Keil 的关键字 using 与寄存器组的切换

渤海三叠浪

基础知识：

基础 1

单片机复位后的状态：

单片机的复位操作使单片机进入初始化状态，其中包括使程序计数器PC＝0000H，这表明程序从0000H地址单元开始执行。单片机冷启动后，片内RAM为随机值，运行中的复位操作不改变片内RAM区中的内容，21个特殊功能寄存器复位后的状态为确定值，见下表。
值得指出的是，记住一些特殊功能寄存器复位后的主要状态，对于了解单片机的初态，减少应用程序中的初始化部分是十分必要的。
说明：表中符号*为随机状态；
A＝00H，表明累加器已被清零；

特殊功能寄存器	初始状态	特殊功能寄存器	初始状态
A	00H	TMOD	00H
B	00H	TCON	00H
PSW	00H	TH0	00H
SP	07H	TL0	00H
DPL	00H	TH1	00H
DPH	00H	TL1	00H
P0~P3	FFH	SBUF	不定
IP	***00000B	SCON	00H
IE	0**00000B	PCON	0*******B

PSW＝00H，表明选寄存器0组为工作寄存器组；
SP＝07H，表明堆栈指针指向片内RAM 07H字节单元，根据堆栈操作的先加后压法则，第一个被压入的内容写入到08H单元中；
P0-P3＝FFH，表明已向各端口线写入1，此时，各端口既可用于输入又可用于输出；
IP＝×××00000B，表明各个中断源处于低优先级；
IE＝0×××00000B，表明各个中断均被关断；

(顺便提一句与 using 无关的内容：《MCS—51 系列单片微型计算机及其应用》P24 说：SBUF 初始状态是“不变”。个人感觉还是用“不变”更好些！)

好！让我们看看在 ***keil 是默认设置***下，startup.a51 都起了些什么作用？

首先编个程序，如下：

```
void main(void)
{
    while (1) {
    }
}
```

看看该程序汇编代码，如下：

```
C:0x0000    020003    LJMP    C:0003
C:0x0003    787F      MOV     R0,#0x7F
C:0x0005    E4        CLR     A
C:0x0006    F6        MOV     @R0,A
C:0x0007    D8FD      DJNZ    R0,C:0006
C:0x0009    758107    MOV     SP(0x81),#0x07
C:0x000C    02000F    LJMP    main(C:000F)
     4: void main(void)
C:0x000F    80FE      SJMP    main(C:000F)
```

这个汇编第一个作用就是将 0x00~0x7F 这段内存清零!(请清零 0x7F，然后清零 0x7E，。。。。最后清零 0x00)，这个汇编第二个作用就是设置了堆栈的起始位置。可以看到 sp 的值是 0x07。

那么就可能提出疑问了？既然单片机复位后 SP 值就是 0x07，在 startup.A51 里面再次让 sp = 0x07，这不是多此一举么？答：对于这个程序来说确实是多此一举。但是当程序中定义 占空间的变量了，那么 sp 的值就不是 0x07 了!至于 sp 的值到底是几？就由编译器进行内存分配后，由 startup.a5 里面的程序来计算了(注：这句话可能说的不够准确)!

比如说这个程序，如下，

```
unsigned char i;
void main(void)
{
    while (1) {
    }
}
```

对应的汇编如下，

```
C:0x0000    020003   LJMP     C:0003
C:0x0003    787F     MOV      R0,#0x7F
C:0x0005    E4       CLR      A
C:0x0006    F6       MOV      @R0,A
C:0x0007    D8FD     DJNZ     R0,C:0006
C:0x0009    758108   MOV      SP(0x81),#i(0x08)
C:0x000C    02000F   LJMP     main(C:000F)
     4: void main(void)
C:0x000F    80FE     SJMP     main(C:000F)
```

看到了吧，SP = 0x08 了。因为设置了个全局变量 i，于是编译器在该全局变量之后的设置 sp。

基础 2

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。

实战演练：

看看寄存器组转换的小演示。。

优化级别 8 编译器 c51.8.18

```c
#include <reg52.h>
#include <string.h>
#include "user.h"
INT8U Hello;

void tryOne(INT8U a, INT8U b);
void tryTwo(INT8U c, INT8U d, INT8U e);

void main(void)
{

    while (1) {
        tryOne(1, 2);
        while (Hello == 6);
    }
}

void tryOne(INT8U a, INT8U b) using 1
{
    tryTwo(a, b, 6);
}

void tryTwo(INT8U c, INT8U d, INT8U e) using 1
{
    Hello = c + d + e;
}
```

Watches

Name	Value
Hello, 0x0A	6
<type F2 to...	

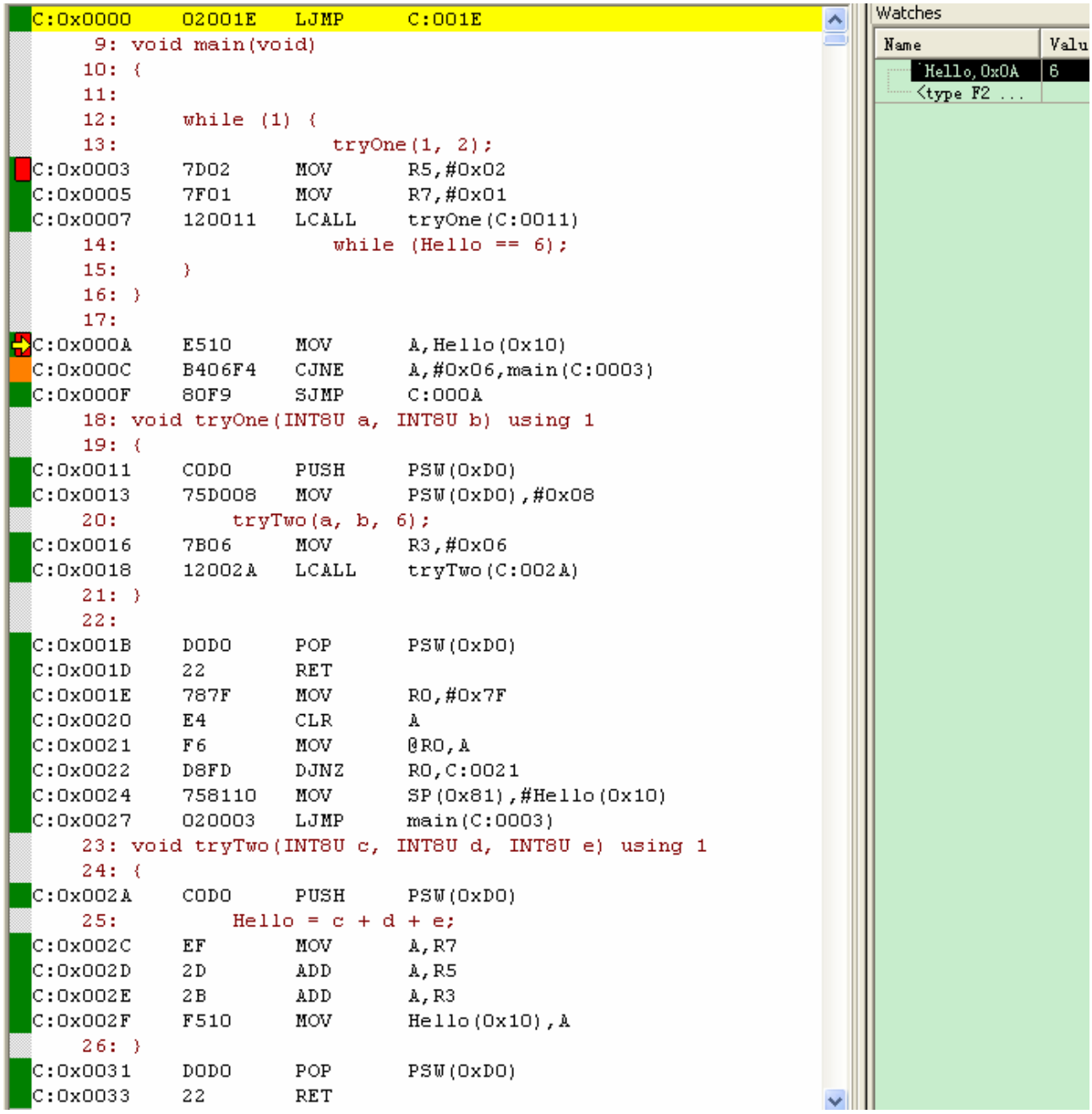
看出来了吧！出问题了！Hello 竟然等于 6.
如果不用我们不强制用 using 这个关键字的话，Hello 应该等于 9 的。

---------- 这里贴个 output windows 的图（优化级别 8）！----------

```
compiling main.C...
linking...
Program Size: data=18.0 xdata=0 code=52
creating hex file from "Port1 (1)"...
"Port1 (1)" - 0 Error(s), 0 Warning(s).
```

猜想一下：肯定是由于用了 同一个寄存器组 1，所以后面的将前面的值覆盖了！

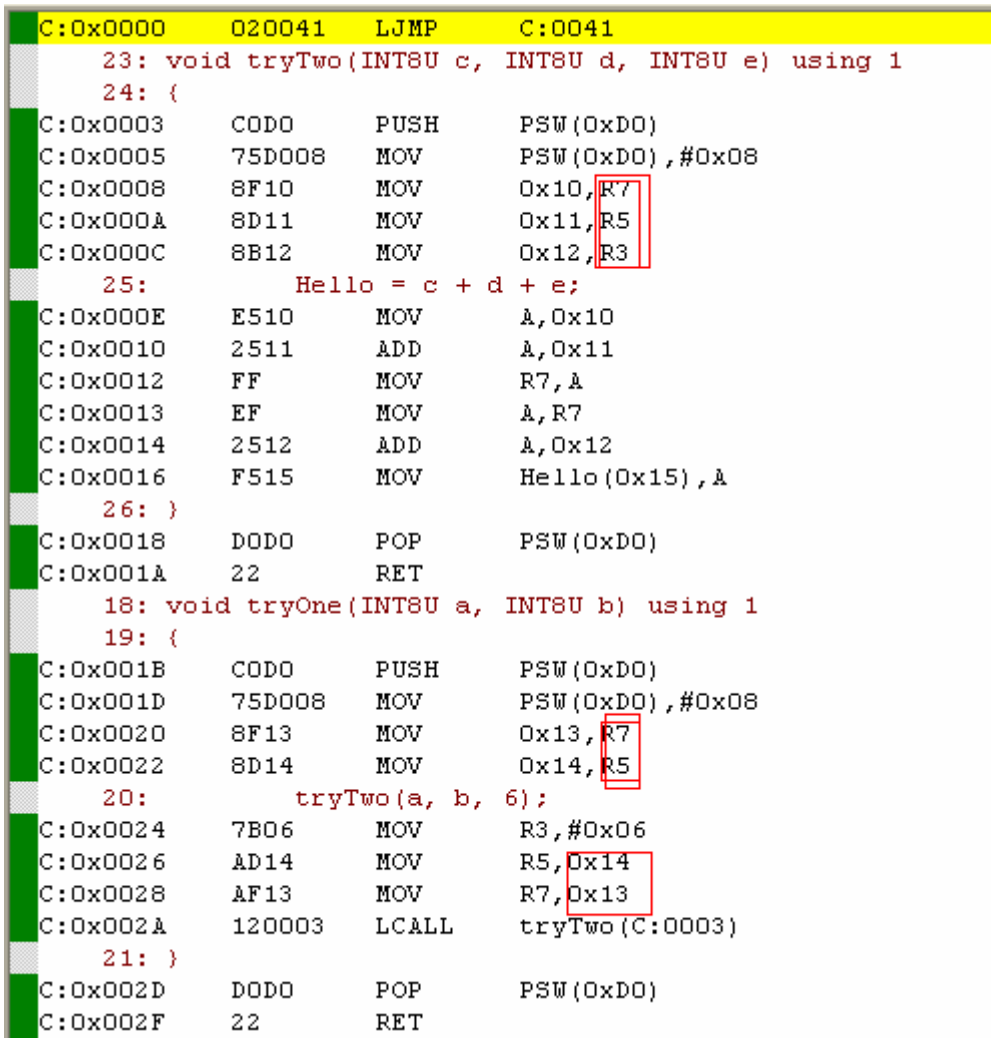
看看汇编吧

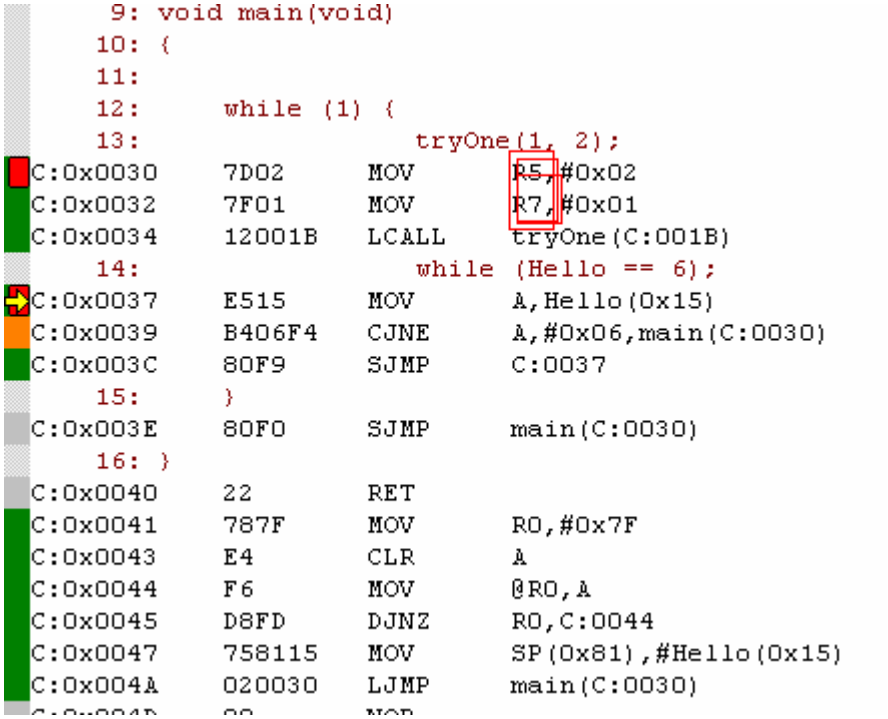


在汇编中看到 PSW = 0x08 也就是 RS1 = 0, RS0 = 1 选择了寄存器组 1.

好像这个汇编看的不是很清晰啊！和我猜想一下所描述的似乎不太一致！

对了！应该是优化级别有点高，将汇编都优化了！好现在将优化级别设置为 0


C:0x0000 020041 LJMP C:0041
23: void tryTwo(INT8U c, INT8U d, INT8U e) using 1
24: {
C:0x0003 C0D0 PUSH PSW(0xD0)
C:0x0005 75D008 MOV PSW(0xD0),#0x08
C:0x0008 8F10 MOV 0x10,R7
C:0x000A 8D11 MOV 0x11,R5
C:0x000C 8B12 MOV 0x12,R3
25: Hello = c + d + e;
C:0x000E E510 MOV A,0x10
C:0x0010 2511 ADD A,0x11
C:0x0012 FF MOV R7,A
C:0x0013 EF MOV A,R7
C:0x0014 2512 ADD A,0x12
C:0x0016 F515 MOV Hello(0x15),A
26: }
C:0x0018 D0D0 POP PSW(0xD0)
C:0x001A 22 RET
18: void tryOne(INT8U a, INT8U b) using 1
19: {
C:0x001B C0D0 PUSH PSW(0xD0)
C:0x001D 75D008 MOV PSW(0xD0),#0x08
C:0x0020 8F13 MOV 0x13,R7
C:0x0022 8D14 MOV 0x14,R5
20: tryTwo(a, b, 6);
C:0x0024 7B06 MOV R3,#0x06
C:0x0026 AD14 MOV R5,0x14
C:0x0028 AF13 MOV R7,0x13
C:0x002A 120003 LCALL tryTwo(C:0003)
21: }
C:0x002D D0D0 POP PSW(0xD0)
C:0x002F 22 RET


9: void main(void)
10: {
11:
12: while (1) {
13: tryOne(1, 2);
C:0x0030 7D02 MOV R5,#0x02
C:0x0032 7F01 MOV R7,#0x01
C:0x0034 12001B LCALL tryOne(C:001B)
14: while (Hello == 6);
C:0x0037 E515 MOV A,Hello(0x15)
C:0x0039 B406F4 CJNE A,#0x06,main(C:0030)
C:0x003C 80F9 SJMP C:0037
15: }
C:0x003E 80F0 SJMP main(C:0030)
16: }
C:0x0040 22 RET
C:0x0041 787F MOV R0,#0x7F
C:0x0043 E4 CLR A
C:0x0044 F6 MOV @R0,A
C:0x0045 D8FD DJNZ R0,C:0044
C:0x0047 758115 MOV SP(0x81),#Hello(0x15)
C:0x004A 020030 LJMP main(C:0030)

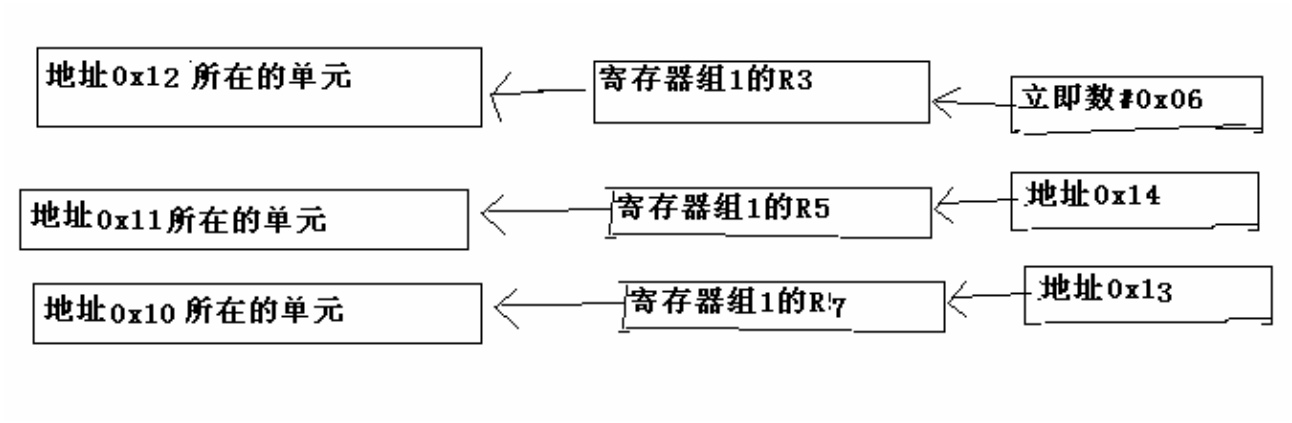
这下子相当清楚了吧！

好，来分析一下整个函数的流程

先说 tryOne 函数：

（注：地址 0x13 0x14 事实上就属于工作寄存器组 2 的区域了！可见对于没有使用过的寄存器组，是被编译器当做是普通地址用了！）

然后说 tryTwo 函数：



TryTwo 函数里面的

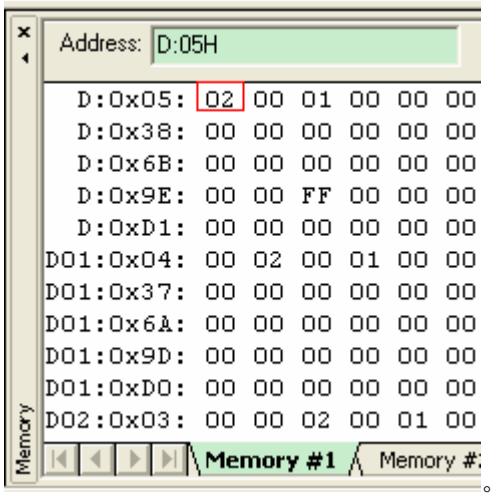
```
25:          Hello = c + d + e;
C:0x000E     E510      MOV       A,0x10
C:0x0010     2511      ADD       A,0x11
C:0x0012     FF        MOV       R7,A
C:0x0013     EF        MOV       A,R7
C:0x0014     2512      ADD       A,0x12
C:0x0016     F515      MOV       Hello(0x15),A
```

，就是说将

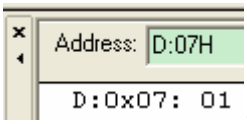
地址0x12 所在的单元 、 地址0x11所在的单元 、

地址0x10 所在的单元 相加了!!!

好函数运行 1 圈，再看看 寄存器组0的R5 的值是多少？



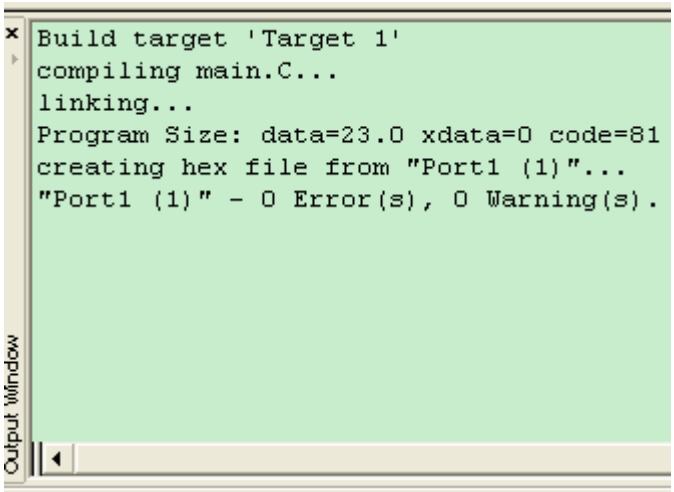
。

再看看寄存器组0的R7的值是多少？ 

。

那么，一切都搞得清楚了!!我的猜想是有问题的！不能称作是覆盖！

---------- 这里贴个 output windows 的图！（优化级别 0）----------



2009-11-30

渤海三叠浪