1. 前言

学习任何一种编程语言都不是一件简单的事情，不可能一朝一夕完成。任何一种编程语言都遍布陷阱（尽管各种编程语言的发明人尽量避免这种情况的出现）。作为编程人员，必须要清楚程序的来龙去脉，否则程序的执行结果就有可能与编写人员的初衷大相径庭，甚至造成不必要的财产损失和人身伤害。

本文不会讲述 STEP7 软件及编程语言的详细使用方法，关于这部分内容请用户参考相关手册。

在本文中，笔者将会列举一些 STEP7 编程常见的错误，其中一些例子来自工程实例。笔者并未将其直接用于本书，而是将其精简成易于理解，易于发现错误的例子（删除工艺因素及无关逻辑程序干扰）。希望通过对这些例子的讲解，用户能够减少在编程方面的错误，能够快速地在纷繁复杂的程序中查找出错误所在，加快开发项目的进度。

相关手册地址连接：

S7-300 和 S7- 400 的梯形图 (LAD) 编程
http://support.automation.siemens.com/CN/view/zh/18654395
S7-300 和 S7-400 的语句表 (STL) 编程
http://support.automation.siemens.com/CN/view/zh/18653496
S7-300 和 S7-400 的 功能块图 (FBD) 编程
http://support.automation.siemens.com/CN/view/zh/18652644
使用 STEP 7 V5.3 编程
http://support.automation.siemens.com/CN/view/zh/18652056

2. STEP7 编程常见的错误例子

2.1. 简单错误

下面列举三个简单的 STEP7 编程常见的错误例子，这三种错误是非常常见的，但却非常容易被忽视：

2.1.1. 地址超范围

```
A       M3000.0
=       Q0.0
```

上面这两条语句相当简单，语法没有任何错误，可以下载到部分 S7-400 系列 CPU 当中，但是，如果用户试图将这两条语句下载到 CPU315-2DP 当中，那将无法完成，因为 CPU315-2DP 不支持

M3000.0 这么大的地址空间。

2.1.2. 对指令不熟悉

```
A     M     0.0
FP    M     0.0
=     Q     0.1
```

同样以上三条语句是没有任何语法错误的，本意是当 M0.0 由 0 变为 1 的上升沿位置时，Q0.1 导通一次。但由于 FP 指令是用了与 A 指令相同的地址 M0.0，所以此处 Q0.1 的状态是不会有任何变化的。

2.1.3. 地址重叠

```
A  M0.0
   =     M11.0
   ……
   L     0
   T     MW10
```

与前面的两个例子一样，这四条语句是没有语法错误的。但如果这些语句分布在程序的不同位置，由于 MW10 包括了 M11.0,用户就要考虑对 M11.0 的“写操作”与对 MW10 的“写操作”是否存在逻辑冲突的问题了。

通过以上三个简单的例子，可以得出如下结论：即使再简单的程序，也存在着出现编程错误的可能性。所以在后续的章节中，将单独列举一些 STEP7 编程中常见的错误。

2.2. 循环程序错误

用户在编程过程中，循环程序的使用是非常常见的。下面例子的初衷为实现一个 500 次的循环操作（Network2 添加的用户程序可以被执行 500 次）。但是由于没有考虑下述的几个因素，使得程序不能正确执行。

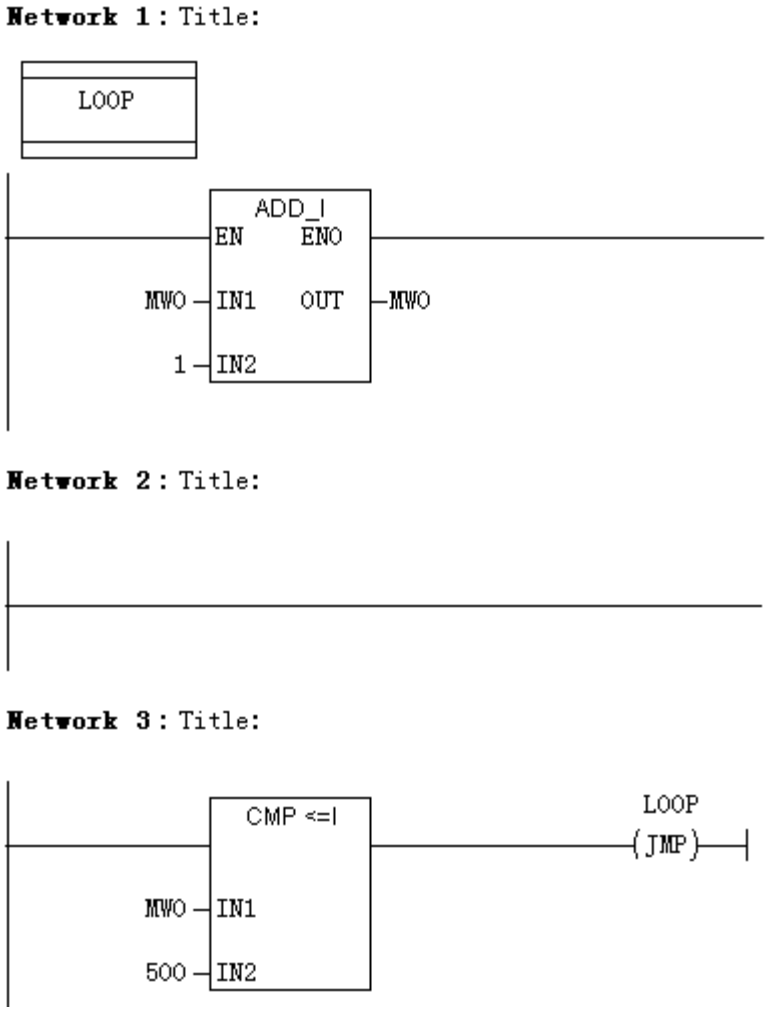

图 2-1 循环程序示例(a)

2.2.1. 循环程序初始化

首先，程序中没有 MW0 初始化的语句，一旦循环程序被多次调用将发生错误，所以应在 LOOP 标号前应增加如下语句：

```
L     0
T     MW0
```

将编程环境切换为 STL 格式后，可得到如下程序：

```
Network 1: Title:
      L     0
      T     MW     0
      NOP   0

Network 2: Title:


LOOP: L     MW     0
      L     1
      +I
      T     MW     0
      NOP   0

Network 3: Title:

// 此处可以添加用户程序

Network 4: Title:
      L     MW     0
      L     500
      <=I
      JC    LOOP
```

图 2-2 循环程序示例（b）

2.2.2. 循环程序执行时间

上图为比较常见的循环程序例子，在 network3 处添加了用户程序后，是不是所有的工作就完成了呢？答案是否定的，我们还需要考虑 CPU 的扫描执行周期问题。假设 network3 处所添加的程序为一个或数个子程序，这些程序执行时需要的时间为 A，用户其它程序执行需要的时间为 B。

那么，A*500+B 的时间总和一定要小于下图中的 Scan cycle monitoring time[ms],否则，就有可能由于程序扫描时间超出了 CPU 的监控时间而导致 CPU 停机，用户应当将硬件组态中的程序扫描监控时间更改为大于此时间的数值。

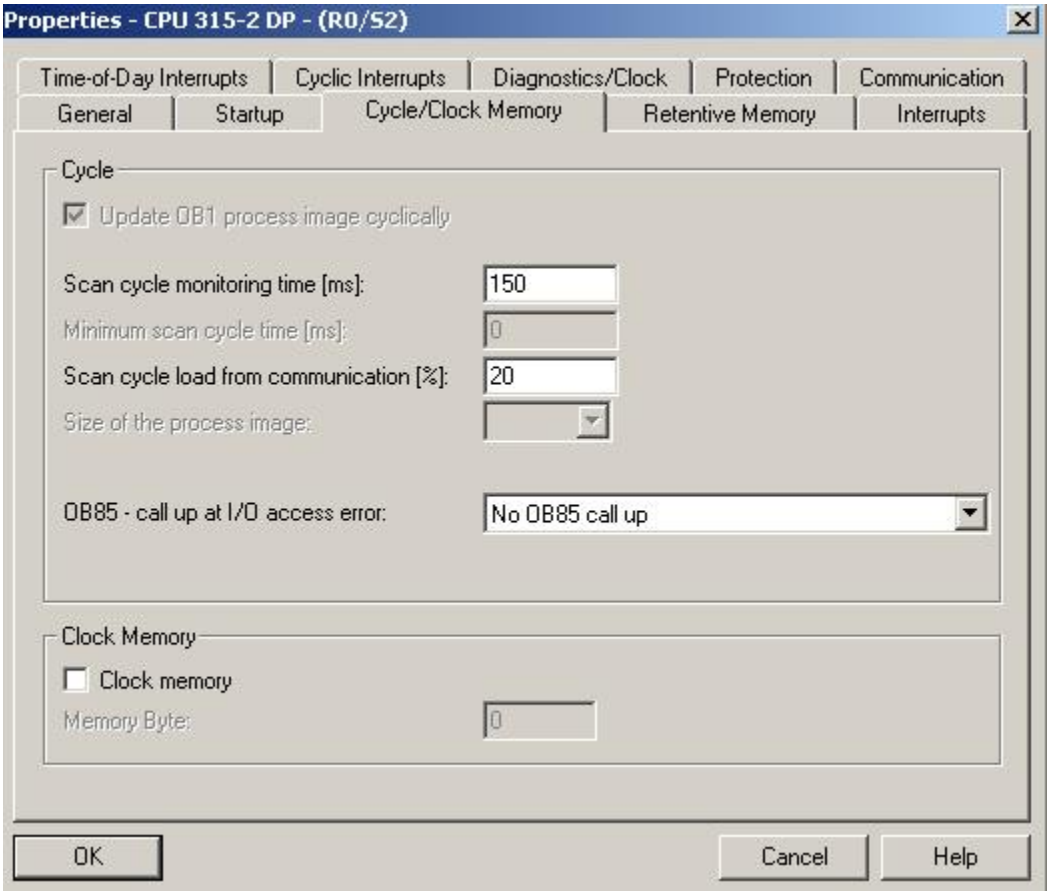

图 2-3 CPU 扫描监控时间

2.2.3. 循环程序累加控制

这里之所以将循环程序以 STL 格式显示，目的在于提示另一个在编写循环程序经常出现的错误。有些编程人员可能会使用 MB0 代替上述程序中的 MW0 ，或者不使用加法指令而使用 INC 指令。由于这两种用法都是字节操作，如果使用 MB0,由于 MB0 最大为 255，永远也不会大于 500，所以程序会出现死循环的情况。而 MW0 作为有符号整数可以最大支持到+32767，包括了 0-500 的计数范围，所以不会出现错误。因此，在下文中将强调数据类型的匹配的问题。

2.3. 数据类型匹配不严谨

对于任何一门计算机语言，都存在着数据类型匹配问题，作为一名有着良好编程习惯的编程人员应当严格遵循的数据匹配规则。

2.3.1. STL 指令数据类型匹配

由于 STL 相对于 LAD 对于数据类型匹配检查并不严格，所以要求用户在在使用 STL 编程的时候尤其要注意到这一点。

特别提示：无论是否会获得正确结果，下列程序是不应当出现在程序当中的：

```
L     0
L     1.000000e+000
```

```
    +R
T   MD      0
```

特别提示：而如下的程序的细微区别也应当引起注意：

```
    L       10
    L       L#10
L   MB10
L   MW10
```

这些操作后的执行结果可以参考 L 指令的 STEP 帮助文件及数据类型说明。

Contents of ACCU 1	ACCU1-H-H	ACCU1-H-L	ACCU1-L-H	ACCU1-L-L
before execution of load instruction	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
after execution of **L MB10** (L <Byte>)	00000000	00000000	00000000	<MB10>
after execution of **L MW10** (L <word>)	00000000	00000000	<MB10>	<MB11>
after execution of **L MD10** (L <double word>)	<MB10>	<MB11>	<MB12>	<MB13>

图 2-4 L 指令操作结果

注意：装载不同的数据类型，累加器 ACCU1 中的内容改写情况是不同的，例如装载 MW10 时，ACCU1 的高 16 位将作添加 0 的处理。

2.3.2. 数据类型匹配与浮点数运算

在流量累计编程中经常会遇到实数加法问题，实数加法运算的注意事项也应当引起编程人员的重视，请看下例程序（假设其在 OB35 中被调用，目的为每隔一定时间间隔就累计一次流量）

```
    L       MD0             //累计流量存储值
    L       MD4             //流量瞬时值
    +R
T   MD      0
```

以上的程序是否存在问题？很多人会认为没有问题，但实际情况是此程序在运行一段时间后就将出现错误。此程序在运行之初是正常的，因为累计流量初始值及流量瞬时值都为一个很小的浮点数，两数相加后，结果正确。但是当一段时间后，累计流量的数值逐渐增大，当它与瞬时流量的数值相差很远的时候，两者执行加法操作后，瞬时流量的数值将被忽略掉（如 9999990.0 与 0.2 做加法操作）。其实具备计算机常识的人都应当清楚这一点，这是由于浮点数的存储机制造成的，是所有计算机方面编程都需要考虑的问题。这个问题可以通过使用二次累加或多次累加的方法来解决。

特别提示：避免数量级相差太多的浮点数之间进行运算。

2.3.3. 浮点数运算与比较指令

用户程序中很多情况下会对浮点数运算的结果与预先设定值进行比较，例如下面的程序比较

MD 与 10.0 是否相等，如果相等的输出 Q0.0 为 1。

```
L       MD      0
L       1.000000e+001
==R
=       Q       0.0
```

此程序存在两个问题：

- 由于浮点数运算存在误差，所以 MD0 有可能非常接近于 10.0，但总是不等于 10.0，（例如 9.999999e+000）
- 如果 MD0 在 10.0 附近数值波动，有可能造成输出 Q0.0 输出频繁波动，造成输出点损坏

因此，用户应当使用一个范围比较的方法代替等于数值比较的方法（例如 9.9<MD0<10.1）。对于输出点 Q0.0，可以使用 R，S 指令并增加比较延时指令，来减少输出点的波动情况。

```
A(
L       MD      0
L       9.900000e+000
>R
)
A(
L       MD      0
L       1.010000e+001
<R
)
=       Q       0.0
```

2.4. 语句执行先后顺序

由于 PLC 扫描程序是由头至尾依次执行的，所以编程人员必须重视程序语句执行顺序对逻辑结果的影响。对于下面的程序相信大家都会判断 MW0 中最后的执行结果。但是，随着程序的复杂性的增加，以及其它干扰因素的出现，对于语句执行先后顺序引起的错误，容易被编程人员所忽视。

```
L       0
T       MW0
……  (其它省略的程序段，此时 MW0 的数值为 0)
L       10
T       MW0
……  (其它省略的程序段，此时 MW0 的数值为 10)
L  20
```

```
T       MW0
……  (其它省略的程序段，此时 MW0 的数值为 20)
```

对于下面程序的例子，大多数人会认为是正确的，但实际监控结果却不是这样。

程序原目的：

T1 定时器每秒导通一次，C1 及 C3 会每隔一秒钟，进行一次加 1 操作。

故障现象：

实际监控结果：C1 工作正常，C3 并未继续计数。

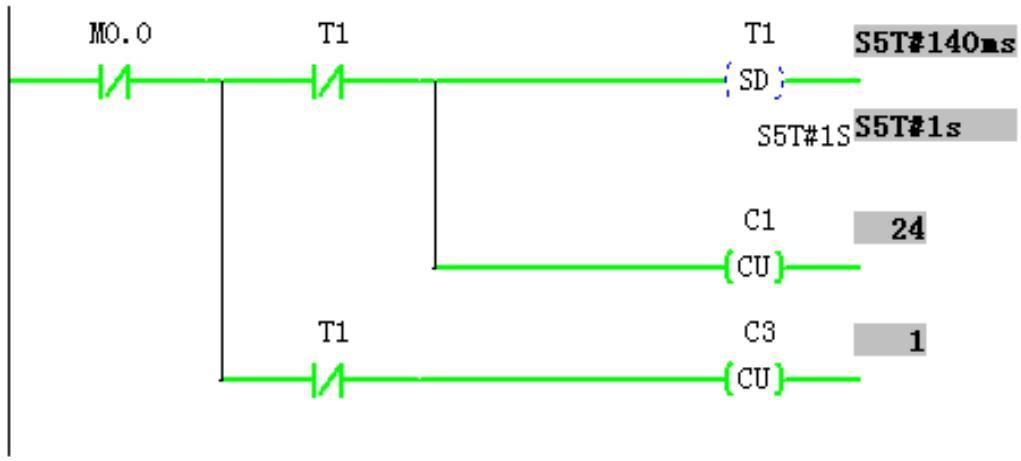

图 2-5 程序语句顺序错误

将上面的程序更改为 STL 格式，可以更清楚看出语句在程序中执行的顺序。

```
AN    M 0.0
=           L 0.0
A           L 0.0  //
AN    T 1
L           S5T#1S
SD    T 1          //此处如果 T1 定时到，则会重新计时，
CU    C 1          //C1 得到了上升沿，计数
A     L 0.0  //此处的语句将开始新的逻辑，
AN    T 1          //无法将 RLO 的 0 至 1 的变化送至 C3
CU    C 3          //C3 无法得到了上升沿，不计数
```

正确的程序反而更加精减，C1 与 C3 并列出现：

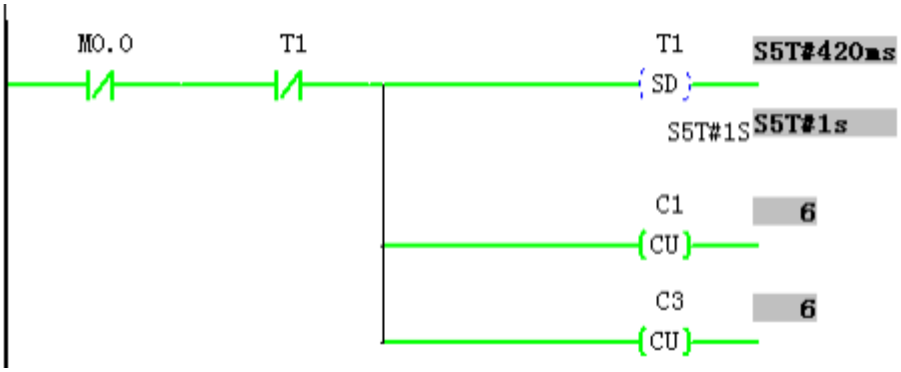

图 2-6 程序语句顺序正确

正确的程序在 STL 格式下的程序为：

```
AN    M0.0
AN    T1
L     S5T#1S
SD    T1
CU    C1
CU    C3
```

特别提示：由于 LAD 对程序的语法规则检查是自动完成的，所以编程人员可能在使用时更随意一些，但没有语法规则错误的程序并不一定会按照编程人员的本意来执行。

2.5. 上升（下降）沿不工作

上升（下降）沿不工作也是一种常见的错误，尽管手册中“P”或“N”指令允许的数据类型为：I，Q，M，L，D。但如果没有特殊目的，仅建议使用 M 及 DB 数据类型。如下图：

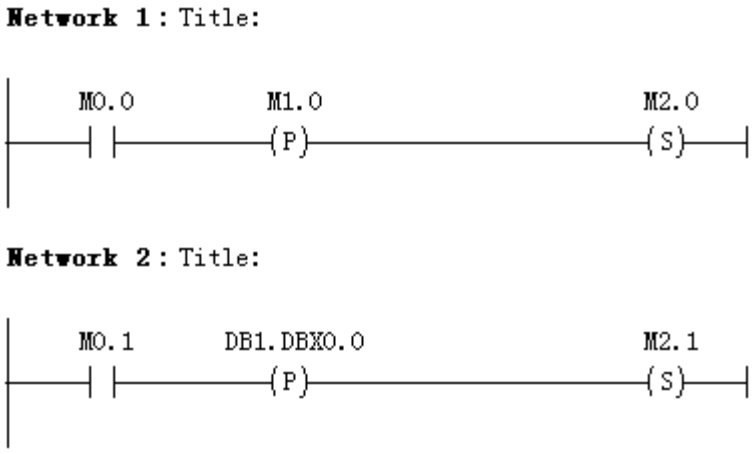

图 2-7 上升（下降）沿指令使用正确

西门子“P”指令要求使用与前面指令不相同的地址，下图为错误用法：

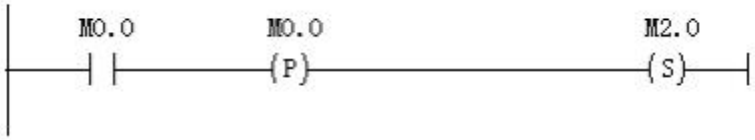

图 2-8 上升（下降）沿指令使用错误 1

“ P ” 指令要避免使用其它程序有写操作的地址，下图为错误用法：

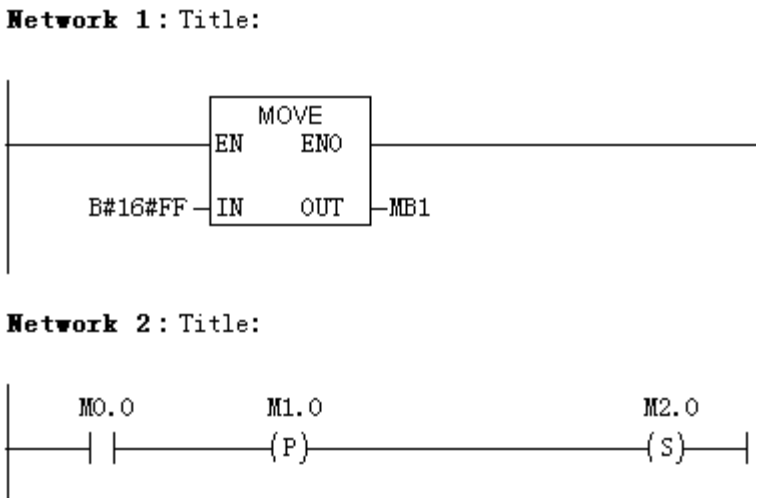

图 2-9 上升（下降）沿指令使用错误 2

“ P ” 指令不应使用临时变量作为存储地址（临时变量会随着系统堆栈变化，这一点将会在后续章节详细讲述），下图为错误用法：

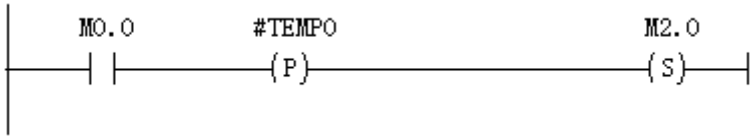

图 2-10 上升（下降）沿指令使用错误 3

2.6. 定时器不工作

有些用户在使用定时器编程后，发现定时器并没有按照自己的意图去计时工作，出现了不计时错误，进而去怀疑硬件是否故障，CPU 是否工作正常等等，浪费了大量的时间和精力。实际上这是由于用户对定时器特性不了解所造成的误解。下面的例子将说明这个问题。

程序原目的：

I4.2 的上升沿触发 T50，T60 的定时，并在 T60 定时结束后，复位 M12.0

故障现象：

I4.2 可以触发 T50，T60 的定时，但有时即使 I4.2 再次将 M12.0 置位为 1 ，T50 不计时。现象见下图：

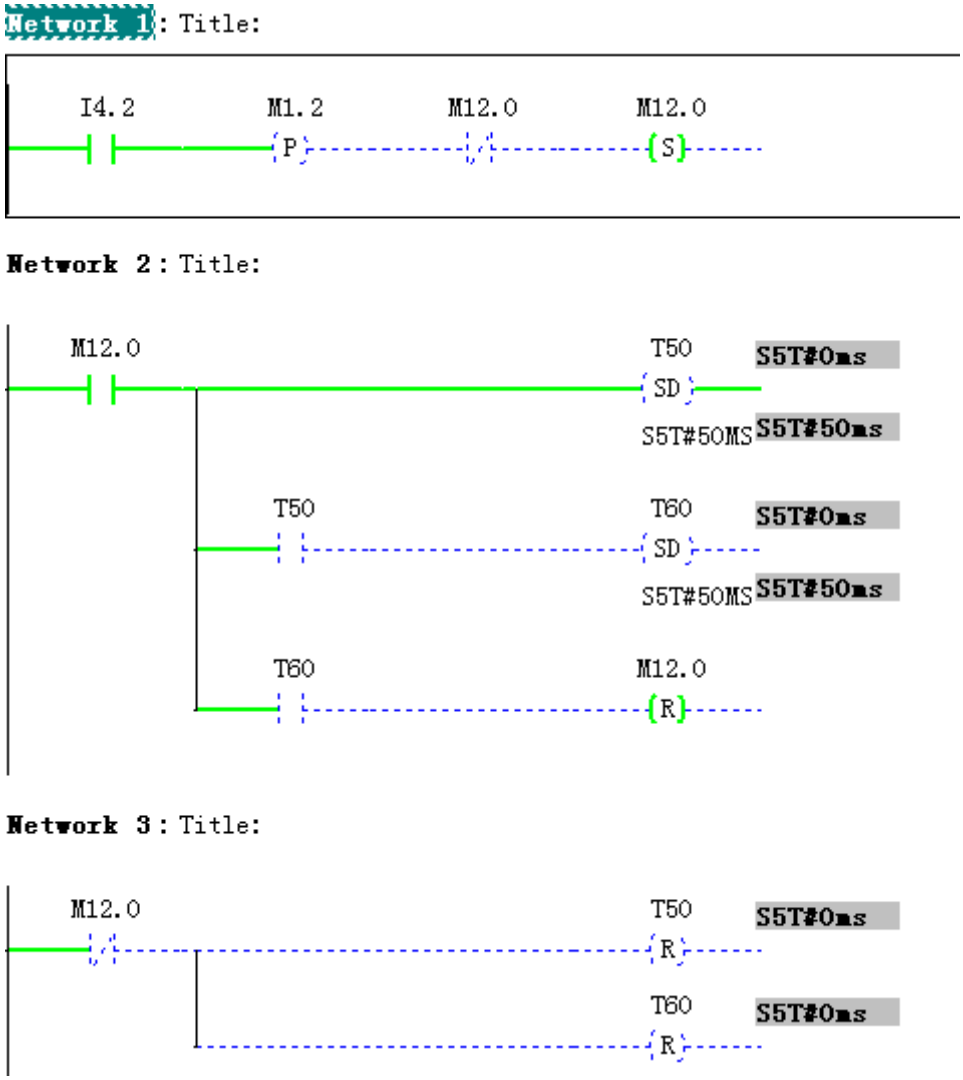

图 2-11 定时器使用错误

故障分析：

首先要明确这个故障现象既不是硬件故障，也不是语句错误所引起的，而是对定时器使用不正确引起的故障。

现在我们分析此故障是如何产生的：

1. 某个扫描周期，
    a. I4.2 的上升沿置位 M12.0，I4.2 恢复为 0
2. 数个扫描周期后，扫描周期 N
    a. 当 T60 计时到时，Network2 中 M12.0 被复位（注意是在 SD T50 语句的后面），此扫描周期末 M12.0 由 1 变为了 0
    b. Network3 中 T50,T60 被复位
3. 扫描周期 N+1

a. 如果此时 I4.2 恰恰出现上升沿置，尽管 M12.0 在上个扫描周期曾经变为 0，但在本扫描周期开始就变为了 1，定时器 T50 在上个扫描周期接受到的 M12.0 状态为 1，定时器 T50 在本扫描周期接受到的 M12.0 状态也为 1 所以 T50 将不会工作。

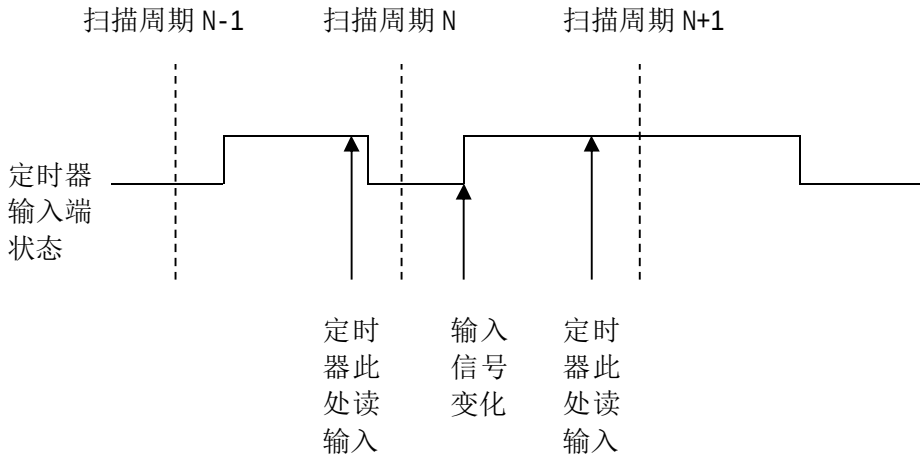

图 2-12 定时器错误分析

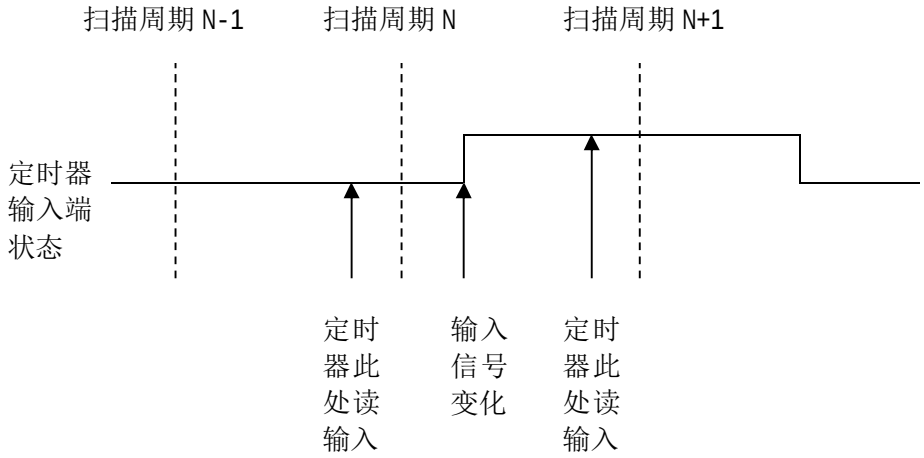

图 2-13 定时器正确使用示意

在上图中，定时器在扫描周期 N 与扫描周期 N+1 之间正确地接收到了上升沿的变化，所以能够正常工作。

故障总结：

定时器计时需要正确地接收到输入端上升沿的变化，如果没有严格遵守这一逻辑顺序，常见的故障现象为定时器不计时工作。这种故障现象可能很隐蔽，本例的原始程序在实际工作中几天

才会出现一次故障现象。由于原始程序包括大量的附加逻辑，子程序，语句位置也比较分散，所以排除此故障现象所用的时间超过了 3 天。

特别提示：此程序改正的方法非常多，例如在置位 M12.0 指令前增加一些限制条件，用户可以自己尝试。

2.7. 定时器的定时与程序扫描周期

在 S7 系列 CPU 中，定时器的最小时基为 10 毫秒。也就是说，S7 系列 CPU 的最小定时时间为 10 毫秒。如果用户程序代码量比较大，程序扫描周期过了 10 毫秒，可能会出现如下情况：尽管定时时间已经到达，但 CPU 还没有执行到相关的程序逻辑。

特别提示：当用户程序中需要非常短的定时功能时，需要考虑程序扫描周期对定时器状态读取的影响。由于 CPU 中的定时中断是由硬件来保证的，并且有高于 OB1 的优先级，所以在这种情况下，建议用户使用定时中断的功能来替代定时器的功能。

2.8. 软定时器的使用

有些时候，在编程中会遇到 CPU 提供的硬件定时器不够使用的问题，这时可以使用系统提供的软定时器，例如 SFB4(TON)。此功能块需要一个背景数据块，

特别提示：使用 SFB4 需要注意的一个问题，CPU 启动后的软定时器复位问题。

由于 SFB 的定时、计时值是存在 DB 中的，由于当 CPU 断电或停机后，DB 中的数据是保持的。如果定时器计时到在停机前已经计时，那么当 CPU 重新运行后（定时器输入端仍然为 1 的情况），定时器会在原来计时位置继续计时。为了避免这种情况的出现，可以在 OB100 中添加如下语句来初始化 SFB4 的背景数据块（参数 PT 应当为 T#0MS）。具体信息可以参考 STEP7 的在线帮助。

软定时器初始化程序：

```
CALL “TON”, DB1
 IN: =
 PT: =T#0MS
 Q: =
 ET: =
NOP   0
```

2.9. 计数器不工作

计数器使用的常见故障与定时器非常相似，可以参考定时器的章节

软计数器的使用注意事项类似于软定时器的使用，具体信息可以参考 STEP7 的在线帮助。

2.10. 数据块错误

对于 S7-300/400 系列 CPU 而言，数据块是其重要的系统资源之一。这些数据块在项目的初始阶段并不存在，编程人员自行定义后并下载至 CPU 后，即可使用。正是由于这样的特点，数据块

在使用当中存在着一些注意事项，如果忽视这些注意事项，则会引起编程错误。

2.10.1. 打开数据块错误

由于数据块是用户自己定义的，所以在下载至 CPU 之前，CPU 中是不存在着数据块的。如果用户程序中引用的数据块在 CPU 中并不存在，将导致编程错误发生。这种情况可以通过用户对自己程序的了解来避免，即不调用不存在的数据块，也可以通过使用 SFC24“TEST_DB”来检测数据块是否存在，再决定是否调用它。

2.10.2. 数据块寻址长度错误

由于数据块的长度或大小也是用户自己定义的，所以用户程序中对数据块的引用不能超出数据块所定义的范围。例如：在 DB1 中只定义了 10 个 BYTE 数据类型，那么程序中对 DB1.DBB10（第 11 个 BYTE）的访问将导致编程错误发生。这种情况可以通过用户对自己程序的了解来避免，即不引用不存在的数据地址，也可以通过使用 SFC24“TEST_DB”来检测数据块的大小，再决定如何调用它。

2.10.3. 数据块寻址不严谨错误

有些类型的 PLC 对于用户自定义的存储区域的寻址方式是有一定限制的，例如对于整数类型的数据地址是不支持位寻址方式的。而西门子 PLC 由于其数据块具有绝对的数据地址，所以可以实现对不同的数据类型地址使用各种方式寻址。例如：DB1.DBW0 被定义为一个整数数据类型，但程序中可以对 DB1.DBX0.0-DB1.DBX1.7 进行位寻址。西门子这种灵活强大的寻址功能给用户提供了非常多的选择，很多算法及数据转换功能可以非常轻松的实现，但与此同时，用户务必要注意如果不遵循数据类型进行寻址，是否会对数据内容的造成影响。

2.10.4. 数据块寄存器使用错误

西门子 S7-300/400 系列 CPU 拥有两个数据块寄存器，它们保存着当前打开的数据块编号：

DB 寄存器保存着打开的共享数据块编号

DI 寄存器保存着打开的背景数据块编号

特别提示：DI 寄存器主要用于 FB 引用背景数据块，但也常用于程序中同时打开两个数据块的操作。如下图的程序即完成了将 DB1.DBW0（16#2222）传送到 DB4.DBW0 中的工作。用户请注意程序中的格式（如 T DIW0）,详细信息请参考 STEP7 编程手册中的寻址方式部分。

			RLO	STA	STANDARD	DB1	DB2	STATUS WORD
OPN	DB	1	0	1	0	1	--	0_0000_0100
OPN	DI	4	0	1	0	1	4	0_0000_0100
L	DBW	0	0	1	2222	1	4	0_0000_0100
T	DIW	0	0	1	2222	1	4	0_0000_0100

图 2-14 同时打开两个数据块

上述例子也可以使用如下语句直接完成：

```
L     DB1.DBW    0
T     DB4.DBW    0
```

RLO	STA	STANDARD	DB1	DB2	STATUS WORD
1	1	2222	1	--	0_1000_0110
1	1	2222	4	--	0_1000_0110

图 2-15 两个数据块之间数据传送

那么上述两者有何区别呢？从两图中的对比可以看出：

```
L     DBW0          //不改变 DB 寄存器的内容
T     DIW0          //不改变 DI 寄存器的内容
L     DB1.DBW0      //改变 DB 寄存器的内容
T     DB4.DBW0      //改变 DB 寄存器的内容,相当于增加了 OPN DB4 的指令。
```

也就是说，对于数据块中地址的完整引用，将影响到 DB 寄存器的内容。那么对于下面的程序，我们将很容易发现其中的错误。

程序原目的：

将 3333 传送至 DB1.DBW0

将 4444 传送至 DB1.DBW2

```
FC8 : Title:
Network 1: Title:
      OPN   DB     1
      L     3333
      T     DBW    0

Network 2: Title:
      L     DB4.DBW    0
      L     0
      ==I
      =     Q      0.0

Network 3: Title:
      L     4444
      T     DBW    2
```

RLO	STA	STANDARD	DB1	DB2
0	1	0	1	--
0	1	3333	1	--
0	1	3333	1	--
0	1	0	4	--
0	1	0	4	--
1	1	0	4	--
1	1	0	4	--
1	1	4444	4	--
1	1	4444	4	--

图 2-16 错误传送程序

故障分析：

由于程序的 Network5 中使用了对于 DB4.DBW0 的比较指令，此指令改变了 DB 寄存器的内容，因而我们将得到如下的错误结果，DB1.DBW2 并没有得到正确数值，正确数值 4444 被错误地送到了

DB4.DBW2 中：

		Address	Symbol	Display format	Status value
1		DB1.DBW 0		DEC	3333
2		DB1.DBW 2		DEC	0
3		DB4.DBW 0		DEC	0
4		DB4.DBW 2		DEC	4444

图 2-17 错误传送程序结果

2.10.5. 改变 DB 寄存器和地址寄存器 AR1 操作

那么在 STEP7 编程中，都有哪些情况会引起 DB 寄存器中的内容改变呢？下面列出了可能引起 DB 寄存器或 AR1 内容改变的一些操作：

使用下述复杂语言结构会导致 DB 寄存器 和地址寄存器 AR1 内容（关于 AR1 的使用将在后续章节中讲解）改变：

1. 以绝对地址对 DB 块的访问 (如：DB20.DBW10)
2. FB 的多重背景调用
3. STRUCT 数据类型作为 FC 或 FB 里的地址
4. STRUCT 数据类型作为 FC 或 FB 的实参

如果用户在编程时使用了上面提到的选项，则必须手动恢复寄存器的内容，否则可能产生错误。关于这个问题的细节可以在 STEP 7 帮助中"Avoiding errors when calling modules" 项下找到。

2.10.6. STRUCT 数据类型作为 FC 或 FB 里的地址

下图的例子显示了如下两种情况对于 DB 寄存器的影响情况。

1. FB 的多重背景调用，
2. STRUCT 数据类型作为 FC 或 FB 里的地址

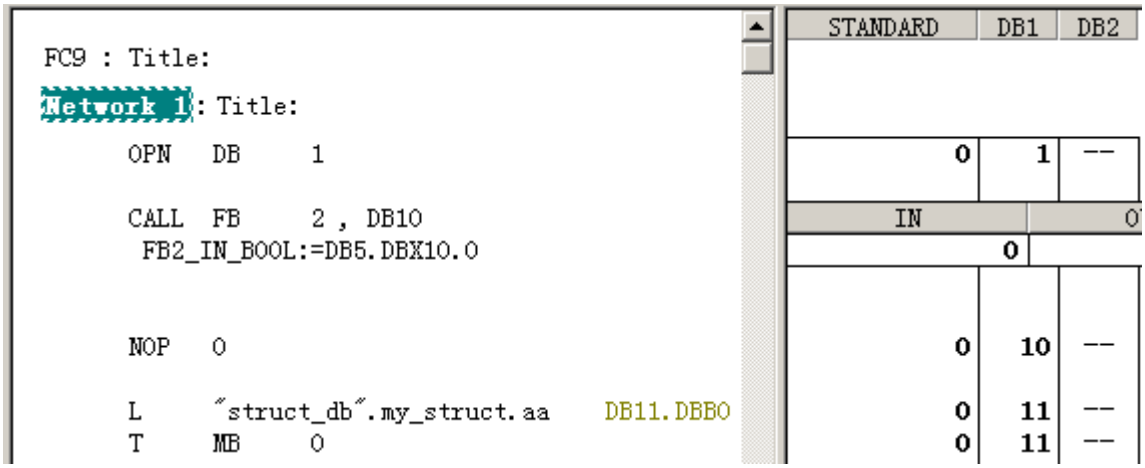

图 2-18 程序对 DB 寄存器的影响

程序执行过程：

1. 在执行 OPN DB1 后， DB 寄存器内容为 1
2. 在调用 FB2,背景数据块为 DB10 后，DB 寄存器内容为 10，DB 寄存器被改变
3. 在引用“stuct_db”.my_struct.aa 后，DB 寄存器内容为 11，DB 寄存器被改变

注意：FB2 的输入参数（BOOL 类型）DB5.DBX10.0，虽然此种格式为数据块的完整引用方式，但作为输入参数类型使用时，并不改变此处 DB 寄存器的值

2.10.7. STRUCT 数据类型作为 FC 或 FB 的实参被调用

下图的例子显示了如下情况对于 DB 寄存器的影响情况：

STRUCT 数据类型作为 FC 或 FB 里的实参

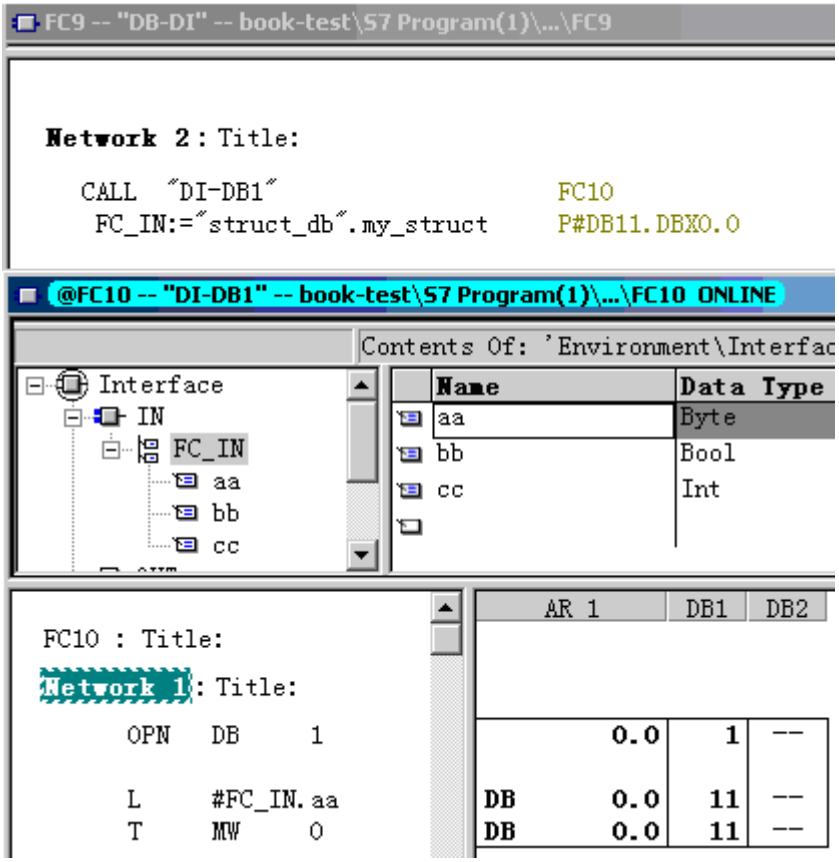

图 2-19 访问 STRUCT 对 DB 寄存器的影响

程序执行过程：

1. 在 FC9 中调用 FC10,并将形参“stuct_db”.my_struct 赋给输入参数 FC_IN
2. 在 FC10 中,FC_IN 为 STRUCT 数据类型
3. 在 FC10 中执行 OPN DB1 后， DB 寄存器内容为 1
4. 引用 FC_IN.aa 后，DB 寄存器内容为 11，DB 寄存器内容被改变

2.10.8. 通常情况，DB 寄存器不受影响

在通常情况下，DB 寄存器是不受影响的，下图为 FC9 中调用 FC11 及 SFC20 后，DB 寄存器不受影响。

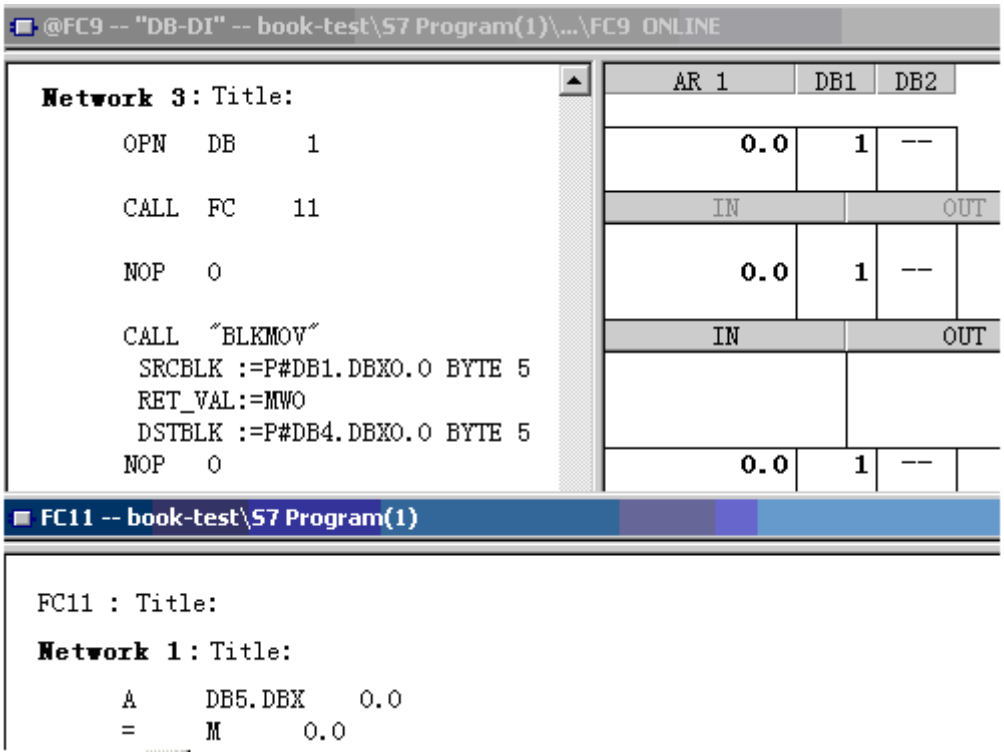

图 2-20 对 DB 寄存器无影响的程序

程序执行过程：

1. 在 FC9 中执行 OPN DB1 后， DB 寄存器内容为 1
2. 在 FC9 中调用 FC11 及 SFC20
3. 在 FC11 中, 引用了地址 DB5.DBX0.0
4. 在 SFC20 参数中使用了 ANY 数据类型
5. FC9 中，调用 FC11,SFC20 结束后，DB 寄存器内容为 1，未受影响

2.10.9. 数据块的完整引用方式的优缺点

至此，也许有的用户会对数据块的完整引用方式存在疑问，既然此种引用方式可以明确当前的数据寄存器，并且编程简单，是不是可以在程序中全部使用数据块完整引用方式呢？答案当然是否定的，原因如下：

1. 相对于数据块的完整引用方式，数据块简略方式寻址提供了更灵活的编程方式及功能（特别是关于指针寻址方式）
2. 数据块完整寻址方式每次都对 DB 寄存器进行操作，增加了程序执行时间，不适用于大数据量编程方式

特别提示：当程序涉及到快速，多次对数据访问时，可以使用 M 存储区代替 DB 数据区，这样可以

得到更快的程序执行时间，但要注意 CPU 中的 M 存储区资源要远少于 DB 数据区资源，要节约使用。（对于 31x 系列 CPU,执行一次 T DBW0 指令需要约 1.6μS，而执行一次 T MW0 指令仅需要约 0.2μS）

特别提示：上面的例子告诉我们，尽管对于数据块完整地址引用会改变 DB 寄存器中的内容，但通过编程者的重视，这个错误是完全可以避免的。

2.11. 在 FC 的使用当中常见的错误

2.11.1. ENO 的误解

对于初学者来说，容易在 ENO 的使用上出错误，由于不清楚 ENO 来龙去脉，经常认为只要调用 FC 是无条件的，那么 ENO 也是永远导通的，实则不然。下图中的例子将说明这个问题。

程序原目的：进行模拟量转换，并无条件地将 MD10 的数值传送至 MD14。

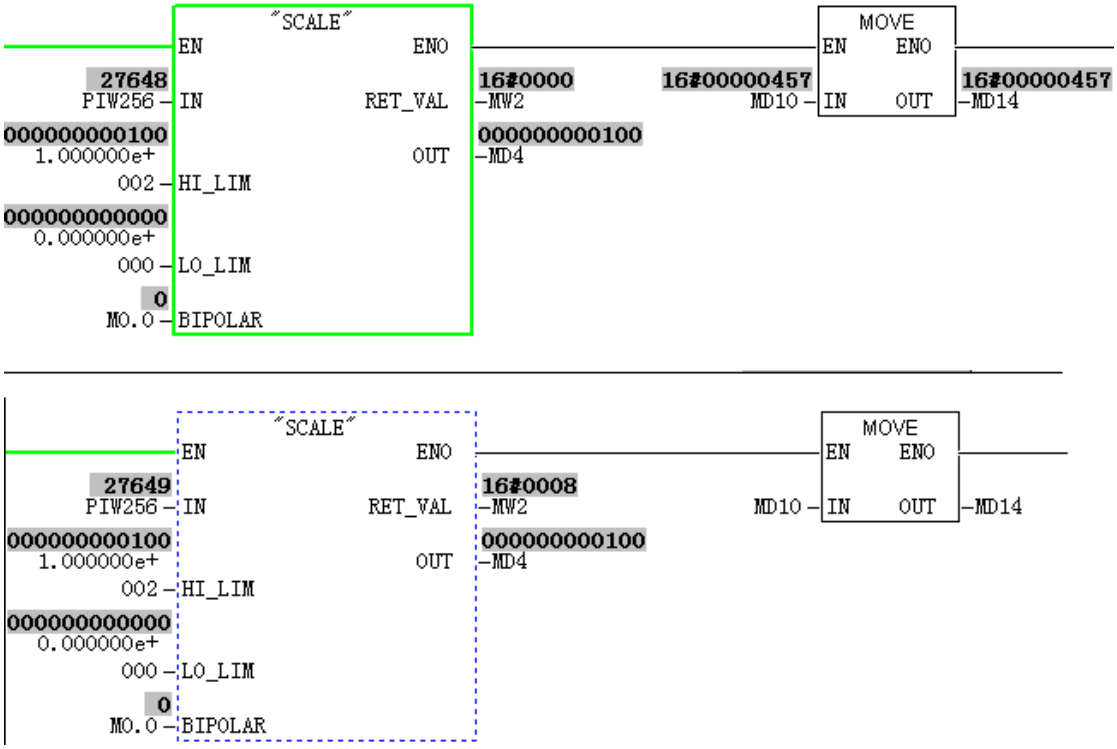

图 2-21 ENO 的使用

在图中可以看出，当 FC105 的输入端 PIW256 在正常范围内的时候，MOVE 指令被执行（ENO=EN），当 PIW256 超出了上限值后，MOVE 指令不再被执行（ENO≠EN）。此故障还是比较隐蔽的，因为大部分情况下，输入可能都处于正常范围。

故障分析：在 STEP7 的 LAD 编程手册中有对 EN/ENO 机制的详细描述，这里不再赘述。仅列出主要部分内容：

ENO 的值取决于公式：ENO = EN 与非 （error ）

- 如果程序调用没有错误（error = 0），则 ENO = EN。
- 如果程序调用有错误（error = 1），则 ENO = 0。

EN/ENO 机制用于：

- 数学运算指令
- 传输及转换指令
- 移位及循环移位指令
- 块调用

EN/ENO 机制不能用于：

- 比较指令
- 计数器
- 定时器

如果用户要创建自己在 FBD 或 LAD 中调用的块，那么必须确保退出块时，置位 BR 位。这并不是一个自动处理过程。不能使用 BR 作为存储位，因为 EN/ENO 机制会不断重写 BR 位。作为代替，可使用一个临时变量（例如 #error）来保存发生的所有错误，并用 0 初始化此变量。在块内部程序中，如果用户某处程序执行为错误状态，则可以将此变量置 1。并且在块结尾编写以下程序段：

```
end: AN error
SAVE
```

确保在任何情况下都处理本程序段，这表示禁止在块内使用 BEC，并禁止跳过本程序段。

2.11.2. 停止对 FC 调用后引起的故障：

下面的程序非常简单，看似正确，但存在着一定隐患。

程序原目的：

在 OB1 中当 M0.0 为 1 的时候，调用 FC13

在 OB1 中当 M0.0 为 01 的时候，不调用 FC13

FC13 中包含简单逻辑及定时器的使用

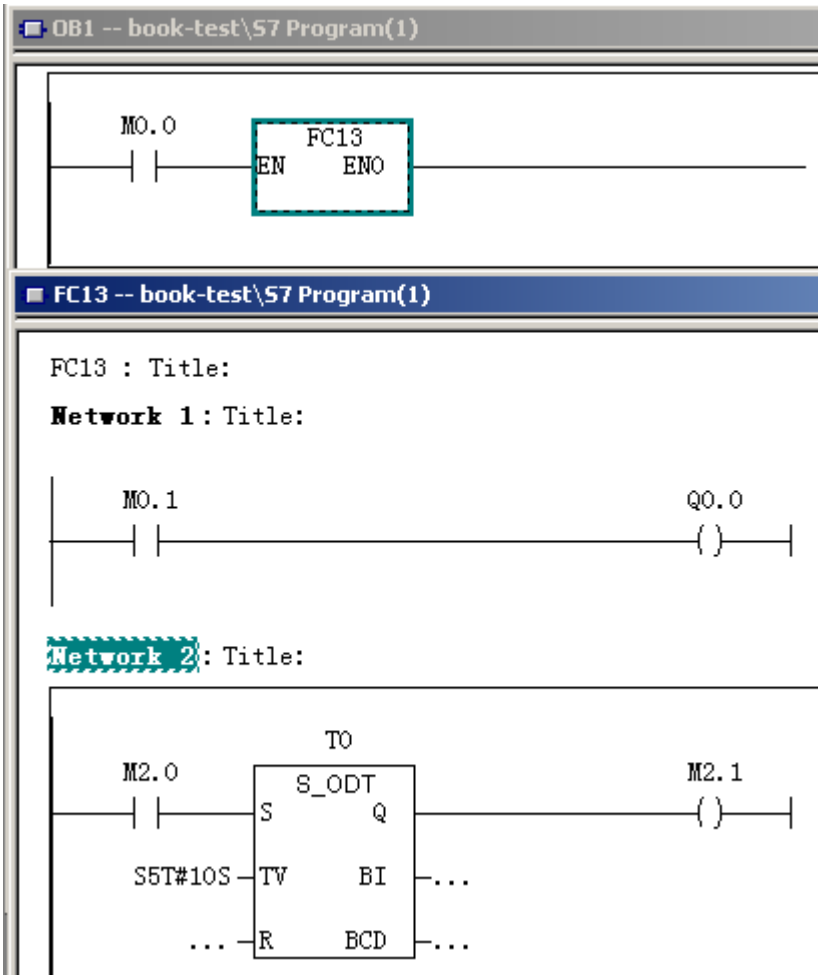

图 2-22 普通 FC 程序内容举例

故障现象：

假设某个时刻，M0.0,M0.1,M2.0 都为 1 ，并且维持数值 1 的时间超过了 10 秒，那么 Q0.1,M2.1 也都会为 1。

假设此时 M0.0 变为 0，FC13 不再被调用，结果如图：

Address	Symbol	Display format	Status value
M 0.0		BOOL	false
M 0.1		BOOL	true
Q 0.0		BOOL	true
M 2.0		BOOL	true
T 0		SIMATIC_TIME	S5T#0ms
M 2.1		BOOL	true

图 2-23 普通 FC 程序执行结果

对于初学者来说，容易忽视的问题为：M0.0 变为 0 后，FC13 中的 Q,M,会保持原来状态，T,C,会继续工作，如果 M0.0 再次变为 1，并且此时 M2.0 也为 1，由于定时器保持着计时到达的状态，

M2.1 会立刻变为 1，这种情况可能会导致某些在 FC 被调用后必须延时执行的逻辑立刻被执行。在实际应用中，如此逻辑为某个设备启动信号，那么这个设备可能会跳过延时或保护逻辑而马上运行！ 对于编程人员如果能够确保上述现象不构成对人身及财产的伤害，则可以不采取措施，否则应当加入限制程序，例如下面的复位语句。

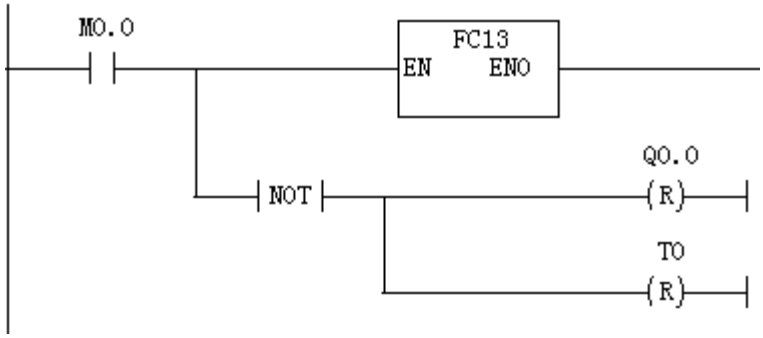

图 2-24 FC 程序复位举例

2.11.3. FC 中临时变量的使用：

很多初学者容易将 FC 及 FB 相混淆,认为 FB 仅仅是比 FC 多了一个背景数据块，这种认识是非常危险的。在 STEP7 的关于 FC 的描述是这样的：

FC 是一个没有存储空间的逻辑块。FC 的临时变量存储在本地数据堆栈中，这些数据在 FC 执行完毕后将会丢失。为了永久的保存数据，FC 可以使用共享数据块。

由于 FC 没有自己的存储空间，所以必须指定实参给它的参数（这就是为什么 FC 的输入输出管脚必须填写参数的原因）。FC 的临时变量（位于本地数据堆栈中）是无法指定初始值的（由于本地数据堆栈是由系统自动动态使用的）。为了更形象的说明这一点，我们来看下面的例子，此例子对 L 堆栈在程序调用时的分配进行了详细的讲解：

L 堆栈永远以地址“ 0 ”开始。在 L 堆栈中，会为每个 FC 提供一定地址空间，作为存放每个块所拥有的固有数据或局部数据。当某个块终止时，那么它的空间随之也被重新释放出来。 指针总是指向当前打开块的第一个字节。

运行等级	**L 堆栈中的字节数**	**指针**
调用 OB1 (带有 20 个字节的系统固有数据和局部数据的 10 个附加字节)	30	0
调用 FC1 (带有 30 个字节的局部数据) 30 个字节 (OB1) + 30 个字节 (FC1)	60	30
调用 FC20 (带有 20 个字节的局部数据)		

	60 个字节 (OB1 + FC1) +20 个字节 FC10	80	60
	调用 FC21 (带有 20 个字节的局部数据) 60 个字节 (OB1 + FC1) +20 个字节 FC11	80	60
调用 FC2 (带有 50 个字节的局部数据) 30 个字节 (OB1) +50 个字节 (FC2)		80	30
	调用 FC30 (带有 10 个字节的局部数据) 80 个字节 (OB1 + FC2) +10 个字节 FC20	90	80

表 2-1 L 堆栈在程序执行过程当中的动态分配

由上面的例子可以看出：对于 FC20 曾经使用过的系统中 L 堆栈 60-80 区间（FC20 中地址范围为 LB0-LB19）在 FC20 调用结束后，被提供给 FC21 使用（FC21 中地址范围同样为 LB0-LB19）。

对于 FC 的临时变量认识不清晰，用户在对临时变量的使用当中，也经常会出现一些错误，下面将使用一个例子非常直观地说明上面的问题

程序原目的：

在 OB1 在程序中调用 FC20 后立即调用 FC21

FC20 中将 20 赋值给临时变量 FC20_TEMP1，将 21 赋值给临时变量 FC20_TEMP2

FC21 中将 FC21_TEMP1，FC21_TEMP2 相加

程序分析：我们发现 FC20 中的临时变量曾经出现的数值（20，21）被 FC21 中的临时变量 FC21_TEMP1，FC21_TEMP2 得到了，如果直接使用这两个临时变量进行加法操作，可以得到结果 41。对于编程人员来说，临时变量必须要在所在程序段中赋值，而后使用。用户对此例中的 FC21_TEMP1，FC21_TEMP2 必须先做清零处理，否则其在使用前即可能拥有数值。

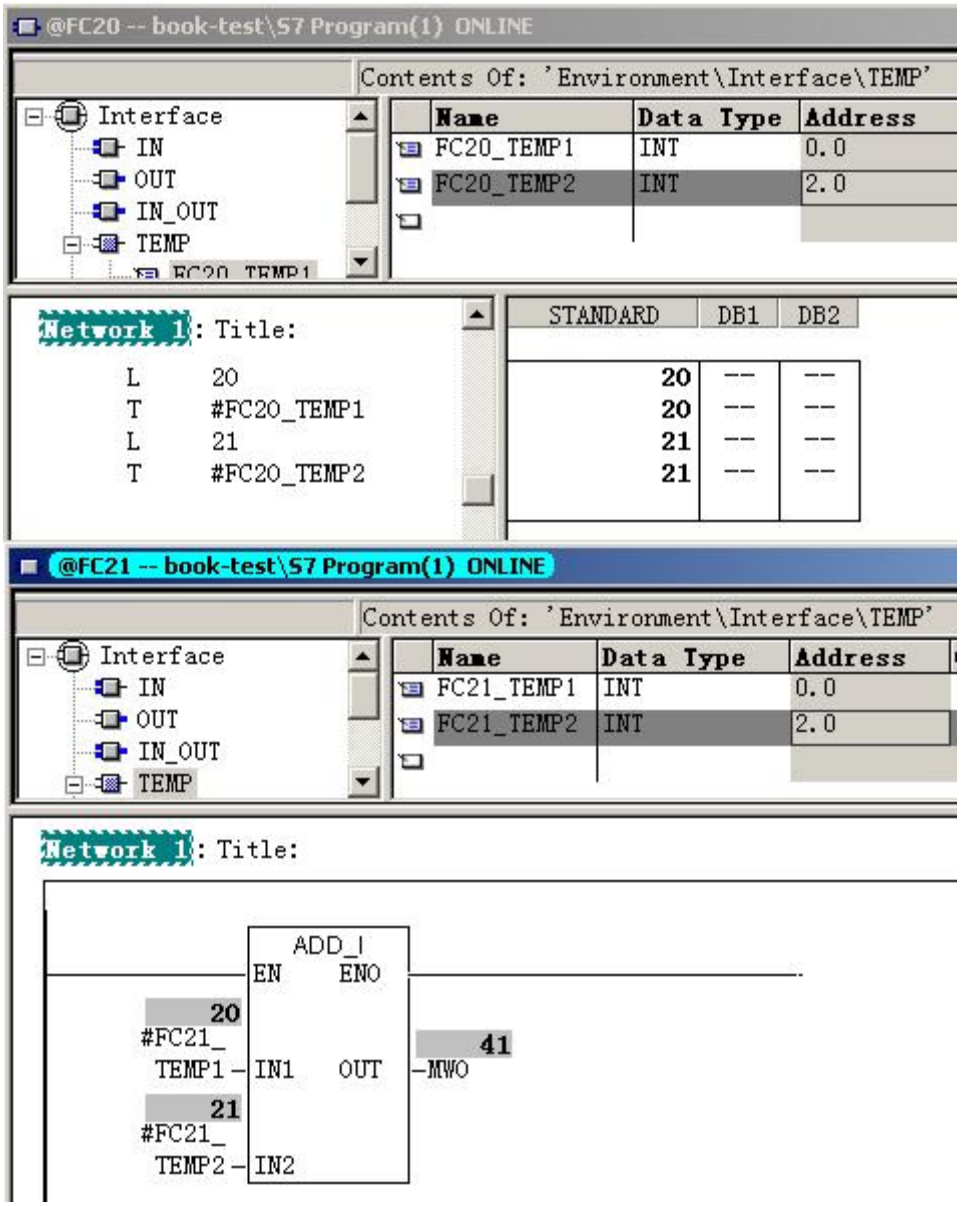

图 2-25 不同 FC 程序临时变量之间的影响

结论：对于 FC 或 FB 中的临时变量，不要希望将本次调用的数值可以存储在里面以供下次程序调用使用，因为这些临时变量所使用的 L 堆栈空间在 FC 或 FB 调用结束就释放给系统了，其它后续程序可以任意使用。所以下列用法都是错误的：

- 将临时变量用于上升/下降沿指令
- 将临时变量用于自保持逻辑
- 临时变量未在所在程序段中赋值，直接使用

警告：不要试图利用 L 堆栈的这种特点进行功能或功能块之间的数据传递。程序逻辑改变，语句执行顺序改变，临时数据区长度定义改变，中断程序都会影响 L 堆栈中的数据存储顺序。

2.11.4. FC 输出处理

对于 FC 的使用，另一个的常见的错误是对输出的错误处理：导致这个错误的原因还是对 FC 认识的不清楚。再次强调：相比较于 FB，FC 是一个没有存储空间的逻辑块。如果没有数据被写至 FC 的 OUT 参数，FC 将会输出一个随机值！对于 FB，因为其可以使用背景数据块来存储 OUT 参数的数值，即使某次调用没有对 OUT 参数进行写操作，OUT 参数依然可以输出上一次的旧值。

下面的程序将说明这一点：

程序原本目的：

在 OB1 中调两次 FC22,将 MW0，MW2 作为输入参数,DB1.DBX0.0,DB1.DBX0.1 分别作为输出参数赋给 FC22

FC22 检测当输入大于 10 时，置位输出为 1

FC22 检测当输入小于-10 时，复位输出为 0

FC22 的输出的动作死区为-10 至 10

此程序乍看是没有错误的，但是，如果 OB1 中调用了两次 FC22，而且 MW2 位于死区（-10 至 10）之间时，MW0 的数值改变将不仅仅改变 DB1.DBX0.0 的状态，同时会影响输出 DB1.DBX0.1 的数值。

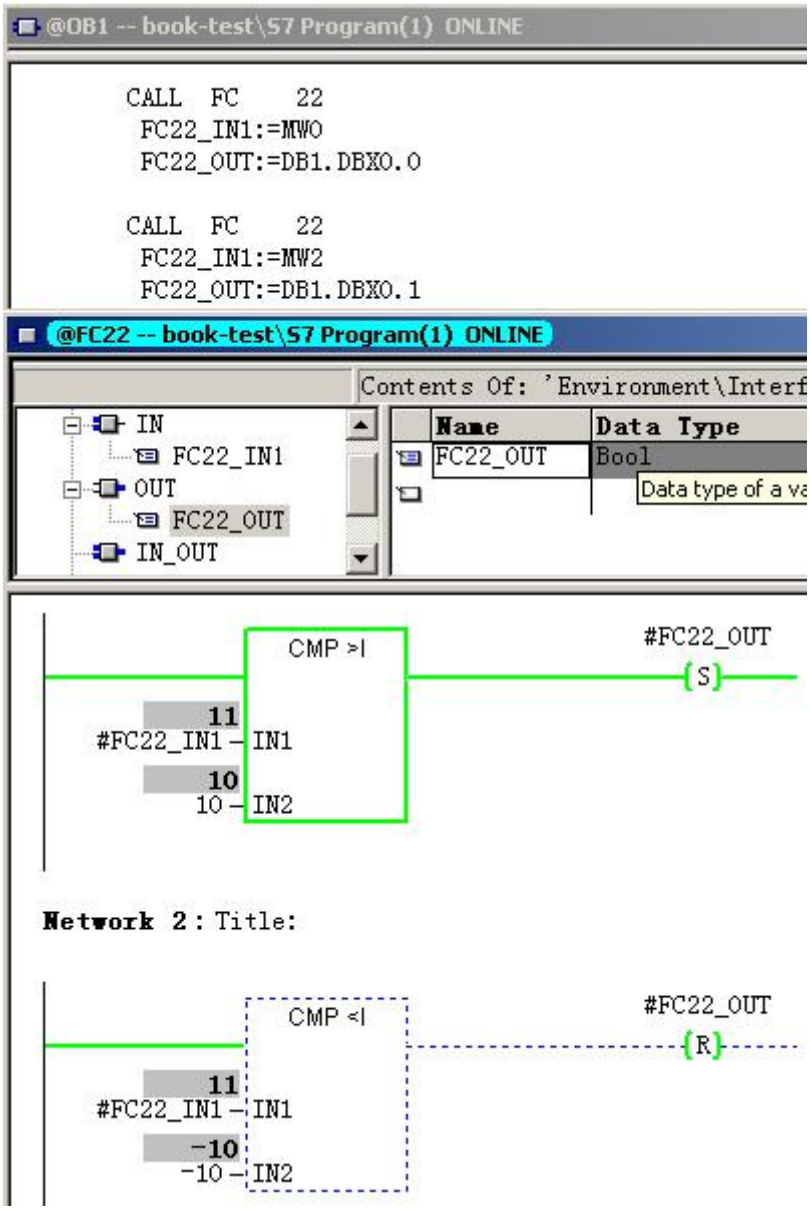

图 2-26 FC 程序输出编程错误举例

故障分析：在上面的例子，OB1 中调用了两次 FC22，而且 MW2 位于死区（-10 至 10）之间时，其输出在 FC22 没有被赋值，DB1.DBX0.1 正常情况下不应当改变数值。但是本例中，MW0 的数值改变将不仅仅改变 DB1.DBX0.0 的状态，同时会影响输出 DB1.DBX0.1 的数值。如下图。

Address	Symbol	Display format	Status value	Modify value
MW 0		DEC	11	11
MW 2		DEC	0	0
DB1.DBX 0.0		BOOL	true	
DB1.DBX 0.1		BOOL	true	

图 2-27 FC 程序输出编程错误结果

结论：

对于 FC 的输出变量，必须要在每次执行 FC 时赋给一个确定的值，否则输出有可能会输出一个随机值。下列用法都是错误的：

- 将输出变量用于上升/下降沿指令
- 将输出变量用于自保持逻辑
- 输出变量未在所在程序段中赋值

警告：不要因为在 FC 编程中遇到没有对输出赋值，而程序执行正确，就忽略了对 FC 输出编程的注意事项，否则将承担这个错误有可能带来的风险。

建议：

- 用 IN/OUT 变量代替 OUTPUT 变量
- 不论何时调用块，FC 中的 OUT 参数都必须被赋值

2.12. 调用 FB 引起错误

2.12.1. FB 的输出处理

由于 FB 有自己的背景数据块，因此其除了临时变量外，其所有过程数据都可以保存在自己的背景数据块内。因此对于 FB 的使用，只是输出变量的处理与 FC 不同，不需要每次都赋给确定的数值。但其它前者 FC 时用的注意事项都适用于 FB。

2.12.2. 在 FB 中使用 AR2

另外由于在 FB 中，系统将会使用 AR2 用于背景数据块的访问，所以如果在 FB 中使用了 AR2 ，务必要注意其用法。下面的例子将说明这一点。

程序原目的：

同时对 DB12 及 DB13 中的数据进行操作，将 DB12 中的多个数据进行加 1 操作复制到 DB13 中(本例仅为了举例说明问题，虽然未涉及复杂算法，也使用了间接寻址) ：

```
DB13.DBB0=DBW12.DBB0+1
DB13.DBB1=DBW12.DBB1+1
DB13.DBB2=DBW12.DBB2+1
```

下面的程序可以放在 FC 中执行，但不可放在 FB 中执行。

```
OPN   DB    12
OPN   DI    13

L     P#0.0
LAR1
LAR2

L     DBB [AR1,P#0.0]
INC   1
T     DIB [AR2,P#0.0]

L     DBB [AR1,P#1.0]
INC   1
T     DIB [AR2,P#1.0]

L     DBB [AR1,P#2.0]
INC   1
T     DIB [AR2,P#2.0]
```

图 2-28 AR2 应用于 FC

发现此错误很容易，因为当程序存盘后会变成如下状态，其中涉及到 AR2 的数据块访问被替换为了背景数据块的变量（图中为输入参数 FB1_IN）：


```
Network 1: Title:
      OPN   DB    12
      OPN   DI    13

      L     P#0.0
      LAR1
      LAR2

      L     DBB [AR1,P#0.0]
      INC   1
      T     #FB1_IN

      L     DBB [AR1,P#1.0]
      INC   1
      T     DIB [AR2,P#1.0]

      L     DBB [AR1,P#2.0]
      INC   1
      T     DIB [AR2,P#2.0]
```

图 2-29 AR2 应用于 FB

2.13. 关于 FC/FB 使用的总结：

对于同时要打开两个数据块，并且涉及间接寻址的程序，推荐使用 FC 实现

并且在 FC 程序开始增加如下语句存储数据块寄存器及地址寄存器的内容：

```
L     DBNO
T     #DBNO_TEMP      //用户自定义的 WORD 临时变量，存储数据块寄存器数值
L     DINO
T     #DINO_TEMP      //用户自定义的 WORD 临时变量，存储数据块寄存器数值

TAR1
T     #AR1_TEMP       //用户自定义的 DWORD 临时变量，存储地址寄存器数值
TAR2
T     #AR2_TEMP       //用户自定义的 DWORD 临时变量，存储地址寄存器数值
//在以上程序之后，此处可以添加用户自身程序。
```

在 FC 程序结束位置增加如下语句恢复数据块寄存器及地址寄存器的内容：

```
OPN   DB [#DBNO_TEMP]
OPN   DI [#DINO_TEMP]
L     #AR1_TEMP
LAR1
L     #AR2_TEMP
LAR2
```

警告：以上建议仅代表笔者个人观点，例子程序也是免费的。任何用户都可以免费复制或传播此程序例子。程序的作者或拥有者对此程序不承担任何功能性或兼容性的责任。此程序的使用者风险自负

2.14. OB 块引起错误

2.14.1. OB 未装载

在 STEP7 程序中，所有的用户程序都将在组织块中被调用。而针对于不同事件，CPU 将会调用不同的组织块，当某事件发生时，如果 CPU 中没有下载相对应的组织块，CPU 将进入 STOP 状态（例如 DP 从站通信故障时，CPU 中如果没有 OB86,CPU 将进入 STOP 状态）。

2.14.2. 调用中断程序引起的故障

调用中断程序引起的故障并不多见，但也应当引起足够的重视，否则也会降低项目程序的可靠性。

1. OB35 的使用：

由于定时中断程序具有稳定的执行时间间隔特性，所以 OB35 也许是除去 OB1 之外，人们最经常使用的组织块了。但是如果其执行间隔如果被修改得非常短(例如 1 毫秒)，那么就必须考虑 OB35 中的程序是否能够在 1 毫秒之内执行完毕。如果 OB35 的执行时间超过了 1 毫秒，这时 OB35 又再次被系统触发，系统将会出错。

2． OB40 的使用：

类似的情况也会在 OB40 中出现，例如某个通道触发了 OB40 硬件中断，在 OB40 尚未执行完毕之前，此通道如果再次触发了硬件中断，此中断将会丢失，不被系统响应。但是，如果某个通道触发了 OB40 硬件中断，在 OB40 尚未执行完毕之前，其它通道触发了硬件中断，此中断不会丢失，将被系统响应。此情况也应引起注意。

2.15. 项目一致性检查（数据块，FB，FC 更新）

由于 STEP7 的程序是由各种类型的块组成的，用户可以分别对这些块进行编辑或修改，并可以单独对某个块进行下载操作，而不影响其它块。在使用这种方便灵活的操作的同时，用户也务必要注意其带来的风险。

例如：

在 OB1 中调用 FC1，当 FC1 的输入/输出参数作出修改后，如果仅仅下载 FC1，而不下载 OB1，那么 CPU 将会出现编程错误。所以当用户修改 FB/FC，DB，UDT 以后，应当对涉及到这些调用的程序进行更新，避免发生错误。STEP7 对此提供了一致性检查的功能，可以在选中程序的 BLOCK 目录后，由菜单的 Edit-Check Block Consistency 调出此功能。

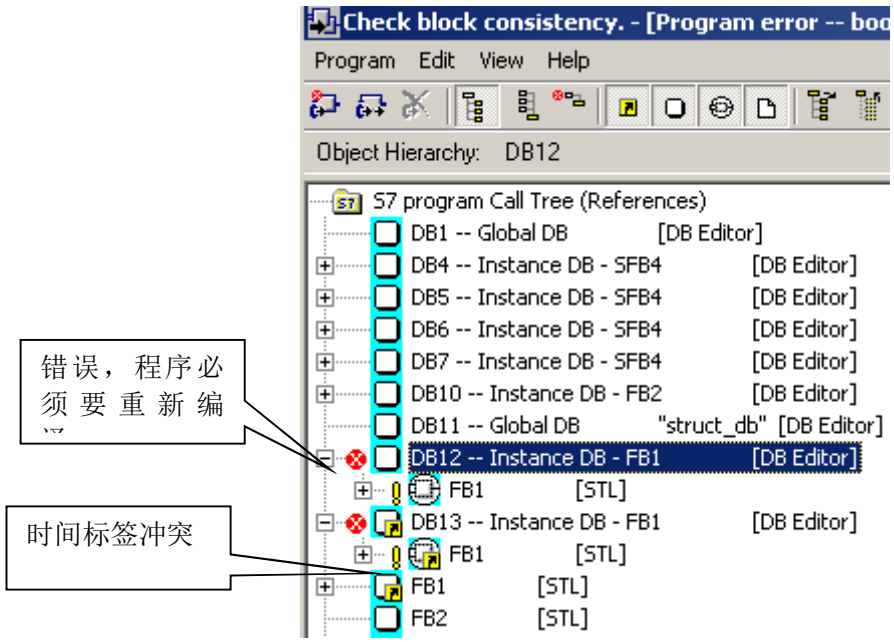

图 2-30 块的一致性检查界面

在 STEP7 程序中，如果出现 FB/FC 的参数改变/时间标签不一致的情况，所调用的 FB/FC 会以

红色显示。此时右键点击次 FB/FC，在菜单中将会出现 Update Block Call 菜单，使用此功能后，调用程序将自动更新 FB/FC 的调用。使用此种方式，比手动删除此 FB/FC，再手动重新输入各种参数要方便的多。

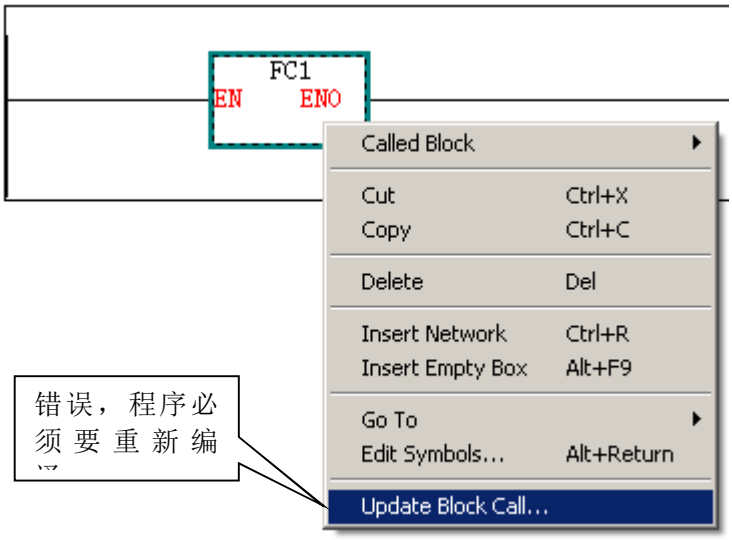

图 2-31 块调用的更新

2.16. 例子程序的使用事项

用户在附件中的例子程序在使用中需要遵循以下注意事项：

- 示例代码大部分编写在 FC/FB 当中，使用 OB1 调用
- 当用户希望验证某个 FC/FB 的功能时，首先应当将相关的 FC/FB/DB 下载到 PLC/PLCSIM 当中，并在 OB1 中将调用此 FC/FB 的注释符号删除，下载 OB1,此 FC/FB 即被调用
- 尽量避免同时验证多个 FC/FB 的功能，因为这些 FC/FB 可能存在着逻辑上的冲突
- 对于不需要验证功能的 FC/FB,在 OB1 中应当保持其作为注释出现的状态

本文附件程序使用的说明：

FC1：循环程序示例

FC2：浮点数运算程序

FC3：程序执行顺序例子

FC4：上升延指令

FC6：定时器触发例子

FC7：同时操作两个数据块的循环程序例子

FC8：数据块寄存器例子 1

FC9：数据块寄存器例子 2

FC10：数据块寄存器例子 3

FC11：数据块寄存器例子 4

FC12：ENO 使用例子

FC13：停止调用 FC 后引起故障的例子

FC20：FC 中临时变量使用的例子 1

FC21：FC 中临时变量使用的例子 2

FC22：FC 输出使用的的例子

FC23：系统变量保存的例子

FB1：在 FB 使用 AR2 的例子

2.17. 重要声明：

- ✧ 由于例子是免费的，任何用户可以免费复制或传播此程序例子。程序的作者对此程序不承担任何功能性或兼容性的责任，使用者风险自负。
- ✧ 西门子不提供此程序例子的错误更改或者热线支持。