本节先介绍
AT&T 汇编语言语法与 INTEL 汇编语法的差别，然后介绍 GCC 内嵌汇编语法。阅读本节需要读者具有 INTEL
汇编语言基础。

1 AT&T 与 INTEL 的汇编语言语法的区别

1.1 指令大小写

INTEL 格式的指令使用大写字母，而 AT&T
格式的使用小写字母。
例：

INTEL	AT&T
MOV EAX,EBX	movl %ebx,%eax

1.2 指令操作数赋值方向

在 INTEL 语法中，第一个表示目的操作数，第二个表示源操作数，赋值方向从右向左。AT&T 语法第一个为源操作数，第二个为目的操作数，方向从左到右，合乎自然。

例：

INTEL	AT&T
MOV EAX,EBX	movl %ebx,%eax

1.3　指令前缀

在 INTEL 语法中寄存器和立即数不需要前缀；

AT&T 中寄存器需要加前缀“%”；立即数需要加前缀“$”。

例：

INTEL	AT&T
MOV EAX,1	movl $1,%eax

符号常数直接引用，不需要加前缀，如：

movl value , %ebx

value 为一常数；

在符号前加前缀 $，表示引用符号地址，

如

movl $value, %ebx

是将 value 的地址放到 ebx 中。

总线锁定前缀“lock”：

总线锁定操作。“lock”前缀在 Linux 核心代码中使用很多，特别是 SMP 代码中。当总线锁定后其它 CPU 不能存取锁定地址处的内存单元。

远程跳转指令和子过程调用指令的操作码使用前缀“l“，分别为 ljmp，lcall，与之相应的返回指令伪 lret。

例：

INTEL	AT&T
	lcall $secion:$offset
JMP FAR SECTION:OFFSET	ljmp $secion:$offset
RET FAR SATCK_ADJUST	lret $stack_adjust

1.4　间接寻址语法

INTEL 中基地址使用“[”、“]”，而在 AT&T“(”、“)”；

另外处理复杂操作数的语法也不同，

INTEL 为 Segreg:[base+index*scale+disp]

，而在 AT&T 中为%segreg:disp(base,index,sale)，其中 segreg

，index,scale，disp 都是可选的，在指定 index 而没有显式指定 Scale 的情况下使用默认值 1。Scale,disp 不需要加前缀“&”。

INTEL AT&T
Instr foo,segreg:[base+index*scale+disp] instr %segreg:disp(base,index,scale),foo

1.5
指令后缀

AT&T 语法中大部分指令操作码的最后一个字母表示操作数大小，“b”表示 byte 一个字节）；“w”表示 word（2，个字节）；“l”表示 long（4，个字节）。
INTEL 中处理内存操作数时也有类似的语法如：
BYTE PTR、WORD PTR、DWORD PTR。

例：

INTEL	AT&T
mov al, bl	movb %bl,%al
mov ax,bx	movw %bx,%ax
mov eax, dword ptr [ebx]	movl (%ebx), %eax

AT&T 汇编指令中，操作数扩展指令有两个后缀，一个指定源操作数的字长，另一个指定目标操作数的字长。AT&T 的符号扩展指令的为“movs”，零扩展指令为“movz
”（相应的 Intel 指令为“movsx”和“movzx”）。因此，“movsbl %al,%edx”表示对寄存器 al
中的字节数据进行字节到长字的符号扩展，计算结果存放在寄存器 edx
中。下面是一些允许的操作数扩展后缀：

l
bl：，字节->长字 l
bw：，字节->字 l
wl：，字->长字

跳转指令标号后的后缀表示跳转方向，“f”表示向前（forward），
“b，”表示向后（back）。
例：

jmp 1f
jmp 1f

1.6 指令

INTEL 汇编与 AT&T 汇编指令基本相同，差别仅在语法上。关于每条指令的语法可以参考 I386Manual。

2 GCC 内嵌汇编

2.1 简介

内核代码绝大部分使用 C
语言编写，只有一小部分使用汇编语言编写，例如与特定体系结构相关的代码和对性能影响很大的代码。GCC 提供了内嵌汇编的功能，可以在 C 代码中直接内嵌汇编语言语句，大大方便了程序设计。

简单的内嵌汇编很容易理解

例：

```
__asm__ __volatile__("hlt");
```

“__asm__”表示后面的代码为内嵌汇编，“asm”是“__asm__”的别名。
“__volatile__”表示编译器不要优化代码，后面的指令保留原样，
“volatile”是它的别名。括号里面是汇编指令。

2.2 内嵌汇编举例在内嵌汇编中，

可以将 C 语言表达式指定为汇编指令的操作数，而且不用去管如何将 C 语言表达式的值读入哪个寄存器，以及如何将计算结果写回 C 变量，你只要告诉程序中 C 语言表达式与汇编指令操作数之间的对应关系即可， GCC 会自动插入代码完成必要的操作。

使用内嵌汇编，要先编写汇编指令模板，然后将 C 语言表达式与指令的操作数相关联，并告诉 GCC 对这些操作有哪些限制条件。例如在下面的汇编语句：

```
__asm__ __violate__ ("movl %1,%0" : "=r" (result) : "m" (input));
```

“movl %1,%0”是指令模板；“%0”和“%1”代表指令的操作数，称为占位符，内嵌汇编靠它们将 C 语言表达式与指令操作数相对应。指令模板后面用小括号括起来的是 C 语言表达式，本例中只有两个：“result”和“input”，他们按照出现的顺序分别与指令操作数“%0”，“%1，”对应；注意对应顺序：第一个 C 表达式对应“%0”；第二个表达式对应“%1
”，依次类推，操作数至多有 10 个，分别用“%0”，“%1”....“%9，”表示。在每个操作数前面有一个用引号括起来的字符串，字符串的内容是对该操作数的限制或者说要求。“result”前面的限制字符串是“=r”，其中“=”表示“result”是输出操作数，“r”表示需要将“result”与某个通用寄存器相关联，先将操作数的值读

入寄存器，然后在指令中使用相应寄存器，而不是“result”本身，当然指令执行完后需要将寄存器中的值存入变量“result”，从表面上看好像是指令直接对“result”进行操作，实际上 GCC 做了隐式处理，这样我们可以少写一些指令。“input”前面的“r”表示该表达式需要先放入某个寄存器，然后在指令中使用该寄存器参加运算。

我们将上面的内嵌代码放到一个 C 源文件中，然后使用 gcc –c–S 得到该 C 文件源代码相对应的汇编代码，然后查看一下汇编代码，看看 GCC 是如何处理的。

C 源文件如下内容如下，注意该代码没有实际意义，仅仅作为例子。

```
extern     int input,result;

void test(void)
{
       input = 1;
       __asm__ __volatile__ ("movl %1,%0" : "=r" (result) : "r"
      (input));
      return ;
}
```

对应的汇编代码如下；

行号	代码	解释
1		
7		
8	movl $1, input	对应 C 语言语句 input = 1;
9	input, %eax	
10	#APP	GCC 插入的注释，表示内嵌汇编开始
11	movl %eax,%eax	我们的内嵌汇编语句
12	#NO_APP	GCC 插入的注释，表示内嵌汇编结束
13	movl %eax, result	将结果存入 result 变量
14	—	
18		

。。。。。。

从汇编代码可以看出，第 9 行和第 13 行是 GCC，自动增加的代码，GCC 根据限定字符串决定如何处理 C 表达式，本例两个表达式都被指定为“r”型，所以先使用指令：

```
movl    input, %eax
```

将 input 读入寄存器%eax；GCC，也指定一个寄存器与输出变量 result 相关，本例也是%eax，等得到操作结果后再使用指令：

movl %eax, result

将寄存器的值写回 C 变量 result 中。从上面的汇编代码我们可以看出与 result 和 input，相关连的寄存器都是%eax，GCC 使用%eax，替换内嵌汇编指令模板中的%0，%1　movl %eax,%eax
显然这一句可以不要。但是没有优化，所以这一句没有被去掉。
由此可见，C 表达式或者变量与寄存器的关系由 GCC 自动处理，我们只需使用限制字符串指导 GCC 如何处理即可。限制字符必须与指令对操作数的要求相匹配，否则产生的汇编代码将会有错，读者可以将上例中的两个“r”，都改为“m”(m，表示操作数放在内存，而不是寄存器中)，编译后得到的结果是：

movl input, result

很明显这是一条非法指令，因此限制字符串必须与指令对操作数的要求匹配。例如指令 movl 允许寄存器到寄存器，立即数到寄存器等，但是不允许内存到内存的操作，因此两个操作数不能同时使用“m”作为限定字符。

2.3　语法

内嵌汇编语法如下：

```
__asm__(
        汇编语句模板:
        输出部分:
        输入部分:
        破坏描述部分)
```

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。例如：

```
__asm__ __volatile__( "cli": : :"memory")
```

2.3.1　汇编语句模板

汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n”或“\n\t”分开。指令中的操作数可以使用占位符引用 C 语言变量，操作数占位符最多 10 个，名称如下：%0，%1...，%9。指令中使用占位符表示的操作数，总被视为 long 型（4，个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在%和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h1。

2.3.2 输出部分

输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和 C 语言变量组成。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。

例：

```
__asm__ __volatile__("pushfl ; popl %0 ; cli":"=g" (x) )
```

描述符字符串表示对该变量的限制条件，这样 GCC 就可以根据这些条件决定如何 分配寄存器，如何产生必要的代码处理指令操作数与 C 表达式或 C 变量之间的联系。

2.3.3 输入部分

输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符由限定字符串和 C 语言表达式或者 C 语言变量组成。

例 1：

```
__asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));
```

例二（bitops.h）：

```
Static __inline__ void __set_bit(int nr,
volatile void * addr)
{
      __asm__(
            "btsl%1,%0"   :
             "=m"(ADDR)    :
            "Ir"(nr));
}
```

后例功能是将(*addr)的第nr位设为1。第一个占位符%0与C,语言变量ADDR对应，第二个占位符%1 与 C，语言变量 nr 对应。因此上面的汇编语句代码与下面的伪代码等价：

btsl nr, ADDR，该指令的两个操作数不能全是内存变量，因此将 nr 的限定字符串指定为“Ir”，将 nr，与立即数或者寄存器相关联，这样两个操作数中只有 ADDR 为内存变量。

2.3.4 限制字符

2.3.4.1 限制字符列表

限制字符有很多种，有些是与特定体系结构相关，此处仅列出常用的限定字符和 i386 中可能用到的一些常用的限定符。它们的作用是指示编译器如何处理其后

的 C 语言变量与指令操作数之间的关系，例如是将变量放在寄存器中还是放在内存中等，下表列出了常用的限定字母。
分类限定符描述 通用寄存器
“a”将输入变量放入 eax
这里有一个问题：假设 eax 已经被使用，那怎么办？
其实很简单：因为 GCC 知道 eax 已经被使用，它在这段汇编代码的起始处插入一条 语句 pushl %eax，将 eax 内容保存到堆栈，然后在这段代码结束处再增加一条 语句 popl %eax，恢复 eax 的内容

“b”将输入变量放入 ebx
“c”将输入变量放入 ecx
“d”将输入变量放入 edx
“s”将输入变量放入 esi
“d”将输入变量放入 edi
“q”将输入变量放入 eax，ebx ,ecx ，edx 中的一个 “r”将输入变量放入通用寄存器，也就是 eax ，ebx，ecx,edx，esi，edi 中的一个 “A”把 eax 和 edx，合成一个 64 位的寄存器(uselong longs)
“m”内存变量
“o”操作数为内存变量，但是其寻址方式是偏移量类型，也即是基址寻址，或者是基址加变址寻址
“V”操作数为内存变量，但寻址方式不是偏移量类型
“,” 操作数为内存变量，但寻址方式为自动增量
“p”操作数是一个合法的内存地址（指针）
寄存器或内存
“g” 将输入变量放入 eax，ebx，ecx ，edx 中的一个或者作为内存变量
“X”操作数可以是任何类型
立即数
“I” 0-31 之间的立即数（用于 32 位移位指令）
“J” 0-63 之间的立即数（用于 64 位移位指令）
“N” 0-255 ，之间的立即数（用于 out 指令）
“i” 立即数
“n” 立即数，有些系统不支持除字以外的立即数，这些系统应该使用“n”而不是“i” 匹配 “0”，“1 ，”... “9 ” 表示用它限制的操作数与某个指定的操作数匹配，也即该操作数就是指定的那个操作数， 例如用“0 ”去描述“％1”操作数，那么“%1”引用的其实就是“%0”操作数，注意作为 限定符字母的 0－9 ，与指令中的“％0”－“％9”的区别，前者描述操作数，后者代表操作数。后面有详细描述
& 该输出操作数不能使用过和输入操作数相同的寄存器后面有详细描述操作数类型
“=” 操作数在指令中是只写的（输出操作数）
“+” 操作数在指令中是读写类型的（输入输出操作数）
浮点数
“f”浮点寄存器
“t”第一个浮点寄存器
“u”第二个浮点寄存器

“G”标准的 80387
浮点常数
% 该操作数可以和下一个操作数交换位置
例如 addl 的两个操作数可以交换顺序（当然两个操作数都不能是立即数）
部分注释，从该字符到其后的逗号之间所有字母被忽略
* 表示如果选用寄存器，则其后的字母被忽略
现在继续看上面的例子，
"=m" (ADDR)表示 ADDR 为内存变量（“m”），而且是输出变量（“=”）；"Ir" (nr)表示 nr，为 0－31 之间的立即数（“I”）或者一个寄存器操作数（“r”）。

2.3.4.2 匹配限制符

I386 指令集中许多指令的操作数是读写型的（读写型操作数指先读取原来的值然后参加运算，最后将结果写回操作数），例如 addl %1,%0，它的作用是将操作数%0 与操作数%1 的和存入操作数%0，因此操作数%0 是读写型操作数。老版本的 GCC 对这种类型操作数的支持不是很好，它将操作数严格分为输入和输出两种，分别放在输入部分和输出部分，而没有一个单独部分描述读写型操作数，因此在 GCC 中读写型的操作数需要在输入和输出部分分别描述，靠匹配限制符将两者关联到一起注意仅在输入和输出部分使用相同的 C 变量，但是不用匹配限制符，产生的代码很可能不对，后面会分析原因。

匹配限制符是一位数字：“0”、“1”.....“9，”，分别表示它限制的 C 表达式分别与占位符%0，%1，.....%9 对应的 C 变量匹配。例如使用“0”作为%1，的限制字符，那么%0 和%1 表示同一个 C，变量。

看一下下面的代码就知道为什么要将读写型操作数，分别在输入和输出部分加以描述。该例功能是求 input+result 的和，然后存入 result：

```
extern int input,result;

void test_at_t()
{
      result= 0;
       input = 1;
       __asm__
       __volatile__ ("addl %1,%0":"=r"(result): "r"(input));

}
```

对应的汇编代码为：

```
       movl $0,_result
       movl $1,_input
       movl _input,%edx /APP
       addl %edx,%eax /NO_APP
```

```
        movl %eax,%edx
        movl %edx,_result
```

input 为输入型变量，而且需要放在寄存器中，GCC 给它分配的寄存器是%edx，在执行 addl 之前%edx，的内容已经是 input 的值。可见对于使用“r”限制的输入型变量或者表达式，在使用之前 GCC 会插入必要的代码将他们的值读到寄存器；“m”型变量则不需要这步。读入 input 后执行 addl，显然%eax 的值不对，需要先读入 result 的值才行。再往后看：movl %eax,%edx 和 movl %edx,_result 的作用是将结果存回 result，分配给 result 的寄存器与分配给 input 的一样，都是%edx。 综上可以总结出如下几点：

1. 使用“r”限制的输入变量，GCC 先分配一个寄存器，然后将值读入寄存器，最后用该寄存器替换占位符；

2. 使用“r”限制的输出变量，GCC 会分配一个寄存器，然后用该寄存器替换占位符，但是在使用该寄存器之前并不将变量值先读入寄存器，GCC 认为所有输出变量以前的值都没有用处，不读入寄存器（可能是因为 AT&T 汇编源于 CISC 架构处理器的汇编语言在 CISC 处理器中大部分指令的输入输出明显分开，而不像 RISC 那样一个操作数既做输入又做输出，例如 add r0,r1,r2，r0，和 r1 是输入，r2 是输出，输入和输出分开，没有使用输入输出型操作数，这样我们就可以认为 r2 对应的操作数原来的值没有用处，也就没有必要先将操作数的值读入 r2，因为这是浪费处理器的 CPU 周期），最后 GCC 插入代码， 将寄存器的值写回变量；

3. 输入变量使用的寄存器在最后一处使用它的指令之后，就可以挪做其他用处，因为已经不再使用。例如上例中的%edx。在执行完 addl 之后就作为与 result 对应的寄存器。因为第二条，上面的内嵌汇编指令不能奏效，因此需要在执行 addl 之前把 result 的值读入寄存器，也许再将 result 放入输入部分就可以了（因为第一条会保证将 result 先读入寄存器）。修改后的指令如下（为了更容易说明问题将 input 限制符由“r，”改为“m”）：

```
extern int input,result;

void test_at_t()
{

        result = 0;
        input = 1;
        __asm__
__volatile__ ("addl %2,%0":"=r"(result):"r"(result),"m"(input));

}
```

看上去上面的代码可以正常工作，因为我们知道%0 和%1 都和 result 相关，应该使用同一个寄存器，但是 GCC 并不去判断%0 和%1，是否和同一个 C 表达式或变量相关联（这样易于产生与内嵌汇编相应的汇编代码），因此%0 和%1 使用的寄存器可能不同。我们看一下汇编代码就知道了。

```
movl $0,_result
movl $1,_input
movl _result,%edx /APP
addl _input,%eax /NO_APP
movl %eax,%edx
movl %edx,_result
```

现在在执行 addl 之前将 result 的值被读入了寄存器%edx，但是 addl 指令的操作数%0 却成了%eax，而不是%edx，与预料的不同，这是因为 GCC 给输出和输入部分的变量分配了不同的寄存器，GCC 没有去判断两者是否都与 result 相关，后面会讲 GCC 如何翻译内嵌汇编，看完之后就不会惊奇啦。使用匹配限制符后，GCC 知道应将对应的操作数放在同一个位置（同一个寄存器或者同一个内存变量）。使用匹配限制字符的代码如下：

```
extern int input,result;

void test_at_t()
{
        result = 0;
        input = 1;
        __asm__  __volatile__
("addl %2,%0":"=r"(result):"0"(result),"m"(input));

}
```

输入部分中的 result 用匹配限制符“0”限制，表示%1 与%0，代表同一个变量，输入部分说明该变量的输入功能，输出部分说明该变量的输出功能，两者结合表示 result 是读写型。因为%0 和%1，表示同一个 C 变量，所以放在相同的位置，无论是寄存器还是内存。

相应的汇编代码为：

```
movl $0,_result
movl $1,_input
movl _result,%edx
movl %edx,%eax /APP
addl _input,%eax /NO_APP
movl %eax,%edx
movl %edx,_result
```

可以看到与 result 相关的寄存器是%edx，在执行指令 addl 之前先从%edx 将 result 读入%eax，执行之后需要将结果从%eax 读入%edx，最后存入 result 中。这里我们可以看出 GCC 处理内嵌汇编中输出操作数的一点点信息：addl 并没有使用%edx，可见它不是简单的用 result 对应的寄存器%edx 去替换%0，而是先分配一个寄存器，执行运算，最后才将运算结果存入对应的变量，因此 GCC 是先看该占位符对应的变量的限制符，发现是一个输出型寄存器变量，就为它分配一个寄存器，此时没有去管对应的 C 变量，最后 GCC，知道还要将寄存器的值写回变量，与此同时，它发现该变量与%edx 关联，因此先存入%edx，再存入变量。

至此读者应该明白了匹配限制符的意义和用法。在新版本的 GCC 中增加了一个限制字符“+”，它表示操作数是读写型的，GCC 知道应将变量值先读入寄存器，然后计算，最后写回变量，而无需在输入部分再去描述该变量。

例;

```
extern int input,result;

void test_at_t()
{

        result = 0;
        input = 1;
        __asm__
__volatile__ ("addl %1,%0":"+r"(result):"m"(input));

}
```

此处用“+”替换了“=”，而且去掉了输入部分关于 result 的描述，产生的汇编代码如下：

```
        movl $0,_result
        movl $1,_input
        movl _result,%eax /APP
        addl _input,%eax /NO_APP
        movl %eax,_result
L2:
        movl %ebp,%esp
```

处理的比使用匹配限制符的情况还要好，省去了好几条汇编代码。

2.3.4.3 “&”限制符

限制符“&”在内核中使用的比较多，它表示输入和输出操作数不能使用相同的寄存器，这样可以避免很多错误。举一个例子，下面代码的作用是将函数 foo 的返回值存入变量 ret 中：

```
__asm__ ( “call foo;movl %%edx,%1”, :”=a”(ret) : ”r”(bar) );
```

我们知道函数的 int 型返回值存放在%eax 中，但是 gcc 编译的结果是输入和输出同时使用了寄存器%eax，如下：

```
movl bar, %eax
#APP
call foo
movl %ebx,%eax
#NO_APP
movl %eax, ret
```

结果显然不对，原因是 GCC 并不知道%eax 中的值是我们所要的。避免这种情况的方法是使用“&”限定符，这样 bar 就不会再使用%eax 寄存器，因为已被 ret 指定使用。

```
_asm__ ( “call foo;movl %%edx,%1”,:”=&a”(ret) : ”r”(bar) );
```

2.3.5 破坏描述部分

2.3.5.1 寄存器破坏描述符

通常编写程序只使用一种语言：高级语言或者汇编语言。高级语言编译的步骤大致如下：

|

预处理；

|

编译

|

汇编

|

链接

我们这里只关心第二步编译（将 C 代码转换成汇编代码）：因为所有的代码都是用高级语言编写，编译器可以识别各种语句的作用，在转换的过程中所有的寄存器都由编译器决定如何分配使用，它有能力保证寄存器的使用不会冲突；也可以利用寄存器作为变量的缓冲区，因为寄存器的访问速度比内存快很多倍。

如果全部使用汇编语言则由程序员去控制寄存器的使用，只能靠程序员去保证寄存器使用的正确性。但是如果两种语言混用情况就变复杂了，因为内嵌的汇编代码可以直接使用寄存器，而编译器在转换的时候并不去检查内嵌的汇编代码使用了哪些寄存器（因为很难检测汇编指令使用了哪些寄存器，例如有些指令隐式修改寄存器，有时内嵌的汇编代码会调用其他子过程，而子过程也会修改寄存器），因此需要一种机制通知编译器我们使用了哪些寄存器（程序员自己知道内嵌汇编代码中使用了哪些寄存器），否则对这些寄存器的使用就有可能导致错误，修改描述部分可以起到这种作用。当然内嵌汇编的输入输出部分指明的寄存器或者指定为“r”，“g”型由编译器去分配的寄存器就不需要在破坏描述部分去描述，因为编译器已经知道了。破坏描述符由逗号格开的字符串组成，每个字符串描述一种情况，一般是寄存器名；除寄存器外
还有“memory”。
例如：“%eax”，“%ebx”，“memory”等。

下面看个例子就很清楚为什么需要通知 GCC 内嵌汇编代码中隐式（称它为隐式是因为 GCC 并不知道）使用的寄存器。 在内嵌的汇编指令中可能会直接引用某些寄存器，我们已经知道 AT&T 格式的汇编语言中，寄存器 名以“%”作为前缀，为了在生成的汇编程序中保留这个“%”号，在 asm 语句中对寄存器的 引用必须用“%%”作为寄存器名称的前缀。原因是“%”在 asm，内嵌汇编语句中的作用与“\”在 C 语言中的作用相同，因此“%%”转换后代表“%”。
例（没有使用修改描述符）：

```
int main(void)
{
    int input, output,temp;
   input = 1;

      __asm__ __volatile__ ("movl $0, %%eax;\n\t
                movl %%eax, %1;\n\t
                movl %2, %%eax;\n\t
                movl %%eax, %0;\n\t"
                :"=m"(output),"=m"(temp)    /* output */
                :"r"(input)                  /* input */
                );
                return 0;
}
```

这段代码使用%eax 作为临时寄存器，功能相当于 C 代码：
“temp = 0;output=input”，
对应的汇编代码如下：

```
        movl $1,-4(%ebp)
        movl -4(%ebp),%eax          /APP
```

```
        movl $0, %eax;
        movl %eax, -12(%ebp);
        movl %eax, %eax;
        movl %eax, -8(%ebp);        /NO_APP
```

显然 GCC 给 input 分配的寄存器也是%eax，发生了冲突，output 的值始终为 0，而不是 input。 使用破坏描述后的代码：

```
int main(void)
{
      int input, output,temp;

       input = 1;

  __asm__ __volatile__
              ( "movl $0, %%eax;\n\t
                movl %%eax, %1;\n\t
                movl %2, %%eax;\n\t
                movl %%eax, %0;\n\t"
                :"=m"(output),"=m"(temp)    /* output */
                :"r"(input)     /* input */
                :"eax"); /* 描述符 */

  return 0;
}
```

对应的汇编代码：

```
        movl $1,-4(%ebp)
        movl -4(%ebp),%edx          /APP
        movl $0, %eax;
        movl %eax, -12(%ebp);
        movl %edx, %eax;
        movl %eax, -8(%ebp); /NO_APP
```

通过破坏描述部分，GCC 得知%eax 已被使用，因此给 input 分配了%edx。在使用内嵌汇编时请记住一点：尽量告诉 GCC 尽可能多的信息，以防出错。如果你使用的指令会改变 CPU 的条件寄存器 cc，需要在修改描述部分增加“cc”。

2.3.5.2 memory 破坏描述符

“memory”比较特殊，可能是内嵌汇编中最难懂部分。为解释清楚它，先介绍一下编译器的优化知识，再看 C 关键字 volatile。最后去看该描述符。

2.3.5.2.1 编译器优化介绍

内存访问速度远不及 CPU 处理速度，为提高机器整体性能，在硬件上引入硬件高速缓存 Cache，加速对内存的访问。另外在现代 CPU 中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用 CPU 的指令流水线，提高执行速度。以上是硬件级别的优化。再看软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。编译器优化常用的方法有：将内存变量缓存到寄存器；调整指令顺序充分利用 CPU 指令流水线，常见的是重新排序读写指令。对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题的解决办法是在从硬件（或者其他处理器）的角度看必须以特定顺序执行的操作之间设置内存屏障（memory barrier），linux 提供了一个宏解决编译器的执行顺序问题。

void Barrier(void)

这个函数通知编译器插入一个内存屏障，但对硬件无效，编译后的代码会把当前 CPU 寄存器中的所有修改过的数值存入内存，需要这些数据的时候再重新从内存中读出。

2.3.5.2.2 C 语言关键字 volatile

C 语言关键字 volatile（注意它是用来修饰变量而不是上面介绍的 __volatile__）表明某个变量的值可能在外部被改变，因此对这些变量的存取不能缓存到寄存器，每次使用时需要重新存取。该关键字在多线程环境下经常使用，因为在编写多线程的程序时，同一个变量可能被多个线程修改，而程序通过该变量同步各个线程，例如：

```
DWORD __stdcall threadFunc(LPVOID signal)
{
        int* intSignal=reinterpret_cast(signal);
        *intSignal=2;
        while(*intSignal!=1)
        sleep(1000);
        return 0;

}
```

该线程启动时将 intSignal 置为 2，然后循环等待直到 intSignal 为 1，时退出。显然 intSignal 的值必须在外部被改变，否则该线程不会退出。但是实际运行的

时候该线程却不会退出，即使在外部将它的值改为 1，看一下对应的伪汇编代码就明白了：

```
mov ax,signal
label:
if(ax!=1)
    goto label
```

对于 C 编译器来说，它并不知道这个值会被其他线程修改。自然就把它 cache 在寄存器里面。记住，C 编译器是没有线程概念的！这时候就需要用到 volatile。volatile 的本意是指：这个值可能会在当前线程外部被改变。也就是说，我们要在 threadFunc 中的 intSignal 前面加上 volatile 关键字，这时候，编译器知道该变量的值会在外部改变，因此每次访问该变量时会重新读取，所作的循环变为如下面伪码所示：

```
label:
mov ax,signal
if(ax!=1)
   goto label
```

2.3.5.2.3 Memory

有了上面的知识就不难理解 Memory 修改描述符了，Memory 描述符告知 GCC：

（1）不要将该段内嵌汇编指令与前面的指令重新排序；也就是在执行内嵌汇编代码之前，它前面的指令都执行完毕。

（2）不要将变量缓存到寄存器，因为这段代码可能会用到内存变量，而这些内存变量会以不可预知的方式发生改变，因此 GCC 插入必要的代码先将缓存到寄存器的变量值写回内存，如果后面又访问这些变量，需要重新访问内存。如果汇编指令修改了内存，但是 GCC 本身却察觉不到，因为在输出部分没有描述，此时就需要在修改描述部分增加“memory”，告诉 GCC 内存已经被修改，GCC 得知这个信息后，就会在这段指令之前，插入必要的指令将前面因为优化 Cache 到寄存器中的变量值先写回内存，如果以后又要使用这些变量再重新读取。

例：

```
.....
Char test[100];
char a;
```

```
char c;
c = 0;
test[0] = 1;
……..
a = test [0];
……
__asm__(
"cld\n\t"
"rep\n\t"
"stosb"
: /* no output */

: "a" (c),"D" (test),"c" (100)
                    :
"cx","di","memory");
……….
// 我们知道 test[0] 已经修改，所以重新读取
a=test[0];
  ……
```

这段代码中的汇编指令功能与 memset 相当，也就是相当于调用了 memset(test,0,100)；它使用 stosb 修改了 test 数组的内容，但是没有在输入或输出部分去描述操作数，因为这两条指令都不需要显式的指定操作数，因此需要增加“memory”通知 GCC。现在假设：GCC 在优化时将 test[0] 放到了%eax 寄存器，
那么 test[0] = 1 对应于%eax=1，a = test [0]被换为 a=%eax ，如果在那段汇编指令中不使用“memory”，Gcc，不知道现在 test[0] 的值已经被改变了（如果整段代码都是我们自己使用汇编编写，我们自己当然知道这些内存的修改情况，我们也可以人为的去优化，但是现在除了我们编写的那一小段外，其他汇编代码都是 GCC 生成的，它并没有那么智能，知道这段代码会修改 test[0]），结果其后的 a=test[0]，转换为汇编后却是 a=%eax，因为 GCC 不知道显式的改变了 test 数组，结果出错了。如果增加了“memory”修饰符，GCC 知道：“这段代码修改了内存，但是也仅此而已，它并不知道到底修改了哪些变量”，因此他将以前因优化而缓存到寄存器的变量值全部写回内存，从内嵌汇编开始，如果后面的代码又要存取这些变量，则重新存取内存（不会将读写操作映射到以前缓存的那个寄存器）。这样上面那段代码最后一句就不再是 %eax=1，而是 test[0] = 1。 这两条对实现临界区至关重要，第一条保证不会因为指令的重新排序将临界区内的代码调到临界区之外（如果临界区内的指令被重排序放到临界区之外，What will happen?），第二条保证在临界区访问的变量的值，肯定是最新的值，而不是缓存在寄存器中的值，否则就会导致奇怪的错误。例如下面的代码：

```
int del_timer(struct timer_list * timer)
{
```

```
        int ret = 0;
        if (timer->next) {
              unsigned long flags;
              struct timer_list * next;
              save_flags(flags);
              cli();// 临界区开始
              if ((next = timer->next) != NULL) {
                    (next->prev = timer->prev)->next = next;
                    timer->next = timer->prev = NULL;
                    ret = 1;
              }     // 临界区结束

              restore_flags(flags);
        }
        Return  ret;
}
```

它先判断 timer->next 的值，如果是空直接返回，无需进行下面的操作。如果不是空，则进入临界区进行操作，但是 cli()的实现（见下面）没有使用“memory”，timer->next 的值可能会被缓存到寄存器中，后面 if ((next =timer->next) != NULL)会从寄存器中读取 timer->next 的值，如果在 if (timer->next)之后，进入临界区之前，timer->next 的值可能被在外部改变，这时肯定会出现异常情况，而且这种情况很难 Debug。但是如果 cli 使用“memory”，那么 if ((next = timer->next) !=NULL)语句会重新从内存读取 timer->next 的值，而不会从寄存器中取，这样就不会出现问题啦。

2.4 版内核中 cli 和 sti 的代码如下：

```
#define __cli()  __asm__  __volatile__("cli": : :"memory")
#define __sti()  __asm__  __volatile__("sti": : :"memory")
```

通过上面的例子，读者应该知道，为什么指令没有修改内存，但是却使用“memory”修改描述符的原因了吧。应从指令的上下文去理解为什么要这样做。
使用“volatile”也可以达到这个目的，但是我们在每个变量前增加该关键字，不如使用“memory”方便。

2.4 GCC 如何编译内嵌汇编代码

GCC 编译内嵌汇编代码的步骤如下：

1.输入变量与占位符

根据限定符和破坏描述部分，为输入和输出部分的变量分配合适的寄存器，如果限定符指定为立即数(“i”)，或内存变量(“m”)，则不需要该步骤，如果限定符没有具体指定输入操作数的类型(如“g”)，GCC 会视需要决定是否将该操作数

输入到某个寄存器。这样每个占位符都与某个寄存器、内存变量或立即数形成了一一对应的关系。对分配了寄存器的输入变量需要增加代码将它的值读入寄存器。另外还要根据破坏描述符的部分增加额外代码。

2.指令模板部分

然后根据这种一一对应的关系，用这些寄存器、内存变量或立即数来取代汇编代码中的占位符。

3.变量输出

按照输出限定符的指定将寄存器的内容输出到某个内存变量中，如果输出操作数的限定符指定为内存变量(“m”)，则该步骤被省略。