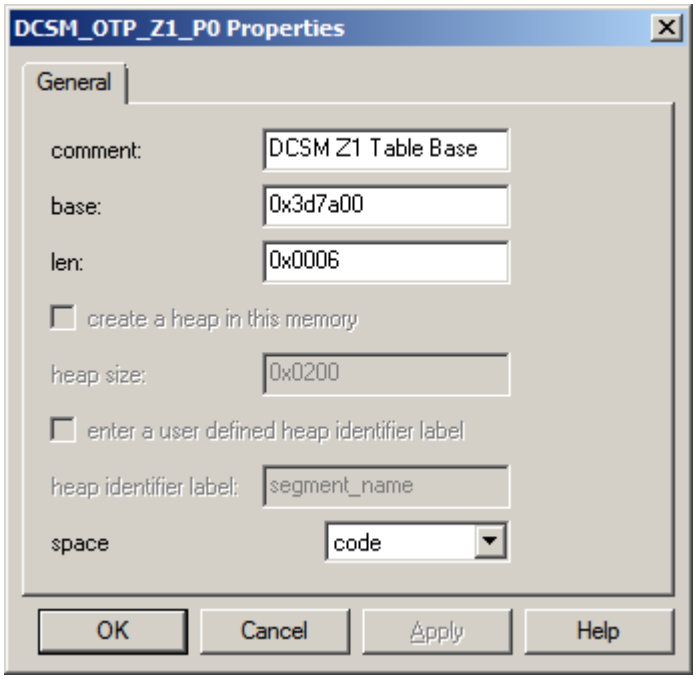

Figure 9. 针对 DCSM_OTP_Z1_P0 内存的 DSP/BIOS MEM 属性

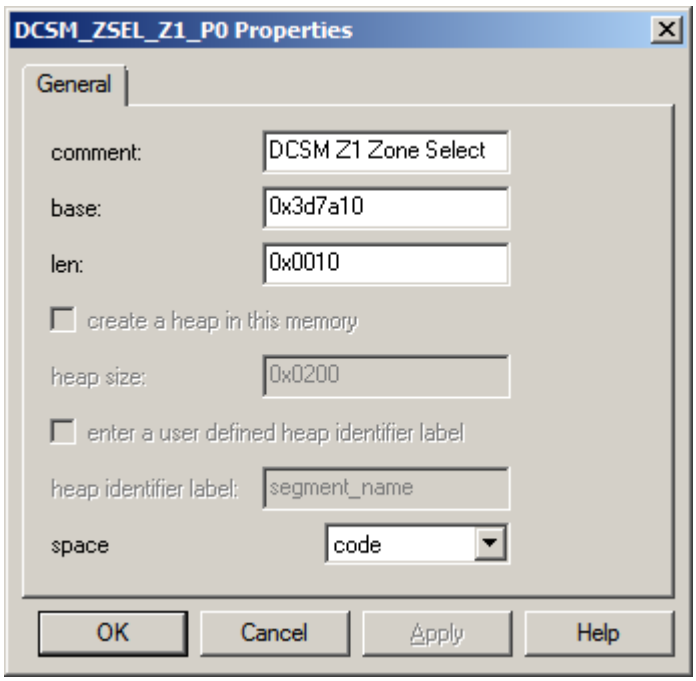

Figure 10. 针对 DCSM_ZSEL_Z1_P0 内存的 DSP/BIOS MEM 属性

然后，四个被创建的安全段被连接至这些具有用户连接器命令文件的内存。 如下所示，这与用于非 DSP/BIOS 项目的 .cmd 文件的 SECTIONS 部分一样：

```
/*********************************************************************
* 用户连接器命令文件（DSP/BIOS 项目）
*********************************************************************/

SECTIONS
{
   dcsm_otp_z1:     > DCSM_OTP_Z1_P0,     PAGE = 0
   dcsm_zsel_z1:    > DCSM_ZSEL_Z1_P0,    PAGE = 0
   dcsm_otp_z2:     > DCSM_OTP_Z2_P0,     PAGE = 0
   dcsm_zsel_z2:    > DCSM_ZSEL_Z2_P0,    PAGE = 0
}
```

6 DSP 复位后，从闪存执行您的代码

（DSP/BIOS 和非 DSP/BIOS 项目）

F28xxx 器件包含一个 ROM 引导加载程序，此程序可在器件复位后，将代码执行转移到闪存。参考 [18 – 24] 中有 ROM 引导加载程序的详细信息。 当引导模式选择引脚被配置为“跳转至闪存”模式时，ROM 引导加载程序将分支到位于闪存内的跳转到闪存目标地址的指令。 对于 F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件，这个地址为 0x3F7FF6，对于 F2805x 器件为 0x3F7FFE，而对于 F2833x 和 F2823x 器件为 0x33FFF6。 始终在器件数据表 [1 – 8] 中，或 ROM 引导加载程序文档 [18 – 24] 中确认针对您专用器件的地址。 用户应该将一个分支到他们代码开始位置的指令放置在这个地址。 一个长分支指令（汇编代码内的 LB）占用两个 16 位字，并且器件内存映射被设计成适应这种配置。

总的来说，分支指令将分支到位于 C 语言汇编程序运行支持库内的 C 语言环境初始化例程的开始。 针对这个例程的项目符号为*_c_int00*。 在这个设置例程运行前，不能执行任何 C 语言代码。 或者，有时需要在开始您的 C 语言应用前执行少量的汇编代码（例如，禁用安全装置定时器外设）。 在这个情况下，分支指令应该分支到您汇编代码的开始。 无论如何，需要在闪存中准确地确定这个分支指令。 完成这一操作的最简单的方法就是使用汇编代码。 下面的示例创建了一个被称为 *codestart* 的已命名初始化段，这个段包含一个到 C 语言环境设置例程的长分支指令。 Codestart 段应该被放置在具有用户连接器命令文件的内存中。

```
;*************************************************************************
; CodeStartBranch.asm
;*************************************************************************

    .ref _c_int00

    .sect "codestart"
    LB _c_int00                     ; 分支到代码起始位置

    .end                            ; 文件 CodeStartBranch.asm 的末尾
```

对于非 DSP/BIOS 项目，用户应该在用户连接器命令文件的 MEMORY 部分内的 PAGE 0 上定义一个名为（例如）*BEGIN_FLASH*的内存。 然后，段 *codestart* 可被连接到这个内存。 以下示例使用的 0x3F7FF6 地址应用于 F281x，F280x，F2801x，F28044，F2802x，F2803x 和 F2806x 器件。

; 用户连接器命令文件（非 DSP/BIOS 项目）

```
/*************************************************************************
; 用户连接器命令文件（非 DSP/BIOS 项目）
*************************************************************************/

内存
{
PAGE0:    /* 程序存储器 */
   BEGIN_FLASH : origin = 0x3F7FF6, length = 0x000002
PAGE1:    /* 数据存储器 */
}

SECTIONS
{
/*** 跳转到闪存引导模式进入点 ***/
codestart:      > BEGIN_FLASH,  PAGE = 0
}
```

对于 DSP/BIOS 项目，用户应该使用 DSP/BIOS 配置工具内的内存段管理器来定义名为 *BEGIN_FLASH*（例如）内存。 Figure 11 显示了针对这个内存的内存段管理器属性，这个内存具有针对 F281x, F280x, F2801x, F2802x, F2803x, F28044, or F2806x 器件的“基”地址入口集。

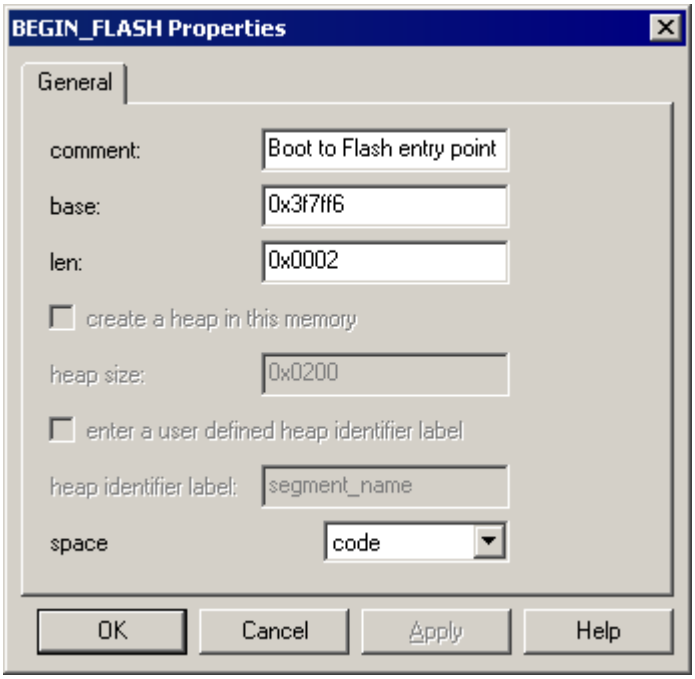

Figure 11. 针对跳转至闪存进入点的 DSP/BIOS MEM 属性

然后， 段 *codestart* 可在用户连接器命令文件内被连接至这个内存。 对于 DSP/BIOS 项目，连接器命令文件将显示为：

```
/*************************************************************************
* 用户的连接器命令文件（DSP/BIOS 项目）
*************************************************************************/

SECTIONS
{
/*** 跳转至闪存引导模式进入点***/
codestart:      > BEGIN_FLASH,  PAGE = 0
}
```

7 C 语言环境引导期间禁用安全装置定时器

（DSP/BIOS 和非 DSP/BIOS 项目）

C 语言编译程序运行支持库内的 C 语言环境初始化函数，*_c_int00*，执行全局和静态变量的初始化。 对于每个已初始化全局变量，这涉及到将一个数据从.*cinit* 段（位于片载闪存存储器内）复制到.*ebss* 段（位于 RAM 内）。 例如，当一个全局变量在源代码中被声明为：

```
int x=5;
```

数字 “5” 被放置在已初始化段.*cinit* 内，而 .ebss 段内的空间被保留用于符号 “x.”。 然后*_c_int00* 例程在运行时将“5”复制到位置“x”。 当大量的已初始化全局和静态变量出现在软件中时，安全装置定时器可在 C 语言环境引导例程完成并且调用 *main()*前超时（在这里，安全装置可被配置和处理，或被禁用。） 这个问题也许在 RAM 中的代码开发中不是很明显，这是因为从一个已连接的.*cinit* 段到 RAM 的数据复制以很快的速度发生。 然而，当.*cinit* 段被连接到内部闪存时，由于内部闪存存储器缺省为最大数量的等待状态，复制每个数据字将花费多个周期（等待状态在用户代码达到 *main()*前不会被配置）。 此外，执行数据复制的代码从闪存执行，这进一步增加了完成数据复制所需的时间（取代码和数据读取必须共用到闪存的访问）。 与安全装置超时周期缺省为它的最小可能值的这样一个事实组合在一起，安全装置超时变得完全可能。

您可以使用 CCS 调试程序来在您的系统中检测这个问题是否出现。 在 main() 的起始位置设定一个断点，也可在 _c_int00 的开始位置设定一个断点。 复位处理器，然后运行。 您应该命中 _c_int00 上的断点。 如果您不这么操作的话，您将具有一个引导模式配置问题。 如果您达到 _c_int00，重新运行。 您应当到达 main() 内的断点的时间。 如果不这么做的话，安全装置在您到达那时已超时。

纠正安全装置超时问题的最简单方法是在启动 C 语言引导例程之前禁用安全装置。 随后，安全装置可在运行至*main()*之后被重新启用，并开始您的正常代码执行流程。 通过将 WDCR 寄存器内将 WDDIS 位设置为 1 来禁用安全装置。 为了在引导例程之前禁用安全装置，必须使用汇编代码（这是因为 C 语言环境还未建立）。 在段 6 中，*codestart* 汇编代码段执行一个分支指令，此指令跳转到 C 语言-环境初始化例程，*_c_int00* 例程。 为了禁用安全装置，这个分支应该改为跳转到安全装置禁用代码，然后可以分支到*_c_int00* 例程。 以下的代码示例执行上述任务任务：

```
;*****************************************************************************
; 文件:  CodeStartBranch.asm
; 器件:  TMS320F28xxx
; 作者:  David M. Alter, 德州仪器 (TI)) 公司
; 发展历史:  02/11/05 - 最初版本 (D.Alter)
;*****************************************************************************

WD_DISABLE    .set   1      ;set to 1 to disable WD, else set to 0(置位为 1 来禁用 WD, 否则置位为 0)

    .ref _c_int00

;*****************************************************************************
; 函数:codestart 段
; 说明:  分支到代码起始点
;*****************************************************************************
    .sect "codestart"
    .if WD_DISABLE == 1
        LB wd_disable       ;Branch to watchdog disable code(分支至安全装置禁用代码)
    .else
        LB _c_int00         ;Branch to start of boot.asm in RTS library(分支至 RTS 库内的 boot.asm 的起
始位置)
    .endif
;end codestart section(终止 codestart 段)

;*****************************************************************************
; 函数:wd_disable
; 说明:  禁用安全装置定时器
;*****************************************************************************
    .if WD_DISABLE == 1

    .text
wd_disable:
    EALLOW                  ;Enable EALLOW protected register access(启用 EALLOW 受保护寄存器访问)
    MOVZ DP, #7029h>>6      ;Set data page for WDCR register(为 WDCR 寄存器设置数据页)
    MOV @7029h, #0068h      ;Set WDDIS bit in WDCR to disable WD(在 WDCR 内将 WDDIS 位置位以禁用 WD)
    EDIS                    ;Disable EALLOW protected register access(禁用 EALLOW 受保护寄存器访问)
    LB _c_int00             ;Branch to start of boot.asm in RTS library(分支到 RTS 库内的 boot.asm 的起
始位置)

    .endif

;end wd_disable(终止 wd_disable)
;*****************************************************************************

    .end                    ; end of file CodeStartBranch.asm(CodeStartBranch.asm 文件的末尾)
```

8 C 语言代码示例

8.1 概述

随这份报告一同提供了针对每个 F281x，F280x，F2802x，F2803x，F2805x，F2806x 和 F2833x 子系列内扩展集器件的包含 CCS v5 项目的代码下载。 除了 F2802x，F2803x 和 F2805x，每个器件类型有四个独特的项目：

- F28xxx_example_nonBIOS_ram.pjt - 从片载 RAM 中运行的非 DSP/BIOS 项目
- F28xxx_example_nonBIOS_flash.pjt - 从片载闪存运行的非 DSP/BIOS 项目
- F28xxx_example_BIOS_ram.pjt - 从片载 RAM 上运行的 DSP/BIOS 项目
- F28xxx_example_BIOS_flash.pjt - 从片载闪存运行的 DSP/BIOS 项目

F2802x，F2803x 和 F2805x 示例不包括 F28xxx_example_BIOS_ram.pjt，这是因为从实用角度来将，这些器件上的 RAM 在不采用闪存存储器保存大容量代码的情况下不足以支持 DSP/BIOS。此外，虽然本报告关注的是从闪存中运行代码，RAM 示例被提供以实现完整性并可被用于开发工作的早期阶段。

这些只是示例，并且仅仅进行了简单测试。 不保证它们对于应用使用的适用性。 这些示例在以下开发工具版本中构建和测试：

CCS v5.3.0，代码生成工具 v6.1.1，DSP/BIOS v5.42.0.07

请注意，即使对早期版本的修订较少，但是 CCS 项目将无法在 CCS 的早期版本中正确运行。CCS 版本通常是向后兼容，而非向前兼容。 此项目看起来似乎成功导入，但是它们也许缺少某些项目选项和/或产生构建错误。 如果您正在使用早期 CCS v5 版本，请至少更新至 v5.3.0。或者，您可以用老版本的 CCS 创建一个新的 CCS 项目，并且将相关的示例源代码置入新项目中。代码本身将运转良好。 它只是也许无法使用早期 CCS 版本的 CCS 项目文件本身（即，.project，.cproject 和 .ccsproject）。

源代码使用 C2000 外设头文件的以下版本来访问器件外设寄存器：

DSP281x 外设头文件结构 v1.20
DSP280x 外设头文件结构 v1.70
DSP2833x 外设头文件结构 v1.33
F2802x 外设头文件结构 v2.10
DSP2803x 外设头文件结构 v1.26
F2805x 外设头文件结构 v1.00
F2806x 外设头文件结构 v1.35

这里包含了所有需要的头文件包内的文件（请参考 [25 - 32]）。 然而，我们鼓励用户获得完整的头文件包以了解额外信息。 对于较新的器件，头文件可通过 ControlSuite 工具获得，此工具可从 http://www.ti.com/tool/controlsuite 内下载。 对于早期器件，它们可从 TI 网站，http://www.ti.com 内下载。

此项目在 eZdspF2812，eZdspF2808 和 eZdspF28335 开发工具上开发，F2808，F28335，F28027，F28035，F28055 和 F28069 试验板套件，F28027 和 F28069‘Piccolo’控制棒。 然而，它们也在下面的其它 F28xxx 电路板上运行：

F281x 示例： 由于它们完全从内部存储器内运行，并且只使用所有 3 个器件的公共闪存存储器，这些示例将在 F281x 电路板上运行。 如果在一个不同的电路板上运行，请注意，代码将 GPIOA0/PWM1 和 GPIOF14/XF_XPLLDIS* 引脚配置为输出。 此外，请注意，代码的确将 F2812 上的外部内存接口配置为 DSP 初始化过程的一部分。 由于大多数外部内存接口不存在于 F2810 和 F2811 器件上（XCLKOUT 引脚例外），这两个器件上不需要这个初始化过程（虽然此过程是无害的）。

F280x 示例： 这将在任何 F2808 电路板上运行。 通过调整 .cmd 文件内针对非-DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2802x，F2801x 和 F2844 电路板上运行。 根据电路板上晶振或振荡器以及器件的运行频率的不同，也许需要调整锁相环 (PLL) 设置。 如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A 和 GPIO34 引脚配置为输出。

F2833x 示例： 这将在任何 F28335 电路板上运行。 通过调整 .cmd 文件内针对非-DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2833x 或 F2823x 电路板上运行。 根据电路板上晶振或振荡器以及器件的运行频率的不同，也许需要调整锁相环 (PLL) 设置。 此外，对于 F2823x 器件，您应该改变 CCS v5 内的项目构建选项以禁用浮点支持（前往 Project →属性，选择 Build→C2000 编译程序 →Runtime Model 目录，并改变‘Specify floating point support’（指定浮点支持）框为空。如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A，GPIO32 和 GPIO34 引脚配置为输出。

F2802x 示例： 这将在任何 F28027 电路板上运行。 通过调整 .cmd 文件内针对非 DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2802x 电路板上运行。 根据器件的运行频率的不同，也许需要调整锁相环 (PLL) 设置。 如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A 和 GPIO34 引脚配置为输出。

F2803x 示例： 这将在任何 F28035 电路板上运行。 通过调整 .cmd 文件内针对非 DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2803x 电路板上运行。 如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A 和 GPIO34 引脚配置为输出。

F2805x 示例： 这将在任何 F28055 电路板上运行。 通过调整 .cmd 文件内针对非 DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2805x 电路板上运行。 如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A 和 GPIO34 引脚配置为输出。

F2806x 示例： 这将在任何 F28069 电路板上运行。 通过调整 .cmd 文件内针对非 DSP/BIOS 项目的内存定义（RAM 和闪存），或者调整针对 DSP/BIOS 项目的 . tcf 文件，它们也可被修改为在其他 F2806x 电路板上运行。 如果在一个不同的电路板上运行，用户应该注意到代码将 GPIO0/ePWM1A 和 GPIO34 引脚配置为输出。

每个代码项目执行一样的函数：

- 图示 F28xxx 器件初始化。 PLL 被配置成每个器件所允许的最大时钟速度。
- 启用 Code Composer Studio 的实时仿真模式。
- 切换 F2812 上的 GPIOF14 引脚，F2808，F28027，F28035，F28055 和 F28069 上的 GPIO34 引脚，以及 F28335 上的 GPIO32 和 GPIO34 引脚。 这将使开发板上的 LED 闪烁（对于 F28335，根据使用的电路板，只将一个 GPIO 连接到 LED）。 在非 DSP/BIOS 项目中，这在 ADCINT ISR 内完成。 在 DSP/BIOS 项目中，使用了一个周期函数。
- 将 ADC 配置为采样 ADCINA0 通道。 在 F2812，F2808，F28335 和 F28069 器件上，其完成速率为 50kHz。 25kHz 速率用于较慢的 60MHz 器件： F28027，F28035 和 F28055。 由于 DSP/BIOS 示例中的 CPU 负载限制，需要 25kHz 速率，这说明了您为什么不应当像在这些示例中完成的那样（即，ADC SWI）将一个高速中断置于 DSP/BIOS 控制之下！ 或者，直接在 HWI 内执行您的高频中断例程。 DSP/BIOS 仍然能够管理整个系统和更低频 ISR。
- 处理 ADC 中断。 ADC 结果被放置在长度为 50 字的循环缓冲器中。
- 在 PWM1 引脚（对于 F281x）上发出 2kHz 对称 PWM，或者映射到 GPIO0 引脚的 ePWM1A 信号（对于 F2808，F28335，F28027，F28035，F28055 和 F28069）。
- 配置捕捉单元 #1。 在 F2808，F28335，F28027，F28055 和 F28069 器件上，eCAP 信号被影射到 GPIO5 引脚上。
- 处理捕捉 #1 中断。 读取捕捉结果并计算脉宽。

8.2 目录结构

每个代码项目在全部所需的文件方面是完全独立完整的（除了 C 语言编译程序运行支持 (RTS) 库，此支持库取自 CCS 项目所使用的代码生成工具的文件夹）。 Table 3 提供单个 CCS 项目目录结构的说明。

Table 3. CCS 示例代码目录说明

文件目录	内容
<项目_根目录>	包含 CCS 创建的项目文件： .ccsproject，.cdtbuild，.cdtproject，.project 和 DSP/BIOS .tcf。
<项目_根目录>\.settings	包含 CCS 创建的项目设置文件
<项目_根目录>\cmd	包含用户连接器命令文件（.cmd 文件）
<项目_根目录>\DSP28xxx_headers\cmd 或者 <项目_根目录>\F28xxx_headers\cmd	包含针对目标器件的外设头文件结构所需的连接器命令文件。

<项目_根目录>\DSP28xxx_headers\include 或者 <项目_根目录>\F28xxx_headers\include	包含外设头文件结构中针对目标器件的所需 include 文件。
<项目_根目录>\include	包含 include 文件（.h 文件）
<项目_根目录>\src	包含源代码文件（.c 和 .asm 文件）

8.3 其他信息

1) 构建一个项目后，.out 文件将被放置在 <项目_根目录>\Debug （调试）目录中。

2a) 如果使用 RAM 示例，您的电路板应该针对“跳转至 H0 SARAM”(F2812) 或者“跳转至 M0 SARAM”（F2808，F28027，F28035，F28055，F28069，F28335）引导模式进行配置。 检查针对您的电路板的参考手册以确认任何所需的跳线设置，并且还需查看用于您的器件的引导 ROM 用户指南（请参考 [18 – 24]）。 下面概括出了一些开发板。 检查电路板跳线/拨码开关为：

eZdspF2812: JP1 2-3 (MP/MC*)
JP9 1-2 (PLL)
JP7 2-3（引导模式选择）
JP8 2-3（引导模式选择）
JP11 1-2（引导模式选择）
JP12 2-3（引导模式选择）

eZdspF2808: DIP SW1: 1 = ON
2 = OFF
3 = ON

eZdspF28335: DIP SW1: 1 = ON
2 = ON
3 = OFF
4 = ON

F2808 实验板套件： 这个电路板的当前版本针对跳转至闪存引导模式实线连接。 调试程序可被用来将 PC 设定为开始从代码入口点执行（例如*_c_int00*）。 在 CCS v5 中，运行→ 重启将把 PC 设定至这个入口点。

F28335 实验板套件： 这个电路板的当前版本针对跳转至闪存引导模式实线连接。 调试程序可被用来将 PC 设定为开始从代码入口点执行（例如*_c_int00*）。 在 CCS v5 中，运行→ 重启将把 PC 设定至这个入口点。

F28027 实验板套件或 ControlStick： 当仿真器被连接至 F28027 时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导模式选择选项。

F28035 实验板套件： 当仿真器被连接至 F28035 时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导模式选择选项。

F28055 实验板套件： 当仿真器被连接至 F28055 时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导模式选择选项。 请注意，在 F28055 ISO 控制卡上，JTAG 仿真与电路板的处理器侧电气隔离。 这个设计的一个副产品是控制处理器和板载仿真芯片之间 TRSTn 信号连接的 DIP SW4.1。 为了实现 JTAG 功能，SW4.1 必须处于 ON 位置（向上）（ON 位置实现 TRSTn 连接）。 更多信息请见 F28055 ISO 控制卡文档。

F28069 实验板套件或 ControStick： 当仿真器被连接至 F28069 时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导模式选择选项。

如果上面的方法看起来不起作用，检查针对您电路板的参考手册以确认任何所需的跳线设置，并且还需查看针对您的器件的引导 ROM 用户指南（请参考 [18-24]）。

2b) 如果使用闪存示例，您的电路板应该针对“跳转至闪存”引导模式进行配置。 检查针对您电路板的参考手册以确认任何所需的跳线设置，并且还请查看针对您器件的引导 ROM 用户指南（请参考 [18 - 24]）。 下面概括了一些开发板。 检查电路板跳线/拨码开关为：

eZdspF2812: JP1 2-3 (MP/MC*)
JP9 1-2 (PLL)
JP7 1-2（引导模式选择）
JP8 无关（引导模式选择）
JP11 无关（引导模式选择）
JP12 无关（引导模式选择）

eZdspF2808: DIP SW1: 1 = OFF
2 = OFF
3 = OFF

eZdspF28335: DIP SW1: 1 = OFF
2 = OFF
3 = OFF
4 = OFF

F2808 实验板套件： 这个电路板实线连接为跳转至闪存引导模式。

F28335 实验板套件： 这个电路板缺省为跳转至闪存引导模式。

F28027 实验板套件或 ControlStick： 当仿真器被连接时，调试程序被用来选择引导模式。请在 CCS v5 内的 Scripts 菜单内选中选择选项。 当仿真器被断开时，如果 OTP 内的 OTP_KEY 和 OTP_BMODE 位置还未进行其它设定，此电路板将在跳转至闪存模式内引导，并且只有在实验板套件上才是如此：

DIP SW1: 1 = ON

2 = ON

F28035 实验板套件： 当仿真器被连接时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导模式选择选项。 当仿真器被断开时，如果 OTP 内的 OTP_KEY 和 OTP_BMODE 位置还未进行其它设定，此电路板将在跳转至闪存模式内引导，并且 DIP SW2 被设定为：

DIP SW2: 1 = ON

2 = ON

F28055 实验板套件： 当仿真器被连接时，调试程序被用来选择引导模式。 请在 CCS v5 内的 Scripts 菜单内选中引导选择选项。 当仿真器被断开时，如果 OTP 内的 OTP_KEY 和 OTP_BMODE 位置还未进行其它设定，此电路板将在跳转至闪存模式内引导，并且以下的 DIP 开关被设定为：

DIP SW1: 1 = ON

2 = ON

DIP SW4: 1 = OFF

2 = 无关

由于 F28055 ISO 控制卡上的 JTAG 仿真与电路板处理器一侧电气隔离，所以需要 SW4 设置。这个设计的一个副产品是，为了使处理器在独立模式中（未连接仿真器），SW4.1 必须处于 OFF 位置（向下）。 SW4.1 控制处理器和板载仿真器芯片间 TRSTn 信号的连接，而 OFF 位置断开此连接。 更多信息请见 F28055 ISO 控制卡文档。

F28069 实验板套件或 ControlStick 当仿真器被连接时，调试程序被用来选择引导模式。 请在 CCS v5 中的 Scripts 菜单中选中引导模式选择选项。 当仿真器被断开时，如果 OTP 内的 OTP_KEY 和 OTP_BMODE 位置还未进行其它设定，此电路板将在跳转至闪存模式内引导，并且只有在实验板套件上才是如此：

DIP SW2: 1 = ON

2 = ON

如果上面的方法看起来不起作用，检查针对您电路板的参考手册以确认任何所需的跳线设置，并且还需查看针对您的器件的引导 ROM 用户指南（参考 [18 - 24]）。

3) 并未对连接位置给予太多关注。 编写这些项目的目的是简化对它们的操作。 如果这些项目被用作代码开发的起始点，也许需要多连接进行调整以获得更好的性能（例如，避免内存块访问冲突，或者更好地管理内存块的使用）。

4) 对于非 DSP/BIOS 项目，在文件 *DefaultIsr_nonBIOS.c* 内定义了一个完整的中断处理例程集合。 每个中断直接从它的硬件 ISR 内执行。 然而，除了 ADCINT 和 ECAP1INT（或者 F2812 上的 CAPINT1）之外，每个 ISR 实际上在一个死循环之前执行一个 ESTOP0 指令（仿真停止）来在调试期间捕捉假中断。 在生产代码中，您有可能想将未使用的中断引导至您自己设计的某些类错误处理例程（与仅仅进行代码捕捉相反）。 请注意，每个 iSR 正在使用“中断”关键字，此关键字告诉编译程序在函数进入/离开时执行一个运行环境保存/恢复。

5) 对于 DSP/BIOS 项目，在文件 *DefaultIsr_BIOS.c* 中定义了（硬件）中断处理例程的一个完整集合。 通过使用 DSP/BIOS 配置工具内的 HWI 管理器，每个 ISR 可被到连接至所需的中断。这样，代码示例只分派两个在用的 ISR，明确地讲是 ADCINT1 和 ECAP1INT。 此外，DSP/BIOS 中断调度程序被用来处理运行环境保存/恢复，这就是为什么 ISR 不使用“中断”关键字（非 DSP/BIOS 情况中那样）。 在这些示例中，ECAP1INTISR（或用于 F2812 的 CAPINT1）在 *DefaultIsr_BIOS.c* 内直接执行（作为一个减少延迟的示例），而 *DefaultIsr_BIOS.c* 内的 ADC 中断函数存储一个 SWI 来执行 ADC 例程。 这些只是示例。 请注意，ECAP1INT（和 CAPINT1）ISR 正在使用 DSP/BIOS 调度程序来执行运行环境保存/恢复（正如配置工具中 HWI 管理器选择的那样）。 如果要求绝对最小延迟（用于某些时间关键 ISR），用户可以禁用针对那个中断的中断调度程序，并且将“中断”关键字添加到 ISR 函数声明中。 请注意，这样做将防止用户在 ISR 中采用任何 DSP/BIOS 功能。 此外，用户应该考虑对任务堆栈尺寸要求的可能影响，这是因为时间关键 ISR 可在任何现有堆栈（系统堆栈或任一堆栈）的运行环境内运行，这取决于关键中断发生时，哪个线程被激活。 最后，DSP/BIOS 配置工具内的 HWI 管理器将所有未使用的中断缺省为名为 HWI_unused() 的例程。 这个例程基本基本上与错误捕捉代码使用一个无限循环。对于生产代码，您应该将所有未使用的中断您自己设计的某些类错误处理例程。

参考书目

1. 《TMS320F2810，TMS320F2811，TMS320F2812，TMS320C2810，TMS320C2811，TMS320C2812 数字信号处理器数据手册》（文献编号：SPRS174）
2. 《TMS320F2809，TMS320F2808，TMS320F2806，TMS320F2802，TMS320F2801，TMS320C2802，TMS320C2801，TMS320F2801x DSPs 数据手册》（文献编号：SPRS230）
3. 《TMS320F28044 数字信号处理器数据手册》（文献编号：SPRS357）
4. 《TMS320F28335，TMS320F28334，TMS320F28332，TMS320F28235，TMS320F28234，TMS320F28232 数字信号处理器数据手册》（文献编号：SPRS439）
5. 《TMS320F28020，TMS320F28021，TMS320F28022，TMS320F28023，TMS320F28026，TMS320F28027 Piccolo 微控制器数据手册》（文献编号：SPRS523）
6. 《TMS320F28030，TMS320F28031，TMS320F28032，TMS320F28033，TMS320F28034，TMS320F28035 Piccolo 微控制器数据手册》（文献编号：SPRS584）
7. 《TMS320F28050，TMS320F28051，TMS320F28052，TMS320F28053，TMS320F28054，TMS320F28055 Piccolo 微控制器数据手册》（文献编号：SPRS797）
8. 《TMS320F28062，TMS320F28063，TMS320F28064，TMS320F28065，TMS320F28066，TMS320F28067，TMS320F28068，TMS320F28069 Piccolo 微控制器数据手册》（文献编号：SPRS698）
9. 《TMS320C28x CPU 和指令集参考指南》（文献编号：SPRU430）
10. 《TMS320C28x 浮点单元和指令集参考指南》（文献编号：SPRUE02）
11. 《TMS320C28x 汇编语言工具用户指南》（文献编号：SPRU513）
12. 《TMS320C28x 优化 C/C++ 编译程序用户指南》（文献编号：SPRU514）
13. 《TMS320x281x DSP 系统控制和中断参考指南》（文献编号：SPRU078）
14. 《TMS320x280x，2801x，2804x DSP 系统控制和中断参考指南》（文献编号：SPRU712）
15. 《TMS320x2833x 系统控制和中断参考指南》（文献编号：SPRUFB0）
16. 《TMS320x2802x 系统控制和中断参考指南》（文献编号：SPRUFN3）
17. 《TMS320x2803x 系统控制和中断参考指南》（文献编号：SPRUGL8）
18. 《TMS320x2805x Piccolo 技术参考手册》（文献编号：SPRUHE5）
19. 《TMS320x2806x Piccolo 技术参考手册》（文献编号：SPRUH18）
20. 《TMS320x281x DSP 引导 ROM 参考指南》（文献编号：SPRU095）
21. 《TMS320x280x, 2801x, 2804x 引导 ROM 参考指南》（文献编号：SPRU722）
22. 《TMS320x2833x，2832x 引导 ROM 参考指南》（文献编号：SPRU963）
23. 《TMS320x2802x Piccolo 引导 ROM 参考指南》（文献编号：SPRUFN6）
24. 《TMS320x2803x Piccolo 引导 ROM 参考指南》（文献编号：SPRUG00）
25. 《F281x C/C++ 头文件和外设示例》（文献编号：SPRC097）
26. 《F280x C/C++ 头文件和外设示例》（文献编号：SPRC191）
27. 《F2804x C/C++ 头文件和外设示例》（文献编号：SPRC324）
28. F2833x/C2823x C/C++ 头文件和外设示例（在 ControlSuite 内提供）
29. 《F2802x C/C++ 头文件和外设示例》（在 ControlSuite 内提供）
30. 《F2803x C/C++ 头文件和外设示例》（在 ControlSuite 内提供）
31. 《F2805x C/C++ 头文件和外设示例》（在 ControlSuite 内提供）
32. 《F2806x C/C++ 头文件和外设示例》（在 ControlSuite 内提供）

修订历史记录

修订版本	日期	更改人	之前版本主要变化的说明
SPRA958L	2013 年 1 月 23 日	D. Alter	- 轻微文档润色。 - 添加了包括代码示例在内的 F2805x 器件支持。 - 扩展了部分 5 以涵盖 F2805x 上的双区域代码安全模块。 - 删除了 CCS v4 代码示例。 - 将 F28069 示例改为 80MHz 至 90MHz 运行。 - 所有代码使用 CCS v5.3.0，C 编译程序 v6.1.1 和 DSP/BIOS v.5.42.0.07 进行测试。

重要声明

德州仪器(TI) 及其下属子公司有权根据 JESD46 最新标准, 对所提供的产品和服务进行更正、修改、增强、改进或其它更改， 并有权根据 JESD48 最新标准中止提供任何产品和服务。客户在下订单前应获取最新的相关信息, 并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的组件的性能符合产品销售时 TI 半导体产品销售条件与条款的适用规范。仅在 TI 保证的范围内，且 TI 认为 有必要时才会使用测试或其它质量控制技术。除非适用法律做出了硬性规定，否则没有必要对每种组件的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用 TI 组件的产品和应用自行负责。为尽量减小与客户产品和应 用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 组件或服务的组合设备、机器或流程相关的 TI 知识产权中授予 的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服 务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它 知识产权方面的许可。

对于 TI 的产品手册或数据表中 TI 信息的重要部分，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况 下才允许进行复制。TI 对此类篡改过的文件不承担任何责任或义务。复制第三方的信息可能需要服从额外的限制条件。

在转售 TI 组件或服务时，如果对该组件或服务参数的陈述与 TI 标明的参数相比存在差异或虚假成分，则会失去相关 TI 组件 或服务的所有明示或暗示授权，且这是不正当的、欺诈性商业行为。TI 对任何此类虚假陈述均不承担任何责任或义务。

客户认可并同意，尽管任何应用相关信息或支持仍可能由 TI 提供，但他们将独力负责满足与其产品及在其应用中使用 TI 产品 相关的所有法律、法规和安全相关要求。客户声明并同意，他们具备制定与实施安全措施所需的全部专业技术和知识，可预见 故障的危险后果、监测故障及其后果、降低有可能造成人身伤害的故障的发生机率并采取适当的补救措施。客户将全额赔偿因 在此类安全关键应用中使用任何 TI 组件而对 TI 及其代理造成的任何损失。

在某些场合中，为了推进安全相关应用有可能对 TI 组件进行特别的促销。TI 的目标是利用此类组件帮助客户设计和创立其特 有的可满足适用的功能安全性标准和要求的终端产品解决方案。尽管如此，此类组件仍然服从这些条款。

TI 组件未获得用于 FDA Class III（或类似的生命攸关医疗设备）的授权许可，除非各方授权官员已经达成了专门管控此类使 用的特别协议。

只有那些 TI 特别注明属于军用等级或“增强型塑料”的 TI 组件才是设计或专门用于军事/航空应用或环境的。购买者认可并同 意，对并非指定面向军事或航空航天用途的 TI 组件进行军事或航空航天方面的应用，其风险由客户单独承担，并且由客户独 力负责满足与此类使用相关的所有法律和法规要求。

TI 已明确指定符合 ISO/TS16949 要求的产品，这些产品主要用于汽车。在任何情况下，因使用非指定产品而无法达到 ISO/TS16949 要求，TI不承担任何责任。

	产品		应用
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP应用处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity	德州仪器在线技术支持社区	www.deyisupport.com

邮寄地址： 上海市浦东新区世纪大道 1568 号，中建大厦 32 楼 邮政编码： 200122