GCC 内联汇编基础

来源：中国自学编程网　发布日期：2008-09-08

这篇文章阐述内联汇编的使用方法。显然，阅读这篇文章您需要具备 X86 汇编语言和 C 语言的基础知识。

Contents

1. 简介

[主要是版权/反馈/勘误/感谢等信息。没有翻译。--译者注，本文中方括号中的都是译者注]

2. 概要

我们现在学习 GCC 内联汇编，那么内联汇编到底是什么？
[我们首先先来看看内联函数有什么好处]
我们可以让编译器将函数代码插入到调用者代码中，指出函数在代码中具体什么位置被执行。这种函数就是内联函数。内联函数似乎很像一个宏？的确，他们之间有很多相似之处。

那么内联函数到底有什么好处呢？

内联函数降低了函数调用的开销。[不仅仅节省堆栈] 如果某些函数调用的实参相同，那么返回值一定是相同的，这就可能给编译器留下了简化的空间。因为返回值相同了就不必把内联函数的代码插入到调用者的代码中[直接用这个返回值替换就好了]。这样可以减少代码量，视不同的情况而定。声明一个函数是内联函数，使用关键字 inline。

现在我们回到内联汇编上来。内联汇编就是一些汇编语句写成的内联函数。它方便，快速，对系统编程非常有用。我们主要目标是研究 GCC 内联函数的基础格式和使用方法。声明一个内联汇编函数，我们使用关键字 asm。

内联汇编的重要性首先体现在它的操作 C 语言变量和输出值到 C 语言变量的能力。由于这些特性，内联汇编常被用作汇编指令和调用它的 C 程序之间的接口。

3. GCC 汇编格式。

GCC (GNU Compiler for Linux) 使用 AT&T UNIX 汇编语法.这里我们将用 AT&T 汇编格式来写代码。如果你不熟悉 AT&T 汇编语法也没有关系，下面将有介绍。AT&T 和 Intel 汇编语法有很多的不同之处。我将给出主要的不同点。

1) 源操作数和目的操作数的方向
AT&T 和 Intel 汇编语法相反。Intel 语法中第一个操作数作为目的操作数，第二个操作数作为源操作数。相反，在 AT&T 语法中，第一个操作数是源操作数，第二个是目的操作数，例如：
Intel 语法: "OP-code dst src"
AT&T 语法: "Op-code src dst"

2) 寄存器命名

[在 AT&T 语法中] 寄存器名字加上%前缀，例如，如果要使用 eax, 写作: %eax.

3) 立即数

AT&T 语法中，立即数以'$'符号作为前缀。静态 C 变量前也要加上'$'前缀。在 Intel 语法中,16 进制的常数加上'h'后缀，但是在 AT&T 中,常量前要加上'0x'。 对于一个 16 进制常数(在 AT&T 中)，首先以$开头接着是 0x,最后是常数。

4) 操作数大小

在 AT&T 语法中，操作数占内存大小决定于汇编命令操作符的最后一个字符的内容。 操作符以'b', 'w'和 'l'为后缀指明内存访问长度是 byte(8-bit), word(16-bit)还是 long(32-bit). 而 Intel 语法在操作数前加上'byte ptr', 'word ptr'和'dword ptr'的内存操作数(这个操作数不是汇编命令操作符)来达到相同目的.

因此, Intel "mov al, byte ptr foo" 用 AT&T 语法就是 :"movb foo, %al"

5) 内存操作数

在 Intel 的语法中，基址寄存器用'['和']'扩起来，但是在 AT&T 中改用'('和')'。 此外，在 Intel 语法中一个间接内存寻址：

section:[base + index * scale + disp]，在 AT&T 中则为:

section:disp(base, index, scale)

总是需要记住的一点就是，当一个常数被用作 disp 或者 scale 时，就不用加'$'前缀。

现在我们已经提到了 AT&T 和 Intel 语法的一些主要不同点。 我只是提到了一小部分。全部内容可以参考 GNU 汇编文档。为了更好理解这些不同，请看下面的例子：

Intel Code AT&T Code

mov eax,1 movl $1, %eax

mov ebx,0ffh movl $0x0ff,%ebx,

int 80h int $0x80

mov ebx,eax movl %eax,%ebx

mov eax,[ecx] movl (%ecx),%eax

mov eax,[ecx+3] movl 3(ecx),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. 基本形式的内联汇编

基本形式的内联汇编格式非常直观:

asm(“assembly code”);

例如:

```
asm("movl %ecx, %eax"); /* 把 ecx 内容移动到 eax */
__asm__("movb %bh , (%eax)"); /* 把 bh 中一个字节的内容移动到 eax 指向的内存 */
```

你可能注意到了这里使用了 asm 和 __asm__关键字.二者皆可。这样如果 asm 关键字和程序其他变量有冲突就可以使用__asm__了。如果有超过一行的的指令，每行要加上双引号，并且后面加上\n\t. 这是因为 GCC 将每行指令作为一个字符串传给 as(GAS)，使用换行和 TAB 可以给汇编器传送正确的格式化好的代码行。

例如:

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

如果我们的代码涉及到一些寄存器(例如改变了其内容)并且从汇编代码返回后并没有修复这些改变，一些意想不到的情况可能发生。因为 GCC 不知道你已经将寄存器内容改了，这将给我们带来麻烦，尤其在编译器作了一些优化的情况下。如果不告诉 GCC，编译器将认为寄存器中事实上已经被改掉了的内容没有被改过，程序将当作它没有被改过而继续执行。 我们能做的就是不要使用这些带来其他附加影响的语句或者当我们退出的时候还原这些内容，否则只有等待程序崩溃了。这里提到的这种情况就是我们将要在下节中阐述的扩展形式的内联汇编。

5. 扩展形式的内联汇编

前面介绍的基础形式的内联汇编方法只涉及到嵌入汇编指令。在高级形式中，我们将可以指定操作数，它允许我们指定输入输出寄存器[内联函数使用这些寄存器作为存储输入输出变量]和程序中涉及到的 clobbered 寄存器列表[clobbered registers:内联汇编程序可能要改变其内容的寄存器]。也并不是一定要要显式指明使用具体的寄存器，我们也可以把它留给 GCC 去选择，这样 GCC 还可能更好的进行优化处理。高级内联汇编的基本格式如下:

```
asm ( assembler template
```

```
: output operands /* optional */
: input operands /* optional */
: list of clobbered registers /* optional */
);
```

其中 assembler template 包含汇编指令部分。括号中每个操作数用 C 表达式常量串描述。不同部分之间用冒号分开。相同部分中的每个小部分用逗号分开。操作数多少被限定为 10 或者由机器决定的一个最大值[这句话翻译的不好，贴出原文: The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.]。如果没有输出部分但是有输入部分，就必须在输出部分之前连续写两个冒号。

例如 ：

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

现在我们来分析上面的代码的功能。上面代码循环 count 次把 fill_value 的值到填充到 edi 寄存器指定的内存位置。并且告诉 GCC 寄存器 eax[这里应该是 ecx]和 edi 中内容可能已经被改变了。 为了有一个更清晰的理解，我们再来看一个例子：

```
int a=10, b;
asm ("movl %1, %%eax;
movl %%eax, %0;"
:"=r"(b) /* output */
:"r"(a) /* input */
:"%eax" /* clobbered register */
);
```

上面代码所做的就是用汇编代码把 a 的值赋给 b。值得注意的几点有:

1) “b”是输出操作数，用%0 来访问,”a”是输入操作数，用%1 来访问。

2) “r” 是一个 constraint, 关于 constraint 后面有详细的介绍。这里我们只要记住这里 constraint ”r”让 GCC 自己选择一个寄存器去存储变量 a。输出部分的 constraint 前必须要有个 ”=”，用来说明是一个这是一个输出操作数，并且只写。

3) 你可能看到有的寄存器名字前面写了两个%，这是用来帮助 GCC 区分操作数和寄存器。操作数只需要一个%前缀。

4) 在第三个冒号后面的 clobbered register 部分， %eax 说明在内联汇编代码中将要改变 eax 中的内容，GCC 不要用他存储其他值。

当这段代码执行结束后，”b”的值将会被改掉，因为它被指定作为输出操作数。换句话说，在”asm”内部对 b 的改动将影响到 asm 外面.

下面我们将对各个部分分别进行详细的讨论：

5.1 汇编模板

汇编模板部分包含嵌入到 C 程序中的汇编指令。格式如下：

每条指令放在一个双引号内，或者将所有的指令都放着一个双引号内。每条指令都要包含一个分隔符。合法的分隔符是换行符(\n)或者分号。用换行符的时候通常后面放一个制表符”\t”。我们已经知道为什么使用换行符+制表符了[前面部分有解释]。其中，访问 C 操作数用%0,%1…等等。

5.2 操作数

C 语言表达式 [大多情况是 C 变量] 将作为”asm”内部使用的操作数。每一个操作数都以双引号开始。对于输出操作数，还要写一个修改标志(=)。constraint 和修改标志都放在双引号内。接下来部分就是 C 表达式了[放在括号内].举例来说:

标准形式如下:

"constraint" (C expression) [如: “=r”(result)]

对于输出操作数还有一个修改标志(=)。 constraint 主要用来指定操作数的寻址类型 (内存寻址或寄存器寻址)，也用来指明使用哪个寄存器。

如果有多个操作数，之间用逗号分隔。

在汇编模板中，每个操作数都用数字引用[这些操作数]，引用规则如下，如果总共有 n 个操作数(包括输入输出操作数)，那么第一个输出操作引用数字为 0，依次递增，然后最后一个操作数是 n-1。关于最多操作数限制参见前面的小结。

输出操作数表达式必须是左值，输入操作数没有这个限制。注意这里可以使表达式[不仅仅限于变量]。高级汇编形式常用在当编译器不知道这个机器指令存在的时候。;-)如果输出表达式不能直接寻址(比如是 bit-field), constraint 就必须指定一个寄存器.这种情况下，GCC 将使用寄存器作为 asm 的输出。然后保存这个寄存器的值到输出表达式中。

如上所述，一般输出操作数必须是只写的；GCC 将认为在这条指令之前，保存在这种操作数中的值已经过期和不再需要了 。高级形式的 asm 也支持输入输出或者读写操作数。

现在我们来看一些例子，把一个数字乘以 5 使用汇编指令 lea

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

这里输入操作数是 ‘x’，不指定具体使用那个寄存器，GCC 会自己选择输入输出的寄存器来操作。如果我们也可以让 GCC 把输入和输出寄存器限定同一个。只需要使用读写操作数，使用合适的 constraint，看下具体方法:

```
asm(“lea (%0,%0,4),%0”
: “=r” (five_times_x)
: “0” (x)
);
```

上面使输入和输出操作数存在相同的寄存器中，我们不知道 GCC 具体使那个寄存器，但是我们也可以指定一个，像这样:

```
asm(“lea (%0,%0,4),%0”
```

: “=c” (five_times_x)
: “c” (x)
);
上面的三个例子中，我没有都没有在 clobber list 中放入任何寄存器的值，这是为什么？在前两个例子中，GCC 决定使用那个寄存器并且自己知道哪儿改变了。第三个例子中我们也没有必要把 ecx 放在 clobber list 中是因为 GCC 知道 X 将存入其中。因为 GCC 知道 ecx 的值，所以我们也不用放入 clobber list.

5.3 Clobber List
一些指令破坏了一个寄存器值，我们就不得不在 asm 里面第三个冒号后的 Clobber List 中标示出来，通知 GCC 这个里面的值要被改掉。这样 GCC 将不再假设之前存入这些寄存器中的值是合法的了。我们不需要把输入输出寄存器在这个部分标出，因为 GCC 知道 asm 将使用这些寄存器。(因为它们已经显式的被作为输入输出标出) 。如果此外指令中还用到其他寄存器无论显示还是隐式的使用到(没有在输入输出中标示出的)，这些指令必须在 clobbered list 中标明。

如果指令中以不可预见形式修改了内存值，要加上”memory”到 clobbered list 中。这使得 GCC 不去缓存在这些内存值。还有，如果内存被改变而没有被列在输入和出部分 要加上 volatile 关键字。

如果需要可以对 clobbered 寄存器多次读写。来看一个乘法的例子；调用函数 _foo 要求接受在 eax 和 ecx 值作为参数。
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或者一些类似优秀的代码，你一定见过很多在 asm 或者__asm__后的函数声明前加了 volatile 或者__volatile__。之前我提到过关于 asm 和__asm__，但是 volatile 有什么用途呢?

如果汇编代码必须在我们放的位置被执行(例如不能被循环优化而移出循环)，那就在 asm 之后()之前，放一个 valatile 关键字。 这样可以禁止这些代码被移动或删除，我们可以这样声明:
asm volatile (... : ... : ... : ...);
如果担心有变量冲突使用__volatile__关键字。

如果汇编语句只是做一些运算而没有什么附加影响。最好不要使用 volatile。不用 volatile 时会给 GCC 做代码优化留下空间。

在“常用技巧”章节中给出了很多例子，在那里你也可以详细看到 clobber-list 的使用。

6. 深入 constraints。

此时你可能理解了 constraint 对内联汇编有很大的影响。但是我们到目前为止才接触到说了关于 constraint 的一小部分。constraint 可以指出一个操作数是在寄存器中，在那个寄存器中，指出操作数是一个内存引用或具体内存地址。无论操作数是直接常量，或者可能是什么值。

6.1 常用 constraints
虽然有很多 constraints，但是常用的只有少数。下面我们就来看下这些限制条件。
1. 寄存器操作数限制条件: r
如果操作数指定了这个限制，操作数将使用通用寄存器来存储。看下面的例子:
asm (“movl %%eax, %0” : “=r” (myval));

变量 myval 被保存在一个寄存器中，eax 中的值被拷贝到这个寄存器中，并且在内存中的 myval 的值也会按这个寄存器值被更新。当 constraints ”r” 被指定时，GCC 可能在任何一个可用的通用寄存器中保存这个值。当然如果你要指定具体使用那个寄存器就要指定具体使用哪个寄存器的 constraints。如下表:
r Register(s)

a %eax, %ax, %al
b %ebx, %bx, %bl
c %ecx, %cx, %cl
d %edx, %dx, %adl
S %esi, %si
D %edi, %di

2. 内存操作数 constraint: m
当操作数在内存中时，任何对其操作将直接通过内存地址进行。和寄存器 constraint 相反，内存操作是先把值存在一个寄存器中，修改后再将值回写到这个内存地址。寄存器 constraint 通常只用在对速度要求非常严格的场合。因为内存 constraint 可以更有效率的将一个 C 语言变量在 asm 中跟新[不需要寄存器中转]，而且可能你也不想用一个寄存器来暂存这个变量的值。例如:
asm (“sidt” %0” : : “m”(loc));

3. 匹配 constraint
在某些情况下，一个变量可能用来保存输入和输出两种用途。这种情况下我们就用匹配 constraint
asm (“incl %0” :”=a”(var) : “0”(var));
我们在之前章节中已经看过类似的例子。这个例子中 eax 寄存器被用来保存输入也用来保存输出变量。输入变量被读入 eax 中，incl 执行之后 eax 被跟新并且又保存到变量 var 中。这儿的 constraint ”0”指定使用用和第一个输出相同的寄存器。就是说，输入的变量应该只能放在 eax 中。这个 constraint 可以在下面的情况下被使用:
a) 输入值从一个变量读入,这个变量将被修改并且修改过的值要写回同一个变量；
b) 没有必要把输入和输出操作数分开。
使用匹配 constraint 最重要的好处是对变量寄存器地使用更高效。

其他 constraint
1. “m”: 使用一个内存操作数，内存地址可以是机器支持的范围内。
2. “o”: 使用一个内存操作数，但是要求内存地址范围在在同一段内 。例如，加上一个小的偏移量来形成一个可用的地址。
3. “V”: 内存操作数，但是不在同一个段内。换句话说,就是使用”m”的所有的情况除了”o”
4. “i”: 使用一个立即整数操作数(值固定)；也包含仅在编译时才能确定其值的符号常量。
5. “n”: 一个确定值的立即数。很多系统不支持汇编时常数操作数小于一个字。这时候使用 n 就比使用 i 好。
6. “g”: 除了通用寄存器以外的任何寄存器，内存和立即整数。

下面的是 x86 特有的 constraint：
"r" : Register operand constraint, look table given above.
"q" : Registers a, b, c or d.
"I" : Constant in range 0 to 31 (for 32-bit shifts).
"J" : Constant in range 0 to 63 (for 64-bit shifts).
"K" : 0xff.
"L" : 0xffff.
"M" : 0, 1, 2, or 3 (shifts for lea instruction).
"N" : Constant in range 0 to 255 (for out instruction).
"f" : Floating point register
"t" : First (top of stack) floating point register

"u" : Second floating point register
"A" : Specifies the `a’ or `d’ registers. This is primarily useful for 64-bit integer values intended to be returned with the `d’ register holding the most significant bits and the `a’ register holding the least significant bits.

6.2 constraint 修改标记
在使用 constraint 的时候，为了更精确的控制约束，GCC 提供了一些修改标记，常用的 修改标记有：
1. “=”指这个操作数是只写的；之前保存在其中的值将废弃而被输出值所代替。

2. “&” Means that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address. An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. [这段不太明白，以后明白了在翻译]。

对 constraint 的解释还远没有完。但是例子可以帮助我们更好的理解内联汇编。下一小结中我们将来看一些例子。这里例子中我们能看到更多的关于 clobber-list 和 constraint。

7．常用技巧

到目前为止，已经讲完了 GCC 内联汇编基础知识。现在让我们来看一些简单的例子。 内联汇编函数可以很方便通过宏的形式来写。可以在内核代码中看到很多这样的内联汇编函数(在/usr/src/linux/asm/*.h)

1. 我们从一个简单的例子起。 我们来写把两个数字加起来的一个程序。

```
int main(void)
{
Int foo = 10, bar = 15;
__asm__ __volatile__ (“ addl %%ebx, %%eax”
: ”=a”(foo)
: ”a”(foo), “b”(bar)
);

prinft(“foo+bar=%d\n”, foo);
return 0;
}
```

这里我们强制让GCC将 foo 值存在%eax, bar 存在 5ebx 中，并且让输出放在%eax 中。其中”=”表明这是一个输出寄存器。再看看其他方法来加这两个数。

```
__asm__ __volatile__ (
“lock;\n”
“addl %1,%0;\n”
:”=m”(my_var)
:”ir”(my_int), “m”(my_var)
:
);
```

这是一个原子加法操作。可以去除指令 lock 移除原子性。在输出部分”=m”指出 my_var 作为输出并且在内存中。类似的”ir”指出 my_int 是一个整型数并且要保存到一个寄存器中(可以想象上面关于 constraint 的表)。这里没有 clobber list

2. 我们在一些寄存器活变量上来执行一些动作来对比下这些值。

```
__asm__ __volatile__ ( “decl %0; sete %1”
: “=m” (my_var), “=q” (cond)
: “m” (my_var)
```

: ”memory”
);
上面的程序将 my_var 减一并且如果减一后结果为零就将 cond 置位。我们可以再汇编语句之前加上”lock;\n\t”变成原子操作。

同样，我们可以用”incl %0”来代替”decl %0”来增加 my_var 的值。
这里值得注意的几点是
1) my_var 是一个存在内存中的变量。
2) cond 是一个存在任何通用寄存器中(eax,ebx,ecx,edx)的这时由于限制条件”=q”决定的。
3) clobber list 中指定了 memory。说明代码将改变内存值。

3. 如何设置和清除寄存器中的某一位？这就是下一个我们要看的技巧。
__asm__ __volatile__(“btsl %1, %0”
: “=m” (ADDR)
: “Ir” (pos)
: “cc”
);
这里在变量 ADDR(一个内存变量)在’pos’的位置值被设置成了 1.。我们可以时候 btrl 来清除由 btsl 设置的位.pos 变量的限定字符”Ir” 指明 pos 放在寄存器中并且值为 0-31(I 是一个 x86 相关 constraint).例如我们可以设置或者清除 ADDR 变量中从第 0 到第 31 位的值。因为这个要改变其中的值，所以我们加上”cc”在 clobberlist 中

4. 现在我里来看一些更加复杂但是有用的函数。字符串拷贝函数
```
static inline char* strcpy (char* dest, const char* src)
{
int d0, d1, d2;
__asm__ __volatile__( "1:\tlodsb\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;
}
```
源地址存在 ESI 寄存器中，目的地址存在 EDI 中。接着开始复制，直到遇到 0 结束复制。约束条件”&S”,”&D”,”&a”指明我们使用的是 ESI,EDI 和 EAX 寄存器。并且这些寄存器是很明显的 clobber 寄存器。("=&S" (d0), "=&D" (d1), "=&a" (d2) 这里用这三个寄存器作输出，GCC 很明显知道他们将被 clobber 所以后面的 clobber list 不用再写了)它们的内容在函数执行后会改变。这里还有很明显可以看出为什么 memory 被放在 clobber list 中。 (d0, d1, d2 被更新)

我们再来看一个相似的函数，用来移动一块双字。注意这个函数通过宏来定义的。
```
#define mov_blk(src, dest, numwords) \
```

```
__asm__ __volatile__ ( \
"cld\n\t" \
"rep\n\t" \
"movsl" \
: \
: "S" (src), "D" (dest), "c" (numwords) \
: "%ecx", "%esi", "%edi" \
)
```

这里没有输出，块移动过程导致 ECX,ESI,EDI 内容的改变，所以我们必须把它们放在 clobber list 中。

在 linux 中，系统调用是用 GCC 内联汇编的形式实现的。就让我们来看看一个系统调用是如何实现的。所有的系统调用都是用宏来写的(linux/unistd.h). 例如，一个带三个参数的系统调用的定义如下:

```
#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); \
}
```

一旦一个带三个参数的系统调用发生，上面的这个宏用来执行系统调用。系统调用号放在 eax 中，每个参数放在 ebx,ecx,edx 中，最后”int 0x80”执行系统调用。返回值放在 eax 中。

所有的系统调用都是用上面类似的方式实现的.Exit 是带一个参数的系统调用。我们看下这个实现的代码，如下:

```
{
asm("movl $1,%%eax; /* SYS_exit is 1 */
xorl %%ebx,%%ebx; /* Argument is in ebx, it is 0 */
int $0x80" /* Enter kernel mode */
);
}
```

Exit 的调用号是 1 参数为 0,所以我们把 1 放到 eax 中和把 0 放到 ebx 中,通过 int $0x80 exit(0)就被执行了。
这就是 exit 如何工作的。

8.结束语

这篇文章讲述了 GCC 内联汇编的基础内容。一旦你理解了基础原则，你自己一步步的看下去就没有什么困难了。我们通过一些例子可以更好的帮助我们理解一些在内联汇编中常用特性。

GCC 内联是一个很大的主题，这片文章要讲的还远远不够。但大多数我们提到的语法都可以在官方文档 GNU Assembler 中看到。完整的 constraint 可以在 GCC 官方文档中找到。

当然 Linux 内核大范围内使用了 GCC 内联。因此我们可以从中找到各种各样的例子。这对我们很有帮助。

如果你找到任何低级的排版打字错误或者过时的信息，请联系我。