<http://blog.csdn.net/liangkaiming/article/details/6259189>

arch/arm/tools/mach-types中的大概379行：存放的是Uboot传递给内核的机器号

我的[mini2440](https://www.baidu.com/s?wd=mini2440&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWrj04PWuhPyPBPHI-mHcd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1D3rj6Yn1R1Ps" \t "_blank)对应的是：u-boot-2009.03/include/asm-arm/mach-types.h

内联汇编

**asm　volatile("Instruction List"**

**: Output**

**: Input**

**: Clobber/Modify);**

* =，表示只写，也是output必须具备的（看见=，就是output）
* r表示寄存器
* m表示内存
* flags为变量名

\_\_asm\_\_ \_\_volatile\_\_内嵌汇编用法简述 在阅读C/C++原码时经常会遇到内联汇编的情况，下面简要介绍下\_\_asm\_\_ \_\_volatile\_\_内嵌汇编用法。因为我们华清远见教学平台是ARM体系结构的，所以下面的示例都是用ARM汇编。

带有C/C++表达式的内联汇编格式为：

\_\_asm\_\_　\_\_volatile\_\_("Instruction List" : Output : Input : Clobber/Modify);

其中每项的概念及功能用法描述如下：

1、 \_\_asm\_\_

\_\_asm\_\_是GCC 关键字asm 的宏定义：

#define \_\_asm\_\_ asm

\_\_asm\_\_或asm 用来声明一个内联汇编表达式，所以任何一个内联汇编表达式都是以它开头的，是必不可少的。

2、Instruction List

Instruction List 是汇编指令序列。它可以是空的，比如：\_\_asm\_\_ \_\_volatile\_\_(""); 或 \_\_asm\_\_ ("");都是完全合法的内联汇编表达式，只不过这两条语句没有什么意义。但并非所有Instruction List 为空的内联汇编表达式都是没有意义的，比如：\_\_asm\_\_ ("":::"memory");

就非常有意义，它向GCC 声明：“内存作了改动”，GCC 在编译的时候，会将此因素考虑进去。 当在"Instruction List"中有多条指令的时候，可以在一对引号中列出全部指令，也可以将一条 或几条指令放在一对引号中，所有指令放在多对引号中。如果是前者，可以将每一条指令放在一行，如果要将多条指令放在一行，则必须用分号（；）或换行符（\n）将它们分开. 综上述：（1）每条指令都必须被双引号括起来 (2)两条指令必须用换行或分号分开。

例如： 在ARM系统结构上关闭中断的操作

int disable\_interrupts (void)   
        {   
                unsigned long old,temp;   
                \_\_asm\_\_ \_\_volatile\_\_("mrs %0, cpsr\n"   
                                "orr %1, %0, #0x80\n"   
                                "msr cpsr\_c, %1"   
                                : "=r" (old), "=r" (temp)   
                                :   
                                : "memory");   
                return (old & 0x80) == 0;   
        }

3. \_\_volatile\_\_

\_\_volatile\_\_是GCC 关键字volatile 的宏定义

#define \_\_volatile\_\_ volatile

\_\_volatile\_\_或volatile 是可选的。如果用了它，则是向GCC 声明不允许对该内联汇编优化，否则当 使用了优化选项(-O)进行编译时，GCC 将会根据自己的判断决定是否将这个内联汇编表达式中的指令优化掉。

4、 Output

Output 用来指定当前内联汇编语句的输出

例如：从arm协处理器p15中读出C1值

static unsigned long read\_p15\_c1 (void)   
        {   
                unsigned long value;   
                \_\_asm\_\_ \_\_volatile\_\_(   
                                "mrc p15, 0, %0, c1, c0, 0 @ read control reg\n"   
                                : "=r" (value) @编译器选择一个R\*寄存器   
                                :   
                                : "memory");   
        #ifdef MMU\_DEBUG   
                printf ("p15/c1 is = %08lx\n", value);   
        #endif   
                return value;   
        }

5、 Input

Input 域的内容用来指定当前内联汇编语句的输入Output和Input中，格式为形如“constraint”(variable)的列表（逗号分隔)

例如：向arm协处理器p15中写入C1值

static void write\_p15\_c1 (unsigned long value)   
        {   
        #ifdef MMU\_DEBUG   
                printf ("write %08lx to p15/c1\n", value);   
        #endif   
                \_\_asm\_\_ \_\_volatile\_\_(   
                                "mcr p15, 0, %0, c1, c0, 0 @ write it back\n"   
                                :   
                                : "r" (value) @编译器选择一个R\*寄存器   
                                : "memory");   
                read\_p15\_c1 ();   
        }

6.、Clobber/Modify

有时候，你想通知GCC当前内联汇编语句可能会对某些寄存器或内存进行修改，希望GCC在编译时能够将这一点考虑进去。那么你就可以在Clobber/Modify域声明这些寄存器或内存。这种情况一般发生在一个寄存器出现在"Instruction List"，但却不是由Input/Output操作表达式所指定的，也不是在一些Input/Output操作表达式使用"r"约束时由GCC 为其选择的，同时此寄存器被"Instruction List"中的指令修改，而这个寄存器只是供当前内联汇编临时使用的情况。

例如：

\_\_asm\_\_ ("mov R0, #0x34" : : : "R0");

寄存器R0出现在"Instruction List中"，并且被mov指令修改，但却未被任何Input/Output操作表达式指定，所以你需要在Clobber/Modify域指定"R0"，以让GCC知道这一点。

因为你在Input/Output操作表达式所指定的寄存器，或当你为一些Input/Output操作表达式使用"r"约束，让GCC为你选择一个寄存器时，GCC对这些寄存器是非常清楚的——它知道这些寄存器是被修改的，你根本不需要在Clobber/Modify域再声明它们。但除此之外， GCC对剩下的寄存器中哪些会被当前的内联汇编修改一无所知。所以如果你真的在当前内联汇编指令中修改了它们，那么就最好在Clobber/Modify 中声明它们，让GCC针对这些寄存器做相应的处理。否则有可能会造成寄存器的不一致，从而造成程序执行错误。

如果一个内联汇编语句的Clobber/Modify域存在"memory"，那么GCC会保证在此内联汇编之前，如果某个内存的内容被装入了寄存器，那么在这个内联汇编之后，如果需要使用这个内存处的内容，就会直接到这个内存处重新读取，而不是使用被存放在寄存器中的拷贝。因为这个 时候寄存器中的拷贝已经很可能和内存处的内容不一致了。

这只是使用"memory"时，GCC会保证做到的一点，但这并不是全部。因为使用"memory"是向GCC声明内存发生了变化，而内存发生变化带来的影响并不止这一点。

例如：

int main(int \_\_argc, char\* \_\_argv[])   
        {   
        int\* \_\_p = (int\*)\_\_argc;   
        (\*\_\_p) = 9999;   
        \_\_asm\_\_("":::"memory");   
        if((\*\_\_p) == 9999)   
        return 5;   
        return (\*\_\_p);   
        }

本例中，如果没有那条内联汇编语句，那个if语句的判断条件就完全是一句废话。GCC在优化时会意识到这一点，而直接只生成return 5的汇编代码，而不会再生成if语句的相关代码，而不会生成return (\*\_\_p)的相关代码。但你加上了这条内联汇编语句，它除了声明内存变化之外，什么都没有做。但GCC此时就不能简单的认为它不需要判断都知道 (\*\_\_p)一定与9999相等，它只有老老实实生成这条if语句的汇编代码，一起相关的两个return语句相关代码。

另外在linux内核中内存屏障也是基于它实现的include/asm/system.h中

# define barrier() \_asm\_\_volatile\_("": : :"memory")

主要是保证程序的执行遵循顺序一致性。呵呵，有的时候你写代码的顺序，不一定是最终执行的顺序，这个是处理器有关的。