对于基于 ARM 的 RISC 处理器，GNU C 编译器提供了在 C 代码中内嵌汇编的功能。这种非常酷的特性提供了 C 代码没有的功能，比如手动优化软件关键部分的代码、使用相关的处理器指令。

这里设想了读者是熟练编写ARM汇编程序读者，因为该片文档不是ARM汇编手册。同样也不是 C 语言手册。

这篇文档假设使用的是 GCC 4 的版本，但是对于早期的版本也有效。

GCC asm 声明让我们以一个简单的例子开始。就像 C 中的声明一样，下面的声明代码可能出现在你的代码中。

```
/* NOP 例子 */

asm("mov r0,r0");
```

该语句的作用是将 r0移动到 r0中。换句话讲他并不干任何事。典型的就是 NOP 指令，作用就是短时的延时。

请接着阅读和学习这篇文档，因为该声明并不像你想象的和其他的 C 语句一样。内嵌汇编使用汇编指令就像在纯汇编程序中使用的方法一样。可以在一个 asm 声明中写多个汇编指令。但是为了增加程序的可读性，最好将每一个汇编指令单独放一行。

```
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。你第一次看起来可能有点怪异，但是当 C 编译器编译 C 语句的是候，它就是按照上面（换行和制表）生成汇编的。到目前为止，汇编指令和你写的纯汇编程序中的代码没什么区别。但是对比其它的 C 声明，asm 的常量和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 : output operand list : input operand list : clobber
list);
```

汇编和 C 语句这间的联系是通过上面 asm 声明中可选的 output operand list 和 input operand list。Clobber list 后面再讲。

下面是将 C 语言的一个整型变量传递给汇编，逻辑左移一位后在传递给 C 语言的另外一个整型变量。

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

每一个 asm 语句被冒号（:）分成了四个部分。

- 汇编指令放在第一部分中的“”中间。

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

- 接下来是冒号后的可选择的 output operand list，每一个条目是由一对[]（方括号）和被他包括的符号名组成，它后面跟着限制性字符串，再后面是圆括号和它括着的 C 变量。这个例子中只有一个条目。

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

- 接着冒号后面是输入操作符列表，它的语法和输入操作列表一样

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

- 破坏符列表，在本例中没有使用

就像上面的 NOP 例子，asm 声明的4个部分中，只要最尾部没有使用的部分都可以省略。但是有有一点要注意的是，上面的4个部分中只要后面的还要使用，前面的部分没有使用也不能省略，必须空但是保留冒号。下面的一个例子就是设置

ARM Soc 的 CPSR 寄存器，它有 input 但是没有 output operand。

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

即使汇编代码没有使用，代码部分也要保留空字符串。下面的例子使用了一个特别的破坏符，目的就是告诉编译器内存被修改过了。这里的破坏符在下面的优化部分在讲解。

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

为了增加代码的可读性，你可以使用换行，空格，还有 C 风格的注释。

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

          : [result]"=r" (y) /* Rotation result. */

          : [value]"r" (x) /* Rotated value. */

          : /* No clobbers */

);
```

在代码部分%后面跟着的是后面两个部分方括号中的符号，它指的是相同符号操作列表中的一个条目。

%[result]表示第二部分的 C 变量 y，%[value]表示三部分的 C 变量 x；

符号操作符的名字使用了独立的命名空间。这就意味着它使用的是其他的符号表。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。在早期的 C 代码中，循环移位的例子必须要这么写：

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

在汇编代码中操作数的引用使用的是%后面跟一个数字，%1代表第一个操作数，%2代码第二个操作数，往后的类推。这个方法目前最新的编译器还是支持的。但是它不便于维护代码。试想一下，你写了大量的汇编指令的代码，要是你想插入一个操作数，那么你就不得不从新修改操作数编号。

优化 C 代码有两种情况决定了你必须使用汇编。1st，C 限制了你更加贴近底层操作硬件，比如，C 中没有直接修改程序状态寄存器（PSR）的声明。2nd 就是要写出更加优化的代码。毫无疑问 GNU C 代码优化器做的很好，但是他的结果和我们手工写的汇编代码相差很远。

这一部分有一点很重要，也是被别人忽视最多的就是：我们在 C 代码中通过内嵌汇编指令添加的汇编代码，也是要被 C 编译器的优化器处理的。让我们下面

做个试验来看看吧。

下面是代码实例。

```
bigtree@just:~/embedded/basic-C$ arm-linux-gcc -c test.c
bigtree@just:~/embedded/basic-C$    arm-linux-objdump    -D
test.o
```

编译器选择 r3作为循环移位使用。它也完全可以选择为每一个 C 变量分配寄存器。Load 或者 store 一个值并不显式的进行。下面是其它编译器的编译结果。

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

编译器为每一个操作数选择一个相应的寄存器，将操作过的值 cache 到 r4中，然后传递该值到 r2中。这个过程你能理解不？

有的时候这个过程变得更加糟糕。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和嵌入汇编所处的上下文。如果在内嵌汇编语句中不使用任何输出部分，那么 C 代码优化器很有可能将该内嵌语句完全删除。比如 NOP 例子，我们可以使用它作为延时操作，但是对于编译器认为这影响了程序的执行速速，认为它是没有任何意义的。

上面的解决方法还是有的。那就是使用 volatile 关键字。它的作用就是禁止优化器优化。将 NOP 例子修改过后如下：

```
/* 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++;
```

优化器肯定是要从新组织代码的，两个 i++并没有对 if 的条件产生影响。更进一步的来讲，i 的值增加2，仅仅使用一条 ARM 汇编指令。因而代码要重新组织如下：

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

这样节省了一条 ARM 指令。结果是：这些操作并没有得到许可。

这些将对你的代码产生很到的影响，这将在下面介绍。下面的代码是 c 乘 b，其中 c 和 b 中的一个或者两个可能会被中断处理程序修改。进入该代码前先禁止中断，执行完该代码后再开启中断。

```
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");
```

但是不幸的是针对上面的代码，优化器决定先执行乘法然后执行两个内嵌汇编，或相反。这样将会使得我们的代码变得毫无意义。

我们可以使用 clobber list 帮忙。上面例子中的 clobber list 如下：

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

上面的 clobber list 将会将向编译器传达如下信息，修改了 r12和程序状态寄存器的标志位。Btw，直接指明使用的寄存器，将有可能阻止了最好的优化结果。通常你只要传递一个变量，然后让编译器自己选择适合的寄存器。另外*寄存器名*，*cc*（condition registor 状态寄存器标志位），*memory* 都是在 clobber list 上有效的关键字。它用来向编译器指明，内嵌汇编指令改变了内存中的值。这将强迫编译器在执行汇编代码前存储所有缓存的值，然后在执行完汇编代码后重新加载该值。这将保留程序的执行顺序，因为在使用了带有 *memory* clobber 的 asm 声明后，所有变量的内容都是不可预测的。

```
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");
```

使所有的缓存的值都无效，只是局部最优（suboptimal）。你可以有选择性的添加 dummy operand 来人工添加依赖。

```
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
```

上面的第一个 asm 试图修改变量先 b，第二个 asm 试图修改 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^_^。

Input and output operands 前面我们学到，每一个 input 和 output operand，由被方括号[]中的符号名，限制字符串，圆括号中的 C 表达式构成。

这些限制性字符串有哪些，为什么我们需要他们？你应该知道每一条汇编指令只接受特定类型的操作数。例如：跳转指令期望的跳转目标地址。不是所有的内存地址都是有效的。因为最后的 opcode 只接受24位偏移。但矛盾的是跳转指令和数据交换指令都希望寄存器中存储的是32位的目标地址。在所有的例子中，C 传给 operand 的可能是函数指针。所以面对传给内嵌汇编的常量、指针、变量，编译器必须要知道怎样组织到汇编代码中。

对于 ARM 核的处理器，GCC 4 提供了一下的限制。

Constraint	Usage in ARM state	Usage in Thumb state
f	Floating point registers f0 .. f7	Not available
h	Not available	Registers r8..r15
G	Immediate floating point constant	Not available
H	Same a G, but negated	Not available
I	Immediate value in data processing instructions e.g. ORR R0, R0, #operand	Constant in the range 0 .. 255 e.g. SWI operand
J	Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand
l	Same as r	Registers r0..r7 e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand
O	Not available	Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand
r	General register r0 .. r15 e.g. SUB operand1, operand2, operand3	Not available
w	Vector floating point registers s0 .. s31	Not available
X	Any operand	

限制字符可能要单个 modifier 指示。要是没有 modifier 指示的默认为 read-only

operand。

Modifier	Specifies
=	Write-only operand, usually used for all output operands
+	Read-write operand, must be listed as an output operand
&	A register that should be used for output only

Output operands 必须为 write-only，相应 C 表达式的值必须是左值。Input operands 必须为 read-only。C 编译器是没有能力做这个检查。

比较严格的规则是：不要试图向 input operand 写。但是如果你想要使用相同的 operand 作为 input 和 output。限制性 modifier（+）可以达到效果。例子如下：

```
asm("mov %[value], %[value], ror #1" : [value] "+r" (y))
```

和上面例子不一样的是，最后的结果存储在 input variable 中。

可能 modifier + 不支持早期的编译器版本。庆幸的是这里提供了其他解决办法，该方法在最新的编译器中依然有效。对于 input operators 有可能使用单一的数字 n 在限制字符串中。使用数字 n 可以告诉编译器使用的第 n 个 operand，operand 都是以0开始计数。下面是例子：

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

限制性字符串“0”告诉编译器，使用和第一个 output operand 使用同样 input register。

请注意，在相反的情况下不会自动实现。如果我没告诉编译器那样做，编译器也有可能为 input 和 output 选择相同的寄存器。第一个例子中就为 input 和 output 选择了 r3。

在多数情况下这没有什么，但是如果在 input 使用前 output 已经被修改过了，这将是致命的。在 input 和 output 使用不同寄存器的情况下，你必须使用&modifier 来限制 output operand。下面是代码示例：

```
asm volatile("ldr %0, [%1]" "\n\t"
             "str %2, [%1, #4]" "\n\t"
```

```
            : "=&r" (rdv)
            : "r" (&table), "r" (wdv)
            : "memory");
```

在以张表中读取一个值然后在写到该表的另一个位置。

其他内嵌汇编作为预处理宏要是经常使用使用部分汇编，最好的方法是将它以宏的形式定义在头文件中。使用该头文件在严格的 ANSI 模式下会出现警告。为了避免该类问题，可以使用__asm__代替 asm，__volatile__代替 volatile。这可以等同于别名。下面就是个例程：

```
#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 桩函数宏定义包含的是相同的代码。这在大型 routine 中是不可以接受的。这种情况下最好定义个桩函数。

```
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 变量的符号名默认的情况下，GCC 使用同函数或者变量相同的符号名。你可以使用 asm 声明，为汇编代码指定一个不同的符号名

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

这个声明告诉编译器使用了符号名 clock 代替了具体的值。

替换 C 函数的符号名为了改变函数名，你需要一个原型声明，因为编译器不接受在函数定义中出现 asm 关键字。

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

调用函数 *calc()*将会创建调用函数 *CALCULATE* 的汇编指令。

强制使用特定的寄存器局部变量可能存储在一个寄存器中。你可以利

用内嵌汇编为该变量指定一个特定的寄存器。

```
void Count(void) {

register unsigned char counter asm("r3");


... some code...

asm volatile("eor r3, r3, r3");

... more code...

}
```

汇编指令“eor r3, r3, r3”，会将 r3清零。Waring：该例子在到多数情况下是有问题的，因为这将和优化器相冲突。因为 GCC 不会预留其它寄存器。要是优化器认为该变量在以后一段时间没有使用，那么该寄存器将会被再次使用。但是编译器并没有能力去检查是否和编译器预先定义的寄存器有冲突。如果你用这种方式指定了太多的寄存器，编译器将会在代码生成的时候耗尽寄存器的。

临时使用寄存器

如果你使用了寄存器，而你没有在 input 或 output operand 传递，那么你就必须向编译器指明这些。下面的例子中使用 r3作为 scratch 寄存器，通过在 clobber list 中写 r3，来让编译器得知使用该寄存器。由于 ands 指令跟新了状态寄存器的标志位，使用 cc 在 clobber list 中指明。

```
asm volatile(

    "ands r3, %1, #3" "\n\t"

    "eor %0, %0, r3" "\n\t"

    "addne %0, #4"

    : "=r" (len)

    : "0" (len)

    : "cc", "r3"
```

```
);
```

最好的方法是使用桩函数并且使用局部临时变量。

寄存器的用途比较好的方法是分析编译后的汇编列表，并且学习 C 编译器生成的代码。下面的列表是编译器将 ARM 核寄存器的典型用途，知道这些将有助于理解代码。

Register	Alt. Name	Usage
r0	a1	First function argument Integer function result Scratch register
r1	a2	Second function argument Scratch register
r2	a3	Third function argument Scratch register
r3	a4	Fourth function argument Scratch register
r4	v1	Register variable
r5	v2	Register variable
r6	v3	Register variable
r7	v4	Register variable
r8	v5	Register variable
r9	v6 rfp	Register variable Real frame pointer
r10	sl	Stack limit
r11	fp	Argument pointer

r12	ip	Temporary workspace
r13	sp	Stack pointer
r14	lr	Link register Workspace
r15	pc	Program counter

转自：http://blogold.chinaunix.net/u2/69404/showart_1922655.html