请注意赛普拉斯已正式并入英飞凌科技公司。
此封面页之后的文件标注有“赛普拉斯”的文件即该产品为此公司最初开发的。请注意作为英飞凌产品组合的部分，英飞凌将继续为新的及现有客户提供该产品。

文件内容的连续性
事实是英飞凌提供如下产品作为英飞凌产品组合的部分不会带来对于此文件的任何变更。未来的变更将在恰当的时候发生，且任何变更将在历史页面记录。

订购零件编号的连续性
英飞凌继续支持现有零件编号的使用。下单时请继续使用数据表中的订购零件编号。

AN60486

PSoC® 1 M8C ImageCraft C 代码优化

作者：Archana Yarlagadda
相关项目：无
相关器件系列：CY8C2xxxx（所有 PSoC 1 器件）
软件版本：PSoC Designer™ 5.4
相关应用笔记：AN75320、AN60630、AN2017

若有任何问题，或需要得到本应用笔记的相关帮助，请通过 msur@cypress.com 邮箱联系本文作者。

AN60486 展示了如何对 PSoC 1 C 代码进行优化，使其执行速度更快、编写更简洁，同时还介绍了能够帮助您优化代码的各种 PSoC Designer 项目设置和 ImageCraft 编译器设置。此外，本文档还提供了一些能够有效编码的指南。本应用笔记假设您已经熟悉 PSoC 1、PSoC Designer 集成设计环境（IDE）和 C 编程语言。相关的指导内容，请参考 AN75320 — PSoC 1 入门。

目录

简介 1
项目级优化 2
第一种设置：可重定位代码的开始地址 2
第 2 种设置：配置初始化 3
第 3 种设置：Sublimation（净化）和 Condensation（缩聚）选项 4
第 4 种设置：Treat const as ROM（将常量视为 ROM）与 Treat const as RAM（将常量视为 RAM）选项 6
ImageCraft 专业版编译器选项 6
提示和指南 6
指南 1：避免在中断服务子程序中调用函数 6
指南 2：限制数学函数 8
指南 3：使用数组索引和指针 10
指南 4：使用 switch 语句和 If-Else 语句 11
指南 5：将部分代码写入汇编器中 12
指南 6：在 PSoC 1 中的位操作 12
指南 7：计算 C 代码闪存的使用率和执行时间 13
结论 14

简介

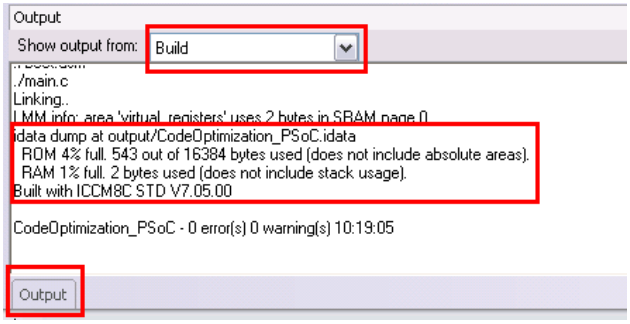
您可以通过多种方法减少 PSoC Designer 项目中使用的闪存（ROM）空间。这样，有时您便可以使用更小的 PSoC 器件，从而降低成本。

PSoC Designer 提供了 ImageCraft 编译器的标准版本。您同样可以通过 ImageCraft 网站购买具有更多优化功能的专业版本。本应用笔记中包含的代码段适用于这两种编译器。本文档所用的示例是通过标准版本编译而成；而使用专业版本可获得的其他优化功能，我们也进行了介绍。

要使用本文档中说明的技术，您应该了解如何开发和构建 PSoC Designer 项目，并具备 C 编程语言的基础知识。请参考 AN75320，了解 PSoC 1 入门。

开发 PSoC Designer 项目时，如果想要知道该项目占用了多少 ROM 空间，可以在 **Output**（输出）状态窗口中的 **Build**（构建）选项卡查看，如图 1 中所示。在该示例中，所占用的代码空间为 543 个字节。

图 1. 显示 ROM 和 RAM 使用情况的 PSoC Designer 构建信息



下面各章节首先探讨项目级优化的各种设置，然后介绍一些编码指南。

项目级优化

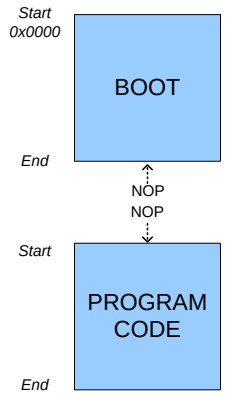
您可以在 **Project > Settings** 下方找到 PSoC Designer 代码优化设置。

第一种设置：可重定位代码的开始地址

构建 PSoC Designer 项目时，ImageCraft 编译器将 C 文件转换为汇编文件。然后，汇编程序将它们转换为可重定位的目标文件。最后，链接器合并这些可重定位的目标文件，以生成可执行的.*hex* 文件。

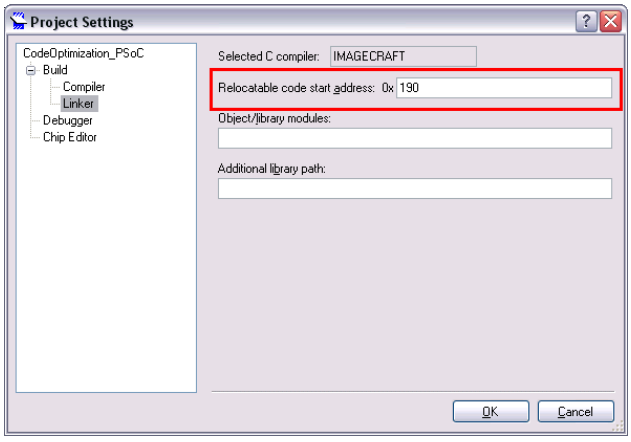
除了您的代码外，.*hex* 文件也包括自动生成的初始化代码（启动代码）。随后便是一系列的 NOP 指令，这些指令能够预留一些闪存字节，以备启动代码需要扩展。请参见图 2。您可以强制将您的代码紧接在启动代码之后，从而节省闪存空间。

图 2. PSoC Designer 项目的闪存映射图



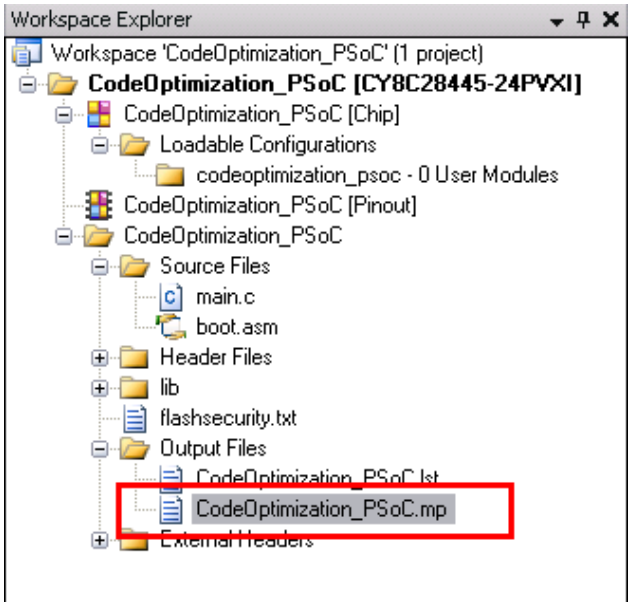
通过 PSoC Designer，您可以选择 **Project > Settings** 来指定代码段的起始地址。在所弹出的窗口中，请选择“Linker”（链接器），如图 3 所示。

图 3. 可重定位的代码起始地址选择



在文本框中输入的地址取决于启动代码的大小，您可以在映射文件（.*mp*）中找到启动代码的大小，如图 4 所示。

图 4. PSoC Designer 中的映射文件



该映射文件显示了不同代码区域的起始地址和结束地址，如图 5 所示。启动代码在“TOP”区域，而您的代码位于“lit”区域。您可以将图 3 中所示的 **Relocatable code start address**（可重定位的代码起始地址）设置为“TOP”区域的结束地址，以便最有效地利用闪存空间。例如，默认的设置是 0x190（“lit”区域的起始地址）。如果您将其更改为 0x151（“TOP”区域的结束地址），您的 ROM 可以增加 63 （0x190 – 0x151 = 0x3F）个字节。

图 5. *.mp* 文件中的“lit”和“TOP”区域

```
Project:      output/CodeOptimization_PSoC
Build Number: 0
Date:         Mon Mar 19 10:19:03 2012
Compiler:     ICCM8C
Version:      7.05.00

Area                              Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  lit    0190   00F2 =    242. bytes (rel,con,rom,lit)

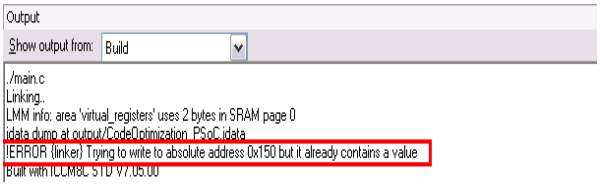
      Addr  Global Symbol
     -----  --------------------------------
      0190  LoadConfigTBL_codeoptimization_psoc_Bank0
      01F3  LoadConfigTBL_codeoptimization_psoc_Bank1
      0282  __lit_end

Area                              Start  End    Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  TOP    0000   0151 =    337. bytes (abs,ovr,rom,code)

      Addr  Global Symbol
     -----  --------------------------------
      0080  __Start
      014E  _bGetPowerSetting
      014E  bGetPowerSetting
```

如果将可重定位的代码起始地址设置为小于“TOP”的结束地址，则系统将显示错误信息。例如，如果将该值设置为 0x150，而启动代码以 0x151 结束，则系统将显示错误信息，如图 6 所示。

图 6. 可重定位的代码起始地址小于启动代码的大小时出现的错误信息



第 2 种设置：配置初始化

代码优化的另一种方法是控制如何在启动时初始化 PSoC 寄存器。可以通过下面两种方法设置寄存器值：**Loop**（循环）或 **Direct write**（直接写入），如图 7 所示。依次选择 **Project > Settings**，然后在出现的窗口中选择“Chip Editor”（芯片编辑器）即可选中该选项。

采用 **Loop** 方法时，系统会创建一个包含寄存器地址和寄存器值的表格，并借助一个函数将表格中的值写入到各自的地址中。采用 **Direct write** 方法时，将直接完成寄存器的写操作，每个寄存器一条 MOV 指令。

选定的方法会在自动生成的文件 *PSOCConfigTBL.asm* 中编写。图 8 显示的是这两种方法的代码。

图 7. 配置初始化选择

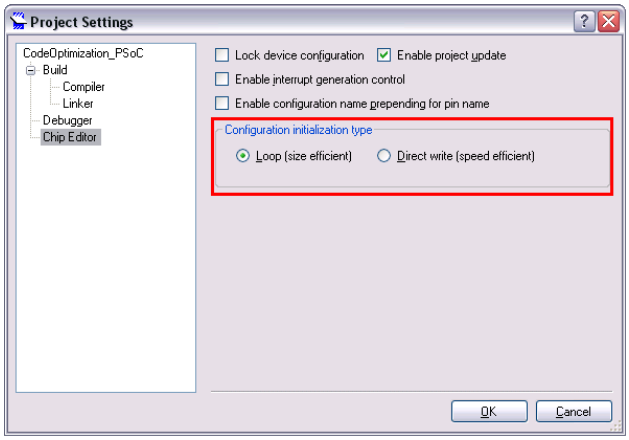


图 8. 循环和直接写入方法的代码差异

```
LoadConfigTBL_codeoptimization_psoc_Bank0:
;  Global Register values Bank 0
    db      60h, 28h        ; AnalogColumn
    db      66h, 00h        ; AnalogCompare
    db      63h, 05h        ; AnalogReferer
    db      65h, 00h        ; AnalogSyncCon
    db      e6h, 00h        ; DecimatorCont
    db      e7h, 00h        ; DecimatorCont
    db      d6h, 00h        ; I2CConfig reg
    db      b0h, 00h        ; Row_0_InputMu
```

Loop

```
LoadConfigTBL_codeoptimization_psoc:
    M8C_SetBank0
;  Global Register values Bank 0
    mov reg[60h], 28h       ; AnalogColumn
    mov reg[66h], 00h       ; AnalogCompar
    mov reg[63h], 05h       ; AnalogRefere
    mov reg[65h], 00h       ; AnalogSyncCo
    mov reg[e6h], 00h       ; DecimatorCon
    mov reg[e7h], 00h       ; DecimatorCon
    mov reg[d6h], 00h       ; I2CConfig re
```

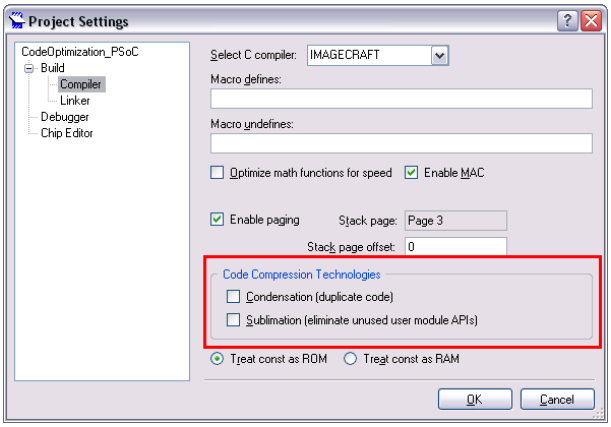
Direct Write

从图 8 中可以看到，在每次寄存器写入时，**Direct write** 方法（MOV reg[expr]，expr = 3 字节）会比使用 **Loop** 方法（addr，数值 = 2 字节）多占用一个字节。但是 **Loop** 方法还包括占用 94 个字节的表格读取函数。因此，如果需要加载超过 94 个寄存器（除最简单的 Designer 项目外，其他项目通常是这种情况），则选择 **Loop** 方法占用的闪存空间会更少。使用图 8 显示的配置来初始化的寄存器数量取决于设计中使用的资源（用户模块）的数量。使用的资源越大，被初始化的寄存器数量也会越多。因此，如果设计增加了许多用户模块以节省闪存，且您愿意花一点时间来启动器件，则请使用 **Loop** 方法。但是如果设计中没有太多的用户模块，或需要快速启动器件，则应该选择 **Direct write** 方法。

第 3 种设置：Sublimation（净化）和 Condensation （缩聚）选项

净化和缩聚是 ImageCraft 标准版提供的压缩技术。依次选择 **Project > Settings**，然后在弹出的窗口中选择“Compiler”（编译器）即可设置这两个选项，如图 9 所示。

图 9. Sublimation 和 Condensation 选项



Sublimation（净化）

当选择 **Sublimation** 选项时，编译器会清除用户模块 API（接口代码）中未使用的函数，以节省闪存空间。

例如，您可以将 PGA 和 PWM 用户模块置于项目中，并为这两个用户模块仅调用“Start”函数。如图 10 和图 11 所示，通过清除这些用户模块 API 中的其他未使用函数，可以节省 142 个字节的闪存空间。

图 10. 未使用净化技术时的 ROM 使用情况

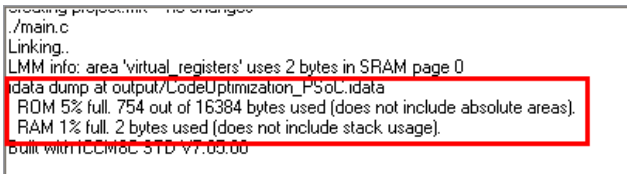
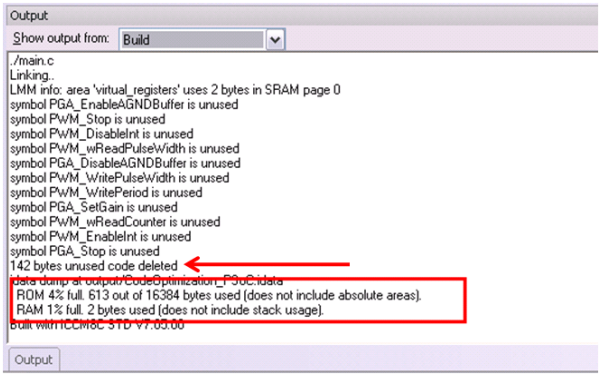
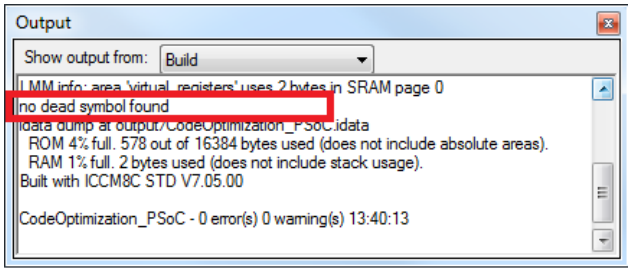


图 11. 使用净化技术时的 ROM 使用情况



如果所有用户模块 API 中的所有函数均用于项目中，则系统将显示“no dead symbol found”（没有发现无效符号）信息，如图 12 所示。

图 12. 使用净化技术时“no dead symbol found”信息



Condensation（缩聚）

当选择 **Condensation**（缩聚）选项时，将生成一个针对项目中重复出现的代码段的函数，且该代码的每个实例均被函数调用所替换。在代码 1 显示的简单示例中，同一代码在项目的不同位置上重复出现三次。使能缩聚功能时，该代码的一个实例将转换为一个函数，无论该实例发生在什么位置，都将被替换为该函数的函数调用，如图 13 所示。创建的所有此类函数/程序都被置于代码的末端且位于“<created procedures>”标签之下，如图 13 所示。您可以看到，针对代码 1 创建的程序正是从“<created procedures>”标签开始的，因此，无论在什么位置使用该实例，实例都将被调用“<created procedures>”标签的指令所替换。“<created procedures>”标签下“0520”和“052B”位置上的函数是项目中另两处重复代码，这些代码被转换成可调用的函数。

代码 1

```
/* Instance one */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

/* Instance two */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];
```

```
/* Instance three */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];
```

图 13..*lst* 文件中针对代码 1 的缩聚

```
1984        0454: 90 C2     CALL   <created procedures>
1985   (0029)
1986   (0030)  /* Instance one */
1987   (0031)  shadowRegs[PORT_2] |= 0x01;
1988   (0032)  PRT2DR = shadowRegs[PORT_2];
1989   (0033)

2050        04AD: 90 69     CALL   <created procedures>
2051   (0060)
2052   (0061)
2053   (0062)  /* Instance two */
2054   (0063)  shadowRegs[PORT_2] |= 0x01;
2055   (0064)  PRT2DR = shadowRegs[PORT_2];

2139   (0108)      /* Instance three */
2140   (0109)      shadowRegs[PORT_2] |= 0x01;
2141        0505: 62 D0 00 MOV    REG[0xD0],0x0
2142        0508: 90 0E     CALL   <created procedures>
2143   (0110)      PRT2DR = shadowRegs[PORT_2];

2218   <created procedures>:
2219        0518: 2E 02 01 OR     [shadowRegs+2],0x1
2220        051B: 51 02     MOV    A,[shadowRegs+2]
2221        051D: 60 08     MOV    REG[0x8],A
2222        051F: 7F        RET
2223        0520: 62 D0 00 MOV    REG[0xD0],0x0
2224        0523: 26 02 FE AND    [shadowRegs+2],0xFE
2225        0526: 51 02     MOV    A,[shadowRegs+2]
2226        0528: 60 08     MOV    REG[0x8],A
2227        052A: 7F        RET
2228        052B: 71 10     OR     F,0x10
2229        052D: 43 08 01 OR     REG[0x8],0x1
2230        0530: 41 09 FE AND    REG[0x9],0xFE
2231        0533: 70 CF     AND    F,0xCF
2232        0535: 7F        RET
```

通过 **Condensation** 选项，您可以节省大量 ROM 空间，如图 17 所示。若想要对代码应用缩聚技术，“text”区域（位于.*mp* 文件中，如图 14 所示）内的代码要超过 256 个字节。如果“text”区域的代码小于 256 个字节，将会显示如下信息：“program code in ‘text’ area too small for worthwhile code compression”（‘text’区域中的程序代码太小，不适合进行代码压缩）（图 15）。如果没有发现任何能够进行缩聚的重复代码实例，将会显示如下信息：“no worthwhile duplicate found”（没有发现适当的重复代码）（图 16）。

注意：图 15、图 16 和图 17 中显示的是针对不同代码的信息，在缩聚代码过程中用于模拟不同的场景。

图 14. *.mp* 文件中的“text”区域

```
Area                                  Addr   Size   Decimal Bytes (Attributes)
------------------------------------  ----   ----   ------- ----- ------------
                                      text   0445   0143 =   323. bytes (rel,con,rom,code)

       Addr  Global Symbol
      -----  --------------------------------
       0445  __text_start
       0445  _main
       0588  __xidata_start
       0559  <created procedures>
       0588  __text_end
       058E  __xidata_end
```

图 15. “text”区域代码小于 256 个字节的缩聚技术

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
program code in 'text' area too small for worthwhile code compression
idata dump at output/CodeOptimization_PSoC.idata
  ROM 6% full. 863 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

图 16. 没有发现重复代码时的缩聚技术

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
no worthwhile duplicate found
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1055 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

图 17. 代码 1 的缩聚技术（代码 1 是一个较大项目的一部分，该项目的 text 区域大于 256 个字节。）

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'data' item of 5 bytes allocated in SRAM page 0
LMM info: area 'virtual_registers' uses 4 bytes in SRAM page 0
374 bytes before Code Compression, 323 after. 12% reduction.
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1063 out of 16384 bytes used (does not include absolute areas).
  RAM 2% full. 9 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

注意：虽然缩聚会明显降低 ROM 空间的使用量，但它会将重复代码代替为函数调用，因此会增加程序执行的延迟。

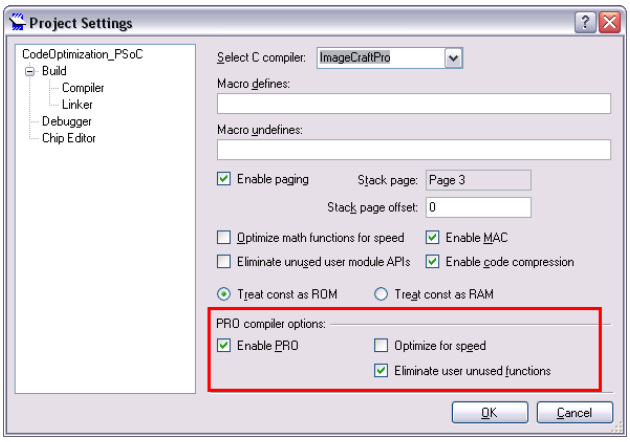
第 4 种设置：Treat const as ROM（将常量视为 ROM）与 Treat const as RAM（将常量视为 RAM）选项

该选项（参考图 9）通过将常量保留在闪存（ROM）内减少 RAM 的使用率；它并不是大幅降低闪存的使用率。大多数项目都使用 **Treat const as ROM** 选项。**Treat const as RAM** 选项主要用于与 ImageCraft 编译器先前版本的向后兼容。

ImageCraft 专业版编译器选项

除了 ImageCraft 标准版编译器提供的优化选项外，ImageCraft 专业版编译器还提供了其他几个选项，如图 18 所示。

图 18. ImageCraft 专业版编译器选项



ImageCraft 标准版中提供的 **Sublimation** 和 **Condensation** 选项分别相对应于 ImageCraft 专业版中的 **Eliminate unused user module APIs**（消除未使用的用户模块 API）和 **Enable code compression**（使能代码压缩功能）。

ImageCraft 专业版还提供了 **Eliminate user unused functions**（消除用户未使用的函数），当用户的项目非常庞大且很难找到用户未使用的函数时，该选项非常有用。简单的示例是项目的版本；未使用的 API 可能仍然保留在代码中，占据了 ROM 空间。勾选该选项后，编译器会确保将这类未使用的 API 从代码中清除，以便将省出的 ROM 空间供其他代码使用。

ImageCraft 专业版还提供另一个选项，即 **Optimize for speed**（优化代码的运行速度）。勾选该选项时，代码将经过编译实现更快的执行速度。ROM 的使用率可增加或减少，具体取决于优化被应用于哪个位置。例如，代码中较小的循环被扩展为顺序指令，这样便提高 ROM 的使用率。冗余的分配或传送操作被移除，这样便降低 ROM 的使用率。

提示和指南

下面的一些编码技术能够使您的固件更为有效。这些方法可以与 PSoC Designer 项目和 PSoC 1 配合使用，且能够应用于类似的 8 位处理器、IDE 和编译器中，比如 PSoC 3 和 PSoC Creator™。

指南 1：避免在中断服务子程序中调用函数

在编译中断服务子程序（ISR）的 C 代码时，ImageCraft 编译器把 ISR 将使用的所有虚拟寄存器（编译器用于存储临时值的寄存器）压入堆栈。如果 ISR 代码中包括函数调用，则编译器无法识别哪些寄存器将被调用的函数修改。ImageCraft C 编译器最多使用 15 个虚拟寄存器，以在堆栈中存储临时数据。

在 RAM 超过 256 个字节的 PSoC 芯片中，4 个页面指针也与这 15 个虚拟寄存器一同存储和恢复。代码 2 中的示例不需要编译器保存任何虚拟寄存器（图 19）。如

代码 3 所示，当使用函数调用执行同一功能时，需要使用额外的 15 个虚拟寄存器，如图 20 所示。每个存储器需要以下 6 个字节，分别是 MOV [2 个字节] + PUSH [1 个字节] + POP [1 个字节] + MOV [2 个字节]。

代码 3 需额外使用 90 个字节，由于有很多 MOV/PUSH 指令，ISR 执行会出现延迟或需要更长时间。

代码 2

```
BYTE bVar1;

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    bVar1 = 1;
}
```

图 19. 为 ISR 生成的代码（代码 2）（无调用函数）

```
(0012) #pragma interrupt_handler SleepTimerHandler;
(0013)
(0014) void SleepTimerHandler(void)
(0015) {
_SleepTimerHandler:
    0445: 71 C0     OR    F,0xC0
    0447: 08        PUSH  A
    0448: 5D D0     MOV   A,REG[0xD0]
    044A: 08        PUSH  A
(0016)      bVar1 = 1;
    044B: 62 D0 00  MOV   REG[0xD0],0x0
    044E: 55 09 01  MOV   [bVar1],0x1
    0451: 18        POP   A
    0452: 60 D0     MOV   REG[0xD0],A
    0454: 18        POP   A
    0455: 7E        RETI
(0017) }
```

代码 3

```
BYTE bVar1;
void TestFunc()
{
    bVar1 = 1;
}

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    TestFunc();
}
```

图 20. 为 ISR 生成的代码（代码 3）（调用函数）

```
990  _SleepTimerHandler:
991     02C7: 08        PUSH  A
992     02C8: 51 10     MOV   A,[__r0]
993     02CA: 08        PUSH  A
994     02CB: 51 0F     MOV   A,[__r1]
995     02CD: 08        PUSH  A
996     02CE: 51 0E     MOV   A,[__r2]
997     02D0: 08        PUSH  A
998     02D1: 51 0D     MOV   A,[__r3]
999     02D3: 08        PUSH  A
1000    02D4: 51 0C     MOV   A,[__r4]
1001    02D6: 08        PUSH  A
1002    02D7: 51 0B     MOV   A,[__r5]
1003    02D9: 08        PUSH  A
1004    02DA: 51 0A     MOV   A,[__r6]
1005    02DC: 08        PUSH  A
1006    02DD: 51 09     MOV   A,[__r7]
1007    02DF: 08        PUSH  A
1008    02E0: 51 08     MOV   A,[__r8]
1009    02E2: 08        PUSH  A
1010    02E3: 51 07     MOV   A,[__r9]
1011    02E5: 08        PUSH  A
1012    02E6: 51 06     MOV   A,[__r10]
1013    02E8: 08        PUSH  A
1014    02E9: 51 05     MOV   A,[__r11]
1015    02EB: 08        PUSH  A
1016    02EC: 51 04     MOV   A,[__rX]
1017    02EE: 08        PUSH  A
1018    02EF: 51 03     MOV   A,[__rY]
1019    02F1: 08        PUSH  A
1020    02F2: 51 02     MOV   A,[__rZ]
1021    02F4: 08        PUSH  A
1022 (0032)     TestFunc();
1023    02F5: 9F CC     CALL  _TestFunc|__text_start|_TestFunc
1024    02F7: 18        POP   A
1025    02F8: 53 02     MOV   [__rZ],A
1026    02FA: 18        POP   A
1027    02FB: 53 03     MOV   [__rY],A
1028    02FD: 18        POP   A
1029    02FE: 53 04     MOV   [__rX],A
1030    0300: 18        POP   A
1031    0301: 53 05     MOV   [__r11],A
1032    0303: 18        POP   A
1033    0304: 53 06     MOV   [__r10],A
1034    0306: 18        POP   A
1035    0307: 53 07     MOV   [__r9],A
1036    0309: 18        POP   A
1037    030A: 53 08     MOV   [__r8],A
1038    030C: 18        POP   A
1039    030D: 53 09     MOV   [__r7],A
1040    030F: 18        POP   A
1041    0310: 53 0A     MOV   [__r6],A
1042    0312: 18        POP   A
1043    0313: 53 0B     MOV   [__r5],A
1044    0315: 18        POP   A
1045    0316: 53 0C     MOV   [__r4],A
1046    0318: 18        POP   A
1047    0319: 53 0D     MOV   [__r3],A
1048    031B: 18        POP   A
1049    031C: 53 0E     MOV   [__r2],A
1050    031E: 18        POP   A
1051    031F: 53 0F     MOV   [__r1],A
1052    0321: 18        POP   A
1053    0322: 53 10     MOV   [__r0],A
1054    0324: 18        POP   A
1055    0325: 7E        RETI
```

指南 2：限制数学函数

在许多情况下，C 编译器将通过内联汇编代码执行简单的数学计算。然而，对于复杂的计算，编译器可能向您的代码中添加一个或多个数学库函数。这样操作的优势可能不会立即被体会到。根据运算和变量类型，即使是‘+’、‘-’、‘*’、‘/’、‘%’、‘>’和‘<’等简单的 C 语言运算符也需要库函数才能实现。

尽管库函数非常有效而且实用，但是它们会占用您无法预计的大量代码空间。在许多情况下，通过谨慎地管理代码中的数据类型和运算，您可以避免过度使用库函数。

当程序使用整数算术运算符时，根据使用的变量大小和类型（8、16 或 32 位，有符号数或无符号数）添加数学库函数。有关不同函数所用字节数的详细信息，请依次选择 **Help > Documentation > Designer Specific Documents** 查看 *Libraries User Guide*（库用户指南）。

基础的算术函数（即最常用的函数）包括整数的加、减和移位。当使用其他运算（如乘法）时，针对这些运算的代码也包含在内。

位移叠加法取代乘法或除法

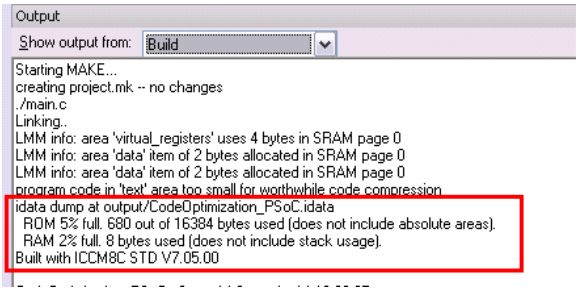
位移位叠加法等技巧取代无符号整数的乘法或除法，从而节省代码空间。在无符号整数中，单一按位右移等于除以 2，而左移等于乘以 2。如下面的代码 5 所示，通过使用位移叠加法，您可以避免使用乘法和除法库函数。

在以下输出相同结果的两个代码段中，执行代码 4 比执行代码 5 多使用 55 个字节，如图 21 和图 22 所示。这个差别是由于向代码中添加了一个“__mul16”函数。

代码 4

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = iTest2 *3;
}
```

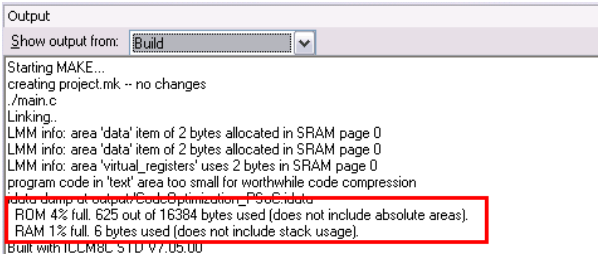
图 21. 代码 4 的 ROM 使用情况



代码 5

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = (iTest2 << 1) + iTest2;
}
```

图 22. 代码 5 的 ROM 使用情况



避免幂函数

幂函数也会添加数学库函数。可以将幂运算简化为乘法或除法运算，例如：$4^2 = 4 \times 4$。 正如前面部分所阐述的内容，乘法或除法运算可以使用移位叠加法来实现。这样，对于有少量幂函数的程序，通过尽可能将幂函数简化为移位叠加法来进行运算，您可以节省程序的代码空间。

避免浮点运算

使用 8 位处理器的浮点运算几乎始终需要库函数。除了基本数学运算函数外，舍入、归一化和检查特殊条件等效用函数也会被添加到代码中。请参考*算术库用户指南*，了解有关浮点运算的更详细信息。为了使您能够进行估算，请参考下面浮点函数所需的代码空间（同样可以在*算术库用户指南*中找到）：

比较（*_fpcmp） = 109 – 125 个字节

加（*_fpadd） = 461 – 478 个字节

减（*_fpsub） = 468 – 485 个字节

乘（*_fpmul） = 406 – 558 个字节

除（*_fpdiv） = 432 – 449 个字节

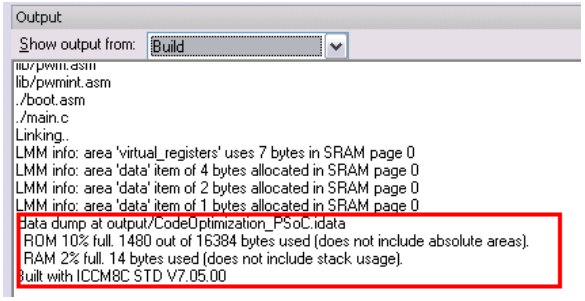
浮点数学函数将整数数学库作为基础使用。这指的是，使用浮点运算时也需要整数数学库。如果变量的范围已知，可以将变量调整为直接使用整数运算，从而代替浮点运算。

例如，在以下两个代码段中，已知的变量范围是小数点后两位。因此，如果您将所有浮点数乘以 100，您便可以使用整数运算来代替浮点运算。在下面示例中，代码 7 中的整数运算法比代码 6 中的等效浮点运算少占用 761 个字节，如图 23 和图 24 所示。

代码 6

```
int iTest2, iTest3;
float fTest1;
void main(void)
{
    fTest1 = iTest2 * 2.42;
    if(fTest1 > 7.5)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

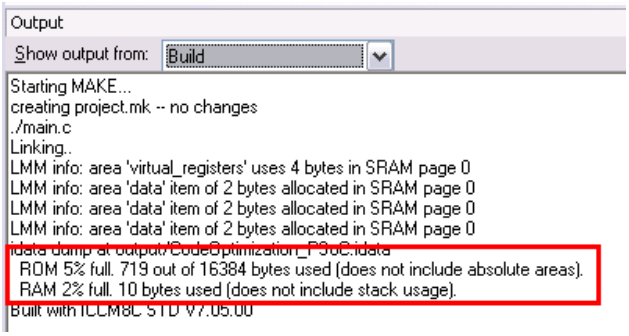
图 23. 代码 6 的 ROM 使用情况



代码 7

```
int iTest1, iTest2, iTest3;
void main(void)
{
    iTest1 = iTest2 * 242;
    if(iTest1 > 750)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

图 24. 代码 7 的 ROM 使用情况



使用查找表（LUT）取代计算

有时候，使用查找表（LUT）取代计算效果更高。这里存在多个权衡因素，比如：速度、精度和代码空间。您的选择取决于应用的类型。

例如，AN2017 – PSoC 1 通过热敏电阻测量温度中的项目提供了浮点算法和 LUT 方法两个选项。在该项目中，使用 LUT 取代浮点运算可节省 4187 个字节的存储器空间（使用查询表时会占用 3779 个字节，而使用浮点方程时会占用 7966 个字节），但精度会有所降低。

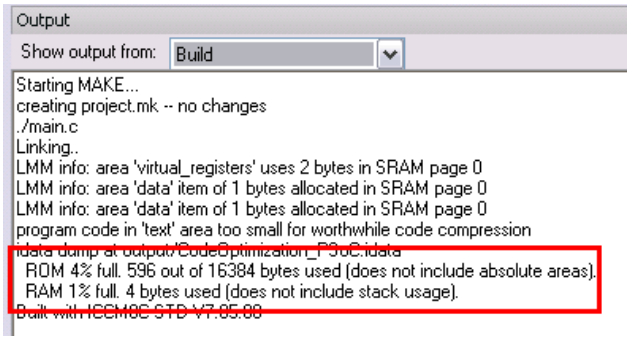
维护数据类型的一致性

当选择变量的数据类型时，请选择该函数所需的最小数据类型。例如，如果变量的最大值不超过 256，则选择的单位应该是 BYTE（字节）而不是 int（整型）。同样，在适用情况下，选择无符号的变量而不是有符号的变量。例如，请参考代码 4 如果在程序中变量“iTest1”和“iTest2”都不超过 255，则可将其声明为“unsigned char”（无符号字符）而非“unsigned int”（无符号整数），如代码 8 所示。 图 25 显示的是 ROM 的使用情况。代码 8 比代码 4 少占用了 84 个字节。

代码 8

```
unsigned char cTest1, cTest2;
void main(void)
{
    cTest1 = cTest2 *3;
}
```

图 25. 代码 8 的 ROM 使用情况



如果您知道有符号变量的正负范围，可以使变量偏移至正值，并使用无符号的数学函数。同样，如果预计变量的值为–10 到+10 之间，则使变量偏移 10，这样便可以使代码 0 对应–10，代码 20 对应+10。

从代码空间的角度来说，类型转换占用的空间非常大，因此，您需要维护所用数据类型的一致性。如果您必须在程序中使用类型转换，那么可以通过变量类型转换来使用相同的数学库，从而节省数学库的代码空间。

指南 3：使用数组索引和指针

在数组索引法中，ImageCraft 直接使用索引访问每个地址。由于地址保持不变，因此无需计算。在使用指针访问的情况下，每次访问都是以指针变量为基础。因此，编译器必须执行更多的计算，才能获取地址。

如果重复使用这种访问方法，访问类型的差别导致存储器使用率的极大差异。

例如，考虑以下两个代码示例。代码 9 比代码 10 所占用 25 个字节，如图 26 和图 27 所示。因此，仔细观察使用的访问方法类型（数组索引还是指针）对于代码优化至关重要。访问类型和变量类型中存在大量的差异。关键要确保地址是常量，在这种情况下，编译器会直接替换地址。当运行过程中必须计算地址时，编译器可使用更多的代码空间。一般情况下，从代码空间的角度来说，使用“->”运算符占用的空间更大。

代码 9

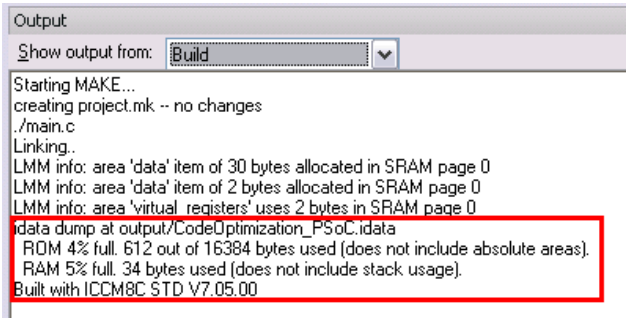
```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myPtr = myTest.myArray;
    myPtr->iData = 100;
}
```

图 26. 代码 9 的 ROM 使用情况



代码 10

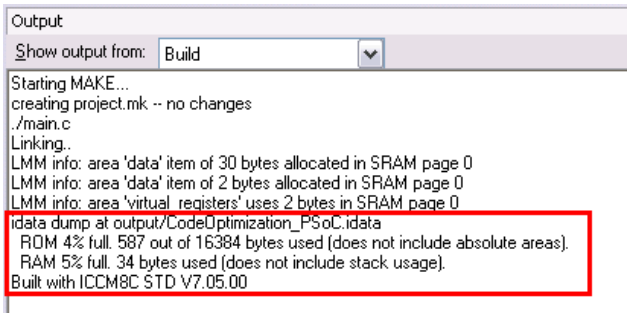
```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myTest.myArray[1].iData = 100;
}
```

图 27. 代码 10 的 ROM 使用情况



指南 4：使用 switch 语句和 If-Else 语句

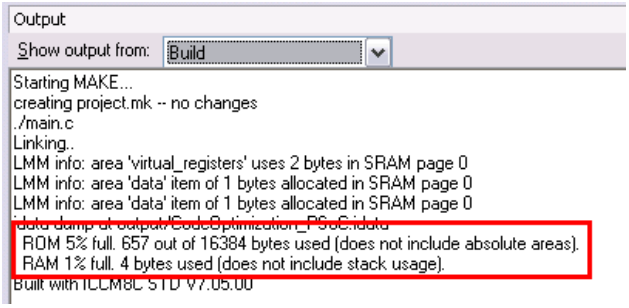
要实现 case 决策，您可以使用 switch 语句或 if-else 语句。这两种语句之间的不同在于，switch 语句始终进行 16 位比较，而 if-else 语句是根据变量类型进行比较。如果是单字节变量（BYTE），与 switch 结构相比，ImageCraft 编译器使用 if-else 结构生成的代码效率更高。这是因为 8 位比较比 16 位比较需要的代码空间少。

Switch 语句使用 9 个额外的字节，外加每个 case 的 5 个字节。例如，代码 11 中显示的带有默认字句的 4 种 case 的 switch 语句比等效代码 12 多占用（9 + 4 × 4）= 25 个字节，如图 28 和图 29 所示。

代码 11

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            bTest2 = 1;
            break;
        }
        case 3:
        {
            bTest2 = 2;
            break;
        }
        case 2:
        {
            bTest2 = 3;
            break;
        }
        default:
        {
            bTest2 = 4;
        }
    }
}
```

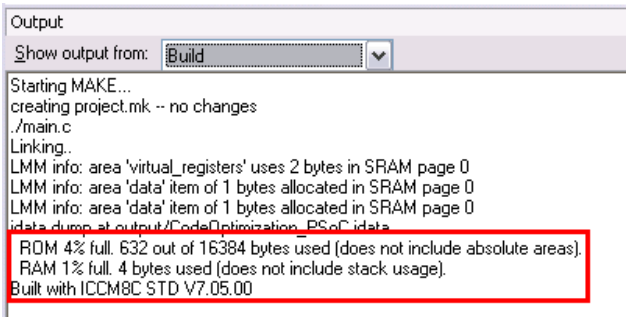
图 28. 代码 11 的 ROM 使用情况



代码 12

```
BYTE bTest1, bTest2;
void main(void)
{
    if(bTest1 == 4)
    {
        bTest2=1;
    }
    else if(bTest1 == 3)
    {
        bTest2 = 2;
    }
    else if(bTest1 ==2)
    {
        bTest2 = 3;
    }
    else
    {
        bTest2=4;
    }
}
```

图 29. 代码 12 的 ROM 使用情况



如果 case 决策针对的是双字节变量（WORD），两种表达式得出的代码大小几乎相同。

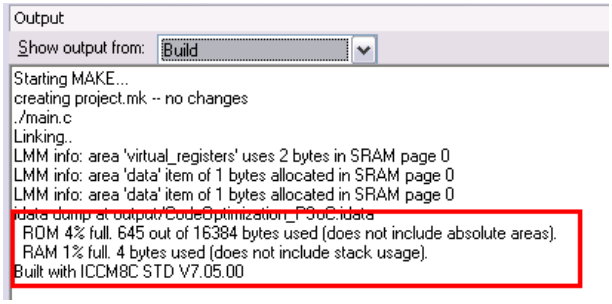
指南 5：将部分代码写入汇编器中

在汇编器中写入程序可避免编译器解译，从而实现全面的用户优化。但是在汇编器中写入完整的程序非常繁琐复杂，因此只将部分代码转换为汇编语言便可以优化代码的大小和性能。请考虑代码 11，参考汇编语言的“bTest2”变量，如代码 13 所示。代码 13 比代码 11 少占用 12 个字节，如图 30 所示。

代码 13

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            asm("MOV [_bTest2], 0x1");
            break;
        }
        case 3:
        {
            asm("MOV [_bTest2], 0x2");
            break;
        }
        case 2:
        {
            asm("MOV [_bTest2], 0x3");
            break;
        }
        default:
        {
            asm("MOV [_bTest2], 0x4");
        }
    }
}
```

图 30. 代码 13 的 ROM 使用情况



指南 6：在 PSoC 1 中的位操作

在 PSoC 1 中，不会定义任何“bit”（位）变量的数据类型。要想进行变量的（各）位操作，可以使用一个掩码和逻辑运算符，或直接将数值分配给该变量。根据操作的要求，掩码可以是一个常量。代码 14 和代码 15 显示了如何在一个变量中进行（各）位的操作。

代码 14

```
BYTE bTest1, bTest2, bTest3;
void main(void)
{
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = (0xA1);

  /* To clear the second and the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
}
```

如果变量具有一个以上的位定义，并且必须进行一位或多个位的操作而不影响变量中的其他位，那么请使用逻辑运算符来进行位操作。这些操作使用 XOR、AND 和 OR 指令。 代码 14 的相应*.asm* 代码在（通过依次选择 **Workspace > Output Files** 的）*<projectname>.lst* 文件中生成，如下所示。

```
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = bTest1 | (0xA1);
  __text_start|_main|_main:
  62 D0 00 MOV   REG[0xD0],0x0
  2E 03 A1 OR    [bTest1],0xA1

  /* To clear the second and the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);
  62 D0 00 MOV   REG[0xD0],0x0
  26 02 BD AND   [bTest2],0xBD

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
  62 D0 00 MOV   REG[0xD0],0x0
  51 04    MOV   A,[bTest3]
  31 92    XOR   A,0x92
  53 04    MOV   [bTest3],A
  address: 8F FF    JMP   address
```

注意: 指令 MOV REG[0xD0],0x0 用于设置“bTest1/bTest2/bTest3”全局变量的 RAM 页码。

如果变量的大小为单比特，或正在对变量中定义的所有位同时进行分配/位操作，则请使用直接分配方法来实现，如代码15所示。此时会使用MOV指令，它比XOR、AND和OR指令快一个周期。

注意: 要想知道某一指令需要使用来执行操作的周期数，请依次选择 **Help > Documentation > Compiler and Programming Documents**，参考 PSoC Designer 中“附录 A5. 指令集总结”的*汇编语言用户指南*部分。

代码15

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;

 /* To invert the variable */
 bTest2 = ~ (bTest2);
}
```

相应的*.asm*代码如下显示：

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;
  __text_start|_main|_main:
  62 D0 00 MOV   REG[0xD0],0x0
  55 02 04 MOV   [bTest1],0x4

  /* To invert the variable */
  bTest2 = ~ (bTest2);
  62 D0 00 MOV   REG[0xD0],0x0
}
  51 03    MOV   A,[bTest2]
  73       CPL   A
  53 03    MOV   [bTest2],A
  address: 8F FF    JMP   address
```

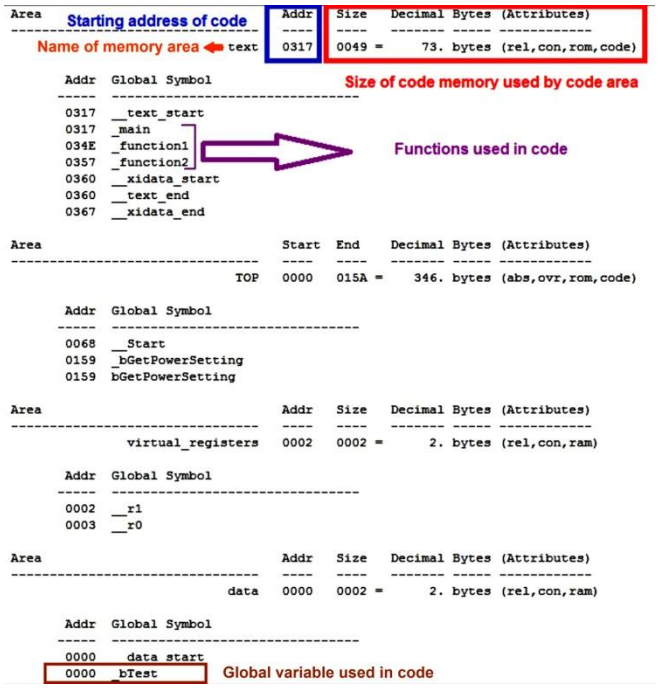
注意: 指令 MOV REG[0xD0],0x0 用于设置变量“bTest1/bTest2”的 RAM 页码。

指南 7：计算 C 代码闪存的使用率和执行时间

当需要考虑代码存储器的使用量和程序执行的时间时，掌握某个函数或代码段所占用的闪存空间大小以及执行时间，对手动优化代码是十分有用的。

要确定某个函数的代码大小，请参考 **Workspace > Output Files** 中的*<projectname>.mp* 文件。*.mp* 文件提供了有关代码区域所占用的代码存储器空间以及在该区域中所定义的函数/常量的详细信息。该文件还提供了 RAM 区域被使用的详细情况，以及这些区域中所定义的全局变量地址，如图 31 所示。

图 31..*mp* 文件中的地址和存储器详情



注意: 欲了解有关代码/RAM 区域的详细信息，请参考*《C 语言编译器用户指南》*中第 6.9 部分，以及*《汇编语言用户指南》*中第 5.1 部分。

要确定某个函数的代码大小，请在*.mp* 文件中搜索该函数名称。如果在自定义区域中尚未取代该函数，那么用户定义的所有函数均默认位于“text”（文本）代码区域内（图 32）。图 32 显示的是代码 16 中定义的“main”、“function1”和“function2”的起始地址。要想确定“main”的代码大小，请使用“main”的地址减去下面函数（这里是“function1”）的地址。“main”的大小为 55 个字节（0x034E – 0x0317 = 0x0037），“function1”的大小为 9 个字节（0x0357 – 0x034E = 0x0009），“function2”的大小为 9 个字节（0x0360 – 0x0357 = 0x0009）。

代码 16

```
int bTest;
void main(void)
{
bTest = ++bTest;
}

void function1(int a)
{      a++;
       return;
}

void function2(int b)
{      b--;
       return;
}
```

图 32. 使用.*mp* 文件计算闪存的使用情况

```
Area                              Addr   Size
--------------------------------  ----   ----
                            text  0317   0049

  Addr  Global Symbol
  ----  --------------------------------
  0317    text start
  0317  _main
  034E  _function1
  0357  _function2
  0360    xidata start
  0360  __text_end
  0367  __xidata_end
```

若要计算代码执行的时间，您必须知道各条指令的执行时间。要想知道各条指令的执行时间，请依次选择 **Help > Documentation > Compiler and Programming Documents**，参考 PSoC Designer 中“附录 A5. 指令集总结”的*汇编语言用户指南*部分。 列表中的“CPU clock cycles”（CPU 时钟周期）列提供了执行指令所需的 CPU 时钟周期数。通过将该值乘以 CPU 周期，可得到一条指令的执行时间。下面各示例显示了计算不同代码的执行时间的方法。

示例 1：如下使用一个空白的“while (1)”循环。

```
void main(void)
{
while (1)
       {
       }
}
```

空白“while (1)”循环的相应.*asm* 代码在 *<projectname>.lst* 文件中生成，具体如下：

```
while (1)
{
}
address: 8X XX     JMP    address
```

执行JMP指令需要5个CPU周期。如果CPU时钟频率为24 MHz，则执行JMP指令会需要：

$$5 \times \frac{1}{24}\ \mu s = 0.208\ \mu s$$

在该速率下，CPU每0.208 *µs*执行一条（JMP）指令，即其速度为(1/0.208) = 4800万条指令每秒。

示例2：
请参考下面示例作为简单的加法。

```
void main(void)
{
BYTE bTest1, bTest2;
while(1)
{
       bTest1 = bTest1 + bTest2;
}
}
```

相应的.*asm*代码如下显示：

```
BYTE bTest1, bTest2;
while(1)
{
       bTest1 = bTest1 + bTest2;
    address1: 52 01     MOV    A,[X+1]
    address2: 05 00     ADD    [X+0],A
}
    address3: 8X XX     JMP    address1
```

执行先前代码总共需要6（MOV） + 8（ADD） + 5（JMP） = 19个时钟周期。这样，平均需要19/3 = 6.33个周期来执行每一条指令。如果CPU时钟频率为24 MHz，则执行整个代码会需要：

$$\frac{19}{24{,}000{,}000} s = 0.792\ \mu s$$

按照该频率，CPU在每0.792/3 = 0.264 *µs*执行一条指令，即（1/0.264） = 3790万条指令一秒。

结论

这些指南能够帮助您优化代码。其中，一些指南专门针对 ImageCraft 编译器，而一些指南则普遍使用。在固件开发期间和后续工作中，遵循这些建议有助于优化代码空间。所有代码段都在 CY8C28xxx 器件和 ImageCraft 标准编译器

版本 7.0.5 中都经过了测试，并捕捉了屏幕快照。 其他 PSoC 1 器件得出的结果也与此类似。

遵照这些指南的同时，当操作更小闪存空间的器件时，在项目开发期间关注代码大小的增加情况是很好的工作方法。一旦代码空间出现大幅增加，请查看映射文件（图 4），以检查最新的代码是否正在运行任何预料之外的函数。如果您熟知引起闪存空间增加的代码的精确位置，优化代码便会更加容易。

更多有关编译器和项目优化选项的详细信息，请参考 *ImageCraft C 编译器指南*和 *ImageCraft 汇编语言指南*。

关于作者

姓名：	Archana Yarlagadda
职务：	应用工程师
背景信息：	赛普拉斯应用工程师，重点工作领域是 PSoC。 诺克斯维尔田纳西大学模拟超大规模集成电路（VLSI）硕士学位
联系方式：	msur@cypress.com

文档修订记录

文档编号：PSoC® 1 M8C ImageCraft C 代码优化 — AN60486

文档编号：001-82493

版本	ECN	变更者	提交日期	变更说明
**	3730495	ZCLI	08/31/2012	本文档版本号为 Rev**，译自英文版 001-60486 Rev*B。
*A	4546205	SNYQ	10/20/2014	本文档版本号为 Rev*A，译自英文版 001-60486 Rev*E。

全球销售和设计支持

赛普拉斯公司拥有一个由办事处、解决方案中心、工厂代表和经销商组成的全球性网络。如果想要查找离您最近的办事处，请访问赛普拉斯所在地。

产品

汽车级产品	cypress.com/go/automotive
时钟与缓冲器	cypress.com/go/clocks
接口	cypress.com/go/interface
照明和电源控制	cypress.com/go/powerpsoc cypress.com/go/plc
存储器	cypress.com/go/memory
PSoC	cypress.com/go/psoc
触摸感应	cypress.com/go/touch
USB 控制器	cypress.com/go/usb
无线/射频	cypress.com/go/wireless

PSoC®解决方案

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

赛普拉斯开发者社区

社区 | 论坛 | 博客 | 视频 | 培训

技术支持

cypress.com/go/support

PSoC 是赛普拉斯半导体公司的注册商标。PSoC Creator 和 PSoC Designer 是赛普拉斯半导体公司的商标。此处引用的所有其他商标或注册商标归其各自所有者所有。

赛普拉斯半导体
198 Champion Court
San Jose, CA 95134-1709

电话号码 ：408-943-2600
传真 ：408-943-4730
网站地址 ：www.cypress.com

©赛普拉斯半导体公司，2010-2014。此处，所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品内嵌的电路外，赛普拉斯半导体公司不对任何其他电路的使用承担任何责任。也不会以明示或暗示的方式授予任何专利许可或其他权利。除非与赛普拉斯签订明确的书面协议，否则赛普拉斯产品不保证能够用于或适用于医疗、生命支持、救生、关键控制或安全应用领域。此外，对于可能发生运转异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。
该源代码（软件和/或固件）均归赛普拉斯半导体公司（赛普拉斯）所有，并受全球专利法规（美国和美国以外的专利法规）、美国版权法以及国际条约规定的保护和约束。赛普拉斯据此向获许可者授予适用于个人的、非独占性、不可转让的许可，用以复制、使用、修改、创建赛普拉斯源代码的派生作品、编译赛普拉斯源代码和派生作品，并且其目的只能是创建自定义软件和/或固件，以支持获许可者仅将其获得的产品依照适用协议规定的方式与赛普拉斯集成电路配合使用。除上述指定的用途外，未经赛普拉斯的明确书面许可，不得对此类源代码进行任何复制、修改、转换、编译或演示。
免责声明：赛普拉斯不针对此材料提供任何类型的明示或暗示保证，包括（但不仅限于）针对特定用途的适销性和适用性的暗示保证。赛普拉斯保留在不做出通知的情况下对此处所述材料进行更改的权利。赛普拉斯不对此处所述之任何产品或电路的应用或使用承担任何责任。对于合理预计可能发生运转异常和故障，并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。
产品使用可能受适用于赛普拉斯软件许可协议的限制。