# ARM GCC 内联汇编参考手册

ARM GCC Inline Assembler Cookbook（最新版）

Tidy Jiang 译.

2016 年 8 月.

# 目录

# 翻译说明

- 原文链接：http://www.ethernut.de/en/documents/arm-inline-asm.html
- 本文档已有中文译文，翻译于2009年：http://blog.chinaunix.net/uid-20706279-id-1888741.html
- 本文档英文原版已有多次更新，更改、新增的内容较多，详细修改历史请参考文档末尾 文档历史 一节
- 本文主体容是按照英文原文翻译的，只有在遇到极个别不理解的地方才参考了中文译文
- 在翻译本文时，本人尽量做到先理解，再翻译，以便更准确地将原文的意思呈现出来
- 如有任何问题，请前往如下地址参与评论、讨论：
    - CSDN 博客：http://blog.csdn.net/tidyjiang/article/details/52138598
    - 个人博客：http://iot-fans.xyz/2016/08/05/zephyr/other-inline-assembler/
    - 个人博客：http://iot-fans.xyz/release/zephyr/arm-gcc-inline-assembler.html
    - GitHub：https://github.com/tidyjiang8/arm-gcc-inline-assembler
- 离线收藏、阅读请前往如下地址下载：
    - 个人博客：http://iot-fans.xyz/release/zephyr/arm-gcc-inline-assembler.pdf
    - GitHub: https://github.com/tidyjiang8/arm-gcc-inline-assembler
- 字数统计：6403 字

# 关于本文档

GNU C 编译器为 ARM 精简指令系统处理器提供了在 C 代码中内嵌汇编的功能。这种非常酷的特性提供了一些 C 代码没有的功能，比如手工优化软件关键代码、使用相关的处理器指令。

本文假设你已经熟悉 ARM 汇编程序，因为本文档不是 ARM 汇编教程，也不是 C 语言教程。

本文中所有的例程只使用 GCC v4 测试过，但是它们多数都应该能在早期版本上工作。

# GCC asm 申明

以一个简单的例程开始。下面的汇编语句可以像其它任何 C 语句一样出现在你的代码中。

```
/* example 1: NOP */
asm("mov r0,r0");
```

它将寄存器 r0 中的内容赋值给 r0。换句话说，它什么也没做。这种语句叫做 NOP（无操作）语句，通常用于实现一个极短的延时功能。

先等等！在将这个例程添加到你的 C 代码之前，请选继续阅读并学习本文后续部分，因为这段代码可能并不像我们所期望那样工作。

在内嵌汇编中使用汇编指令的方法与在纯汇编程序中使用汇编指令的方法一样。可以在一条 asm 语句中写多个汇编指令。但是为了增加程序的可读性，最好将每一条汇编指令单独放一行。

```
/* example 2: */
asm(
"mov     r0, r0\n\t"
"mov     r0, r0\n\t"
"mov     r0, r0\n\t"
"mov     r0, r0"
);
```

换行符和制表符的使用可以使得指令列表看起来变得美观。你第一次看时可能觉得有点怪异，但是当 C 编译器编译 C 语句的时候，它就是按照上面（换行和制表）的格式生成汇编代码的。到目前为止，内嵌汇编指令和你写的纯汇编程序中的代码没什么区别。但事实上，对比其它的 C 语句，asm 语句对常量、寄存器的处理是不一样的。通用的内嵌汇编模版是这样的：

```
asm(
    code
    : 输出操作数列表
    : 输入操作数列表
    : clobber列表
    );
```

sam 语句的第二部分(输出操作数列表)和第三部分(输入操作数列表)是可选的，它们在汇编语言与 C 语言之间提供一个桥梁作用。第三个部分(clobbers 列表)也是可选的，我们将在随后介绍。

再看另一个例程，一个 C 整型变量，向右移一位，并将结果保持到另一个整型变量中。

```
/* example 3: Rotating bits example */
asm(
    "mov %[result], %[value], ror #1"
    : [result] "=r" (y)
    : [value] "r" (x)
    );
```

一条 asm 语句被冒号分为四个部分：

1. 使用字符串字面值描述的汇编指令：

   ```
   "mov %[result], %[value], ror #1"
   ```

2. 可选输出操作数列表。每个条目由方括号内的符号名、约束字符串、圆括号内的 C 表达式三部分组成：

   ```
   [result] "=r" (y)
   ```

3. 可选的输入操作数列表。其语法与输出操作数列表的语法相同。再次声明，这是可选的，且在我们的例程中只使用了一个操作数。

   ```
   [value] "r" (x)
   ```

4. 可选的 clobber 寄存器列表。该列表在本例程中被忽略。

只包含汇编指令的 asm 汇编叫做基本内联汇编，包含有可选部分的 asm 汇编叫做扩展内联汇编。

就像上面的 NOP 例程一样，如果尾部部分不使用，可以省略。但是需要注意，如果某一部分未使用，但是它后面的部分被使用了，那么必须将该部分空出。下面的例程用于设置 ARM CPU 的当前程序状态寄存器(CPSR)，它使用了输入操作数，但没有使用输出操作数：

```
asm(
        "msr cpsr,%[ps]"
        :
        : [ps]"r"(status)
        );
```

即使不使用汇编代码，代码部分也要保留空字符串。在下面的例程中，创建了一个特殊的 clobber，以告诉编译器内存环境可能已改变。再次声明，我们在后面讨论代码优化的时候会解释 clobber 列表。

```
asm(""::"memory");
```

为了增加程序的可读性，你可以插入空白、增加新行甚至添加 C 语言注释：

```
asm("mov    %[result], %[value], ror #1"

          : [result]"=r" (y) /* Rotation result. */
          : [value]"r"   (x) /* Rotated value. */
          : /* No clobbers */
    );
```

在代码部分，你可以通过百分号%和符号名来引用操作数。这将引用操作数列表中的同名实体。在移位例程中：

*%[result]* 引用了输出操作数中的 C 变量 y；*%[value]* 引用了输入操作数中的 C 变量 x。

符号操作的名字使用了独立的命令空间，这意味着本语句的符号名与其它任何符号表不相关。简单一点就是说你不必关心使用的符号名在 C 代码中已经使用了。不过，在同一条 asm 语句中的不同符号的名字不能相同。

如果你看到过早期的 C 代码，循环移位的例子必须要这么写：

```
asm(
        "mov %0, %1, ror #1"
        : "=r" (result)
        : "r" (value)
        );
```

操作数的引用是通过百分号加数字实现的，其中 %0 引用第一个操作数，%1 引用第二个操作数，以此递推。最新的 GCC 发行版中依然支持该格式，但是该格式很容易让人犯错并使代码难以维护。试想一下，如果你写了大量的汇编指令，但是你想插入一个新的输出操作数，你必须手工修改操作数的引用编号。

# C 代码优化

使用汇编语言主要有两个原因：第一，C 语言对硬件底层的处理被受到限制，比如 C 语句不能直接修改处理器的程序状态寄存器；第二，写出高度优化的代码。毫无疑问，虽然 GNU C 优化器的工作做得很好，但是其处理结果依然与手工汇编代码有差距。

本节的主题是我们容易忽略的部分：当使用内联汇编语句添加汇编语言代码时， C 编译器的代码优化器会对这些代码进行优化处理。我们来检查一下循环移位例程可能产生的汇编代码：

```
00309DE5    ldr   r3, [sp, #0]    @ x, x
E330A0E1    mov   r3, r3, ror #1  @ tmp, x
04308DE5    str   r3, [sp, #4]    @ tmp, y
```

编译器选择寄存器 r3 做循环移位使用，它也完全可以选择为每个 C 变量分配寄存器。可能不会显式地加载值或存储结果。下面是由不同版本的编译器使用不同编译选项生成的代码：

```
E420A0E1    mov r2, r4, ror #1    @ y, x
```

编译器为每个操作数选择一个相应的寄存器，使用已经缓存到 r4 中的值，并将 r2 中的结果传递给后面的代码。这个过程你能理解不？

有时候会更糟糕。有时候编译器甚至完全抛弃你嵌入的汇编代码。C 编译器的这种行为，取决于代码优化器的策略或嵌入汇编处的上下文。例如，如果你在后面的 C 程序中不再使用内联汇编中的任何输出操作数，优化器很有可能会删除掉你的内联汇编语句。在最开始的 NOP 例程中就可能出现这样的情况，因为编译器认为这段代码没有意义，只会增加开销、降低程序的执行效率。

上面问题的解决方法是使用 volatile 关键字指示编译器不要优化这段代码。NOP 的例程修改后的代码如下：

```
/* NOP example, revised */
asm volatile("mov r0, r0");
```

下面还有更多的烦恼等着我们。一个设计精细的优化器可能重新排列代码。看下面的代码：

```
i++;
if (j == 1)
    x += 3;
i++;
```

对于上面的代码，优化器认为两个 i++ 语句将不会影响 if 条件语句的执行，而且如果 i 的值增加 2 将节省一条 ARM 汇编指令，因此编译器将重新组织代码：

```
if (j == 1)
    x += 3;
i += 2;
```

因此，我们无法保证编译后的代码与源代码中的语句的顺序相同。

这种行为可能会对我们的代码产生很大的副作用。下面一段代码的作用是计算 b 和 c 的乘积。其中，b 和/或 c 的值可能会被中的例程修改。因此，我们在访问变量前先禁止中断，并在完成计算后再重新使能中断。

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" ::: "r12", "cc");
c *= b; /* This may fail. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc");
```

不幸的是，优化器可能会让乘积指令先执行，再执行两个内联汇编指令，或者相反。这会让我们的汇编代码毫无意义。

对于这个问题，我们可以借助于 clobber 列表。该例程中的 clobber 列表如下：

```
"r12", "cc"
```

这条 colbber 列表将给编译器传达如下信息：我这段汇编代码修改了寄存器 r12 的值，并更新了程序状态寄存器的标志位。顺便说一下，直接指明使用的寄存器，将有可能阻止了最好的优化结果。一般情况下，你应该传递一个变量，让编译器自己选择寄存器。clobber 列表中的关键字，除了寄存器名和 cc(状态寄存器标志位)，还包括 memory。memory 关键字用来指示编译器，该汇编指令改变了内存中的值。这将强制编译器在执行汇编指令前将所有缓存的值保存起来，并在汇编指令执行完后再将这些值加载进去。此外，编译器还必须保留执行顺序，因为在执行完带有 memory 关键字的 asm 语句后，所有变量的内容都是无法预测的。

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" :: : "r12", "cc", "memory");
c *= b; /* This is safe. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc", "memory");
```

使所有缓存值无效可能是次优化的。你也可以添加一个虚拟操作数来创建一个虚拟依赖：

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" : "=X" (b) :: "r12", "cc");
c *= b; /* This is safe. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" :: "X" (c) : "r12", "cc");
```

上述代码中，第一条汇编语句尝试去修改变量 b ，第二条汇编语句尝试使用变量 c 。这将保留三个语句的执行顺序，而不要使缓存的变量无效。

理解优化器对内嵌汇编的影响很重要。如果你读到这里还是云里雾里，最好是在看下个主题之前再把这段文章读几遍^_^。

# 其它要点

## 使用内联汇编作为预处理宏

对于经常需要重用的汇编代码，你可以将它们定义成宏放在头文件中。不过，如果这些头文件用在模块中，将导致编译器在严格 ANSI 模式下时产生警告信息。为了消除警告，需要将 asm 用 **asm**、volatile 用 **volatile** 替换掉。**asm** 和 **volatile** 相当于 asm 和 volatile 的别名。下面的宏可以将一个长整型值从小端转到大段或者相反。

```
#define BYTESWAP(val) \
    __asm__ __volatile__ ( \
        "eor     r3, %1, %1, ror #16\n\t" \
        "bic     r3, r3, #0x00FF0000\n\t" \
        "mov     %0, %1, ror #8\n\t" \
        "eor     %0, %0, r3, lsr #8" \
        : "=r" (val) \
        : "0"(val) \
        : "r3", "cc" \
    );
```

## C 桩函数

当宏被引用时，它将在每个引用处展开为相同的汇编代码，这在大型程序中是不可接受的。在这种情形下，你可以定义一个 C 桩(stub)函数。将上面的宏以 C 函数形式重新实现如下：

```
unsigned long ByteSwap(unsigned long val)
{
asm volatile (
        "eor     r3, %1, %1, ror #16\n\t"
        "bic     r3, r3, #0x00FF0000\n\t"
        "mov     %0, %1, ror #8\n\t"
        "eor     %0, %0, r3, lsr #8"
        : "=r" (val)
        : "0"(val)
        : "r3"
);
return val;
}
```

# 替换 C 变量的符号名

默认情况下，GCC 在 C 和汇编代码中的函数、变量使用相同的符号名。你可以使用一个特殊的格式— asm 语句 — 为汇编代码指定不同的名字：

```
unsigned long value asm("clock") = 3686400;
```

该语句指示编译器在生成汇编代码时使用 clock 作为符号名，而不要使用默认的 value。这只对全局变量有效，因为局部变量（又叫自动变量）在汇编代码中没有符号名。

# 替换 C 变量的符号名

虽然编译器不允许在函数定义中使用 asm 关键字，但是你可以通过原型声明来改变函数的名字：

```
extern long Calc(void) asm ("CALCULATE");
```

调用函数 *Clac()* 时将会调用到汇编指令中的函数 *CALCULATE*。

# 强制使用指定的寄存器

局部变量可以存储在寄存器中。你可以使用内联汇编为局部变量指定一个寄存器。

```
void Count(void) {
register unsigned char counter asm("r3");

... some code...
asm volatile("eor r3, r3, r3" : "=l" (counter));
... more code...
}
```

> 译注：eor 是异或指令，其原型是 EOR

汇编指令 "eor r3, r3, r3" 将清除变量 counter 的值(清零)。需要注意，在大多数情况下使用该指令都不是一个好主意，原因有两点：该指令会与编译器的优化器产生冲突；GCC 不会为相关的寄存器保留完整的备份。如果编译器认为变量不会再被引用，那么对应的寄存器就会被重用(re-used)。编译器没有能力检查这些寄存器是否与其它预定义寄存器之间存在冲突。如果你用这种方式指定了太多的寄存器，编译器甚至可能在产生代码时就将寄存器耗尽。

# 临时使用寄存器

如果你使用了寄存器，但是该寄存器没有出现在操作数中，那么你需要告诉编译器你使用了该寄存器。下面的例子将一个值调整为 4 的倍数，它使用 r3 作为 scratch 寄存器，并将其指定在 clobber 列表中，以告知编译器。此外，ands 指令修改了 CPU 的状态标识，因此也将 cc 添加到 clobber 列表中了。

```
asm volatile(
    "ands    r3, %1, #3"     "\n\t"
    "eor     %0, %0, r3" "\n\t"
    "addne   %0, #4"
    : "=r" (len)
    : "0" (len)
    : "cc", "r3"
  );
```

再次声明，直接将寄存器的用法写死（hard coding）总是一个坏习惯！更好的实现方法是使用 C 桩函数，并使用局部变量作为临时值。

# 使用常量

你可以使用 mov 指令将一个立即数常量值加载到寄存器中。这个常量值通常将限定在 0~255 之间。

```
asm("mov r0, %[flag]" : : [flag] "I" (0x80));
```

但是当在给定范围内移位偶数个比特的时候，也可以使用一个更大的值。换言之，该值可以是$n * 2^x$。其中 n 是上面提到的 0~255，x 是 0~24 中的偶数。由于可以翻转，x 可以被设为 26、28 或 30，在这种情况下，比特 37~32 被翻转到比特 5~0。最后，当使用 mvn 指令代替 mov 指令时，需要使用这些值的二进制补码。

如果你需要跳转到一个由预处理宏定义的固定内存地址处，你可以使用下面的汇编代码：

```
    ldr  r3, =JMPADDR
    bx   r3
```

如果上述常量是一个合法地址（比如 0x20000000），聪明的汇编器就会将上面的代码转换为：

```
    mov  r3, #0x20000000
    bx   r3
```

如果不合法（比如 0x00F000F0），汇编器将会从文字池（literal pool）中加载其值。

```
    ldr  r3, .L1
    bx   r3
    ...
    .L1: .word 0x00F000F0
```

内联汇编与上述汇编代码的的工作方式相同。但是你不需要使用 ldr 指令，你只需要提供一个常量作为寄存器的值：

```
asm volatile("bx %0" : : "r" (JMPADDR));
```

根据常量的实际值，你可以使用 mov 或 ldr 指令的变体。例如，如果 JMPADDR 被定义为 0xFFFFFF00，那么相应的代码类似于：

```
    mvn  r3, #0xFF
    bx   r3
```

真实世界当然比这个更复杂，比如可能有这样的需求：我们需要将一个常量加载到一个特殊寄存器中。假设我们想要调用一个子程序，但是在调用完后返回到另一个地址。当嵌入式固件从 main 返回时，这样做会很有用。在这种情况下，我们需要将值加载到连接寄存器(lr)。汇编代码如下：

```
    ldr  lr, =JMPADDR
    ldr  r3, main
    bx   r3
```

想到如何用内联汇编实现这段代码吗？答案是：

```
asm volatile(
    "mov lr, %1\n\t"
    "bx %0\n\t"
    : : "r" (main), "I" (JMPADDR));
```

但是还有一个问题。我们这里使用的是 mov 指令。当 JMPADDR 的值合法时，代码将能像我们期待那样正常工作。当不合法时，需要使用 ldr 指令代替。但是不幸的是，内联汇编中没有这样的表达式

```
    ldr  lr, =JMPADDR
```

相反，我们必须写成

```
asm volatile(
    "mov lr, %1\n\t"
    "bx %0\n\t"
    : : "r" (main), "r" (JMPADDR));
```

与纯汇编代码相比，我们使用了一条额外的语句作为结尾——使用一个额外的寄存器。

```
    ldr     r3, .L1
    ldr     r2, .L2
    mov     lr, r2
    bx      r3
```

> 注：好晕，但是翻译得应该没问题。

# 寄存器的用途

比较好的学习方法是分析编译后的汇编清单，并学习 C 编译器生成的代码。下面的表格是编译器典型使用的 ARM 寄存器，知道这些将有助于理解代码。

| 寄存器 | 别名 | 用途 |
| --- | --- | --- |
| r0 | a1 | 第一个函数参数<br>Scratch 寄存器 |
| r1 | a2 | 第二个函数参数<br>Scratch 寄存器 |
| r2 | a3 | 第三个函数参数<br>Scratch 寄存器 |
| r3 | a4 | 第四个函数参数<br>Scratch 寄存器 |
| r4 | v1 | 寄存器变量 |
| r5 | v2 | 寄存器变量 |
| r6 | v3 | 寄存器变量 |
| r7 | v4 | 寄存器变量 |
| r8 | v5 | 寄存器变量 |
| r9 | v6<br>rfp | 寄存器变量<br>实际的帧指针 |
| r10 | sl | 栈接线 |
| r11 | fp | 参数指针 |
| r12 | ip | 临时 |
| r13 | sp | 栈指针 |
| r14 | lr | 连接寄存器 |
| r15 | pc | 程序计数 |

# 常见陷阱

# 指令顺序

开发者总是自以为源代码中指定的指令顺序与最终的指令顺序一致。这种写法是错误的，并导致难以查找bug。实际上，优化器会像优化 C 语句那样优化汇编语句。如果有可能，指令的顺序可能会重排。

“优化 C 代码”一节对此进行了详细讨论并提供了解决方案。

# 定义变量作为指定的寄存器

即使将一个变量强制赋值给了一个指定的寄存器，代码运行的结果也可能不是我们所期望的。考虑如下片段：

```
int foo(int n1, int n2) {
  register int n3 asm("r7") = n2;
  asm("mov r7, #4");
  return n3;
}
```

编译器被指示使用 r7 作为局部变量 n3，且使用参数 n2 进行初始化。接着在内联汇编语句中将 r7 设为 4，最后再返回。然后，这完全错了！一定要记住，编译器不知道内联汇编中发生了什么，但是优化器对 C 代码很聪明，将产生如下汇编代码：

```
foo:
  mov r7, #4
  mov r0, r1
  bx  lr
```

返回的结果不是 r7，而是 n2 的值。n2 会在我们的函数中传递给寄存器 r1。发生了什么？尽管最终的代码中包含了我们的内联汇编语句，C 代码优化器认为完全不需要使用 n3。它直接返回了参数 n2 。

仅仅将一个变量分配一个固定的寄存器不意味着 C 编译器将使用这个变量。我们仍然需要告诉编译器，在内联汇编的操作数中，有一个变量被修改了。对于给定了例程，我们需要在 asm 语句的输出操作数中做扩展：

```
asm("mov %0, #4" : "=l" (n3));
```

现在，C 编译器知道 n3 被修改了，并将产生我们期望的结果：

```
foo:
  push {r7, lr}
  mov  r7, #4
  mov  r0, r7
  pop  {r7, pc}
```

# 在 **thumb** 状态执行

需要注意，依赖于所给定的编译选项，编译器可能会切换到 thumb 状态。使用在 thumb 状态时无效的内联汇编指令将导致隐藏的编译错误。

# 汇编代码尺寸

在大多数情况下，编译器能正确的判断汇编指令的尺寸，但是当有汇编宏的时候是例外。因此最好避免之。

你可能会感到困惑：是汇编语言宏，不是 C 预处理宏。使用后者更好。

## 标签

在汇编指令中，你可以使用标签来达到跳转的目的。不过，你不能由一个汇编指令调整到另一个汇编指令。优化器只知道这些跳转将产生坏代码。

> What ? 没明白 !

## 预处理宏

内联汇编指令不能包含预处理宏，因为对于预处理器而言，这些指令仅仅是字符常量。

如果你的汇编代码中必须引用宏中的值，请参考“使用常量”一节。

# 外部链接

如果想要更深入地讨论内联汇编，请参考 gcc 用户手册。最新版的 gcc 用户手册位于：

http://gcc.gnu.org/onlinedocs/

# 版权



# 文档历史

| 日期 (年月日) | 改动 | 致谢 |
| --- | --- | --- |
| 2014/02/11 | 更正第一个常量例程，常量必须作为输入操作数 | spider391Tang |
| 2013/08/16 | 更正特殊寄存器语法的例程代码，在陷阱章节增加若干主题 | Sven Köhler |
| 2012/03/28 | 更正陷阱章节中常量传输的错误，并移动到使用章节 | enh |
|  | 添加预处理宏陷阱 |  |
|  | 添加本历史 |  |