

## 1.MDK 中的 char 类型的取值范围是？

在 MDK 中，默认情况下，char 类型的数据项是无符号的，所以它的取值范围是 0~255。它们可以显式地声明为 signed char 或 unsigned。因此，定义有符号 char 类型变量，必须用 signed 显式声明。我曾读过一本书，其中有一句话：“signed 关键字也是很宽宏大量，你也可以完全当它不存在，在缺省状态下，编译器默认数据位 signed 类型”，这句话便是有争议的，我们应该对自己所用的 CPU 构架以及编译器熟练掌握。

## 2. 赋初值的全局变量和静态变量，初值被放在什么地方？

[cpp] view plaincopy

```
1. unsigned int g_unRunFlag=0xA5;
2. static unsigned int s_unCountFlag=0x5A;
```

这两行代码中，全局变量和静态变量在定义时被赋了初值，MDK 编译环境下，你知道这个初值保存在那里吗？

对于在程序中赋初值的全局变量和静态变量，程序编译后，MDK 将这些初值放到 Flash 中，紧靠在可执行代码的后面。在程序进入 main 函数前，会运行一段库代码，将这部分数据拷贝至相应 RAM 位置。若是你不小心将这些位置的数据擦除掉，嘿嘿...反正我是碰到了。

PS：后来看 ARM 的链接器，才知道 ARM 映像文件各组成部分在存储系统中的地址有两种：一种是在映像文件位于存储器中时（也就是该映像文件开始运行之前，通俗的说就是下载到 Flash 中的二进制代码）的地址，称为加载地址；一种是在映像文件运行时（通俗的说就是给板子上电，开始运行 Flash 中的程序了）的地址，称为运行时地址。赋初值的全局变量和静态变量在程序还没运行的时候，初值是被放在 Flash 中的，这个时候他们的地址称为加载地址，当程序运行后，这些初值会从 Flash 中拷贝到 RAM 中，这时候就是运行时地址了。

3. 最新的 keil MDK (V4.54) 在编辑界面中已经可以支持中文编码了，所以可以在编辑器中直接输入汉字和中文标点符号，再也不会显示乱码或者不显示了。虽然乱写汉字和中文标点编译时依然会报错，但好歹能显示，也从侧面说明中国市场的崛起。开启方法见

<http://blog.csdn.net/zhzht19861011/article/details/7741928> 不再贴了。

我还清楚的记得自己在大学刚开始用 Keil C51 那会，一次不小心在一行代码后面用了个中文分号，在当时这个中文分号是不被显示的，然后编译，编译器报错，我双击报错信息定位到报错的代码行，却怎么也检查不出来错误来，当时着急的心情现在想想还很好笑的，那个时候只能将错误代码行用双斜杠注释掉，才能看到那个中文分号。但从 V4.54 之后，就应该再不会遇到我当时的情况了。

4. 不知道从什么版本开始, keil MDK 的标题栏可以显示工程路径了, 我是从 V4.10 直接升级到 V4.54, V4.10 的标题栏还是下图的这个样子:



如果你同一个工程有多个备份, 你有同时打开了多个备份工程, 要想识别出那个工程是那个备份, 可是件不容易的事情, 还好, keil 更新较快.

5. 这一条真伪未知, 因为我搜索了很久都没有查证.

在一个论坛上看到的, Keil 原来是一个人名, 住在德国, 最初的 keil C51 编译器就是他开发的. 为人低调, 话不多, 但超级认真. 当然, 也超级厉害.


6. Stack 分配到 RAM 的哪个地方?

keil MDK 中, 我们只需要定义各个模式下的堆栈大小, 编译器会自动在 RAM 的空闲区域选择一块合适的地方来分配给我们定义的堆栈, 这个地方位于 RAM 的那个地方呢? 通过查看编译列表文件, 原来 MDK 将堆栈放到程序使用到的 RAM 空间的后面, 比如你的 RAM 空间从 0x4000 0000 开始, 你的程序用掉了 0x200 字节 RAM, 那么堆栈空间就从 0x4000 0200 处开始。具体的 RAM 分配, 其实你可以从编译后生成的列表文件“工程名.map”文件中查看。

7. 有多少 RAM 会被初始化?

大家可能都已经知道, 在进入 main() 函数之前, MDK 会把未初始化的 RAM 给清零的 (在程序中自己定义变量初值的见第二条), 但 MDK 会不会把所有 RAM 都初始化呢? 答案是否定的, MDK 只是把你的程序用到的 RAM 以及堆栈 RAM 给初始化, 其它 RAM 的内容是不管的。如果你要使用绝对地址访问 MDK 未初始化的 RAM, 那就要小心翼翼的了, 因为这些 RAM 的内容很可能是随机的, 每次上电都不同。至少, NXP 的 LPC2000 系列就是这样。

## 8. 还是一个新版本的变化, 还是关于版本 V4. 10 和 V4. 54

V4. 10 版本, 只要你重新打开工程, 点击“Build target files”(就这个图标:) , 编译器就会将所有文件都编译一次, 不管你的文件在这之前有没改动. 但 V4. 54 就不一样了, 再次打开文件, 点击“Build target files”它会只编译改过的文件的. 早该这么做了, 每次打开工程都要编译个十几秒钟, 着实等的难受.

## 9. 好个一丝不苟的编译器

这是个十分奇葩的问题, 碰巧被我遇到了, 我承认是我代码写的不够规范, 但正是这个不规范的代码, 才得以发现这个奇葩的事件. 实在忍不住用了两个奇葩来形容. 把过程简化一下, 如下所述:

假如你的工程至少有两个 .c 文件, 其中一个为 timer.c, 里面有个定时器中断程序, 每 10ms 中断一次, 定义一个变量来统计定时器中断次数:

[cpp] view plaincopy

```
1. unsigned int unIdleCount;
```

还有一个 timer.h 文件, 里面是一些 timer.c 模块的封装, 其中变量 unIdleCount 就被封装在里面:

[cpp] view plaincopy

```
1. extern unsigned int unIdleCount;
```

在 main.c 函数中, 包含 timer.h 文件, 并利用定时器变量 unIdleCount 来精确延时 2 秒, 代码如下:

[cpp] view plaincopy

```
1. unIdleCount=0;
2.
3. while(unIdleCount!=200);    //延时 2S 钟
```

keil MDK V5.54 下编译, 默认优化级别, 编译后下载到硬件平台. 你会发现, 代码在

[cpp] view plaincopy

```
1. while(unIdleCount!=200);
```

处陷入了死循环. 反汇编, 代码如下:

[plain] view plaincopy

```
1.      122:      unIdleCount=0;
2.      123:
3.  0x00002E10  E59F11D4  LDR        R1,[PC,#0x01D4]
4.  0x00002E14  E3A05000  MOV        R5,#key1(0x00000000)
5.  0x00002E18  E1A00005  MOV        R0,R5
6.  0x00002E1C  E5815000  STR        R5,[R1]
7.      124:      while(unIdleCount!=200);    //延时 2S 钟
8.      125:
9.  0x00002E20  E35000C8  CMP        R0,#0x000000C8
10. 0x00002E24  1AFFFFFD  BNE        0x00002E20
```

重点看最后两句汇编代码，寄存器 R0 是当前变量 unIdleCount 的值，汇编指令 CMP 为比较指令，如果 R0 中的内容与 0xC8 不等，则循环。但是这里并没有更新寄存器 R0 的代码，也就是说变量 unIdleCount 的值虽然在变化，但跟 0xC8 一直比较的却是内容不变的 R0。因为之前变量 unIdleCount 被清零，所以 R0 的内容也是 0，永远不等于 0xC8，永远不会跳出循环。

看到这里，也许你已经笑翻了：你这个小白，这很明显是没用 volatile 修饰变量 unIdleCount 造成的！！！不错，比起从 RAM 中读写数据，ARM 或其它硬件从寄存器读取数据要快的多的多的多... 因此编译器会“自作主张”的将某些变量读到寄存器中，再次运算时也优先从寄存器中读取，上面的例子就是这样。解决这样的方法是用关键字 volatile 修饰你不想让编译器优化的变量，明白的告诉编译器：你不准优化我，每次使用我你都要本本分分的从 RAM 中读取或写入 RAM。

所以先不要笑，我是不会犯这种错误的，之所以从这里说起，是为了照顾下还不知道 volatile 关键字的。。。

其实在 timer.c 中我是这样定义统计定时器中断次数变量的：

[cpp] view plaincopy

```
1. unsigned int volatile unIdleCount;
```

但是，在 timer.h 中，我确偷了个懒，声明这个变量的代码如下：

[cpp] view plaincopy

```
1. extern unsigned int unIdleCount;
```

没有使用关键字 volatile，在 keil MDK V5.54 下编译，默认优化级别，然后查看代码的反汇编，如下所示：

[plain] view plaincopy

```

1.    122:    unIdleCount=0;
2.    123:
3.  0x00002E10  E59F11D4  LDR        R1,[PC,#0x01D4]
4.  0x00002E14  E3A05000  MOV        R5,#key1(0x00000000)
5.  0x00002E18  E1A00005  MOV        R0,R5
6.  0x00002E1C  E5815000  STR        R5,[R1]
7.    124:    while(unIdleCount!=200);    //延时 2S 钟
8.    125:
9.  0x00002E20  E35000C8  CMP        R0,#0x000000C8
10. 0x00002E24  1AFFFFFFD BNE        0x00002E20

```

可以看出，这个反汇编代码居然和没加 `volatile` 关键字的时候一模一样！！代码还是会在 `while` 出陷入死循环。

现在，应该知道我要表达的意思了吧，如果引用的变量声明中没有使用 `volatile` 关键字修饰，即便定义这个变量的时候使用了 `volatile` 关键字修饰，MDK 编译器照样优化掉它！

将 `timer.h` 中的声明更改为：

[cpp] view plaincopy

```
1.  extern unsigned int volatile unIdleCount;
```

同样环境下编译，查看反汇编代码，如下所示：

[plain] view plaincopy

```

1.    122:    unIdleCount=0;
2.    123:
3.  0x00002E10  E59F01D4  LDR        R0,[PC,#0x01D4]
4.  0x00002E14  E3A05000  MOV        R5,#key1(0x00000000)
5.  0x00002E18  E5805000  STR        R5,[R0]
6.    124:    while(unIdleCount!=200);    //延时 2S 钟
7.    125:
8.  0x00002E1C  E5901000  LDR        R1,[R0]
9.  0x00002E20  E35100C8  CMP        R1,#0x000000C8
10. 0x00002E24  1AFFFFFFC BNE        0x00002E1C

```

看最后三句汇编代码，发现多了一个载入汇编指令 `LDR`，这个指令在每次循环中都将变量 `unIdleCount` 从 RAM 中读出到寄存器 `R1` 中，然后 `R1` 的值再和 `0xC8` 比较。这才是符合逻辑的需要的代码。

其实如果好好看看编译原理的书,是不会犯这么低级的错误的,编译器是分文件编译,然后链接,文件 A 使用了文件 B 中定义的变量,在编译的时候,文件 A 是完全不知道文件 B 里面有什么东西的,只能通过文件 B 的接口文件(.h 文件)来获得使用变量的属性.

以这个为例子，着重说明下关键字 `volatile`，同时也要掌握编译原理的知识，用好手中的工具。

### 10. 关于 `float` 类型

在 keil 中,在不选择“Optimize for time”编译选项时,局部 `float` 变量占用 8 个字节(编译器默认自动扩展成 `double` 类型),如果你从 Flash 中读取一个 `float` 类型常量并放在局部 `float` 型变量中时,有可能发生意想不到的错误:Cortex-M3 中可能会出现硬 fault. 因为字节对齐问题.

但有趣的是,一旦你使用“Optimize for time”编译选项,局部 `float` 变量只会占用 4 个字节.

### 11. 默认情况下,从按下复位到执行你编写的 C 代码 `main` 函数,keil mdk 做了些什么?

硬件复位后,第一步是执行复位处理程序,这个程序的入口在启动代码里(默认),摘录一段 cortex-m3 的复位处理入口代码:

[plain] view plaincopy

1.	<code>Reset_Handler</code>	<code>PROC</code>	<code>;PROC 等同于 FUNCTION,表示一个函数的开始,与 ENDP 相对?</code>
2.		<code>EXPORT</code>	<code>Reset_Handler [WEAK]</code>
3.	<code>IMPORT</code>	<code>SystemInit</code>	
4.		<code>IMPORT</code>	<code>__main</code>
5.	<code>LDR</code>	<code>R0, =SystemInit</code>	
6.		<code>BLX</code>	<code>R0</code>
7.		<code>LDR</code>	<code>R0, =__main</code>
8.		<code>BX</code>	<code>R0</code>
9.		<code>ENDP</code>	

这里 `SystemInit` 函数是我自己用 C 代码写的硬件底层时钟初始化代码,这个可不算是 keil mdk 给代劳的.初始化堆栈指针、执行完用户定义的底层初始化代码后,发现接下来的代码是调用了 `__main` 函数,这里之所以有 `__main` 函数,是因为在 C 代码中定义了 `main` 函数,函数标签 `main()` 具有特殊含义。`main()` 函数的存在强制链接器链接到 `__main` 和 `__rt_entry` 中的初始化代码。

其中, `__main` 函数执行代码和数据复制、解压缩以及 `ZI` 数据的零初始化。解释一下, C 代码中,已经赋值的全局变量被放在 `RW` 属性的输入节中,这些变量的初值被 keil mdk 压缩后放到 `ROM` 或 `Flash` 中 (`RO` 属性输入节)。什么是赋值的全局变量呢? 如果你在代码中这样定义一个全局变量: `int nTimerCount=20`; 变量 `nTimerCount` 就是已经赋值的变量, 如果是这样定义: `int nTimerCount`; 变量 `nTimerCount` 就是一个非赋值的变量, keil 默认将它放到属性为 `ZI` 的输入节。为什么要压缩呢? 这

是因为如果赋值变量较多，会占用较多的 Flash 存储空间，keil 默认用自己的压缩算法。这个“解压缩”就是将存放在 RO 输入区（一般为 ROM 或 Flash）的变量初值，按照一定算法解压缩后，拷贝到相应 RAM 区。ZI 数据清零是指将 ZI 区的变量所在的 RAM 区清零。使用 UNINIT 属性对执行区进行标记可避免 \_\_main 对该区域中的 ZI 数据进行零初始化。这句话很重要，比如我有一些变量，保存一些重要信息，不希望复位后就被清零，这时就可以用分散加载文件定义一块 UNINIT 属性的区，将不希望零初始化的变量定义到这个区即可。

## 12. 关于新版本 V4.70

期盼已久的功能终于在 V4.7 实现了！！IDE 升级到了 **µVision V4.70.00**，增加了代码和参数自动补全以及动态语法检验,增加了两个性能分析命令。J-LINK 驱动更新到 V4.62，编译器版本更新到 5.03.其中我是最喜欢的是代码和参数自动补全以及动态语法检验。刚回答了一个百度知道提问，提问者问到 V4.70a 为什么不能自动代码和参数补全，这里也说一下，这并不是 4.70a 的问题,4.70a 相对于 4.70 只是修正了"linking in MDK-ARM user guides"问题,使能自动补全的功能要设置一下:点击 Edit-Configuration...,在打开的对话框中选中 Text Completion 标签栏,在此页面中选中 symbols after 复选框即可完全开启(安装后默认并没有选中这个框).补充，你的系统至少要有 vc++2010 运行库，才能看到这个设置。

另外 V4.70 自带的动态语法校验我也非常喜欢,在输入的时候就可以检查出我输入的变量名称,标点,函数名等等对不对了.推荐升级测试一下。

我做嵌入式行业，编程也多和硬件打交道，好多人说编译器只是工具，重要的在于算法和思想。这话说的本来没错，但要有一个条件在先：那就是你真正掌握了你所用的编译器。但就我来看，真正熟悉编译器的却并不多见。当你深入了解一个编译器后，你能像用汇编一样用 C，可以像汇编那样随心所欲的操作 MCU！

了解一个编译器，首先应该有汇编的基础，不要求能用汇编编写程序或做过项目，但至少看的懂！不熟悉汇编的嵌入式程序员是不合格的程序员！

了解一个编译器，最好的方法是看它自带的帮助文件，至少要看过 Compiler User's Guide，至少遇到问题会想到到帮助中查找方法，虽然帮助大多是 E 文。

工作以来一直使用 keil MDK 编译器,对于这个编译器的界面以及设置,可以参考博文：<http://blog.csdn.net/zhzht19861011/article/details/5964827>，在这里先来看一看 keil MDK 编译器的一些细节，看看这些细节，你知道多少。

1. 在所有的内部和外部标识符中，大写和小写字符不同。

2. 默认情况下, `char` 类型的数据项是无符号的。它们可以显式地声明为 `signed char` 或 `unsigned char`。

3.基本数据类型的大小和对齐:

类型	位大小	按字节自然对齐
<code>char</code>	8	1
<code>short</code>	16	2
<code>int</code>	32	4
<code>long</code>	32	4
<code>long long</code>	64	8
<code>float</code>	32	4
<code>double</code>	64	8
<code>long double</code>	64	8
所有指针	32	4
<code>bool</code> (仅用于 C++)	8	1
<code>_Bool</code> (仅用于 C)	8	1
<code>wchar_t</code> (仅用于 C++)	16	2

注: a. 通常局部变量保留在寄存器中, 但当局部变量太多放到栈里的时候, 它们总是字对齐的。例如局部 `char` 变量在栈里以 4 为边界对齐;

b. 压缩类型的自然对齐方式为 1。使用关键字 `__packed` 来压缩特定结构, 将所有有效类型的对齐边界设置为 1。

4. 整数以二进制补码形式表示; 浮点量按 IEEE 格式存储。

5. 有符号量的右移是算术移位, 即移位时要保证符号位不改变。

6. 对于 `int` 类的值: 超过 31 位的左移结果为零; 无符号值或正的有符号值超过 31 位的右移结果为零。负的有符号值移位结果为-1。



7. 整数除法的余数的符号于被除数相同，由 ISO C90 标准得出；

8. 如果整型值被截断为短的有符号整型，则通过放弃适当数目的最高有效位来得到结果。如果原始数是太大的正或负数，对于新的类型，无法保证结果的符号将于原始数相同。所以强制类型转化的时候，对转换的结果一定要清晰。

9. 整型数超界不引发异常；像 `unsigned char test; test=1000;` 这类是不会报错的，赋值或计算时务必小心。

10. 默认情况下，整型数除以零返回零。

11. 对于两个指向相同类型和对齐属性的指针相减，计算结果如下表达式所示：

$((\text{int})a - (\text{int})b) / (\text{int})\text{sizeof}(\text{指向数据的类型})$

12. 在严格 C 中，枚举值必须被表示为整型，例如，必须在 -2147483648 到 +2147483647 的范围内。但 keil MDK 自动使用对象包含 enum 范围的最小整型来实现（比如 char 类型），除非使用编译器命令 `--enum_is_int` 来强制将 enum 的基础类型设为至少和整型一样宽。超出范围的枚举值默认仅产生警告：`#66: enumeration value is out of "int" range`

13. 结构体：struct {

```
char c;

short s;

int x;

} //这个结构体占 8 个字节
```

但是，结构体：

```
struct {

char c;

int x;

short s;
```

```
} //这个结构体占 12 个字节
```

这是为什么？

对于结构体填充，据定义结构的方式，keil MDK 编译器用以下方式的一种来填充结构：

- 定义为 **static** 或者 **extern** 的结构用零填充；
- 栈或堆上的结构，例如，用 `malloc()` 或者 **auto** 定义的结构，使用先前存储在那些存储器位置的任何内容进行填充。不能使用 `memcmp()` 来比较以这种方式定义的填充结构！

14. 编译器不对声明为 **volatile** 类型的数据进行优化。 我发现还有不少刚入门的嵌入式程序员从没见过这个关键字。

15. `__nop()`：延时一个指令周期，编译器绝不会优化它。如果硬件支持 **NOP** 指令，则该句被替换为 **NOP** 指令，如果硬件不支持 **NOP** 指令，编译器将它替换为一个等效于 **NOP** 的指令，具体指令由编译器自己决定。

16. 还有一些编译器知识，我放在了另外一篇博文里，