AN60486

PSoC® 1 M8C ImageCraft C 代码优化

作者：**Archana Yarlagadda**
相关工程项目：无
相关器件系列：**CY8C2xxxx**（所有 **PSoC 1** 器件）
软件版本：**PSoC Designer™ 5.2**
相关应用笔记：**AN60630**、**AN2017**

如果您有任何问题，或者需要获得本应用笔记的相关帮助，请通过 **msur@cypress.com** 联系本文作者

摘要

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，本应用笔记还提供了高效编码的一些指南。本应用笔记假设读者熟悉 PSoC 1、PSoC Designer IDE 和 C 语言编程。要获得有关 PSoC 入门的信息，请访问此处。

目录

简介 1
项目级优化 2
设置 1：可重定位的代码起始地址 2
设置 2：配置初始化方法....3
设置 3：Sublimation 和 Condensation....3
设置 4：Treat const as ROM 与 Treat const as RAM ..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：数组索引与指针....11
指南 4：Switch 语句与 IfElse 12
指南 5：初始化全局变量....13
指南 6：汇编器中的部分代码 13
结论 13

简介

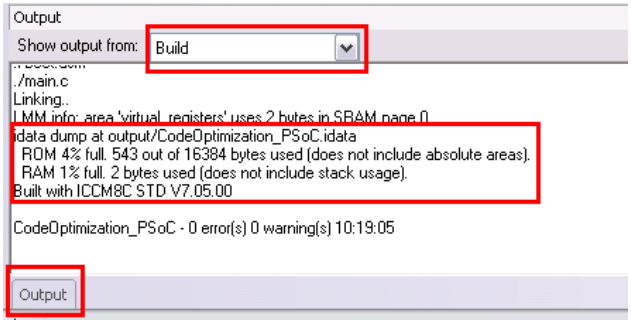
您可以通过多种方法减少 PSoC Designer 项目中使用的闪存 (ROM) 容量。这样，有时您便可以使用更小的 PSoC 器件，从而节省成本。

Designer 中包含 ImageCraft 编译器“标准”版本；如需购买具有更多优化功能的“专业”版本，请点击此处。本应用笔记中包含的代码段适用于这两种编译器。本文档所用示例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，我们将在本文档的稍后部分进行介绍。

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 语言编程的基础知识。请参考此链接，了解 PSoC 的入门信息。

在开发 PSoC Designer 项目时，如果您想要知道该项目占用了多少 ROM 空间，可在 **Output**（输出）状态窗口中的 **Build**（构建）选项卡下方查看。如图 1 所示，此示例中所占用的代码空间是 543 个字节。

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



以下章节将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

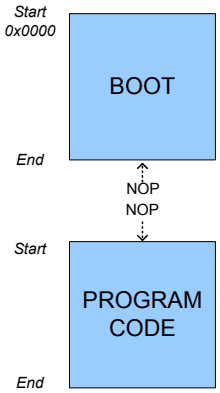
您可以在 **Project（项目）> Settings（设置）**下方找到 PSoC Designer 代码优化设置。

设置 1：可重定位的代码起始地址

在构建 Designer 项目时，ImageCraft 编译器将 C 文件转换为汇编文件。然后，汇编器将这些文件转换为可重定位的目标文件。最后，连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。

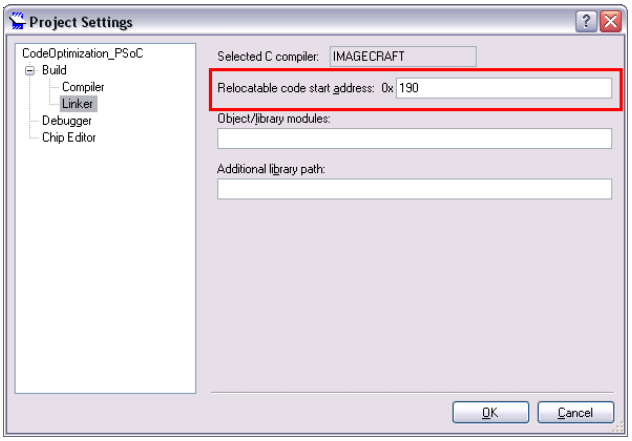
除了您的代码外，*.hex* 文件也包括自动生成的初始化代码（启动代码）。随后便是一系列的 NOP 指令，这些指令能够预留一些闪存字节，以备启动代码需要扩展。请参见图 2。您可以强行将您的代码紧接在启动代码之后，从而节省闪存空间。

图 2. PSoC Designer 项目的闪存映射图



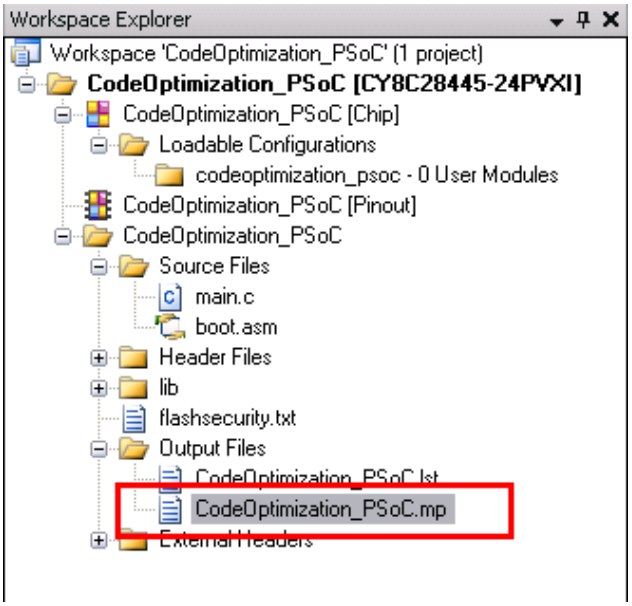
您可以通过 PSoC Designer 指定代码段的起始地址。此选项位于 **Project（项目）> Settings（设置）> Linker（连接器）**下。图 3 显示了相关的弹出窗口。

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根据启动代码的大小而启动，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**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 0xD1（“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
```

如果可重定位的代码起始地址设置的值小于“TOP”的结束地址，则系统将显示错误消息。例如，如果您将此值设置为 0x150，而启动代码以 0x151 结束，系统将显示如图 6 所示的错误消息。

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消息

```
Output
Show output from: Build
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
idata dump at output/CodeOptimization_PSoC.idata
!ERROR (linker) Trying to write to absolute address 0x150 but it already contains a value
Built with ICCM8C STD V7.05.00
```

设置 2：配置初始化方法

代码优化的另一种方法是在启动时初始化 PSoC 寄存器。可以通过两种方法设置寄存器值：**循环**或**直接写入**，如图 7 所示。

采用**循环**方法，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**直接写入**方法，将直接完成寄存器写入操作，每个寄存器一条 MOV 指令。

选定的方法会被编写在自动生成的文件 *PSOCConfigTBL.asm* 中。图 8 显示了这两种方法的代码。

图 7. 配置初始化选择

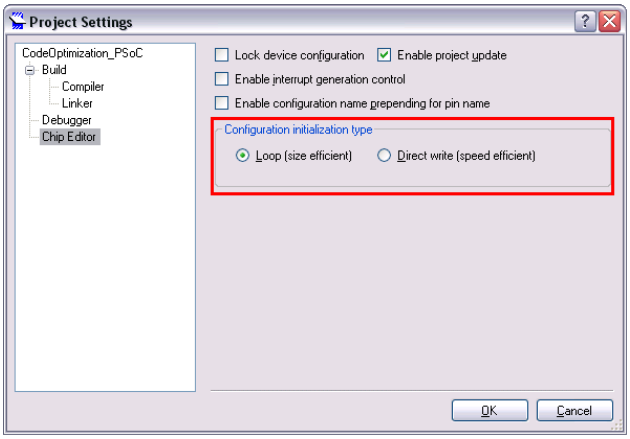


图 8. 循环和直接写入方法的代码差异

```
LoadConfigTBL_codeoptimization_psoc_Bank0:
;  Global Register values Bank 0
    db      60h, 28h        ; AnalogColumn
    db      66h, 00h        ; AnalogCompara
    db      63h, 05h        ; AnalogReferen
    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       ; AnalogCompara
    mov reg[63h], 05h       ; AnalogReferen
    mov reg[65h], 00h       ; AnalogSyncCon
    mov reg[e6h], 00h       ; DecimatorCont
    mov reg[e7h], 00h       ; DecimatorCont
    mov reg[d6h], 00h       ; I2CConfig reg
```

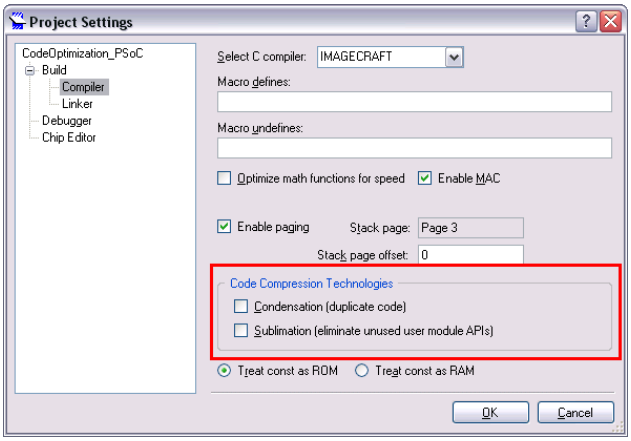
Direct Write

从图 8 中您可以看到，在每次寄存器写入时，直接写入方法（MOV reg[expr], expr = 3 个字节）要比循环方法（addr, value = 2 个字节）多占用一个字节。但是循环方法还包括占用 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），选择循环方法占用的闪存空间更少。

设置 3：Sublimation和Condensation

净化和缩聚是 ImageCraft 标准版提供的压缩技术。您可以在 **Project（项目）> Settings（设置）> Compiler（编译器）** 下设置这两种技术；相关弹出窗口如图 9 所示。

图 9. Sublimation 和 Condensation 选择



Sublimation（净化）

如果您选择 **Sublimation**（净化）选项，编译器将删除用户模块 API（接口代码）中未使用的函数，以节省空间。

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 中的未使用函数，可以节省 142 个字节的闪存空间。

图 10. 未使用 Sublimation 技术时的 ROM 使用情况

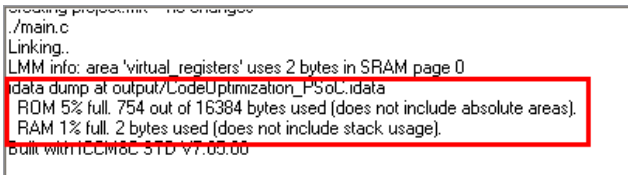
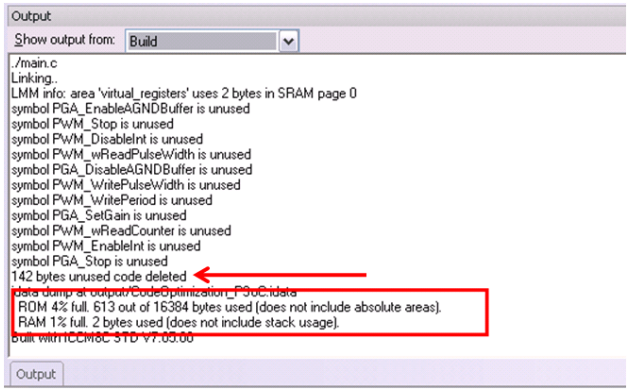


图 11. 使用 Sublimation 技术时的 ROM 使用情况



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）消息。

Condensation（缩聚）

如果您选择 **Condensation**（缩聚）选项，将生成一个针对项目中重复出现的代码段的函数，且此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。启用 condensation 时，此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 12 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 12 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>”标签的命令所替换。“<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 2 */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

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

图 12. “*.lst*”文件中针对代码 1 的 condensation

```
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 16 所示。如果您想对代码应用 condensation，“text”区域（位于“*.mp*”文件中，如图 13 所示）内的代码应该超过 256 个字节。如果“text”区域小于 256 个字节，将会显示如下消息：*“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 14）。如果没有发现能够进行 condensation 的重复代码实例，将会显示如下消息：*“no worthwhile duplicate found”*（没有发现适当的重复代码）（图 15）。

图 13. “*.mp*”文件中的“text”区域

```
80
81  Area                                  Addr   Size   Decimal Bytes (Attributes)
82  --------------------------------      ----   ----   ------- ----- ------------
83                                  text   0445   0143 =   323. bytes (rel,con,rom,code)
84
85        Addr  Global Symbol
86        ----  -------------------------------
87        0445  __text_start
88        0445  _main
89        0588  __xidata_start
90        0559  <created procedures>
91        0588  __text_end
92        058E  __xidata_end
93
```

图 14. “text”区域小于 256 个字节时的 condensation 消息

```
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 15. 没有发现重复代码时的 condensation 消息

```
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 16. 代码 1 的 condensation （代码 1 是一个大项目的一部分，该大项目大于 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
```

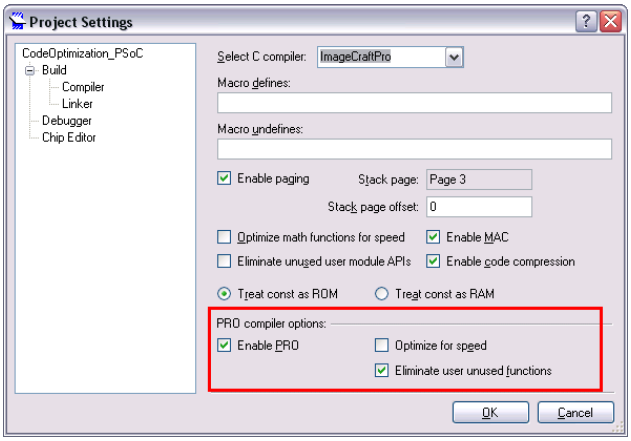
设置 4：Treat const as ROM 与 Treat const as RAM

这一选项（参见图 9）实际上是通过将常量保留在闪存 (ROM) 中来减少 RAM 使用率；并不是大幅降低闪存的使用率。大多数项目使用 **Treat const as ROM**（将常量视为 ROM）选项；**Treat const as RAM**（将常量视为 RAM）选项主要用于与 ImageCraft 编译器之前版本的向后兼容。

ImageCraft 专业版编译器选项

除了 ImageCraft 标准版编译器中提供的优化选项外，ImageCraft 专业版编译器还提供了其他几个选项，如图 17 所示。

图 17. ImageCraft 专业版编译器选项



ImageCraft 标准版中提供的“Sublimation”和“condensation”选项分别对应 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”（启用代码压缩）选项。

ImageCraft 专业版提供了“Eliminate user unused functions”（消除用户未使用的函数）选项，当用户的项目非常庞大且用户未使用的函数很难找到时，此选项非常有用。一个简单的示例是，项目正在进行修改，且之前未使用的 API 可能仍然保留在代码中，占据着 ROM 空间。启用此选项后，编译器将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 专业版提供的另一个选项是优化代码的运行速度。通过启用此选项，代码将经过编译实现更快的执行速度。启用此选项后，ROM 使用率可能有所提高也可能降低，具体取决于应用优化的位置。例如，代码中较小的循环将被扩展为顺序指令，这样便能够提高 ROM 的使用率。冗余的分配或传送操作被移除，这样便降低了 ROM 的使用率。

提示和指南

以下是一些能够使您的固件更为有效的编码技术。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 编译器将 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，因此便将所有寄存器压入堆栈。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 18）。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 19 所示。每个寄存器需要以下这些额外的间接代码：每个存储器总共 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]。因此，代码 3 需额外使用 90 个字节，由于有很多这样的 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 18. 为无函数调用的 ISR 生成的代码（代码 2）

```
958  (0012) #pragma interrupt_handler SleepTimerHandler;
959  (0013)
960  (0014) void SleepTimerHandler(void)
961  (0015) {
962  _SleepTimerHandler:
963      0445: 71 C0     OR    F,0xC0
964      0447: 08        PUSH  A
965      0448: 5D D0     MOV   A,REG[0xD0]
966      044A: 08        PUSH  A
967  (0016)      bVar1 = 1;
968      044B: 62 D0 00  MOV   REG[0xD0],0x0
969      044E: 55 09 01  MOV   [bVar1],0x1
970      0451: 18        POP   A
971      0452: 60 D0     MOV   REG[0xD0],A
972      0454: 18        POP   A
973      0455: 7E        RETI
974  (0017) }
```

代码 3

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

#pragma interrupt_handler
SleepTimerHandler;

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

图 19. 为包含函数调用的 ISR 生成的代码（代码 3）

```
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刻显现。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，但是库函数会占用大量您无法预计的代码空间。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。

当程序使用整数运算时，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，请查看“Libraries User Guide”（库用户指南），该文件位于 **Help（帮助）> Documentation（文档）> Designer Specific Documents（Designer 专用文档）**之下。

基础的算术函数，即最常用的函数，包括整数的加、减和移位。当使用乘法等其他运算时，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

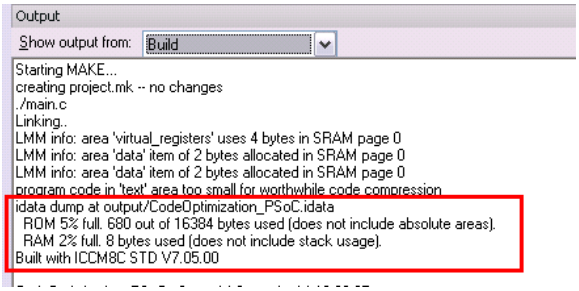
位移位叠加法等技巧取代无符号整数的乘法或除法，从而节省了代码空间。在无符号整数中，单一位元右移等于除以 2，而左移等于乘以 2。如下面的示例所示，通过使用位移叠加法，您可以避免使用乘法和除法库函数。

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 20 和图 21 所示。这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;
}
```

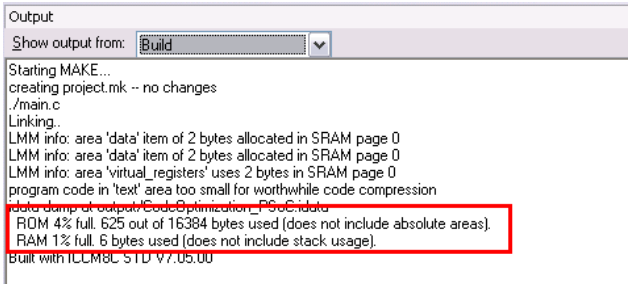
图 20. 代码 4 的 ROM 使用情况



代码 5

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

图 21. 代码 5 的 ROM 使用情况



避免幂函数

幂函数也会增加数学库函数。幂运算可以被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，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。您可以在“库用户指南”中查看浮点库的全部详细信息。为了使您能够进行估算，请参考如下浮点函数所需的代码空间。

比较 (*_fpcmp) = 78 个字节

浮点效用函数 (*_util) = 180 个字节

加 (*_fpadd) = 250 个字节

减 (*_fpsub + *_fpadd) = 9 + 250 = 259 个字节

乘 (*_fpmul+i_mulu8_block_util) = 292 + 29 = 321 个字节

除 (*_fpdiv) = 221 个字节

浮点效用函数（180 个字节）可与除比较函数以外的所有函数共同使用。因此，浮点函数所用的全部内存中包括效用函数的大小。例如，浮点加法函数所用的全部内存是 250 + 180 = 430 个字节。

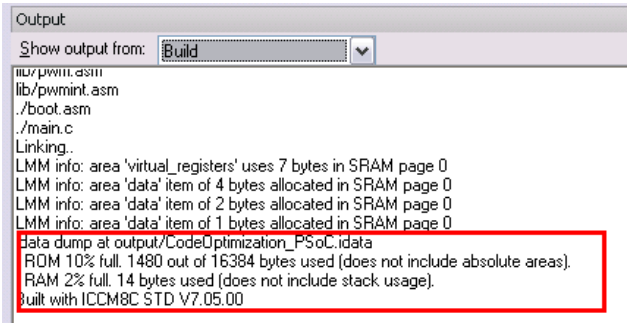
浮点数学函数使用整数数学库作为基础。这指的是，使用浮点运算时也需要整数数学库。如果变量的范围已知，可以将变量调整为直接使用整数运算，从而代替浮点运算。

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。在以下示例中，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 22 和图 23 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;
    }
}
```

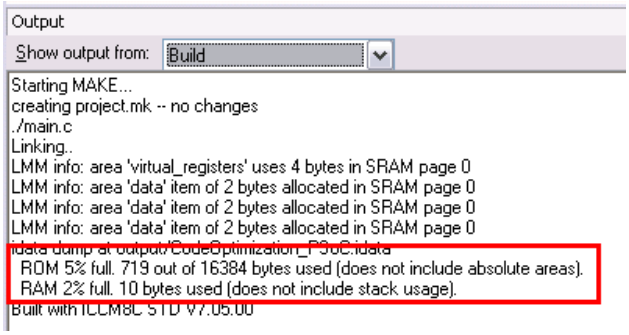
图 22. 代码 6 的 ROM 使用情况



代码 7

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

图 23. 代码 7 的 ROM 使用情况



使用查找表取代计算

有时候，使用查找表 (LUT) 取代计算效率更高。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程序的类型。

例如，*基于热敏电阻的温度计（PSoC Style AN2017）项目提供了浮点算法和 LUT 方法两个选项。在此项目中，使用 LUT 取代浮点运算可节省 1920 个字节的内存，但是精度有所降低。*

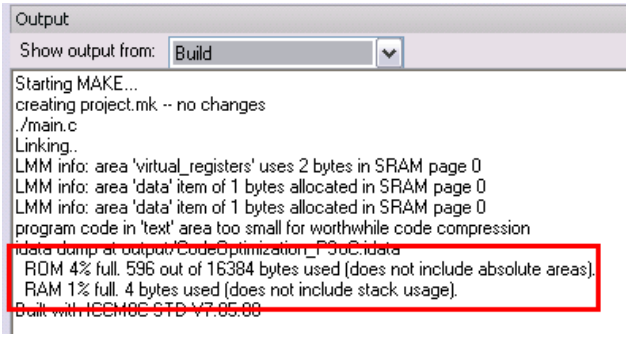
维护数据类型的一致性

当您选择变量的数据类型时，请选择此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，则您选择的单位应该是 BYTE（字节）而不是 int（整型）。同样，在适用情况下，选择无符号变量而非有符号变量。例如，请参考代码 4，如果程序中变量“iTest1”和“iTest2”的值不超过 255，我们便能将它们声明为“unsigned char”（无符号字符）而不是“unsigned int”（无符号整型），如代码 8 所示。图 24 显示了代码 8 的 ROM 使用情况，该代码比代码 4 少占用 84 个字节。

代码 8

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

图 24. 代码 8 的 ROM 使用情况



如果您知道有符号变量的正负范围，您可以使变量偏移至正值，并使用无符号的数学函数。同样，如果预计变量的值是在 –10 和 +10 之间，则使变量偏移 10，这样便可以使 0 对应 –10，20 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，您可以利用变量类型转换，以便使用相同的数学库，从而节省数学库的代码空间。

指南 3：数组索引与指针

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。在指针访问的情况下，每次访问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内存使用率的极大差异。

例如，考虑以下两个代码示例。代码 9 比代码 10 多占用 25 个字节，如图 25 和图 26 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。关键是确保地址是常量，在这种情况下，编译器将会直接替换地址。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

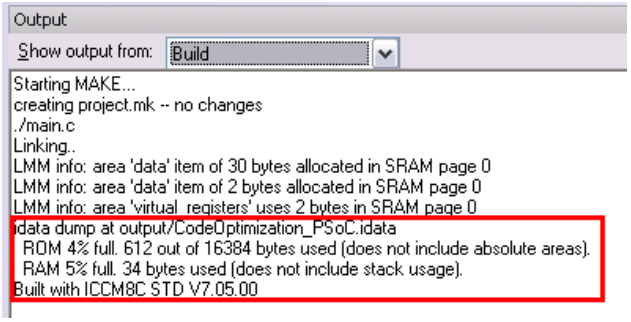
```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

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

sArray myTest;
sData* myPtr;

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

图 25. 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 26. 代码 10 的 ROM 使用情况

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'data' item of 30 bytes allocated in SRAM page 0
LMM info: area 'data' item of 2 bytes allocated in SRAM page 0
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
idata dump at output/CodeOptimization_PSoC.idata
 ROM 4% full. 587 out of 16384 bytes used (does not include absolute areas).
 RAM 5% full. 34 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

指南 4：Switch 语句与 IfElse

要实现条件判断，您可以使用 switch 语句或 if-else 语句。这两种语句之间的不同是，switch case 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 中显示的带有默认子句的 4 种 case 的 switch 语句比等效的代码 12 多占用 (9 + 4 × 4) = 25 个字节，如图 27 和图 28 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 27. 代码 11 的 ROM 使用情况

```
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
LMM info: area 'data' item of 1 bytes allocated in SRAM page 0
LMM info: area 'data' item of 1 bytes allocated in SRAM page 0
idata dump at output/CodeOptimization_PSoC.idata
 ROM 5% full. 657 out of 16384 bytes used (does not include absolute areas).
 RAM 1% full. 4 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

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 28. 代码 12 的 ROM 使用情况

```
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
LMM info: area 'data' item of 1 bytes allocated in SRAM page 0
LMM info: area 'data' item of 1 bytes allocated in SRAM page 0
idata dump at output/CodeOptimization_PSoC.idata
 ROM 4% full. 632 out of 16384 bytes used (does not include absolute areas).
 RAM 1% full. 4 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

如果条件判断针对的是双字节 (WORD)，两种表达式得出的代码大小几乎相同。

指南 5：初始化全局变量

在默认情况下，全局变量的初始值为零。明确地将全局变量初始化为零将会添加更多的代码。因此，对于代码优化而言，请勿明确地将全局变量设为零。

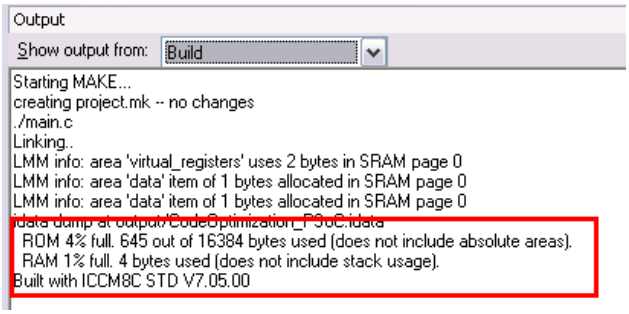
指南 6：汇编器中的部分代码

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可以优化代码的大小和性能。请考虑代码 11，参照代码 13 汇编中 bTest2 变量的赋值。

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 29. 代码 13 的 ROM 使用情况



代码 13 比代码 11 少占用 12 个字节，如图 29 所示。

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，而一些指南则普遍适用。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，当您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。如需所有编译器和项目优化选项的更多详细信息，请参照“C language Compiler user guide”（C 语言编译器用户指南）和“Assembly language user guide”（汇编语言用户指南），这两个文件均位于 **Help（帮助）-> Documentation（文档）-> Compiler and Programming（编译器和编程）**文件下。

关于作者

姓名：	Archana Yarlagadda
职衔：	应用工程师
背景：	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
联系人：	yara@cypress.com

文档修订记录

文档标题：PSoC® 1 M8C ImageCraft C 代码优化 – AN60486

文档编号：001-82493

修订版	ECN	变更方	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

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、工厂和经销商组成的全球性网络。要找到距您最近的办事处，请访问赛普拉斯所在地。

产品

汽车	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
光学导航传感器	cypress.com/go/ons
PSoC	cypress.com/go/psoc
触摸传感	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 5

赛普拉斯开发者社区

社区|论坛|博客|视频|培训

PSoC 是赛普拉斯半导体公司的注册商标。“可编程片上系统”、PSoC Designer 和 PSoC Creator 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 ：408-943-2600
传真 ：408-943-4730
网站 ：www.cypress.com

© 赛普拉斯半导体公司，2010-2012。此处所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品的内嵌电路之外，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。也不根据专利或其他权利以明示或暗示的方式授予任何许可。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。
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。除上述指定的用途之外，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的赛普拉斯软件许可协议限制。