KEIL v8.02

KEIL 如何分配内存

渤海三叠浪著

第一个话题：函数的参数

```
void tryOne(INT8U i, INT8U k);

INT8U j;
void main(void)
{
    while (1) {
        tryOne(1, 2);
    }
}
void tryOne(INT8U i, INT8U k)
{
    j = i + 1 + k;
}
```

选择 3 级别 （注：第 4 级别为寄存器优化）

TYPE BASE LENGTH RELOCATION SEGMENT NAME

```
* * * * * * *   D A T A   M E M O R Y   * * * * * * *
REG      0000H     0008H     ABSOLUTE      "REG BANK 0"
DATA     0008H     0002H     UNIT          _DATA_GROUP_
DATA     000AH     0001H     UNIT          ?DT?KEIL_COMPILER
IDATA    000BH     0001H     UNIT          ?STACK
```

选择 4 级别

TYPE BASE LENGTH RELOCATION SEGMENT NAME

```
* * * * * * *   D A T A   M E M O R Y   * * * * * * *
REG      0000H     0008H     ABSOLUTE      "REG BANK 0"
DATA     0008H     0001H     UNIT          ?DT?KEIL_COMPILER
IDATA    0009H     0001H     UNIT          ?STACK
```

通过对比，可以看出这个_DATA_GROUP 指的是什么东西了！

因寄存器没有被优化而增加的变量叫做_DATA_GROUP

而 static 变量和全局变量叫做?DT?KEIL_COMPILER

下面这段话再次证明了这一点（阅读下面这段话的同时，请注意看上面截图的的 _ 和 ？）

缺省的，C函数在寄存器中最多传递三个参数。余下的参数通过固定存储区传递。可以用**NOREGPARMS**命令取消用寄存器传递参数。如果用寄存器传递参数取消，或参数太多，参数通过固定存储区传递。用寄存器传递参数的函数在生成代码时被**Cx51**编译器在函数名前加了一个下划线（'_'）的前缀。只在固定存储区传递参数的函数没有下划线。参考166页的“用SRC命令”。

刚才有实验了一下 0-2 级别。发现了 SEGMENT NAME 又是不同的名字！哈哈明白了 是我命名的函数的名字

唉！情况太多了！实在没有必要弄得过于清楚！

现在在 8 级优化情况下！

```
INT8U j;
void tryOne(INT8U i, INT8U k, INT8U x)
{
    j = i + 1 + k + x;
}
void main(void)
{
    while (1) {
        tryOne(1, 2, 3);
    }
}
```

```
TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * * *  D A T A  M E M O R Y  * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
DATA    0008H     0001H     UNIT         ?DT?KEIL_COMPILER
IDATA   0009H     0001H     UNIT         ?STACK
```

```
INT8U j;
void tryOne(INT8U i, INT8U k, INT8U x, INT8U y)
{
    j = i + 1 + k + x + y;
}
void main(void)
{
    while (1) {
        tryOne(1, 2, 3, 4);
    }
}
```

```
TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * * *  D A T A  M E M O R Y  * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
DATA    0008H     0004H     UNIT         _DATA_GROUP_
DATA    000CH     0001H     UNIT         ?DT?KEIL_COMPILER
IDATA   000DH     0001H     UNIT         ?STACK
```

可以看出多了一个参数，竟然多出了 4 个固定存储空间！
可见。编译器的编译原理太复杂了！实在是没有可能也没有必要搞明白！
知道大概的意思就可以啦！

第二个话题：普通函数的局部变量

局部变量

如可能，在循环和别的临时计算中用局部变量。作为优化的一部分，编译器尝试保存局部变量在寄存器中。寄存器访问是最快的存储访问。这通常用 **unsigned char** 和 **unsigned int** 变量类型达到。

没有必要试验了，知道这么回事就行！

第三个话题：中断函数的局部变量

中断函数的局部变量如何被编译器处理呢？

找了半天，才**间接**找到了证据！

未调用函数

在开发过程中，通常会有写了但不调用的函数。编译器允许这样而不产生错误，但同时连接/定位器也暂时不处理这些代码，因为支持数据覆盖，只生成一个警告。

中断函数不被调用，他们由硬件调用。一个未调用的程序被连接器作为一个可能的中断程序。这意味着函数的局部变量被分配在一个不可覆盖数据区。这会很快耗尽所有可用的数据区（根据所用的存储类型决定）。

如果不希望用尽存储区，必须检查和未调用或未使用程序相关的连接器警告。可以用连接器的 **IXREF** 命令在连接器影射（.M51）文件包含一个交叉参考列表。

上面这段话间接说明了即使在选择了数据覆盖优化级别（级别>=2）情况下，中断函数的局部变量也会被分配到一个不可覆盖区域。

下面就来举例证明一下。

首先设定编译器的优化级别是 8 级（注：第 2 级别是数据覆盖）

```
05  INT8U j;
06  //void tryTwo(void)
07  //{
08  //  INT8U e;
09  //  INT8U f;
10  //  INT8U g;
11  //}
12  void tryOne(INT8U i, INT8U k, INT8U x, INT8U y)
13  {
14      INT8U z = 2;
15      j = z + j;
16      j = i + 1 + k + x + y;
17  }
18  void main(void)
19  {
20      while (1) {
21          tryOne(1, 2, 3, 4);
22      }
23  }
24
25
```

KEIL COM... | KEIL COM...

```
Build target 'Target 1'
compiling KEIL COMPILER.c...
linking...
Program Size: data=14.0 xdata=0 code=44
"KEIL COMPILER" - 0 Error(s), 0 Warning(s).
```

```
TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * * *  D A T A   M E M O R Y  * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
DATA    0008H     0004H     UNIT         _DATA_GROUP_
DATA    000CH     0001H     UNIT         ?DT?KEIL_COMPILER
IDATA   000DH     0001H     UNIT         ?STACK
```

```
05 INT8U j;
06 void tryTwo(void)
07 {
08     INT8U e;
09     INT8U f;
10     INT8U g;
11 }
12 void tryOne(INT8U i, INT8U k, INT8U x, INT8U y)
13 {
14     INT8U z = 2;
15     j = z + j;
16     j = i + 1 + k + x + y;
17 }
18 void main(void)
19 {
20     while (1) {
21         tryOne(1, 2, 3, 4);
22     }
23 }
24
```

KEIL COM... KEIL COM...

```
Build target 'Target 1'
compiling KEIL COMPILER.c...
KEIL COMPILER.C(8): warning C280: 'e': unreferenced local variable
KEIL COMPILER.C(9): warning C280: 'f': unreferenced local variable
KEIL COMPILER.C(10): warning C280: 'g': unreferenced local variable
linking...
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?TRYTWO?KEIL_COMPILER
Program Size: data=17.0 xdata=0 code=45
"KEIL COMPILER" - 0 Error(s), 4 Warning(s).
```

```
TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * * * *  D A T A  M E M O R Y  * * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
DATA    0008H     0004H     UNIT         _DATA_GROUP_
DATA    000CH     0003H     UNIT         ?DT?TRYTWO?KEIL_COMPILER
DATA    000FH     0001H     UNIT         ?DT?KEIL_COMPILER
IDATA   0010H     0001H     UNIT         ?STACK
```

好！这个问题很重要，那么再弄个中断函数试试！

首先设定编译器的优化级别是 8 级（注：第 2 级别是数据覆盖）

```
06  void tryTwo(void)
07  {
08      INT8U e;
09      INT8U f;
10      INT8U g;
11  }
12  void tryOne(INT8U i, INT8U k, INT8U x, INT8U y)
13  {
14      INT8U z = 2;
15      j = z + j;
16      j = i + 1 + k + x + y;
17  }
18  void main(void)
19  {
20      RCAP2H  = 0x4C; RCAP2L  = 0x00;TR2 = 1;
21      ET2  = 1; EA = 1;
22      while (1) {
23          tryOne(1, 2, 3, 4);
24      }
25  }
26
27
28
29
30  void Timer0_Server(void) interrupt 5
31  {
32      INT8U a;
33      INT8U b;
34      if (TF2) {
35          TF2 = FALSE;
36          a = 1;
37          b = 0;
38      }
39  }
40
41
```

这两句要是不要了，就是6个警告喽！

```
Output Window
Build target 'Target 1'
compiling KEIL COMPILER.c...
KEIL COMPILER.C(8): warning C280: 'e': unrefer
KEIL COMPILER.C(9): warning C280: 'f': unrefer
KEIL COMPILER.C(10): warning C280: 'g': unrefe
linking...
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR
    SEGMENT: ?PR?TRYTWO?KEIL_COMPILER
Program Size: data=19.0 xdata=0 code=72
"KEIL COMPILER" - 0 Error(s), 4 Warning(s).
Build  Command
```

```
TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
-----------------------------------------------------

* * * * * * *   D A T A   M E M O R Y   * * * * * * *
REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
DATA    0008H     0006H     UNIT         _DATA_GROUP_
DATA    000EH     0003H     UNIT         ?DT?TRYTWO?KEIL_COMPILER
DATA    0011H     0001H     UNIT         ?DT?KEIL_COMPILER
IDATA   0012H     0001H     UNIT         ?STACK
```

通过对比，一切都很清晰了！

下面再随便说点别的：

- 名称最多为 256 个字符。C 语言是大小写敏感的。但是为了兼容，目标文件中的所有名称但是大写字母。因此一个源程序的外部目标名的大小写是无关紧要的。

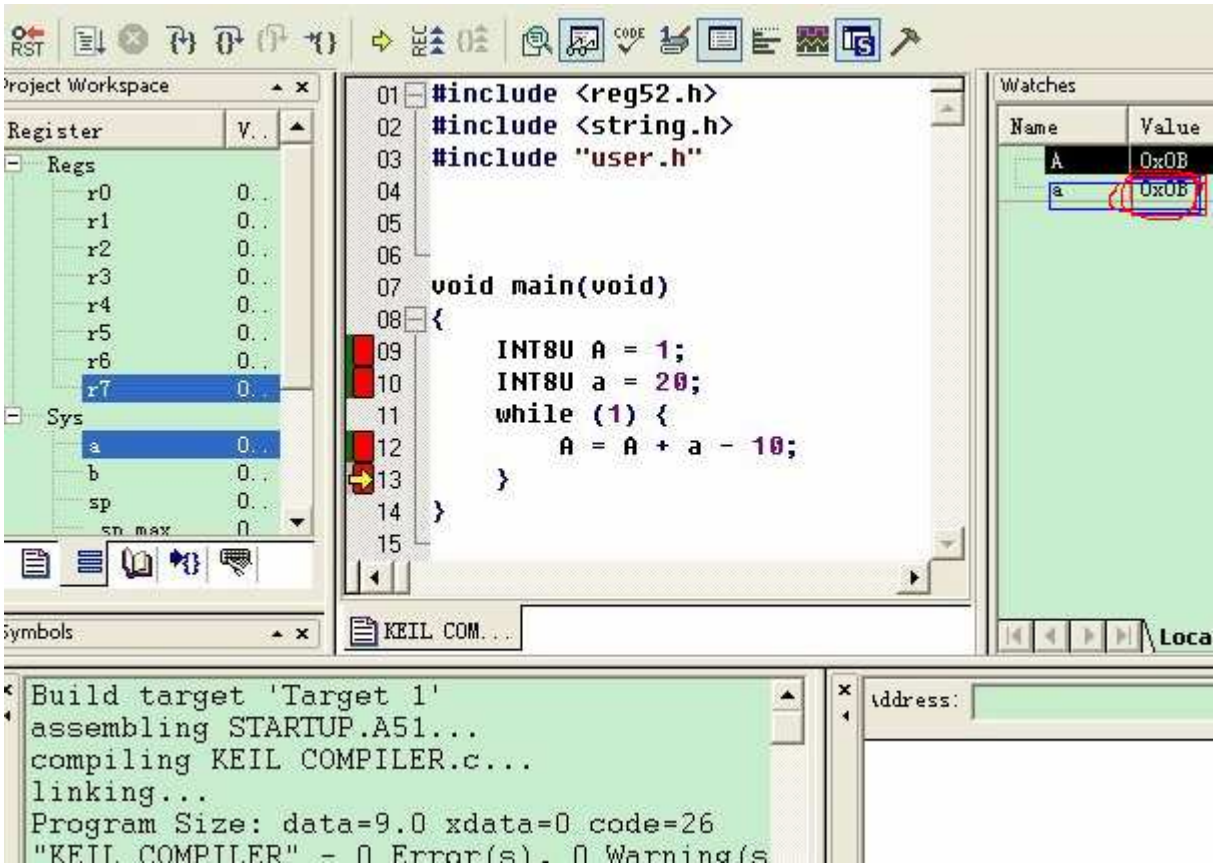
变量命名这个最大为 256 字符，这是从 **版本 6.0** 开始的，以前的版本只支持 32 个！

版本 6.0 差异

- **取除外部和段的限制**
  每个模块的外部符号和段的数目不再限制在 256 内。这个历史的限制是旧的 INTEL 目标文件格式强制的。
- **变量名可以有 256 个字符**
  现在一个变量名可以有 256 个字符，以前只能有 32 个字符。

当然这只是版本 6.0 差异的一部分了，另一部分没粘贴过来！
变量名 256 个，这是我弄过！我试验了大概是 240 多个！反正够用就行了!!

上文说 C 语言（当然指的是 51 的 C 语言喽）对大小写是敏感的！可以试验一下：



怎么样！编译通过了吧！程序也运行正确！我说的是编译器编译的是正确的！但是可以看到软件仿真器的小 a 的值是有问题的！这只是仿真器的问题！编译器还是编译正确的！

```
void main(void)
{
    INT8U A = 1;
    INT8U a = 20;
    while (1) {
        A = A + a - 10;
        while (a - 20);
    }
}
```

通过

```
void main(void)
{
    INT8U A = 1;
    INT8U a = 20;
    while (1) {
        A = A + a - 10;
        while (a - 19);
    }
}
```

和

，就能知道 a 的值是多少了！

当然 a 肯定是 20 了！

好，再看几个版本差异：

版本 5 的差异

- **优化级别 7，8，和 9**
  C51 提供三个新的优化级别。这些新的优化主要集中于代码密度。参考 157 页的“优化”。

- **支持双 DPTR 的命令**
  C51 对 ATMEL，ATMEL WM，和 PHILIPS 提供双 DPTR 支持，用命令 **MODA2** 和 **MODP2**。

- **data，pdata，xdata 自动变量在所有存储模式中可覆盖**
  C51 现在可覆盖所有的 data，pdata，和 xdata 自动变量，无论选了什么存储模式。在以前的 C51 版本中，只有缺省存储模式的自动变量可覆盖。例如，如果一个函数用 SMALL 存储模式编译，C51 版本 5 不覆盖 pdata 或 xdata 变量。

可见，以后再也不用关系优化级别 7、8、9 了！
关于说得存储模式，因为默认是 LARGE 模式，并且我都是选的 LARGE 模式！
注：以上是版本 5 的部分差异！

版本 3.4 差异

- **优化级别 6**
  C51 支持优化级别 6，提供循环旋转。结果代码更有效和运行更快。

哈哈，既然是代码优化，以后也没有必要关系了！

好，继续说**函数参数和堆栈**！

函数参数和堆栈

在传统的 8051 中堆栈指针只能访问内部数据区。**Cx51** 编译器把堆栈定位在内部数据区的所有变量的后面。堆栈指针间接访问内部存储区，可以使用 0xFF 前的所有内部数据区。

传统 8051 的总的堆栈空间是受限的：最多只有 256 字节。除了用堆栈传递函数参数，**Cx51** 编译器对每个函数参数分配一个特定地址。当函数被调用时，调用者在传递控制权前必须拷贝参数到分配好的存储区。函数就可从固定的存储区提取参数。在这个过程中只有返回地址保存在堆栈中。中断函数要求更多的堆栈空间，因为必须切换寄存器组，需要保存一些寄存器值在堆栈中。

上面这段话**很重要！要结合相关内容理解！**

寄存器传递参数

用寄存器传递参数

Cx51 编译器允许用 CPU 寄存器传递三个参数。这可以明显的提高系统性能。参数传递可以用 **REGPARMS** 和 **NOREGPARMS** 控制命令来控制。

下面的表列出了不同参数位置和数据类型所用的寄存器。

参数数目	char，1字节指针	int，2字节指针	long，float	通用指针
1	R7	R6&R7	R4-R7	R1-R3
2	R5	R4&R5	R4-R7	R1-R3
3	R3	R2&R3		R1-R3

如果没有寄存器可用来传递参数，固定存储区被使用。

函数返回值

CPU 寄存器经常用来返回函数值。下面的表列出了返回类型和所用的寄存器。

返回类型	寄存器	说明
bit	CF	
char，unsigned char， 1字节指针	R7	
int，unsigned int，2字节指针	R6&R7	MSB在R6，LSB在R7
long，unsigned long	R4-R7	MSB在R4，LSB在R7
float	R4-R7	32位IEEE格式
通用指针	R1-R3	存储类型在R3，MSB R2，LSB R1

注意:

*如果函数的第一个参数是一个 **bit** 类型，那么别的参数不能用寄存器传递。这是因为寄存器传递参数不符合上面的计划。因此，**bit** 参数应该在参数的最后声明。*

注意到了吧!! 这个注意说得可有点意思！不过我还从来没有用过 bit 作为参数呢！

资料引用部分来自《Keil Software – Cx51 编译器用户手册 中文完整版》（403 页）.pdf