详解西门子间接寻址

【地址的概念】

完整的一条指令，应该包含指令符+操作数（当然不包括那些单指令，比如 NOT 等）。其中的操作数是指令要执行的目标，也就是指令要进行操作的地址。

我们知道，在 PLC 中划有各种用途的存储区，比如物理输入输出区 P、映像输入区 I、映像输出区 Q、位存储区 M、定时器 T、计数器 C、数据区 DB 和 L 等，同时我们还知道，每个区域可以用位（BIT）、字节（BYTE）、字（WORD）、双字（DWORD）来衡量，或者说来指定确切的大小。当然定时器 T、计数器 C 不存在这种衡量体制，它们仅用位来衡量。由此我们可以得到，要描述一个地址，至少应该包含两个要素：

1、存储的区域

2、这个区域中具体的位置

比如：A Q2.0

其中的 A 是指令符，Q2.0 是 A 的操作数，也就是地址。这个地址由两部分组成：

Q：指的是映像输出区

2.0：就是这个映像输出区第二个字节的第 0 位。

由此，我们得出， 一个确切的地址组成应该是：

〖存储区符〗〖存储区尺寸符〗〖尺寸数值〗.〖位数值〗，例如：DBX200.0。

DB X 200 . 0

其中，我们又把〖存储区符〗〖存储区尺寸符〗这两个部分合称为：地址标识符。这样，一个确切的地址组成，又可以写成：

地址标识符 + 确切的数值单元

【间接寻址的概念】

寻址，就是指定指令要进行操作的地址。给定指令操作的地址方法，就是寻址方法。

在谈间接寻址之前，我们简单的了解一下直接寻址。所谓直接寻址，简单的说，就是直接给出指令的确切操作数，象上面所说的，A Q2.0，就是直接寻址，对于 A 这个指令来说，Q2.0 就是它要进行操作的地址。

这样看来，间接寻址就是间接的给出指令的确切操作数。对，就是这个概念。

比如：A Q[MD100] ，A T[DBW100]。程序语句中用方刮号 [ ] 标明的内容，间接的指明了指令要进行的地址，这两个语句中的 MD100 和 DBW100 称为指针 Pointer，它指向它们其中包含的数值，才是指令真正要执行的地址区域的确切位置。间接由此得名。

西门子的间接寻址方式计有两大类型：存储器间接寻址和寄存器间接寻址。

【存储器间接寻址】

存储器间接寻址的地址给定格式是：地址标识符+指针。指针所指示存储单元中所包含的数值，就是地址的确切数值单元。

存储器间接寻址具有两个指针格式：单字和双字。

单字指针是一个 16bit 的结构，从 0-15bit，指示一个从 0-65535 的数值，这个数值就是被寻址的存储区域的编号。

双字指针是一个 32bit 的结构，从 0-2bit，共三位，按照 8 进制指示被寻址的位编号，也就是 0-7；而从 3-18bit，共 16 位，指示一个从 0-65535 的数值，这个数值就是被寻址的字节编号。

指针可以存放在 M、DI、DB 和 L 区域中，也就是说，可以用这些区域的内容来做指针。

单字指针和双字指针在使用上有很大区别。下面举例说明：

L DW#16#35 //将 32 位 16 进制数 35 存入 ACC1

T MD2 //这个值再存入 MD2，这是个 32 位的位存储区域

L +10 //将 16 位整数 10 存入 ACC1，32 位 16 进制数 35 自动移动到 ACC2

T MW100 //这个值再存入 MW100，这是个 16 位的位存储区域

OPN DBW[MW100] //打开 DBW10。这里的[MW100]就是个单字指针，存放指针的区域是 M 区，

MW100 中的值 10，就是指针间接指定的地址，它是个 16 位的值！

L L#+10 //以 32 位形式，把 10 放入 ACC1，此时，ACC2 中的内容为：16 位整数 10

T MD104 //这个值再存入 MD104，这是个 32 位的位存储区域

A I[MD104] //对 I1.2 进行与逻辑操作！

=DIX[MD2] //赋值背景数据位 DIX6.5！

A DB[MW100].DBX[MD2] //读入 DB10.DBX6.5 数据位状态

=Q[MD2] //赋值给 Q6.5

A DB[MW100].DBX[MD2] //读入 DB10.DBX6.5 数据位状态

=Q[MW100] //错误！！没有 Q10 这个元件

从上面系列举例我们至少看出来一点：

单字指针只应用在地址标识符是非位的情况下。的确，单字指针前面描述过，它确定的数值是 0-65535，而对于 byte.bit 这种具体位结构来说，只能用双字指针。这是它们的第一个区别，单字指针的另外一个限制就是，它只能对 T、C、DB、FC 和 FB 进行寻址，通俗地说，单字指针只可以用来指代这些存储区域的编号。

相对于单字指针，双字指针就没有这样的限制，它不仅可以对位地址进行寻址，还可以对 BYTE、WORD、DWORD 寻址，并且没有区域的限制。不过，有得必有失，在对非位的区域进行寻址时，必须确保其 0-2bit 为全 0！

总结一下：

单字指针的存储器间接寻址只能用在地址标识符是非位的场合；双字指针由于有位格式存在，所以对地址标识符没有限制。也正是由于双字指针是一个具有位的指针，因此，当对字节、字或者双字存储区地址进行寻址时，必须确保双字指针的内容是 8 或者 8 的倍数。

现在，我们来分析一下上述例子中的 A I[MD104] 为什么最后是对 I1.2 进行与逻辑操作。

通过 L L#+10 ，我们知道存放在 MD104 中的值应该是：

MD104：0000 0000 0000 0000 0000 0000 0000 1010

当作为双字指针时，就应该按照 3-18bit 指定 byte，0-2bit 指定 bit 来确定最终指令要操作的地址，因此：

0000 0000 0000 0000 0000 0000 0000 1010 = 1.2

详解西门子间接寻址<2>

【地址寄存器间接寻址】

在先前所说的存储器间接寻址中，间接指针用 M、DB、DI 和 L 直接指定，就是说，指针指向的存储区内容就是指令要执行的确切地址数值单元。但在寄存器间接寻址中，指令要执行的确切地址数值单元，并非寄存器指向的存储区内容，也就是说，寄存器本身也是间接的指向真正的地址数值单元。从寄存器到得出真正的地址数值单元，西门子提供了两种途径：

1、区域内寄存器间接寻址

2、区域间寄存器间接寻址

地址寄存器间接寻址的一般格式是：

〖地址标识符〗〖寄存器,P#byte.bit〗，比如：DIX[AR1,P#1.5] 或 M[AR1,P#0.0] 。

〖寄存器,P#byte.bit〗统称为：寄存器寻址指针，而〖地址标识符〗在上帖中谈过，它包含〖存储区符〗+〖存储区尺寸符〗。但在这里，情况有所变化。比较一下刚才的例子：

DIX [AR1,P#1.5]

X [AR1,P#1.5]

DIX 可以认为是我们通常定义的地址标识符，DI 是背景数据块存储区域，X 是这个存储区域的尺寸符，指的是背景数据块中的位。但下面一个示例中的 M 呢？X 只是指定了存储区域的尺寸符，那么存储区域符在哪里呢？毫无疑问，在 AR1 中！

DIX [AR1,P#1.5] 这个例子，要寻址的地址区域事先已经确定，AR1 可以改变的只是这个区域内的确切地址数值单元，所以我们称之为：区域内寄存器间接寻址方式，相应的，这里的[AR1,P#1.5] 就叫做区域内寻址指针。

X [AR1,P#1.5] 这个例子，要寻址的地址区域和确切的地址数值单元，都未事先确定，只是确定了存储大小，这就是意味着我们可以在不同的区域间的不同地址数值单元以给定的区域大小进行寻址，所以称之为：区域间寄存器间接寻址方式，相应的，这里的[AR1,P#1.5] 就叫做区域间寻址指针。

既然有着区域内和区域间寻址之分，那么，同样的 AR1 中，就存有不同的内容，它们代表着不同的含义。

【AR 的格式】

地址寄存器是专门用于寻址的一个特殊指针区域，西门子的地址寄存器共有两个：AR1 和 AR2，每个 32 位。

当使用在区域内寄存器间接寻址中时，我们知道这时的 AR 中的内容只是指明数值单元，因此，区域内寄存器间接寻址时，寄存器中的内容等同于上帖中提及的存储器间接寻址中的双字指针，也就是：

其 0-2bit，指定 bit 位，3-18bit 指定 byte 字节。其第 31bit 固定为 0。

AR：

0000 0000 0000 0BBB BBBB BBBB BBBB BXXX

这样规定，就意味着 AR 的取值只能是：0.0 ——65535.7

例如：当 AR=D4（hex）=0000 0000 0000 0000 0000 0000 1101 0100（b），实际上就是等于 26.4。

而在区域间寄存器间接寻址中，由于要寻址的区域也要在 AR 中指定，显然这时的 AR 中内容肯定于寄存器区域内间接寻址时，对 AR 内容的要求，或者说规定不同。

AR：

1000 0YYY 0000 0BBB BBBB BBBB BBBB BXXX

比较一下两种格式的不同，我们发现，这里的第 31bit 被固定为 1，同时，第 24、25、26 位有了可以取值的范围。聪明的你，肯定可以联想到，这是用于指定存储区域的。对，bit24-26 的取值确定了要寻址的区域，它的取值是这样定义的：

区域标识符

26、25、24 位

P（外部输入输出）

000

I（输入映像区）

001

Q（输出映像区）

010

M（位存储区）

011

DB（数据块）

100

DI（背景数据块）

101

L（暂存数据区，也叫局域数据）

111

如果我们把这样的 AR 内容，用 HEX 表示的话，那么就有：

当是对 P 区域寻址时，AR=800xxxxx

当是对 I 区域寻址时，AR=810xxxxx

当是对 Q 区域寻址时，AR=820xxxxx

当是对 M 区域寻址时，AR=830xxxxx

当是对 DB 区域寻址时，AR=840xxxxx

当是对 DI 区域寻址时，AR=850xxxxx

当是对 L 区域寻址时，AR=870xxxxx

经过列举，我们有了初步的结论：如果 AR 中的内容是 8 开头，那么就一定是区域间寻址；如果要在 DB 区中进行寻址，只需在 8 后面跟上一个 40。84000000-840FFFFF 指明了要寻址的范围是：

DB 区的 0.0——65535.7。

例如：当 AR=840000D4（hex）=1000 0100 0000 0000 0000 0000 1101 0100（b），实际上就是等于 DBX26.4。

我们看到，在寄存器寻址指针 [AR1/2,P#byte.bit] 这种结构中，P#byte.bit 又是什么呢？

【P#指针】

P#中的 P 是 Pointer，是个 32 位的直接指针。所谓的直接，是指 P#中的#后面所跟的数值或者存储单元，是 P 直接给定的。这样 P#XXX 这种指针，就可以被用来在指令寻址中，作为一个“常数”来对待，这个“常数”可以包含或不包含存储区域。例如：

- L P#Q1.0 //把 Q1.0 这个指针存入 ACC1，此时 ACC1 的内容=82000008（hex）=Q1.0
- ★ L P#1.0 //把 1.0 这个指针存入 ACC1，此时 ACC1 的内容=00000008（hex）=1.0
- L P#MB100 //错误！必须按照 byte.bit 结构给定指针。
- L P#M100.0 //把 M100.0 这个指针存入 ACC1，此时 ACC1 的内容=83000320（hex）=M100.0
- L P#DB100.DBX26.4 //错误！DBX 已经提供了存储区域，不能重复指定。
- L P#DBX26.4 //把 DBX26.4 这个指针存入 ACC1，此时 ACC1 的内容=840000D4（hex）=DBX26.4

我们发现，当对 P#只是指定数值时，累加器中的值和区域内寻址指针规定的格式相同（也和存储器间接寻址双字指针格式相同）；而当对 P#指定带有存储区域时，累加器中的内容和区域间寻址指针内容完全相同。事实上，把什么样的值传给 AR，就决定了是以什么样的方式来进行寄存器间接寻址。在实际应用中，我们正是利用 P#的这种特点，根据不同的需要，指定 P#指针，然后，再传递给 AR，以确定最终的寻址方式。

在寄存器寻址中，P#XXX 作为寄存器 AR 指针的偏移量，用来和 AR 指针进行相加运算，运算的结果，才是指令真正要操作的确切地址数值单元！

无论是区域内还是区域间寻址，地址所在的存储区域都有了指定，因此，这里的 P#XXX 只能指定纯

粹的数值，如上面例子中的★。

【指针偏移运算法则】

在寄存器寻址指针［AR1/2,P#byte.bit］这种结构中，P#byte.bit 如何参与运算，得出最终的地址呢？

运算的法则是：AR1 和 P#中的数值，按照 BYTE 位和 BIT 位分类相加。BIT 位相加按八进制规则运算，而 BYTE 位相加，则按照十进制规则运算。

例如：寄存器寻址指针是：[AR1，P#2.6]，我们分 AR1=26.4 和 DBX26.4 两种情况来分析。

当 AR1 等于 26.4，

AR1：26.2

+ P#： 2.6

= 29.7 这是区域内寄存器间接寻址的最终确切地址数值单元

当 AR1 等于 DBX26.4，

AR1：DBX26.2

+ P#： 2.6

= DBX29.7 这是区域间寄存器间接寻址的最终确切地址数值单元

【AR 的地址数据赋值】

通过前面的介绍，我们知道，要正确运用寄存器寻址，最重要的是对寄存器 AR 的赋值。同样，区分是区域内还是区域间寻址，也是看 AR 中的赋值。

对 AR 的赋值通常有下面的几个方法：

1、直接赋值法

例如：

L DW#16#83000320

LAR1

可以用 16 进制、整数或者二进制直接给值，但必须确保是 32 位数据。经过赋值的 AR1 中既存储了地址数值，也指定了存储区域，因此这时的寄存器寻址方式肯定是区域间寻址。

2、间接赋值法

例如：

L [MD100]

LAR1

可以用存储器间接寻址指针给定 AR1 内容。具体内容存储在 MD100 中。

3、指针赋值法

例如：

LAR1 P#26.2

使用 P#这个 32 位“常数”指针赋值 AR。

总之，无论使用哪种赋值方式，由于 AR 存储的数据格式有明确的规定，因此，都要在赋值前，确认所赋的值是否符合寻址规范。

详解西门子间接寻址<3>

使用间接寻址的主要目的，是使指令的执行结果有动态的变化，简化程序是第一目的，在某些情况下，这样的寻址方式是必须的，比如对某存储区域数据遍历。此外，间接寻址，还可以使程序更具柔性，换句话说，可以标准化。

下面通过实例应用来分析如何灵活运用这些寻址方式,在实例分析过程中,将对前面帖子中的笔误、错误和遗漏做纠正和补充。

【存储器间接寻址应用实例】

我们先看一段示例程序：

```
L 100
T MW 100 // 将 16 位整数 100 传入 MW100
L DW#16#8 // 加载双字 16 进制数 8，当把它用作双字指针时，按照 BYTE.BIT 结构，
 结果演变过程就是：8H=1000B=1.0
T MD 2 // MD2=8H
OPN DB [MW 100] // OPN DB100
L DBW [MD 2] // L DB100.DBW1
T MW[MD2] // T MW1
A DBX [MD 2] // A DBX1.0
= M [MD 2] // =M1.0
```

在这个例子中，我们中心思想其实就是：将 DB100.DBW1 中的内容传送到 MW1 中。这里我们使用了存储器间接寻址的两个指针——单字指针 MW100 用于指定 DB 块的编号，双字指针 MD2 用于指定 DBW 和 MW 存储区字地址。

对于坛友提出的 DB[MW100].DBW[MD2] 这样的寻址是错误的提法，这里做个解释：

DB[MW100].DBW[MD2] 这样的寻址结构就寻址原理来说，是可以理解的，但从 SIEMENS 程序执行机理来看，是非法的。在实际程序中，对于这样的寻址，程序语句应该写成：

OPN DBW[WM100]， L DBW[MD2]

事实上，从这个例子的中心思想来看，根本没有必要如此复杂。但为什么要用间接寻址呢？

要澄清使用间接寻址的优势，就让我们从比较中，找答案吧。

例子告诉我们，它最终执行的是把 DB 的某个具体字的数据传送到位存储区某个具体字中。这是针对数据块 100 的 1 数据字传送到位存储区第 1 字中的具体操作。如果我们现在需要对同样的数据块的多个字（连续或者不连续）进行传送呢？直接的方法，就是一句一句的写这样的具体操作。有多少个字的传送，就写多少这样的语句。毫无疑问，即使不知道间接寻址的道理，也应该明白，这样的编程方法是不合理的。而如果使用间接寻址的方法，语句就简单多了。

【示例程序的结构分析】

我将示例程序从结构上做个区分，重新输入如下：

========== 输入 1：指定数据块编号的变量

|| L 100

|| T MW 100

==========输入 2：指定字地址的变量

|| L DW#16#8

|| T MD 2

==========操作主体程序

OPN DB [MW 100]

L DBW [MD 2]

T MW[MD2]

显然，我们根本不需要对主体程序（红色部分）进行简单而重复的复写，而只需改变 MW100 和 MD2 的赋值（绿色部分），就可以完成应用要求。

结论：通过对间接寻址指针内容的修改，就完成了主体程序执行的结果变更，这种修改是可以是动态的和静态的。

正是由于对真正的目标程序（主体程序）不做任何变动，而寻址指针是这个程序中唯一要修改的地方，可以认为，寻址指针是主体程序的入口参数，就好比功能块的输入参数。因而可使得程序标准化，具有移植性、通用性。

那么又如何动态改写指针的赋值呢？不会是另一种简单而重复的复写吧。

让我们以一个具体应用，来完善这段示例程序吧：

将 DB100 中的 1-11 数据字，传送到 MW1-11 中

在设计完成这个任务的程序之前，我们先了解一些背景知识。

【数据对象尺寸的划分规则】

数据对象的尺寸分为：位（BOOL）、字节（BYTE）、字（WORD）、双字（DWORD）。这似乎是个简单的概念，但如果，MW10=MB10+MB11，那么是不是说，MW11=MB12+MB13？如果你的回答是肯定的，我建

议你继续看下去，不要跳过，因为这里的疏忽，会导致最终的程序的错误。

按位和字节来划分数据对象大小时，是以数据对象的 bit 来偏移。这句话就是说，0bit 后就是 1bit，1bit 后肯定是 2bit，以此类推直到 7bit，完成一个字节大小的指定，再有一个 bit 的偏移，就进入下一个字节的 0bit。

而按字和双字来划分数据对象大小时，是以数据对象的 BYTE 来偏移！这就是说，MW10=MB10+MB11，并不是说，MW11=MB12+MB13，正确的是 MW11=MB11+MB12，然后才是 MW12=MB12+MB13！

这个概念的重要性在于，如果你在程序中使用了 MW10，那么，就不能对 MW11 进行任何的操作，因为，MB11 是 MW10 和 MW11 的交集。

也就是说，对于“将 DB100 中的 1-11 数据字，传送到 MW1-11 中”这个具体任务而言，我们只需要对 DBW1、DBW3、DBW5、DBW7、DBW9、DBW11 这 6 个字进行 6 次传送操作即可。这就是单独分出一节，说明数据对象尺寸划分规则这个看似简单的概念的目的所在。

【循环的结构】

要“将 DB100 中的 1-11 数据字，传送到 MW1-11 中”，我们需要将指针内容按照顺序逐一指向相应的数据字，这种对指针内容的动态修改，其实就是遍历。对于遍历，最简单的莫过于循环。

一个循环包括以下几个要素：

1、初始循环指针

2、循环指针自加减

2、继续或者退出循环体的条件判断

被循环的程序主体必须位于初始循环指针之后，和循环指针自加减之前。

比如：

初始循环指针：X=0

循环开始点 M

被循环的程序主体：

循环指针自加减：X+1=X

循环条件判断：X≤10 ，False：GO TO M；True：GO TO N

循环退出点 N

如果把 X 作为间接寻址指针的内容，对循环指针的操作，就等于对寻址指针内容的动态而循环的修改了。

【将 DB100 中的 1-11 数据字，传送到 MW1-11 中】

```
 L L#1 //初始化循环指针。这里循环指针就是我们要修改的寻址指针
 T MD 102
M2: L MD 102
```

```
T #COUNTER_D
OPN DB100
L DBW [MD 102]
T MW [MD 102]
L #COUNTER_D
L L#2 // +2，是因为数据字的偏移基准是字节。
+D
T MD 102 //自加减循环指针，这是动态修改了寻址指针的关键
L L#11 //循环次数=n-1。n=6。这是因为，首次进入循环是无条件的，
但已事实上执行了一次操作。
<=D
JC M2
```

有关于 T MD102 ，L L#11， <=D 的详细分析，请按照前面的内容推导。

【将 DB1-10 中的 1-11 数据字，传送到 MW1-11 中】

这里增加了对 DB 数据块的寻址，使用单字指针 MW100 存储寻址地址，同样使用了循环，嵌套在数据字传送循环外，这样，要完成“将 DB1-10 中的 1-11 数据字，传送到 MW1-11 中”这个任务 ，共需要 M1 循环 10 次 × M2 循环 6 次 =60 次。

```
L 1
T MW 100
L L#1
T MD 102
M1: L MW 100
T #COUNTER_W
M2: 对数据字循环传送程序，同上例
L #COUNTER_W
L 1 //这里不是数据字的偏移，只是编号的简单递增，因此+1
+I
T MW 100
L 9 //循环次数=n-1，n=10
<=I
JC M1
```

通过示例分析，程序是让寻址指针在对要操作的数据对象范围内进行遍历来编程，完成这个任务。

我们看到，这种对存储器间接寻址指针的遍历是基于字节和字的，如何对位进行遍历呢？

这就是下一个帖子要分析的寄存器间接寻址的实例的内容了。

详解西门子间接寻址<4>

L [MD100]

LAR1

与

L MD100

LAR1

有什么区别？

当将 MD100 以这种 [MD100] 形式表示时，你既要在对 MD100 赋值时考虑到所赋的值是否符合存储器间接寻址双字指针的规范，又要在使用这个寻址格式作为语句一部分时，是否符合语法的规范。

在你给出第一个例程的第一句：L [MD100]上，我们看出它犯了后一个错误。

存储器间接寻址指针，是作为指定的存储区域的确切数值单元来运用的。也就是说，指针不包含区域标识，它只是指明了一个数值。因此，要在 [MD100]前加上区域标识如： M、DB、I、Q、L 等，还要加上存储区尺寸大小如：X、B、W、D 等。在加存储区域和大小标识时，要考虑累加器加载指令 L 不能对位地址操作，因此，只能指定非位的地址。

为了对比下面的寄存器寻址方式，我们这里，修改为：L MD[MD100]。并假定 MD100=8Hex，同时我们也假定 MD1=85000018Hex。

当把 MD100 这个双字作为一个双字指针运用时，其存储值的 0-18bit 将会按照双字指针的结构 Byte.bit 来重新“翻译”，“翻译”的结果才是指针指向的地址，因而 MD100 中的 8Hex=1000B=1.0，所以下面的

语句：

L MD[MD100]

LAR1

经过“翻译”就是：

L MD1

LAR1

前面我们已经假定了 MD1=85000018,同样道理,MD1 作为指针使用时,对 0-18bit 应该经过 Byte.bit 结构的“翻译”，由于是传送给 AR 地址寄存器，还要对 24-31bit 进行区域寻址“翻译”。这样，我们得出 LAR1 中最终的值=DIX3.0。就是说，我们在地址寄存器 AR1 中存储了一个指针，它指向 DIX3.0。

L MD100

LAR1

这段语句，是直接把 MD100 的值传送给 AR，当然也要经过“翻译”，结果 AR1=1.0。就是说，我们在地址寄存器 AR1 中存储了一个指针，它指向 1.0，这是由 MD100 直接赋值的。

似乎，两段语句，只是赋值给 AR1 的结果不同而已，其实不然。我们事先假定的值是考虑到对比的关系，特意指定的。如果 MD100=CHex 的呢？

对于前一段，由于 CHex=1100，其 0-3bit 为非 0，程序将立即出错，无法执行。（因为没有 MD1.4 这种地址！！）

后一段 AR1 的值经过翻译以后，等于 1.4，程序能正常执行。

不知道这个算不算：

```
FUNCTION_BLOCK "ADDRESS"
TITLE =
VERSION : 0.1
VAR_INPUT
  IN0 : BYTE ;
  IN1 : BYTE ;
  IN2 : BYTE ;
  IN3 : BYTE ;
  IN4 : BYTE ;
  IN5 : BYTE ;
  IN6 : BYTE ;
  IN7 : BYTE ;
END_VAR
VAR_OUTPUT
  OUT0 : BYTE ;
  OUT1 : BYTE ;
  OUT2 : BYTE ;
  OUT3 : BYTE ;
  OUT4 : BYTE ;
  OUT5 : BYTE ;
  OUT6 : BYTE ;
  OUT7 : BYTE ;
END_VAR
VAR
```

```
  BYTE_0 : BYTE ;
  BYTE_1 : BYTE ;
  BYTE_2 : BYTE ;
  BYTE_3 : BYTE ;
  BYTE_4 : BYTE ;
  BYTE_5 : BYTE ;
  BYTE_6 : BYTE ;
  BYTE_7 : BYTE ;
END_VAR
VAR_TEMP
  IN0_B : BYTE ;
  IN1_B : BYTE ;
  IN2_B : BYTE ;
  IN3_B : BYTE ;
  IN4_B : BYTE ;
  IN5_B : BYTE ;
  IN6_B : BYTE ;
  IN7_B : BYTE ;
  OUT0_B : BYTE ;
  OUT1_B : BYTE ;
  OUT2_B : BYTE ;
  OUT3_B : BYTE ;
  OUT4_B : BYTE ;
  OUT5_B : BYTE ;
  OUT6_B : BYTE ;
  OUT7_B : BYTE ;
END_VAR
BEGIN
NETWORK
TITLE =
//IB0
//I 0.0W1  流量故障
```

```
//I 0.1NR  快熔故障
//I 0.2W2  快熔故障
//I 0.3W1  快熔故障
//I 0.4NR SCR 过热
//I 0.5W2 SCR 过热
//I 0.6W1 SCR 过热
//I 0.7W2  流量故障
      L     #IN0;
      T     #IN0_B;
      SET   ;
      SAVE  ;
      CLR   ;
      A     BR;
      =     L     16.0;
      A     L     16.0;
      A     L     0.0;
      =     L     8.0;
      A     L     16.0;
      A     L     0.1;
      =     L     8.1;
      A     L     16.0;
      A     L     0.2;
      =     L     8.2;
      A     L     16.0;
      A     L     0.3;
      =     L     8.3;
      A     L     16.0;
      A     L     0.4;
      =     L     8.4;
      A     L     16.0;
      A     L     0.5;
      =     L     8.5;
```

```
      A     L     16.0;
      A     L     0.6;
      =     L     8.6;
      A     L     16.0;
      A     L     0.7;
      =     L     8.7;
      A     L     16.0;
      JNB   _001;
      L     #IN0_B;
      T     #OUT0;
_001: NOP   0;
      A     L     16.0;
      JNB   _002;
      L     #IN0_B;
      T     #BYTE_0;
_002: NOP   0;
NETWORK
TITLE =
//IB1
//I 1.0NR  流量故障
//I 1.1W1  接地故障
//I 1.2W2  接地故障
//I 1.3NR  接地故障
//I 1.4W1  浸水故障
//I 1.5W2  浸水故障
//I 1.6NR  浸水故障
//I 1.7W1  控制器故障
      L     #IN1;
      T     #IN1_B;
      SET   ;
      SAVE  ;
      CLR   ;
```

```
        A       BR;
        =       L       16.0;
        A       L       16.0;
        A       L       1.0;
        =       L       9.0;
        A       L       16.0;
        A       L       1.1;
        =       L       9.1;
        A       L       16.0;
        A       L       1.2;
        =       L       9.2;
        A       L       16.0;
        A       L       1.3;
        =       L       9.3;
        A       L       16.0;
        A       L       1.4;
        =       L       9.4;
        A       L       16.0;
        A       L       1.5;
        =       L       9.5;
        A       L       16.0;
        A       L       1.6;
        =       L       9.6;
        A       L       16.0;
        A       L       1.7;
        =       L       9.7;
        A       L       16.0;
        JNB     _003;
        L       #IN1_B;
        T       #OUT1;
_003: NOP     0;
        A       L       16.0;
```

```
      JNB    _004;
      L      #IN1_B;
      T      #BYTE_1;
_004: NOP    0;
NETWORK
TITLE =
//IB2
//I 2.0W2  控制器故障
//I 2.1NR  控制器故障
//I 2.2 主变合闸
//I 2.3 主变故障
//I 2.4 设备允许 NC
//I 2.5 设备允许 NO
//I 2.6 运行允许 NC
//I 2.7 运行允许 NO
      L      #IN2;
      T      #IN2_B;
      SET    ;
      SAVE   ;
      CLR    ;
      A      BR;
      =      L      16.0;
      A      L      16.0;
      A      L       2.0;
      =      L      10.0;
      A      L      16.0;
      A      L       2.1;
      =      L      10.1;
      A      L      16.0;
      A      L       2.2;
      =      L      10.2;
      A      L      16.0;
```

```
      A     L     2.3;
      =     L     10.3;
      A     L     16.0;
      A     L     2.4;
      =     L     10.4;
      A     L     16.0;
      A     L     2.5;
      =     L     10.5;
      A     L     16.0;
      A     L     2.6;
      =     L     10.6;
      A     L     16.0;
      A     L     2.7;
      =     L     10.7;
      A     L     16.0;
      JNB   _005;
      L     #IN2_B;
      T     #OUT2;
_005: NOP   0;
      A     L     16.0;
      JNB   _006;
      L     #IN2_B;
      T     #BYTE_2;
_006: NOP   0;
NETWORK
TITLE =
//IB3
//I 3.0 控制电源故障
//I 3.1 启动
//I 3.2 停止
//I 3.3 复位
//I 3.4
```

```
//I 3.5
//I 3.6
//I 3.7
      L     #IN3;
      T     #IN3_B;
      SET   ;
      SAVE  ;
      CLR   ;
      A     BR;
      =     L     16.0;
      A     L     16.0;
      A     L      3.0;
      =     L     11.0;
      A     L     16.0;
      A     L      3.1;
      =     L     11.1;
      A     L     16.0;
      A     L      3.2;
      =     L     11.2;
      A     L     16.0;
      A     L      3.3;
      =     L     11.3;
      A     L     16.0;
      A     L      3.4;
      =     L     11.4;
      A     L     16.0;
      A     L      3.5;
      =     L     11.5;
      A     L     16.0;
      A     L      3.6;
      =     L     11.6;
      A     L     16.0;
```

```
      A     L     3.7;
      =     L     11.7;
      A     L     16.0;
      JNB   _007;
      L     #IN3_B;
      T     #OUT3;
_007: NOP   0;
      A     L     16.0;
      JNB   _008;
      L     #IN3_B;
      T     #BYTE_3;
_008: NOP   0;
NETWORK
TITLE =
//QB0
//Q 0.0 运行 LED
//Q 0.1 停止 LED 高压启动时闪烁
//Q 0.2 报警 LED
//Q 0.3 准备好 REL
//Q 0.4 运行 REL
//Q 0.5 停止 REL
//Q 0.6 报警 REL
//Q 0.7 故障 REL
      L     #IN4;
      T     #IN4_B;
      SET   ;
      SAVE  ;
      CLR   ;
      A     BR;
      =     L     16.0;
      A     L     16.0;
      A     L     4.0;
```

```
        =       L       12.0;
        A       L       16.0;
        A       L       4.1;
        =       L       12.1;
        A       L       16.0;
        A       L       4.2;
        =       L       12.2;
        A       L       16.0;
        A       L       4.3;
        =       L       12.3;
        A       L       16.0;
        A       L       4.4;
        =       L       12.4;
        A       L       16.0;
        A       L       4.5;
        =       L       12.5;
        A       L       16.0;
        A       L       4.6;
        =       L       12.6;
        A       L       16.0;
        A       L       4.7;
        =       L       12.7;
        A       L       16.0;
        JNB     _009;
        L       #IN4_B;
        T       #OUT4;
_009: NOP     0;
        A       L       16.0;
        JNB     _00a;
        L       #IN4_B;
        T       #BYTE_4;
_00a: NOP     0;
```

```
NETWORK
TITLE =
//QB1
//Q 1.0 主变合闸 REL
//Q 1.1 触发允许 REL
//Q 1.2 风机 REL
//Q 1.3
//Q 1.4
//Q 1.5
//Q 1.6
//Q 1.7
      L     #IN5;
      T     #IN5_B;
      SET   ;
      SAVE  ;
      CLR   ;
      A     BR;
      =     L     16.0;
      A     L     16.0;
      A     L      5.0;
      =     L     13.0;
      A     L     16.0;
      A     L      5.1;
      =     L     13.1;
      A     L     16.0;
      A     L      5.2;
      =     L     13.2;
      A     L     16.0;
      A     L      5.3;
      =     L     13.3;
      A     L     16.0;
      A     L      5.4;
```

```
      =     L     13.4;
      A     L     16.0;
      A     L      5.5;
      =     L     13.5;
      A     L     16.0;
      A     L      5.6;
      =     L     13.6;
      A     L     16.0;
      A     L      5.7;
      =     L     13.7;
      A     L     16.0;
      JNB   _00b;
      L     #IN5_B;
      T     #OUT5;
_00b: NOP   0;
      A     L     16.0;
      JNB   _00c;
      L     #IN5_B;
      T     #BYTE_5;
_00c: NOP   0;
NETWORK
TITLE =
//QB2
//Q 2.0W1 脉冲允许 REL
//Q 2.1W1 并联 REL
//Q 2.2W1 串联 REL
//Q 2.3W1 电压 LED 并联运行时闪烁，LED 亮表头为 UPs1,LED 灭表头为 UPn1
//Q 2.4W2 脉冲允许 REL
//Q 2.5W2 并联 REL
//Q 2.6W2 串联 REL
//Q 2.7W2 电压 LED 并联运行时闪烁，LED 亮表头为 UPs1,LED 灭表头为 UPn1
//
```

```
L       #IN6;
T       #IN6_B;
SET    ;
SAVE   ;
CLR    ;
A       BR;
=       L       16.0;
A       L       16.0;
A       L       6.0;
=       L       14.0;
A       L       16.0;
A       L       6.1;
=       L       14.1;
A       L       16.0;
A       L       6.2;
=       L       14.2;
A       L       16.0;
A       L       6.3;
=       L       14.3;
A       L       16.0;
A       L       6.4;
=       L       14.4;
A       L       16.0;
A       L       6.5;
=       L       14.5;
A       L       16.0;
A       L       6.6;
=       L       14.6;
A       L       16.0;
A       L       6.7;
=       L       14.7;
A       L       16.0;
```

```
      JNB    _00d;
      L      #IN6_B;
      T      #OUT6;
_00d: NOP    0;
      A      L      16.0;
      JNB    _00e;
      L      #IN6_B;
      T      #BYTE_6;
_00e: NOP    0;
NETWORK
TITLE =
//QB3
//Q 3.0NR 脉冲允许 REL
//Q 3.1NR 并联 REL
//Q 3.2NR 串联 REL
//Q 3.3NR 电压 LED 并联运行时闪烁，LED 亮表头为 UPs1,LED 灭表头为 UPn1
//Q 3.4
//Q 3.5
//Q 3.6
//Q 3.7
      L      #IN7;
      T      #IN7_B;
      SET    ;
      SAVE   ;
      CLR    ;
      A      BR;
      =      L      16.0;
      A      L      16.0;
      A      L       7.0;
      =      L      15.0;
      A      L      16.0;
      A      L       7.1;
```

```
      =     L     15.1;
      A     L     16.0;
      A     L      7.2;
      =     L     15.2;
      A     L     16.0;
      A     L      7.3;
      =     L     15.3;
      A     L     16.0;
      A     L      7.4;
      =     L     15.4;
      A     L     16.0;
      A     L      7.5;
      =     L     15.5;
      A     L     16.0;
      A     L      7.6;
      =     L     15.6;
      A     L     16.0;
      A     L      7.7;
      =     L     15.7;
      A     L     16.0;
      JNB   _00f;
      L     #IN7_B;
      T     #OUT7;
_00f: NOP   0;
      A     L     16.0;
      JNB   _010;
      L     #IN7_B;
      T     #BYTE_7;
_010: NOP   0;
END_FUNCTION_BLOCK.
```

step7 中的难点：间接寻址示例，中文详细注释。

这个例子的作者是BaiZH ，我在学习后根据自己的理解加了中文注释，也许对新手有所帮助。感谢BaiZH

无私提供源码，感谢cvlsam不厌其烦的指点，有所领悟不敢独享。

背景知识：http://bbs.e10000.cn/a/a.asp?B=302&ID=97070

欢迎纠错，防止误导。

```
FUNCTION "DBtoDB" : VOID //该功能块的作用是把一个数据块中的指定的一批数据，复制到另一个块的指定位置。
TITLE =                  //标题，这里没有指定
AUTHOR : BaiZH           //作者 感谢您，BaiZH，通过您的这个例子我基本入明白了间接寻址的用法。不过具体在什么情况下使用我还得继续努力。
FAMILY : IR              //分类
NAME : DBtoDB            //名称
VERSION : 0.1            //版本
VAR_INPUT                                                                  //输入型变量声明开始
  SRC_DB : INT ; //Source DB Block Number                                  //整型值，要复制的源数据块块号
  SRC_SttAddr : INT ; //Start Address of the Sending Data in SRC_DB  //源数据块的要复制的数据起始地址
  SendNum : INT ; //Words Number Need Sending                              //要复制的数据量
  DST_DB : INT ; //Destination DB Block Number                             //目标数据块号
  DST_SttAddr : INT ; //Start Address of the Receiving Data in DST_DB //目标数据块中数据起始地址
END_VAR
VAR_IN_OUT                                                                 //输入输出变量声明
  Enable : BOOL ; //Enable Bit                                            //使能此功能块位
END_VAR
VAR_TEMP                                                                   //声明临时变量
  DB_LOAD_TEMP : INT ;                                                     //存放临时数据块块号
  Loop_Val : INT ;                                                         //Send Data Loop Value //循环次数
  DB_SAVE : INT ;                                                          //保存进入此函数前，系统已经打开的数据块号
```

```
  DI_SAVE : INT ;                                                    //同上
  AR1_SAVE : DWORD ;                                                 //保存进入此函数
前，地址寄存器 1 中的值
  AR2_SAVE : DWORD ;                                                 //同上
END_VAR
BEGIN  //在STEP7 的BLOCK中编辑时的程序主要从这里开始
NETWORK
TITLE =Send Data
//Move data from DB to DB
      A     #Enable;    //使能位，ENABLE为 1 执行以下程序
      JCN   END;         //否则跳转到最后
      TAR1  #AR1_SAVE; // Save AR and Opened DB //保存进入此函数前的数据到临时变量中，以备
离开时复原
      TAR2  #AR2_SAVE;
      L     DBNO;     //同上，保存调用前的现场数据，以备调用完毕复原主程序的现场数据
      T     #DB_SAVE;  //一个DBNO，一个DINO，是因为要同时打开两个数据块，只能一个背景数据
块，一个共享数据块。
      L     DINO;
      T     #DI_SAVE;
      L     #SRC_DB; //Open DB  //把要打开的数据块块号通过中间变量#DB_LOAD_TEMP传送。它的
好处引用cvlsam的指点，详细：http://bbs.e10000.cn/a/a.asp?B=302&ID=608300
      T     #DB_LOAD_TEMP;
      OPN   DB [#DB_LOAD_TEMP];
      L     #DST_DB; //Open DB
      T     #DB_LOAD_TEMP;
      OPN   DI [#DB_LOAD_TEMP];
      L     #SRC_SttAddr; //Load Start Address //要复制的数据起始地址
      SLD   3; //左移位，使的地址指针最右边三位保证为 0，确保符合地址格式的要求。详细：
http://bbs.e10000.cn/a/a.asp?B=302&ID=608300
      LAR1  ;
      L     #DST_SttAddr;
      SLD   3;
```

```
      LAR2  ;
      L     #SendNum;  开始循环程序，把复制的数据量放入循环变量中
LP1:  T     #Loop_Val; //Move Data
      L     DBW [AR1,P#0.0];
      T     DIW [AR2,P#0.0];
      +AR1  P#2.0;  //指针移位
      +AR2  P#2.0;
      L     #Loop_Val;
      LOOP  LP1; //循环变量减 1,判断循环条件
      LAR1  #AR1_SAVE; //Recover Original AR and DB//这里在执行完功能后，开始恢复调用前的主程序现场数据。
      LAR2  #AR2_SAVE;
      OPN   DB [#DB_SAVE];
      OPN   DI [#DI_SAVE];
      SET   ; //系统将RLO置 1，代表FB（FC）执行完毕，相当于功能块的ENO使能输出位。再次感谢cvlsam。
      R     #Enable;
END:  NOP   0;
END_FUNCTION
```

非常感谢楼主的共享！

以前我也编写了一个类似功能的程序，而且常常在我的工程上用到，今天也发上来与大家一起交流。当时感觉程序很简单，所以没有加注释。

一、用STL编写的：

```
FUNCTION "数据块传送" : VOID
TITLE =数据块传送
VERSION : 0.1
VAR_INPUT
  DB1_No : INT ;       //源DB号
  DB1_begin : INT ;       //源区字节初值
  DB2_No : INT ;       //目标DB号
  DB2_begin : INT ;       //目标区字节初值
  longness : INT ;       //传送字节长度
```

```
  SW : BOOL ;       //传送开关
END_VAR
VAR_TEMP
  index_01 : INT ;
  index_02 : INT ;
  index_03 : DINT ;
  index_04 : DINT ;
  data : INT ;
  data1 : INT ;
END_VAR
BEGIN
NETWORK
TITLE =循环传送
    A     #SW;
      JNB   P002;
      L     #DB1_No;
      T     #index_01;
      L     #DB2_No;
      T     #index_02;
      L     #DB1_begin;
      ITD   ;
      L     8;
      *D    ;
      T     #index_03;
      L     #DB2_begin;
      ITD   ;
      L     8;
      *D    ;
      T     #index_04;
      L     #longness;
P001: T     #data;
      OPN   DB [#index_01];
```

```
      L     DBB [#index_03];
      OPN   DB [#index_02];
      T     DBB [#index_04];
      L     #index_03;
      L     L#8;
      +D    ;
      T     #index_03;
      L     #index_04;
      L     L#8;
      +D    ;
      T     #index_04;
      L     #data;
      LOOP  P001;
P002: NOP   0;
END_FUNCTION
```

二、用SCL编写的：

```
FUNCTION_BLOCK FB2 //数据块copy
TITLE = '数据块copy';
// Block Parameters
VAR_INPUT
    // Input Parameters
    DB1_BLOCK:INT;//BLOCK_DB;//源始DB
    DB1_begin:INT;//源始DB起始值
    DB2_BLOCK:INT;//BLOCK_DB;//目标DB
    DB2_begin:INT;//目标DB起始值
    longness:INT;//传送字节长度
    SW:BOOL:=0;//传送开关
END_VAR
VAR
    // Static Variables
    index_1:INT;
    index_2:INT;
```

```
    index:INT;
END_VAR
    index_1:=DB1_begin;
    index_2:=DB2_begin;
    index:=longness;
    // Statement Section
    IF SW = 1 THEN
        WHILE index > 0 DO
        // Statement Section
        WORD_TO_BLOCK_DB(INT_TO_WORD(DB2_BLOCK)).DB[index_2]:=WORD_TO_BLOCK_DB(INT_TO_WOR
D(DB1_BLOCK)).DB[index_1];
        index_1:=index_1+1;
        index_2:=index_2+1;
        index:=index-1;
        END_WHILE;
    END_IF;
 END_FUNCTION_BLOCK
```