Code Optimization tips for 8-bit microcontrollers

8 位微控制器的代码优化窍门

作者：赛普拉斯半导体公司，Steve Kolokowsky

假如您是一款非常成功的消费类产品的项目负责人。某天，公司要求您对已上市的所有产品添加最新的"必需"特性以进行升级。但是，当您在编译新代码时却看到了"程序存储器溢出"这条可怕的消息。您本来已经在编译器中打开了所有优化程序，努力让上次升级成功，可现在您却无计可施了。

本文将介绍一些优化技术，帮助设计人员节约多达 10% 的代码空间，从而让容量有限的程序存储器支持更多新特性和补丁。

许多程序员在 32 位处理器上学习编写软件，如 Intel 的 Pentium 处理器或某种 ARM 平台。不过，嵌入式领域的软件编写需要不同的思路。在 32 位 CPU 上，存储比特位的最佳方法通常是使用 32 位变量。对 8 位处理器而言，最好的办法就是采用单字节。像增强型 8051s 等某些处理器可能提供特殊的 1 位变量。

嵌入式处理器通常会超出标准的哈佛架构将存储器分散到不同的存储器空间中，有的相互重叠，有的又是相互分离。例如，8051 中常见的存储器空间包括 CODE、XDATA、DATA、IDATA、BIT 以及寄存器等。当要决定在何处存放变量时，了解每个存储器空间的优缺点显得非常重要，特别是在各个存储空间的容量都有限时更是如此。例如，IDATA 空间可能只能运行 256 个字节，不过它为间接存取进行了优化。虽然 DATA 空间也只能运行 256 个字节，但它包括了位可寻址空间和寄存器。尽管 CODE 和 XDATA 只能通过慢速间接存取机制进行访问，但它们的寻址空间却高达 64K。

许多 8 位 CPU 的编译器包含了很多优化程序，不过，这些优化程序都有其局限性。如果可以，应该尽可能简化表达。例如下面这段代码：

通常要比下述代码多占空间：

X = a * CONSTANT1* CONSTANT2;

因为编译器能将两个常量合并为一个。

优化——三思而后行

经验丰富的木匠都知道做事应该事先作好计划，三思而后行。嵌入式固件工程师也应该遵循这一原则。所有嵌入式编译器都提供了一个可给出有用信息映射文件。如图 1 所示，该映射文件提供了本文所用代码示例的有用信息。图中所示的库 (LIB_CODE) 使用的空间超过了 1K，而且启动代码 (c51startup) 使用的代码超过了 140 字节。

起点	终点	大小	对齐	片断	名称
0053H	0055H	0003H	---	代码	?CO?CYANUSBJTBL?2
0056H	045DH	0408H	字节	代码	?C?LIB_CODE
045EH	04FEH	00A1H	字节	代码	?C_INITSEG
04FFH	04FFH	0001H	---	**GAP**	

0500H	05A5H	00A6H	页面	代码	CY_AN_USB_DSCR
05A6H	0636H	0091H	字节	代码	CY_AN_TARGET
0637H	06C2H	008CH	字节	代码	?C_C51STARTUP
06C3H	06F2H	0030H	字节	CONST	?CO?CYANFWCBWPROC

图 1：映射文件示例

进行优化的另一原因是可以节约时间。在优化之前，衡量程序的性能尤为重要。显而易见，如果源文件过大，肯定会占用大量的存储器空间，但我们很难测定代码的哪些关键部分在消耗宝贵的 MIPS。在此过程中，我们可将程序概要分析 (Profiling) 作为一个重要的工具来加以利用。

我们可利用未使用的单一输出引脚来进行程序概要分析，不过输出引脚越多，分析也就越容易。我们可创建一个宏来设置程序概要分析输出，如下所示，再将宏放在每个例程的起点和终点处。

```
#define ISR_PROFILE{saveit = GPIO33+GPIO34<<1; GPIO33 = 1;  GPIO34 = 1; }

#define RESTORE_PROFILE {GPIO33 = saveit & 1; GPIO34 = (saveit & 2} >>1;}

#define PROFILE_2      {disable_ints(); GPIO33 = 1;  GPIO34 = 1; enable_ints();}

#define PROFILE_1      {disable_ints(); GPIO33 = 1;  GPIO34 = 1; enable_ints();}

#define PROFILE_0      {disable_ints(); GPIO33 = 1;  GPIO34 = 1; enable_ints();}

foo()

{

PROFILE_1;

... your code here
```

```
PROFILE_0;

}


Isr1() interrupt

{

ISR_PROFILE


RESTORE_PROFILE;

}
```

了解支付情况

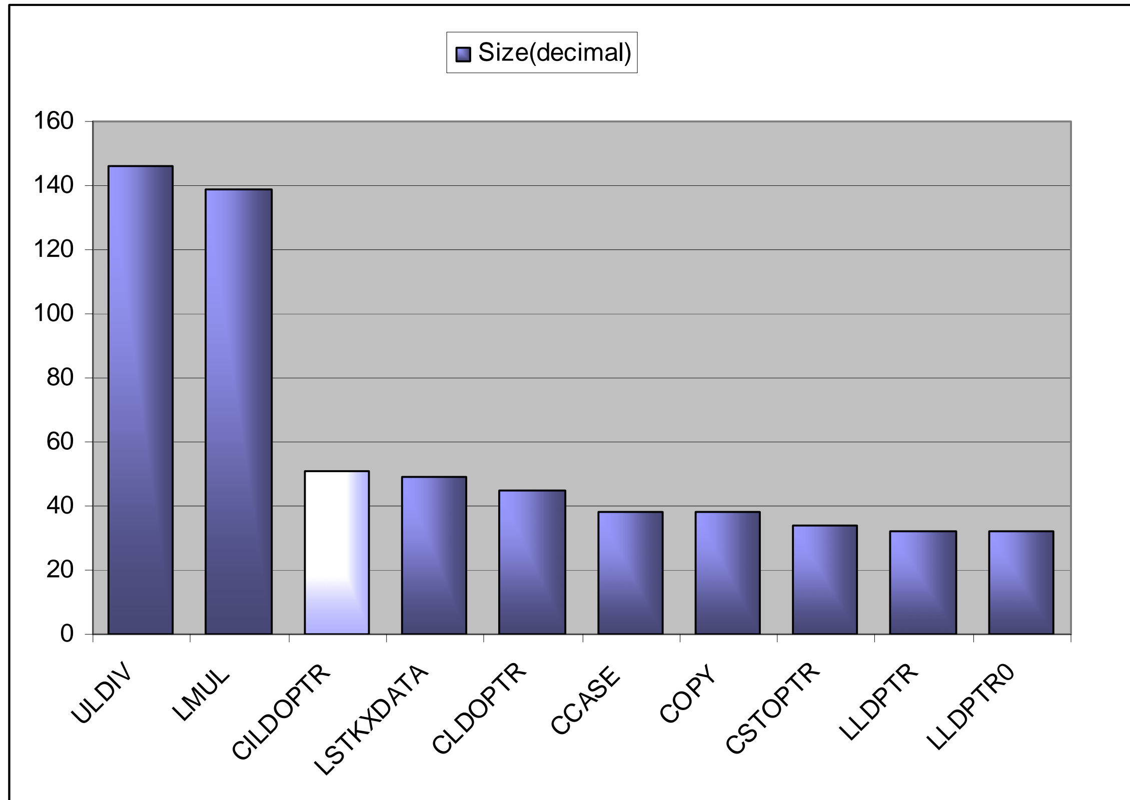

图 2：根据大小区分的库函数

在上述的映射文件中，我们了解到库占用了 1K 的宝贵存储器空间。深入查看映射文件，通过 Excel 进行分析后得到了如图 2 所示的结果。我们从图中移出较小的库函数部分。尽管这些函数名称比较晦涩，不过我们可以对照库参考资料逐一了解其含义。首先，ULDIV 是指无符号数的长除法 (long division)，而图中第二个则是指长乘法 (long multiplication)。

.map 文件的交叉参考表明我们很幸运：上述函数只用于一个文件中。.lst 文件显示了长除法函数的两种使用情况以及长乘法函数的一种使用情况：

```
glNandDevCapacity = CYAN_NAND_DEV_NUMPAGES_BLOCK * CYAN_NAND_UBLKS_PER_ZONE * (uint32_t)glNandNumZones;
```

在该特定案例中，我们知道 zone 的数量是一个二进制数，而另两个值为常量。因此，我们可用重复 8 次的左移位 (left shift) 操作替代长乘法：

```
{

char zoneCtr = glNandNumZones;

glNandDevCapacity  = CYAN_NAND_DEV_NUMPAGES_BLOCK * CYAN_NAND_UBLKS_PER_ZONE;
```

```
while (zoneCtr)

{

glNandDevCapacity <<= 1;

zoneCtr >>= 1;

}

}
```

尽管这个例程相当大，但它仍能减少库的使用并减小代码的整体大小。

*掌握比**编辑**器更多的信息*

成熟的 8 位编译器包括代码编写良好、经过优化的库函数。不过，这些函数须考虑到通过对数据的了解可自行处理的一些不常见情况。映射文件中显示的最大库函数就是这样一个很好的例子。调用两次 ULDIV 例程，以获得输入值除以常量后得到的除数和余数：

```
    zn                  = (adj_lba / CYAN_NAND_UBLKS_PER_ZONE);

    glNandRelativeBlkAddr = (adj_lba % CYAN_NAND_UBLKS_PER_ZONE);

}
```

由于我们在预期值方面比编译器了解的更多，因此我们可以让编译器不使用庞大的长除法函数，而采用较小的 16 位版本来替代。

```
{

xdata unsigned char lastNibble = adj_lba & 0xf;

adj_lba >>= 4;

zn = ((uint16_t)adj_lba / (uint8_t)CYAN_NAND_UBLKS_PER_ZONE/16));

glNandRelativeBlkAddr = ((uint16_t)adj_lba % (uint8_t) (CYAN_NAND_UBLKS_PER_ZONE/16));

glNandRelativeBlkAddr =  (glNandRelativeBlkAddr << 4) + lastNibble;

}
```

激进的的程序优化者甚至可能实现他们自己的二进制长除法例程。

*全局**变**量更好用*

将参数传递给函数是一个很好的代码经验。在 C 程序中，编译器可绝对确保调用的子程序不会修改参数。编译器可处理存储器管理的问题。不过，这将占用难以承受的大量时间和空间。试考虑下面这段代码：

```
Main()

{

    Int effectiveGlobal;

    Foo(effectiveGlobal)

}
```

由于变量在 main() 中已经声明，因此该变量与真正的全局变量之间的真正差别是命名空间 (namespace)。但是，每次调用 foo() 时，编译器都必须在新的位置存储 effectiveGlobal。声明真正的全局变量有助于降低因调用而造成的代码和数据开销。

*向**编译**器提供尽可能多的信息*

8051 可提供 64K 的地址空间 XDATA、256 字节的堆栈与间接寻址空间 IDATA 以及 256 字节的直接寻址空间 DATA 等多个存储器空间。在大多数情况下，代码编写人员都知道指针指向了哪个存储器空间。如果用户指定了存储器空间，编译器就无需包含对例程中的所有三类存储器进行寻址的代码，只需使用一个即可。由于指针无需包含数据空间信息，因此有助于节约数据空间。

在我的 8051 编译器中，上述变量可通过包含 OPTR 字符串的库例程进行存取。在列表和库文件中搜索对 OPTR 的引用可以发现长变量被多次使用，而且由于在代码中假定了指针的大小，其中某些长变量还会导致一些问题。

在变量声明中使用 const 关键词可以实现两方面的优化：第一，编译器不必再存储变量的初始值；第二，编译器能在编译时间而非执行时间执行一些数学运算。查看示例程序的编译输出，以确定对 const与 #define 的处理是否真的一样。以下是我对代码的测试：

经过测试，得到以下输出，表明它并不清楚 const 变量的值。

```
----- FUNCTION main (BEGIN) -----

 FILE: 'main.c'

    7: main()

    8: {

;---- Variable 'i' assigned to Register 'R7' ----

    9:    unsigned char i;

   10:

   11:    i *= c1 + c2;

000096 E508            MOV     A,c1

000098 2509            ADD     A,c2
```

```
00009A FE        MOV   R6,A
00009B EF        MOV   A,R7
00009C 8EF0      MOV   B,R6
00009E A4        MUL   AB
  12:   i *= D1 + D2;
00009F 75F00B    MOV   B,#0BH
0000A2 A4        MUL   AB
0000A3 FF        MOV   R7,A
  13: }
0000A4 22        RET
----- FUNCTION main (END) -------
```

汇编语言

不少嵌入式固件工程师信誓旦旦的表示他们始终能比编译器做得更好，不仅如此，他们还认为应该使用汇编语言重新编写所有代码。然而事实上，现代编译器提供的许多特性已经能赶上人脑的水平了。

变量共享：一些 8 位处理器尚无有效的机制来存取堆栈上的变量。一般的解决方案是创建调用树，并在相互不进行调用的函数间共享变量。在汇编程序中要想保持这种结构相当困难，且容易出错。

可靠性：任何从事专业软件或固件开发工作的人员都能读懂 C 语言程序。如果您需要将代码交给其它开发人员处理，他们无需掌握那些为发挥汇编语言的最大效率而需要的所有技巧便可立即开始修改代码。

可移植性：C 语言最初的开发目的之一就是要提供一种非常抽象，以便可以在多种处理器上应用的语言。这一目标至今仍然非常重要。

代码共享：许多 8 位编译器都能在链接时间之后进行优化，这使得编译器不仅能执行许多人工能完成的优化，而且还能完成一些人工所不能完成的优化。例如，现在许多编译器都能搜索不同函数中共有的代码字符串，并将其合并为一个新的函数。而人类是不可能记住每个编译周期中执行此函数所需要的全部细节的。

汇编语言现在仍占有一席之地。不过，在使用汇编语言之前应首先考虑上述所有因素。

结论

在撰写本文的过程中，我将成熟程序的大小从 0x6000 多字节缩减到了 0x5f2b 字节，节约了 200 多字节。该程序过去曾是多次试图优化程序大小的目标。希望上述小窍门能帮助您进一步减小程序的大小！

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com

© Cypress Semiconductor Corporation, 2007. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™, Programmable System-on-Chip™, and PSoC Express™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.