

# GCC内联汇编入门

2014/11/11 11:24:17



原文为GCC-Inline-Assembly-HOWTO，在google上可以找到原文，欢迎指出翻译错误。

## 中文版说明

由于译者水平有限，故译文出错之处，还请见谅。C语言的关键字不译，一些单词或词组（如colbber等）由于恐怕译后词不达意，故并不翻译，由下面的单词表代为解释，敬请见谅。

英文原文中的单词和词组：

operand：操作数，可以是寄存器，内存，立即数。

volatile：易挥发的，是C语言的关键字。

constraint：约束。

register：本文指CPU寄存器。

asm：“asm”和“\_\_asm\_\_”在C语言中是关键字。原文中经常出现这个单词，是指嵌入到C语言（或者其它语言）的汇编程序片断。

basic inline assembly：指C语言中内联汇编程序的一种形式，和extended asm对应。基本格式如下：

asm("assembly code");

extended assembly：和basic inline assembly对应，比它多了一些特性，如可以指明输入，输出等。基本格式如下：

百度下

找找看

壹佰网络

湖北电信 7星级PHP5虚拟主机

特价优惠中

100MB 电信带宽服务器租用

1000元/月起

女子诱强奸犯回家带套

喝醉酒被整的美女

最全的齐B小短裙合集

关之琳被塞高尔夫球

- 1 30天摆脱酒店低盈利状态
- 2 春季特卖全场女装包邮9.9起
- 3 网购还没用返现？你亏大了
- 4 国家认证的网上药店！
- 5 网易有道智选推广
- 6 有道智选，小成本做大宣传
- 7 国家认证的网上药店！
- 8 国家认证的网上药店！

```
asm ( assembler template  
    : output operands  
    : input operands  
    : list of clobbered registers  
    );
```

clobber list: 实际上就是被使用的寄存器的列表, 用来告诉GCC它们已经被asm使用了, 不要在asm程序外使用它们。不然可能带来不可预见的后果。

clobbered registers: 它和clobber list对应。

assembler template: 就是汇编模板, 所有内联汇编代码都有按一定的格式。

见extended assembly的说明

作者: Sandeep.S

译者: 吴遥

版本号 v0.1 2003年3月01日

翻译版更新日期 2008/06/11

这篇HOWTO解释GCC提供的内联汇编特性的用途和用法。学习这篇文章只须具备两个前提条件, 显然那就是对x86汇编语言和C语言有基本的了解。

## 目 录

### 1. 前言

#### 1.1 版权与许可证

#### 1.2 回馈与更正

#### 1.3 感谢

### 2. 简介

### 3. GCC 汇编语法

#### 4. 基本内联汇编

#### 5. 扩展内联汇编

##### 5.1 汇编程序模板

##### 5.2 操作数

##### 5.3 Clobber列表

5.4 Volatile ... ?

6.更多关于约束条件

6.1 常用的约束

6.2 约束修饰符

7. 一些有用的诀窍

8. 结束语

9. 参考

# 1. 前言

---

## 1.1 版权与许可证

版权所有 (c)2003 Sandeep S.

这篇文档是免费的，你可以在依据自由软件组织GNU通用公共许可证条款下重新发布或者修改它。无论是版本2的许可证还是后来的版本（由你自己选择）。

这份文档的发布是希望它有用，但是并没有任何保证。

## 1.2 回馈与更正

欢迎善意的回馈和批评，我感谢每一个指出本文错误的人并尽快地更正错误。

## 1.3 感谢

我向GNU开发者提供这个功能强大的特性表达最诚挚的感谢。感谢Mr.Pramode C E的帮助。感谢政府工程学院的朋友尤其是Nisha Kurur和Sakeeb S精神上的支持。感谢政府工程学院老师对我的帮助。

另外，还要感谢 Phillip、Brennan、Underwood 和 colin@nyx.net，他们解决了很多难题。

## 2.简介

---

现在我们开始学GCC内联汇编。内联意味着什么？

我们可以指示编译器插入一个函数的代码到调用者的代码中，也就是实际调用产生的地方。这样的函数就是内联函数。看上去很像宏？实际上它们很相似。

内联函数有什么好处呢？

内联的方法减少了函数调用的额外开销。而且如果有实际的参数值是常数，那么在编译的时候编译器知道可能允许参数值的单一化，所以并不是所有的内联函数的代码都要

包含进来。对可执行代码大小的影响是不可预测的，它视乎对特定的情况。声明一个内联函数，我们声明中使用关键字inline。

现在我们站在一个位置来猜什么是内联汇编。它只是一些写在函数内的汇编语言的程序。在系统编程时候它们会显得很便利，快速，非常有用。我们的主要目标是学习GCC

内联汇编函数的基本格式和用法。

内联汇编之所以如此重要主要是因为它操作的能力和让它的输出在C语言变量中可见。（这个句话译得不太好）因为这样的能力，“asm”（译者：asm指内联函数）就像一个汇编指令和包含它的C语言程序之间的接口。

## 3.GCC汇编语法

---

GCC，即Linux平台下的GNU C语言编译器，它使用AT&T&sol（译者：应该是指AT & T语法，但是sol就不知道是什么）；UNIX汇编语法。现在让我们使用AT & T语法来进行汇编编码。如果你对AT & T语法不熟悉也不用担心，我将会教你。这种语法和Intel语法有很大的不同。以下我将给出主要的不同。

### 1、来源地 – 目的地 定序

AT & T语法和Intel语法在操作数的方向上是相反的。Intel语法的第一个操作数是目的地，第二个是来源地。然而AT & T语法的第一个操作数是来源地，第二的是目的地。也即：

“Op-code dst src”在Intel语法中变为“Op-code src dst”AT&T语法。

### 2、寄存器命名

寄存器名字要有前缀“%”。也即如果寄存器eax被使用，应写作%eax。

### 3、立即操作数

AT & T立即操作数之前要有一个“\$”符号。对于静态C语言变量也要有前缀“\$”。在Intel语法里，十六进制常数要有“h”作为后缀。在AT & T语法里我们用“0x”作为代替。所以，对于十六进制的数，我们看到一个“\$”，然后一个“0x”，最后才是常数本身。

#### 4、操作数大小

在AT & T语法里内存操作数的大小取决于操作码名字的最后一个字母。操作码后“b”，“w”和“l”分别指定byte（8字节长度），word（16字节长度）和long（32字节长度）的内存引用。Intel语法采用对内存操作数(不是操作码)加上前缀“byte ptr”，“word ptr”和“dword ptr”的方法来实现。

这样，Intel语法的“mov al, byte ptr foo”等同于AT & T语法的“movb foo, %al”。

#### 5、内存操作数

在Intel语法里寄存器包含在“[”和“]”里，而在AT & T语法里却改为“（”和“）”。另外，在Intel语法一个非直接内存引用是这样的：section&colon;[base &plus; index&ast;scale &plus; disp]，而在AT & T语法里却是这样的：section&colon;disp(base, index, scale)。

有一点要记住的是当一个常数当作disp&sol;scale时，“\$”符号不能前缀。

现在来看一下Intel语法和AT & T语法的主要不同点。我只是写了很少的一部分。如果要了解全部的内容，请参考GNU汇编文档（GNU Assembler documentations）。现在让我们看一些例子来帮助理解。

Intel Code	AT&T Code
mov eax,1	movl \$1,%eax
mov ebx,0ffh	movl \$0xff,%ebx
int 80h	int \$0x80
mov ebx, eax	movl %eax, %ebx
mov eax,[ecx]	movl (%ecx),%eax
mov eax,[ebx&plus;3]	movl 3(%ebx),%eax
mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

## 4.基本内联汇编

---

基本内联汇编的格式是非常简单的，如下：

```
asm("assembly code");
```

例子如下：

```
asm("movl %ecx %eax");/*将ecx的值传给eax了/
```

```
__asm__("movb %bh (%eax)");/*将bh的值传到eax指向的内存处*/
```

你可能已经注意到在这里我使用asm和\_\_asm\_\_两个关键字。它们都是正确的。如果asm关键字与程序里的某些程序发生冲突，那么你可以使用\_\_asm\_\_代替。如果我们有不止一条指令，那么我们每行在双引号里写一条指令，并且在指令最后加上一个'\n'和'\t'。这是因为gcc以字符串的形式发送每条指令给as（GAS），并且通过使用newline & tab的方法发送正确的行格式给汇编器。例子：

```
__asm__ ("movl %eax, %ebx/n/t"  
        "movl $56, %esi/n/t"  
        "movl %ecx, $label(%edx,%ebx,$4)/n/t"  
        "movb %ah, (%ebx)");
```

如果在我们的代码中我们改变了一些寄存器的值并且没有记下这些改变便返回，可能会导致错误的发生。这是因为GCC并不知道寄存器的值改变了，这会给我们带来麻烦，尤其是编译器对程序进行一些优化处理时。假设有这样的情况发生：某些寄存器保存着某些变量的值，而我们没有告诉GCC便改变了它，程序会如常地运行。这就是我们所扩展功能的原因。扩展asm提供我们这种的功能。

## 5.扩展内联汇编

---

在基本汇编内联里，我们只使用了指令。而在扩展汇编内联里，我们能够指定操作数。它允许我们指定输入寄存器，输出寄存器和一系列clobbered registers（译者注：实际就是指一些被内联汇编使用的寄存器，不知道如何翻译，所以下文也是以英文写出）。没有强制性规定一定要指定使用寄存器，我们可以把头痛的事情留给GCC，并且这样可能更有利于GCC对程序优化。基本的格式如下：

```
asm ( assembler template  
    : output operands          /* optional */  
    : input operands           /* optional */  
    : list of clobbered registers /* optional */  
    );
```



汇编程序模板（The assembler template）由汇编指令组成。每一个操作数由在括号内的C语言表达式后的操作数约束字符串描述。第一个冒号把汇编程序模板和第一个输出操作数分开，第二个冒号则把最后一个输出操作数和第一个输入操作数分开，假设有这样的操作数。逗号则在每组中分开操作数。操作数的量最多为10,或者机器描述里最大操作数的的指令模式，这取决于那一个比较大。

如果没有输出操作数却有输入操作数，就要写上两个连续冒号，以说明没有输出操作数。例子：

```
asm ("cld/n/t"
    "rep/n/t"
    "stosl"
    : /* no output registers */
    : "c" (count), "a" (fill_value), "D" (dest)
    : "%ecx", "%edi"
    );
```

现在，看看这代码都做了什么。上面的内联代码把fill\_value的值写到edi指向的内存地址count次。并且告诉GCC寄存器eax和edi不可以再用了（也则是说被使用了）。让我们看多一个例子来更好地理解。

```
int a=10, b;

asm ("movl %1, %%eax;
    movl %%eax, %0;"
    : "r"(b)      /* output */
    : "r"(a)      /* input */
    : "%eax"      /* clobbered register */
    );
```

这里我们使用汇编指令让“b”的值等于“a”的值。下面是

一些要点：

“b”是一个输出操作数，通过%0联系起来；而“a”是一个输入操作数，通过%1联系起来。“r”是对操作数的一个约束。在后面我们将会谈到关于约束的细节。“r”告诉GCC使用任何一个寄存器来存储操作数的值。输出操作数的约束必须包含一个约束修饰符“=”。这个修饰符说明输出

操作数是只写的。

这里有两个“%”在寄存器之前。这样可以帮助GCC辨别操作数和寄存器，操作数只有一个“%”作为前缀。第三个冒号后的clobbered register %eax告诉GCC%eax的值将会在“asm”里被修改，所以GCC不会使用这个寄存器去存储其它的数值。

当“asm”程序执行完后，“b”会映射出更新后的值，因为它被指定为一个输出操作数。换句话说，在“asm”里对“b”的改变将会影响到“asm”的外面的程序。

现在我们来看看各部分的细节。

## 5.1 汇编程序模板

汇编程序模板包含被插入到C语言程序里的汇编指令的集合。它的格式像这样：每一个指令必须包括在双引号里面，或者全部的指令包括在双引号里面。每一个指令还必须以一个定界符（delimiter）结束。合法的定界符可以是newline(/n)和semicolon(&semi;)。'/n'可以接一个tab (/t)。我们都知道了使用newline/tab的理由了吧？对应于C语言表达式的操作数被表示为%0，%1...等等。

## 5.2 操作数

C语言表达式在“asm”里作为汇编指令的操作数。每一个操作数首先要有一个在双引号里的一个操作数约束。对于输出操作数，还必须要有一个约束修饰符（它也是在引号里面），最后才是一个C语言表达式表示这个操作数。也就是说，“constraint”(C expression)是一般的形式，对于输出操作数则会有一个额外的修饰符。约束主要用来决定操作数的寻址模式。约束同样能够用来指定使用哪个寄存器。

如果我们使用不止一个操作数，那么它们用逗号分开。

在汇编程序模板里，每一个操作数通过号码来引用。编码方式如下。如果总共有n个操作数（包括输入和输出），那么第一个输出操作数编号为0，，第二个编号为1，以此类推，最后一个输入操作数编号为n-1。最大的操作数号如上一节所说的。

输出操作数必须是值。而输入操作数则没有这么严格，它们可以是表达式。扩展asm属性经常用于机器指令，编译器本身并不知道它的存在。如果输出表达式不能直接地寻址（例如，它是一个bit范围的值），我们的约束就必须允许使用一个寄存器。那样的话，GCC将会使用这个寄存器作为asm的输出，然后把这个寄存器的内容保存到输出。

正如上面所说的，普通的输出操作数必须是只写的（write-only）；GCC将会假设在指令之前的这样的操作数的值是没有用的，并且不须要被产生。扩展asm也支持input-output或者read-write操作数。

现在让我们专注于一些例子。我们想要一个数乘以5，我们使用lea指令。

```
asm ("leal (%1,%1,4), %0"
    : "=r" (five_times_x)
    : "r" (x)
    );
```

这里我们的输入是“x”。我没有指定使用哪一个寄存器。GCC会选择某个寄存器来作为输入，另外某个来作为输出。如果我们想输入和输出都在同一个寄存器，我们可以命令GCC这样做。这里我们使用读写类型（types of read write）的操作数。通过指定合适的约束，下面我们实现它：

```
asm ("leal (%0,%0,4), %0"
    : "=r" (five_times_x)
    : "0" (x)
    );
```

现在输入和输出操作数都在同一个寄存器里，但是我们还是不知道是哪一个寄存器。如果我们想指定寄存器，可以这样：

```
asm ("leal (%%ecx,%%ecx,4), %%ecx"
    : "=c" (x)
    : "c" (x)
    );
```

在以上的三个例子中，我们没有在clobber列表上指定任何寄存器。为什么呢？在前两个例子中，GCC决定了使哪个寄存器并且它知道什么改变了？在最后一个例子中，我们也不用在clobber列表加入ecx，GCC知道它和x之间传递数值。所以GCC知道ecx的值，不用考虑把它加入clobber列表。

## 5.3 Clobber列表

如果指令连续使用一些寄存器。我们必须把这么寄存器列在clobber列表之中，也则是内联汇编程序里第三个冒号后的范围。这样是为了告诉GCC我们自己将会使用并且修改它们。这样GCC将不会认为这些寄存器里的值是可用的。我们没有必要列出用于输入和输出的寄存器。因为GCC知道asm程序使用到它们（因为它们在约束中明显地指出来）。如果指令使用任何其它的寄存器，无论是显式还是隐式的指出（同是这样的寄存器也没有在输入或者输出约束列表中出现），那么这些寄存器必须在clobber列表中出现。

如果我们的指令会改变寄存器的值，我们必须加上"cc"到clobber列表上。

如果我们的指令在不可预知的情况修改了内存，则要增加这个到clobber 列表增加"memory"。这样的话GCC在执行汇编指令时就不会在寄存器中保存这个内存的值。如果对内存的影响并没有列在input或者output中，那么我们还要增加volatile这个关键字。

我们可以读写无限多次clobber寄存器。考虑到模板里多指令的例子，程序假设子程序\_foo从寄存器eax和ebx接收参数：

```
asm ("movl %0,%%eax;
    movl %1,%%ecx;
    call _foo"
    : /* no outputs */
    : "g" (from), "g" (to)
    : "eax", "ecx"
);
```

## 5.4 Volatile ... ?

如果你熟悉内核代码之类的代码，你一定经常看到许多函数被声明为 volatile 或者 \_\_volatile\_\_，这个声明在asm或者\_\_asm\_\_后面。前面我们说到了很多关于asm和\_\_asm\_\_。那么什么是volatile呢？

如果我们写的汇编语句一个要在我们写的地方执行（例如，一定不可以为了优化从一个循环里移出来），那么把关键字volatile放在asm和括号之间。这样就能够避免移动，删除代码。声明如下：

```
asm volatile ( ... : ... : ... : ... );
```

使用\_\_volatile\_\_关键字时我们必须很小心。

如果我们写的汇编语句只是为了做一些计算，并且对外面不造成任何影响，那么最好还是不要用volatile关键字。这样做有利于gcc对代码的优化和美化。对于这一章的一些有用的技巧，我已经提供了很多内联asm函数的例子。详细内容可以查看clobber列表。



## 6.更多关于约束条件

到目前为止，你可能已经知道了约束跟内联汇编关系密切。但是我们还没有详细地说到约束。约束可以用来说明一个操作数是否存放在一个寄存器中，并且在哪个寄存器中；也可以用来说明是否在内存中，并且是什么类型的地址；说明是否是一个立即数，并且可能取什么值（比如取值范围）等等。

### 6.1 常用的约束

约束有很多种，但是常用的比较不多，现在让我们来了解一下这些约束。

#### 1. 寄存器操作数约束(r)

当操作数被指定使用以下的约束时，它们就会被保存在通用寄存器(General Purpose Registers)，请看下面的例子：

```
asm ("movl %%eax, %0/n" : "=r"(myval));
```

变量myval被保存在寄存器里，寄存器eax的值被复制到这个寄存器，接着变量myval的值从这个寄存器传到内存中，更新内存的值。当"r"约束声明时，gcc会保存变量的值到任何一个可用的通用寄存器中。想要指定某个寄存器，你必须使用专用的寄存器约束符直接指定寄存器的名称。这些约束符如下所示：

r	Register(s)
a	%eax, %ax, %al
b	%ebx, %bx, %bl
c	%ecx, %cx, %cl
d	%edx, %dx, %dl
S	%esi, %si
D	%edi, %di

#### 2. 内存操作数约束(m)

对于寄存器操作数约束，先保存要运算的值到一个寄存器，运算后再把值传到内存中去。内存操作数约束与相反，当操作数在内存中时，任何运算都会直接在内存中执行。寄存器约束经常在指令能够大大地提高进程运行的速度时被使用。而在C语言变量须要在内联汇编语句中更新和不须要使用寄存器保存这个变量值时，内存约束则是最有效的。如下面的例子，idtr的

值被保存在内存loc中：

```
asm("sidt %0/n" : : "m"(loc));
```

### 3. 匹配（数字）约束

有时候，一个变量同时作为输入和输出的变量。这样的情况在内联汇编里可以用“匹配约束”来说明：

```
asm ("incl %0" : "=a"(var): "0"(var));
```

上一节，我们看到了一些关于操作数的例子。而这个例子是为了说明匹配约束，寄存器%eax同时作为输入和输出的变量。输入的var被读到%eax里，在%eax自增后，%eax的值又保存在var里。在这里"0"说明和第0个输入变量使用同样的约束。也即是var的值只保存在%eax。这种约束能够用在：

1. 输入和输出是同一个变量
2. 输入和输出的操作数实例是不重要的。

使用匹配约束最重要的影响就是这样能够更加高效地利用可用的寄存器。

其它被使用的约束有：

1. "m"：允许使用一个内存操作数，通常这个内存的地址可以是机器支持的任何值。
2. "o"：允许使用一个内存操作数，不过这个地址必须是可移位的。例如，这个地址增加一个较小的位移后，这个地址还是可用的。
3. "v"：一个内存操作数，它是不可移位的。
4. "i"：允许使用一个立即整型数（一个常数），这包括一个符号常量，它的值在汇编时知道。
5. "n"：允许使用一个立即整型数，它是一个已知的数字值。
6. "g"：允许使用任意通用寄存器、内存和立即数。

下面的约束是x86专用的：

1. "r"：寄存器操作数约束；
2. "q"：寄存器a、b、c、d；
3. "l"：常量0到31；
4. "J"：常量0到63；
5. "K"：0xff；
6. "L"：0xffff；
7. "M"：0, 1, 2, 3；

8. “N”：0到255的常量；
9. “f”：浮点数寄存器；
10. “t”：第一个浮点数寄存器；
11. “u”：第二个浮点数寄存器；
12. “A”：指定为寄存器a或者d，主要用于64位的值；

## 6.2 约束修饰符

使用约束时，为了使对约束效果的控制更加精确，GCC为我们提供了约束修饰符。常用的约束修饰符如下：

1. "="：意味着操作数对于这个指令只写，之前的值被放弃并且被输入值代替。
2. "&"：意味着操作数在clobber列表中，并且在被用来作为输入操作数之前就已经被改过。因此操作数不能作为输入操作数存放在寄存器里。

上面的列表和说明并不完全，学习例子能够更好地了解到内联汇编的使用方法。下一章我们将会学习一些例子，这些例子里有更多关于clobber列表和约束。



## 7. 一些有用的诀窍

---

上面所说的覆盖了GCC内联汇编的基本原理，现在我们开始关注一些简单的例子。用宏来写内联汇编函数总是方便的。我们能在内核代码里找到很多内联汇编函数。（`/usr/src/linux/include/asm/*.h`）

1、首先我们以一个简单的例子开始。我们要写一个程序对两个数据进行相加。

```
int main(void)
{
    int foo = 10, bar = 15;

    __asm__ __volatile__(
        "addl %%ebx,%%eax"
        : "=a"(foo)
        : "a"(foo), "b"(bar)
        );

    printf("foo+bar=%d\n", foo);

    return 0;
}
```

上面的例子我们把foo存放到%eax，bar到%ebx，并且把结果放到%eax。符号"="说明它是一个输出寄存器。我们也可以用另一种方式给变量加一个数值：

```
__asm__ __volatile__(
    " lock    ;/n"
    " addl %1,%0 ;/n"
    : "=m" (my_var)
    : "ir" (my_int), "m" (my_var)
```

```

: /* no clobber-list */

);

```

这是一个原子式的加法。我们可以去除指令"lock"来去除它的原子性。在输出区域, "m"说明my\_var是一个输出并且存放在内存里。同样的, "ir"说明my\_int是一个数字并且应该放在某些寄存器里。clobber列表上没有寄存器。

2、现在我们执行一些程序在变量或者寄存器上并且比较它们的值。

```

__asm__ __volatile__( "decl %0; sete %1"

: "=m" (my_var), "=q" (cond)

: "m" (my_var)

: "memory"

);

```

my\_var的值自减1, 如果自减的结果是0, 那么变量cond的值被设置。我们能够加入一句"lock ;/n/t"来增加程序的原子性。

同理, 我们可以使用"incl %0"来代替"decl %0", 来自增my\_var。

说明:

(i)my\_var是在内存里的一个变量

(ii)cond是寄存器eax, ebx, ecx和edx里的任一个

(iii)我们能看到"memorg"是在clobber列表上。例如: 代码修改内存的内容。

3、怎么设置, 清空一个寄存器的任一个二进制位。作为一个技巧, 请看下面的例子:

```

__asm__ __volatile__( "btsl %1,%0"

: "=m" (ADDR)

: "Ir" (pos)

: "cc"

);

```

在上面的例子里, 内存"ADDR"的第pos个位被设置为1。我们可以用"btrl"代替"btsl"来清空这个位。pos的约束符"Ir"说明pos在一个寄存器里面, 并且它的值的范围是为0到31。

4、现在来看看一些复杂但是有用的功能——字符串拷贝。

```
static inline char * strcpy(char * dest,const char *src)
{
    int d0, d1, d2;

    __asm__ __volatile__( "1:/tlofsb/n/t"

        "stosb/n/t"

        "testb %%al,%%al/n/t"

        "jne 1b"

        : "=&S" (d0), "=&D" (d1), "=&a" (d2)

        : "0" (src),"1" (dest)

        : "memory");

    return dest;
}
```

拷贝源的地址存放到esi，拷贝目标的地址到edi。紧接着开始拷贝。遇到0时，拷贝结束。约束符"&S"，"&D"，"&a"说明寄存器esi, edi, eax是之前被用到的clobber列表。也即是说，在函数结束之前它们的内容会被改变。内存存在clobber列表的原因也是显而易见的。

下面是一个类似的函数移动一块双字节，函数被声明为宏：

```
#define mov_blk(src, dest, numwords) /

__asm__ __volatile__(

    "cld/n/t" /

    "rep/n/t" /

    "movsl" /

    : /

    : "S" (src), "D" (dest), "c" (numwords) /

    : "%ecx", "%esi", "%edi" /

) /
```

5、在Linux内核里，系统调用就是用内联汇编来实现的。现在来看看一个系统调用是怎么实现的。所有的系统调用都被写成宏（在文件Linux/unistd.h里）。例如，一个被定义为宏的有三个参数的系统调用如下：

```
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) /

type name(type1 arg1,type2 arg2,type3 arg3) /
```

```
{ /
long __res; /

__asm__ volatile ( "int $0x80" /

    : "=a" (__res) /

    : "0" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2)), /

    "d" ((long)(arg3))); /

__syscall_return(type, __res); /

}
```

系统调用号被放到寄存器`eax`里，然后三个参数放到`ebx`, `ecx`, `edx`里。最后是指令`"int $0x80"`调用这个系统调用。返回值被放到`eax`。

每一个系统调用都可以用类似的方法实现。`"Exit"`是一个只有一个参数的系统调用，我们来看看它的代码是怎样的：

```
{
    asm("movl $1,%%eax;      /* SYS_exit is 1 */

        xorl %%ebx,%%ebx;    /* Argument is in ebx, it is 0 */

        int $0x80           /* Enter kernel mode */

    );
}
```

它的调用号是1，参数是0。所以执行`"int $0x80"`时，我们将`eax`设置为1，`ebx`为0。

## 8. 结束语

---

这篇文章介绍GCC内联汇编的基础。只要你能理解它的基本概念，就不难靠自己一步步学习。我们已经学习了一些有助于理解GCC内联汇编特性的常用例子。

GCC内联是一个很大的内容，同时这个文章在这个意义上并不完整。关于语法的更多的细节可以在GNU Assembler的官方文档上找到。

当然，Linux内核里大量地用到了GCC内联汇编。所以我们可以从源代码上找到各种各样的例子。它们能给我们带来很多的帮助。

如果你找到任何的排版出错，或者过时的信息，请让我们都知道。

## 9. 参考

---

1. Brennan's Guide to Inline Assembly
2. Using Assembly Language in Linux
3. Using `as`, The GNU Assembler

4. Using and Porting the GNU Compiler Collection (GCC)

5. Linux Kernel Source



分享到： QQ空间 新浪微博 腾讯微博 人人网 百度空间 百度搜藏 QQ收藏 百度贴吧

上一篇： js获取网页高度  
下一篇： 经常使用的时间同步server地址

关于这篇文章的相关推荐12

- 搭建arm-linux-gcc交叉编译工具链环境(Android
- arm-xilinx-eabi-gcc不是内部或外部命令,也不是
- [转]GCC编译的背后( 预处理和编译 汇编和链接 )
- Linux下clang、gcc、intel编译器最新版本安装笔
- Windows下利用MinGW配置GCC纯净编译环境
- 原创博文, 转载请注明出处。GCC的编译过程分
- 在LINUX AS4下安装GCC--RPM
- ubuntu 11.10 由于gcc版本过高引起的错误, 安装
- 如何在XCode中使用gcc编译生成的.a库文件?
- struct内存对齐: gcc与VC的差别

编程 汇编

- Linux跨GCC版本出现“浮点数例外”的解决办法
- GCC、ARM-LINUX-GCC、ARM-ELF-GCC浅析
- 被忽略了的gcc 浮点选项
- 有关gcc的扩展\_\_attribute\_\_((unused))
- [C] 跨平台使用SSE、AVX指令集心得——以单
- ("ld"输出的double。原本以为是GCC的Bug, 后来
- Eclipse CDT编译错误Internal Builder: Cannot
- ubuntu13.04编译安装升级
- 【原】TQ2440下配置arm\_linux\_gcc交叉编译工
- LINE: 在windows上运行原生linux程序 (3) : bash

云推荐

.Net 文章一周点击	.Net 文章一月点击
<div>iPhone推送修复Push Doctor目前最好最有效（转）</div> <div>《北京爱情故事》中《滴答滴》简谱</div> <div>iPad iOS 5 通知栏显示农历和天气的方法(转载)</div> <div>アプリ： old basement -地下倉庫からの脱出-脱出ゲーム 攻略</div> <div>完整版12306自动刷票教程，顺便学习了一下程序，受益匪浅</div> <div>真正的完美版的WAYOS完美破解版，确实有这号</div>	<div>iPhone推送修复Push Doctor目前最好最有效（转）</div> <div>iPad iOS 5 通知栏显示农历和天气的方法(转载)</div> <div>《北京爱情故事》中《滴答滴》简谱</div> <div>アプリ： old basement -地下倉庫からの脱出-脱出ゲーム 攻略</div> <div>真正的完美版的WAYOS完美破解版，确实有这号东西，哈哈</div> <div>完整版12306自动刷票教程，顺便学习了一下程序</div>


<a href="#">东西，哈哈</a>	<a href="#">，受益匪浅</a>
<a href="#">60个最佳新年2012年日历壁纸</a>	<a href="#">60个最佳新年2012年日历壁纸</a>
<a href="#">奇酷的JavaScript代码（可直接在浏览器地址栏运行）</a>	<a href="#">奇酷的JavaScript代码（可直接在浏览器地址栏运行）</a>
<a href="#">分享27款非常精美的2012新年桌面壁纸</a>	<a href="#">分享27款非常精美的2012新年桌面壁纸</a>
<a href="#">old offender 攻略ヒント集-----謎を解くためのヒント集を掲載 Android</a>	<a href="#">old offender 攻略ヒント集-----謎を解くためのヒント集を掲載 Android</a>

0条评论

最新 最早 最热

还没有评论，沙发等你来抢

社交帐号登录: [微博](#) [QQ](#) [人人](#) [豆瓣](#) [更多»](#)



说点什么吧...

发布

好工具正在使用多说