第 14 章 开放式 PLC 的开发

PLC 以其工作可靠、编程方便被广泛应用于工业控制现场。目前 PLC 常采用梯形图进行编程，广大工程技术人员使用这一工具时基本没有编程方面的困难，因而 PLC 易于在工控现场推广使用。但是 PLC 价格不菲，而在一些应用场合，如果使用单片机探制板来完成同样的功能，成本可能低至其若干分之一。

目前，市场上已可见多款单片机工控板，虽然这些控制板与 PLC 相比有明显的价格优势，但目前很多工程师仍在观望，部分人进行了一些尝试，总体说来，很多做系统集成的公司或者工业现场的工程师仍不愿用单片机工控板替代 PLC。究其原因，除了在硬件抗干扰等方面尚不完全成熟外，单片机工控板需要使用较为复杂的汇编或者 C 语言进行开发，很多人感到畏惧，不愿也不敢去尝试使用。

作者开发了 DKB-1A 型工控板，然后又开发了一套程序，可以使用梯形图为该工控板编写程序。其开发流程如图 14-1 所示。

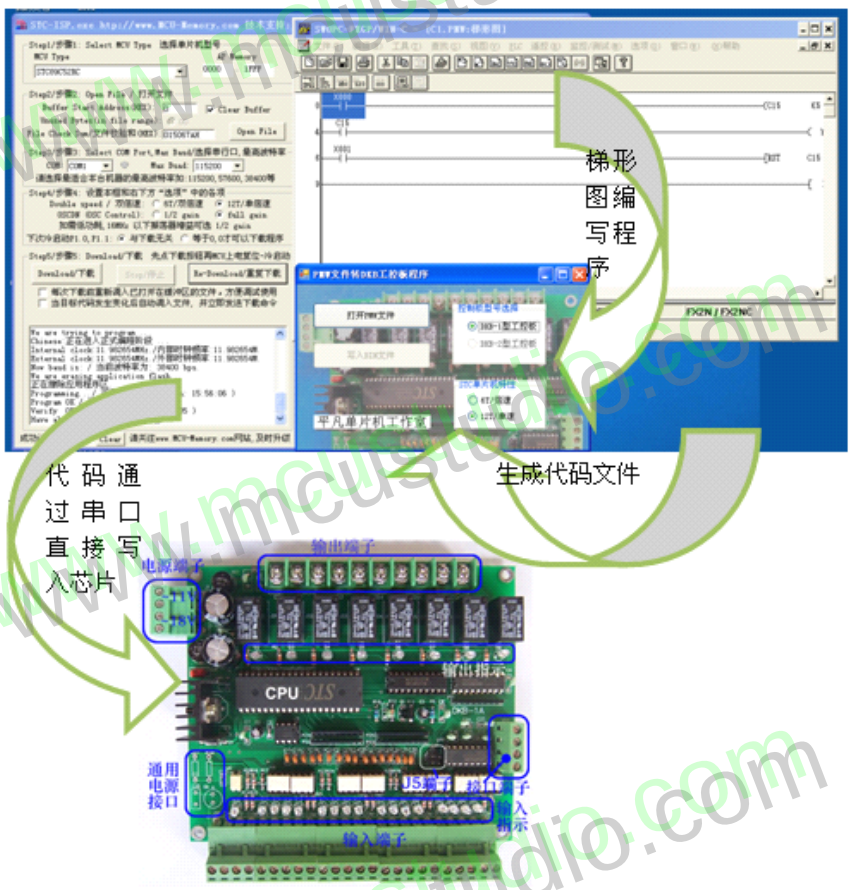

图 14-1 使用梯形图编写工控板程序的流程

由图可见，开发方法非常简单，即画出梯形图→转换成为 Hex 格式文件→将 HEX 格式文件其写入芯片中。采用这种方法进行开发，基本上不需要任何额外的开发成本，而编程又非常方便。这一产品在网站公开后，很多人表示对此有兴趣。

以此为基础，作者进一步开发了“开方式 PLC”这一产品，其外形如图 14-2 所示，性能如下：

- 12 点光耦隔离输入；

- 8 点继电器隔离输出；
- 板上自带 RS232 通信功能；
- 安装有 DS1302 实时钟和后备电池；
- 使用 STC12 系列高速芯片，兼容 51 系列，片内 RAM 达 1280Byte；
- CPU 具有在线可编程功能，通过 RS232 即可编程，使用方便；
- 可安装铁电系列 FLASH（FRAM）；
- 1 路高速计数输入；
- 2 路高速脉冲输出；
- 2 路 AD 转换输入；
- 2 路顶调电位器输入；
- 自带 485 通信功能。

图 14-2 开放式 PLC 的外形

这个开放式 PLC 既可以使用汇编语言、C 语言等编程方法来开发，又可以通过梯形图转换成 HEX 文件的方法来开发，非常方便。下面就介绍一下如何实现将梯形图转换成为 HEX 的过程，供读者参考。

14．1 PLC 简介

通常 PLC 指令较多，各种不同型号的 PLC 指令也各不相同。但用于逻辑量处理的指令并不多，各种型号的 PLC 此类指令也是大同小异。表 1 列出了某型 PLC 常用的指令及其含义。

表 14-1 梯形图指令及其功能描述

指令助记符	功 能 描 述	指令助记符	功 能 描 述
LD	使常开触点与左母线相连	OUT	线圈驱动指令
LDI	使常闭触点与左母线相连	SET	线圈动作保持指令
LDP	上升沿检出运算开始	RST	解除线圈动作保持指令
LDF	下降沿检出运算开始	PLS	线圈上升沿输出指令
AND	继电器常开触点与其他继电器触点串联	PLF	线圈下降沿输出指令
ANI	继电器常闭触点与其他继电器触点串联	MC	公共串联接点用线圈指令
ANDP	继电器常开触点闭合瞬间与前面的触点串联一个扫描周期	MCR	公共串联接点解除指令

ANDF	继电器常开触点断开瞬间与前面的触点串联一个扫描周期	MPS	运算存储
OR	继电器常开触点与其他继电器触点并联	MRD	存储读出
ORI	继电器常闭触点与其他继电器触点并联	MPP	存储读出和复位
ORP	继电器常开触点闭合瞬间与前面的触点并联一个扫描周期	INV	运算结果取反
ORF	继电器常开触点断开瞬间与前面的触点并联一个扫描周期	NOP	无动作
ANB	电路块之间串联	END	程序结束
ORB	电路块之间并联		

本书不对 PLC 指令详细说明，读者如对这些指令的用法有疑问，可以找 PLC 教材阅读。

14．2 梯形图转换方法分析

要使用梯形图的方式来编写单片机程序，关键是将梯形图转化为单片机可以识读的 HEX 格式文件。通常梯形图是用一些专用软件在 PC 机上绘制，如图 14-3 所示是一个简单的梯形图。

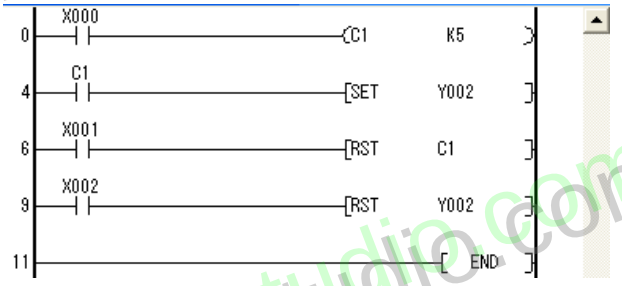

图 14-3 梯形图例子

如果希望采用这种图形化方式来编程，就需要自行开发可以绘制梯形图的软件，限于条件，这种方案不考虑。

自行开发图形编程软件不考虑，那么就要对梯形图进一步进行分析。对应于梯形图，最终会形成一个指令表，如图 14-3 的梯形图，可以转换为如下所示的指令表：

```
LD    X000
OUT   C1   K5
LD    C1
SET   Y002
LD    X001
RST   C1
LD    X002
RST   Y002
END
```

这是一种文本格式，文本处理较之图形要容易一些。因此，可以考虑编写一段程序，读入每一条指令，然后对其进行解释以直接生成 HEX 格式文件或者生成 C 语言源程序，最后编译、链接生成 HEX 格式文件。这种方式较之上述直接编写图形化软件要方便一些，但仍不是最佳选择。

绘制好的梯形图最终必须保存为文件，对文件分析可以得到诸多有用的信息。因此接下

来对各条指令在文件中的保存情况进行分析，以找到每条指令与其保存代码之间的关系。

14．2．1　LD 类指令

LD 类指令功能是使常开触点与左母线连接。LD 指令的操作元件可以是输入继电器 X、输出继电器 Y、辅助继电器 M、状态继电器 S、定时器 T 和计数器 C 中的任何一个。下面首先分析 LD　S 类指令。

ld　　s*　（*的取值从 000~999）

这是取状态继电器 s 触点的指令，s 元件的个数最多为 1000 个，当 PLC 编程指令为：

Ld　　s0

时，其对应的二进制代码为：

00　20

而当 PLC 指令为：

LD　S999

时，其对应的代码为：

0xE7　0x23

不难看出，，代码中第一个数是元件号，第二个数是命令码，可以用来区分是哪一条指令。

接下来再分析 LD　X 类指令

指令：ld　x*　（*的取值从 0~255，用 8 进制表示）

经过分析，其代码为：* 24，即当指令为：LD X0 时，其对应的代码为：00 24，而当指令为：LD　X377（377 为 8 进制数，相当于十进制的 255）时，其对应的代码为：00 FF。

分析了这两条指令，下面的指令都是类似的，就不再一一分析了，表 14-2 给出了指令与其代码之间的对应关系。

表 14-2　Ld 类指令与代码的关系

指令	Ld　s*	Ld　X*	Ld　Y*	Ld　T*	Ld　M*	Ld　C*
代码	00 20 ~ e7 23	00 24~ff 24	00 25~ff 25	00 26~ff 26	00 28~ff 2d	00 2e~ff 2e

说明：S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255。在书写指令时，X 和 Y 元件中的*用八进制书写，其他元件中的*均用十进制书写。如用指令格式书写 PLC 程序时，不会出现 LD X8 或 LD Y9 之类的指令，因为对 X 或 Y 操作时，其后数字必须是八进制。可以出现 LD　　S8 或 LD　　M9 之类的指令，因为这些操作数均用十进制表示。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

LDI 类指令称之为“取反指令”，其功能是使常闭触点与左母线连接。LDI 指令的操作元件与 LD 类指令相同。其指令与代码关系如表 14-3 所示。

表 14-3　LdI 类指令与代码的关系

指令	Ld　s*	Ld　X*	Ld　Y*	Ld　T*	Ld　M*	Ld　C*
代码	00 30 ~ e7 33	00 34~ff 34	00 35~ff 35	00 36~ff 36	00 38~ff 3d	00 3e~ff 3e

说明：S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255。在写指令时，X*，Y*中的*用八进制写法，其他均用十进制写法。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．2　AND 和 ANI 类指令

当继电器的常开触点与其他继电器的触点串联时，使用 AND 指令。AND 指令与代码关系如表 14-4 所示。

表 14-4　AND 类指令与代码的关系

指令	AND　s*	AND　X*	AND　Y*	AND　T*	AND　M*	AND　C*
代码	00 40 ~ e7 43	00 44~ff 44	00 45~ff 45	00 46~ff 46	00 48~ff 4d	00 4e~ff 4e

说明：S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255。在书写指令时，X 和 Y 元件中的*用八进制书写，其他元件中的*均用十进制书写。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

当继电器的常闭触点与其他继电器的触点串联时，使用 ANI 指令。ANI 指令与代码关系如表 14-5 所示。

表 14-5　ANI 类指令与代码的关系

指令	ANI　s*	ANI　X*	ANI　Y*	ANI　T*	ANI　M*	ANI　C*
代码	00 50 ~ e7 53	00 54~ff 54	00 55~ff 55	00 56~ff 56	00 58~ff 5d	00 5e~ff 5e

说明：S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255。在书写指令时，X 和 Y 元件中的*用八进制书写，其他元件中的*均用十进制书写。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．3　OR 和 ORI 类指令

继电器的常开触点与其他继电器的触点并联时，使用 OR 指令。OR 类指令与代码关系如表 14-6 所示。

表 14-6　OR 类指令与代码的关系

指令	OR　s*	OR　X*	OR　Y*	OR　T*	OR　M*	OR　C*
代码	00 60 ~ e7 63	00 64~ff 64	00 65~ff 65	00 66~ff 66	00 68~ff 6d	00 6e~ff 6e

说明：S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255。在书写指令时，X 和 Y 元件中的*用八进制书写，其他元件中的*均用十进制书写。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

继电器的常开触点与其他继电器的触点并联时，使用 ORI 指令。ORI 类指令与代码关系如表 14-7 所示。

表 14-7　ORI 类指令与代码的关系

指令	ORI　s*	ORI　X*	ORI　Y*	ORI　T*	ORI　M*	ORI　C*
代码	00 70 ~ e7 73	00 74~ff 74	00 75~ff 75	00 76~ff 76	00 78~ff 7d	00 7e~ff 7e

说明： S*中的*取值为 000~999，M*中的*取值为 0000~1536，其余*的取值为 0~255;在书写指令时，X 和 Y 元件中的*用八进制书写，其他元件中的*均用十进制书写。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．4　ANB、ORB、MPS、MRD、MPP、INV 指令

在梯形图中，可能会出现电路块与电路块串联，或者电路块与电路块并联的情况，这时

要使用 ANB 或者 ORB 指令；在 PLC 中，有若干个存储运算中间结果的存储器，称为栈存储器。这个栈存储器将触点之间的逻辑运算结果存储后，可以用指令将这个结果读出，再参与其他触点之间的逻辑运算。这一类指令共有 3 条，即 MPS、MRD 和 MPP，分别是进栈指令、读栈指令和出栈指令；INV 是取反指令，即将当前状态取反。从 PLC 指令的角度来看，这 6 条指令并没有较强的关联关系，但是从其代码来看，却有一定的相关性，因此，这 6 条指令列表如表 14-8 所示。

表 14-8　ANB、ORB 等指令与代码的关系

指令	ANB	ORB	MPS	MRD	MPP	INV
代码	F8 FF	F9 FF	FA FF	FB FF	FC FF	FD FF

说明：表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．5　MC 指令与 MCR 指令

MC 指令称为“主控指令”。通过 MC 指令的操作元件 Y 或 M 的常开触点将左母线临时移到一个所需要的位置，产生一个临时左母线，形成一个主控电路块。而 MCR 指令称为“主控复位指令”。MCR 指令的功能是取消临时左母线，即将临时左母线返回原来位置，结束主控电路块。MC 与 MCR 指令与代码的关系如表 14-9 所示。

表 14-9　MC、MCR 等指令与代码的关系

指令	MC N* M**	MC N* Y**	MCR M0
代码	0A 00 * 80 ** 88	0A 00 * 80 ** 85	0B 00 *80

说明：代码中 0A 00 是命令码，代码中 0x80 前的*的值就是指令中 N 后的数值，代码中 0x88 前面的**的值就是指令中 M 后的数值。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．6　OUT 类指令

OUT 指令称为“输出指令”或“驱动指令”，驱动指令的操作元件可以是输出继电器 Y、辅辅助继电器 M、状态继电器 S、定时器 T 和计数器 C 中的任何一个。OUT S、OUT Y 和 OUT M 三类指令与代码的关系如表 14-10 所示。

表 14-10　OUT S、OUT Y、OUT M 指令与代码的关系

指令	OUT S*	OUT　Y*	OUT M*（*从 0~1535）	OUT　M*　（　*　从 1535~3071）
代码	05 00 00 80 ~ 05 00 E7 83	00 C5~ FF c5	00　C8~ FF　CD	02 00 00 A8~ 02 00 FF AD

说明：指令中 s*和 m*中的*是一个数，表示元件号，用十进制表示。Y*中的*是一个数，用八进制表示。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

OUT　C 和 OUT　T 类指令除需要指定元件号以外，还需要指定计数常数、定时常数，因此其代码需要 6 位。例如某条 OUT C 类指令：out　c0　k513

其代码格式为：00　0E　01　80　02　80

其中：　0e 是命令码，0E 前面的数字 00 是计数器元件号，而第 1 个 80 前面的 01 和第 2 个 80 前面的 02 则表示了计数初值，其中 02 为高字节，01 为低字节，即计数初值为 0x201 即十进制的 513。而两个 80 则是识别码。这样就可以归纳出 OUT C 类指令与代码的关系：

指令：OUT C * K**
代码 ：* 0E （**值的低 8 位） 80 （**值的高 8 位） 80
OUT T 类指令
指令：OUT T* K**
代码：* 05 （**值的低 8 位） 80 （**值的高 8 位） 80

14．2．7 SET 与 RST 类指令

SET 指令称为“置位指令”。SET 指令的功能是驱动线圈，使其具有自锁功能，维持接通状态。置位指令的操作元件为输出继电器 Y、辅助继电器 M 和状态继电器 S。它们的指令与代码格式各不相同，下面要别分析。

表 14-11 SET 类指令与代码的对应关系

指令	SET S*	SET Y*	SET m0~ SET M1535	SET M1536~SET M3071
代码	06 00 00 80~ 06 00 E7 83	* D5	00 D8~FF DD	03 00 00 A8~03 00 FF AD

说明：指令中 s*和 m*中的*是一个数，表示元件号，用十进制表示。Y*中的*是一个数，用八进制表示。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

RST 指令称为“复位指令”。RST 指令的功能是使线圈复位。复位指令的操作元件为输出继电器 Y、辅助继电器 M、状态继电器 S、积算定时器 T 和计数器 C。以下逐一分析其指令与代码的对应关系。

表 14-12 RST 类指令与代码的对应关系

指令	RST S*	RST Y*	RST M0~ RST M1535	RST M1536~ RST M3071	RST C*
代码	07 00 00 80~ 07 00 e7 83	* E5	00 e8~ff ed	04 00 00 a8~ 04 00 ff ad	0c 00 * 8e

说明：指令中 s*和 m*中的*是一个数，表示元件号，用十进制表示。Y*中的*是一个数，用八进制表示。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．8 LDP 和 LDF 指令

LDP、LDF 类指令功能与 LD 类指令基本一样，用于常开触点接左母线，但不同的是 LDP 指令让常开触点只在闭合的瞬间接到左母线一个扫描周期，而 LDF 指令让常开触点只在断开的瞬间接到左母线一个扫描周期。LDP 和 LDF 指令的操作元件可以是输入继电器 X、输出继电器 Y、辅助继电器 M、状态继电器 S、定时器 T 和计数器 C 中的任何一个。

表 14-11 LDP 类指令与代码的关系

指令	LDP S*	LDP X*	LDP Y*	LDP T*	LDP M0~ LDP m1535	LDP C*
代码	Ca 01 00 81 Ca 01 e7 83	Ca 01 * 84	Ca 01 * 85	Ca 01 * 86	Ca 01 00 88 Ca 01 ff 8d	Ca 01 * 8e

说明：指令中 s*和 m*中的*是一个数，表示元件号，用十进制表示。Y*中的*是一个数，用八进制表示。表格中代码所在行中的数均为 16 进制数，为简单起见，未加前缀 0x。

14．2．9　NOP 和 END 指令

NOP 指令称为“空操作指令”。NOP 指令可以在调试程序时取代一些不必要的指令。

指令：NOP
代码：0xff　0xff

空操作指令，不做任何动作。

END 指令称为“结束指令”。当遇到结束指令后，结束指令后的指令即不再执行。注意结束指令不是意味着要求 PLC 停机，而是作为一次循环扫描结束的标志，一旦遇到这条指令，说明所有需要分析-执行的指令全部执行完毕，退出指令分析过程，转而将当前输出状态送到输出端口，从输入端口读取输入值并存入内存映像单元中。然后再进行下一轮的取指令-分析和执行指令的循环过程。

指令：END
代码：0f　00

通过对 PLC 指令的分析，我们已掌握了指令与其代码的关系，这种关系完全是一一对应的，因此使用单片机处理 PLC 程序只要处理代码即可。

14．3 使用单片机处理 PLC 程序

PLC 的工作过程是一个不断循环扫描的过程。每一次扫描过程包括：输入采样、程序执行和输出刷新三个阶段，如图 14-4 所示。

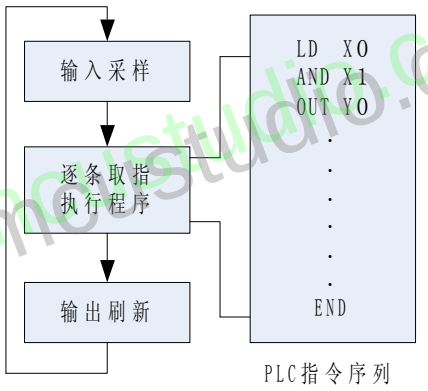

图 14-4　PLC 程序的工作过程示意图

（1）输入采样阶段：　PLC 在输入采样阶段，首先扫描所有输入端，并将各输入端的状态存入对应的输入映像寄存器中。当输入映像寄存器被刷新后，进入程序执行阶段。

（2）程序执行阶段：不论梯形图如何画，是否有分支、块等，最终得到的指令序列是一个一维的指令序列。PLC 按顺序逐句扫描执行程序。当指令中涉及输入状态时，CPU 从输入映像寄存器中读取输入状态，而不是直接去读输入端的状态。当指令需要输出时，CPU 将待输出的数据送到输出映像寄存器，而不是直接进行输出。当指行到结束指令 END 时，结束程序执行阶段，进入输出刷新阶段。

（3）输出刷新阶段：当用户程序执行结束后，输出映像寄存器中所有输出继电器的状态，在输出刷新阶段转存到输出锁存器中，并最终驱动执行机构（晶体管、继电器等动作）。

输出刷新阶段完成后，转到输入采样又开始下一轮循环，只要 PLC 不断电，这个循环就会一直不停地工作下去。

14．3．1 整体流程

使用单片机来处理 PLC 程序时，也必须按照这一工作过程来进行，如图 14-5 所示是这个处理程序的总流程示意图。

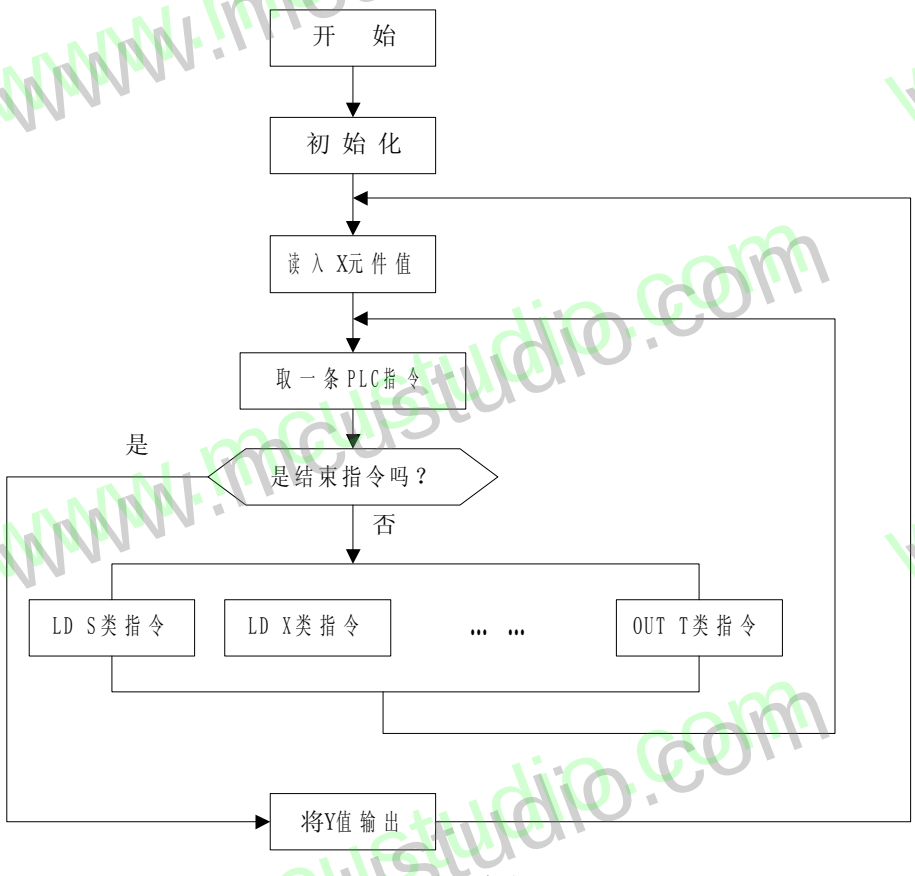

图 14-5 主程序流程图

程序开始运行后，首先对内存、定时器、I/O 口等进行初始化；然后读入 X 元件值，即输入采样；随后进入逐条取指令-执行指令的阶段，每取一条指令先判断该指令是否是 END 指令，如果不是则要对每一条指令进行分析判断并执行，执行完一条指令后转去取下一条指令并分析执行，如此循环不断；当取到 END 指令时，进入输出刷新阶段，将 Y 值通过 I/O 口输出。

从 12.2 节的分析可以看出，每一条指令都有其特定的操作码，因此，对操作码进行判断就能知道这条指令要做的工作；除了诸如 NOP、END 等少数指令外，大部分指令都用操作数表示了操作对象、参数等特性。因此，编程时先找出操作码，然后通过一个 switch 语句来区分，根据指令的用途查找这条指令的操码，并且最终完成这条指令的操作。

根据这样的思路，写出了程序，下面是指令处理的部分程序：

```
    for（;;）
    {    InPut（）;                    //读输入数据，即输入采样。
    /*从这里开始一次读指令-处理指令的过程，直到所有指令执行完毕，遇到结束指令END，才能退出这个无限循环*/
        for（i=0;;）                   //i 的增加由循环内部控制
        {
            pCode[0]=Code[i];
```

```
            i++;
            pCode[1]=Code[i];            //从指令数组中读出 2 字节的指令
            i++;
            if（（pCode[0]==0x0f）&&（pCode[1]==0x00））    //结束指令
                break;
    switch （pCode[1]）   //对指令进行分析和执行
    {
    ……
    }
    OutPut（）;              //输出
    }
```

程序分析：InPut（）函数用于输入采样，这一阶段完成以后，接下来就是程序执行阶段，这实际上是一个“读取指令-执行指令”的过程，这里采用了一个无限循环来完成这一过程，而退出这个无限循环则通过在循环体内通过判断是否遇到结束指令来实现。

本段程序用于测试梯形图指令，因此用了一种简单的方法，直接将有关梯形图指令代码放在数组中。例如有这样的一段 PLC 程序：

指令		代码					说明
ld	x0	0x00	0x24				取 x0 的状态
out	c0 k10	0x00	0x0e	0x0a	0x80	0x00 0x80	设置计数器 C0，计数值为 10
ld	c0	0x00	0x2e				取计数器 c0 的输出触点
out	y0	0x00	0xc5				驱动 Y0 输出
ld	x1	0x01	0x24				取 X1 的状态
rst	c0	0x0c	0x00	0x00	0x00	0x8e	复位 C0
end		0x0f	0x0				结束

要将这段程序让单片机处理，那么就在程序中定义了这样一个数组：

```
uchar Code[]=
{0x00，0x24，0x00，0x0e，0x0a，0x80，0x00，0x80，0x00，0x2e，0x00，0xc5，0x01，0x24，0x0c，0x00，0x00，0x8e，0x0f，0x00};
```

这样就描述了这段 PLC 程序。

读取指令时，将指令代码读入两个变量 pCode[0]和 pCode[1]中，同时将指针加 1，指向下一条指令。如第 1 条指令代码 00 24 其中 24 就是操作码，而 00 就是操作数。

随后就要判断这条指令是否是结束指令了，程序通过下面的程序行来判断：

```
        if（（pCode[0]==0x0f）&&（pCode[1]==0x00））          //结束指令
            break;
```

一段 PLC 程序的所有有效指令执行结束时，将遇到“结束”指令，在 for（i=0;;）这个循环中，判断出现了“结束”指令，就将执行 break 指令，结束循环。接下来执行 OutPut 函数，将本段指令执行过程中得到的结果集中输出，并再次回到 for（;;）大循环中去，进行下一轮循环，这种循环将一直持续不断地进行直到断电为止。

真正在实现 PLC 功能时，不可能通过手工方法将 PLC 指令用数组格式存放在源程序中，而是要将其放在单片机的 ROM 中。如果使用具有 8K 容量的单片机 89S52 或者其他类似的芯片，其中前 4K 可用于存放 PLC 操作平台的代码，而后 4K 则准备用于存放梯形图指令所对应的代码。由于 PLC 的指令大部分为双字节，少量的为 4 字节或者 6 字节，因此，4K 代码空间约可放 1K~2K 条指令的 PLC 程序，如果这一容量不够使用，还可以使用具有更大容量的单片机。至于如何将 PLC 图程序文件中的指令部分提取出来，并且与操作平台代码合

并，形成一个完整的代码，将在本章的最后一节介绍。

存放在 ROM 中的指令读出是很容易的，使用 C 语言编程时用指针即可做到。下面的函数就是读出 ROM 中存放指令的程序：

```
uchar GetChar（uint CodeAddr）
{    uchar code *p;          //指向程序区的指针
     uchar GetDat;
     p=CodeAddr;             //将地址值赋给指针变量
     GetDat=*p;              //读取指针所指 ROM 单元的值
     return GetDat;          //返回读到的值
}
```

调用这段函数时，给出指令所在地址即可读出该地址中所存放的 PLC 指令。

14．3．2 输入采样

InPut（）函数用来进行输入采样，即将 X 元件的状态采集到内部映像单元中，该函数如下。

```
/*********************************************************************
函数功能：读取输入端口的数据，并且送入内存映像单元中
参      数：输出数据
返      回：无
备      注：不同的硬件设计，只需改变这个函数即可
*********************************************************************/
void InPut（）
{
     InPort1=0xff;           //根据准双向 IO 口要求，输入端口置高电平
     InPort2=0xff;           //根据准双向 IO 口要求，输入端口置高电平
     InDat[0]=InPort1;       //端口 1 读到的数据送到 InDat[0]变量中
     InDat[1]=InPort2;       //端口 2 读到的数据送到 InDat[1]变量中
}
```

变量 InDat[0]和 InDat[1]与输入 X 的对应关系如图 14-6 所示。

bit7							bit0	变量名
X7	X6	X5	X4	X3	X2	X1	X0	InDat[0]
X17	X16	X15	X14	X13	X12	X11	X10	InDat[1]

图 14-6　X 元件的内存映象图

在程序执行阶段，所有对 x 的操作，如 LD X0、AND X1 之类的指令，都是从 InDat[0]及 InDat[1]内存单元中获取的端口 1 和端口 2 的映像，而不是端口 1 和端口 2 的实时状态。

14．3．3　PLC 指令的分解

当所取的指令不是结束指令，那么这就是一条需要分析和执行的指令。其中 pCode[1]中保存的是操作码，接下来通过 switch 语句来区分不同的指令，程序如下：

```
switch （pCode[1]）
```

```
{    case 0x20：                                    //LD   S 类指令
     {    tmp=pCode[0]/8；
          bTmp=testbit（sDat[tmp]，pCode[0]%8）；
          if（bTmp）
               setbit（bitVar，pBitVar）；          //置位当前系统位变量
          else
               clrbit（bitVar，pBitVar）；          //清零当前系统位变量
          pBitVar++；
          break；
     }
//case 0x21，0x22，0x23 均是 s 的范围，因本机指持的 S 值有限，这里仅处理 0x20
     case 0x24：                                    //LD   X 类指令
     {
          tmp=pCode[0]/8；
          bTmp=testbit（InDat[tmp]，pCode[0]%8）；
          if（bTmp）
               setbit（bitVar，pBitVar）；          //置位当前系统位变量
          else
               clrbit（bitVar，pBitVar）；          //清零当前系统位变量
          pBitVar++；
          break；
     }
     case 0x25：                                    //LD    Y* 处理
     {    tmp=pCode[0]/8；
          bTmp=testbit（OutDat[tmp]，pCode[0]%8）；
          if（bTmp）
               setbit（bitVar，pBitVar）；          //置位当前系统位变量
          else
               clrbit（bitVar，pBitVar）；          //清零当前系统位变量
          pBitVar++；
          break；
     }
```

程序分析：这段程序中 setbit（）、ClrBit（）、testbit（）是分别用于置 1、清 0 和测试某位是否为 1 的宏，其定义如下：

```
#define   setbit（var，  vBit）      （（var）  |=  （1 <<  （vBit）））
#define   clrbit（var，  vBit）      （（var）  &=～（1 <<  （vBit）））
#define   testbit（var，  vBit）     （（var）  &  （1 <<（vBit）））
```

第一个参数是字节变量，是用于存储位变量的字节，而第二个参数则是指定操作的某一位，其取值范围为 0~7。例：

```
unsigned char BitVar=0；
Setbit（BitVar，1）；
```

则执行完后 BitVar 的值变为：0x02，即 0000 0010B。

了解了这三个宏的工作原理，就可以解读程序了，以 ld　　x*类指令为例：

pCode[0]中保存的是操作数，即梯形图指令中 ld　x*中的*的值，由于其值的范围为

0~255，即一共 256 位，因而最多需 32 个字节才能保存这 256 个位。这里将 pCode[0]/8 的值赋给 tmp，其含义为用来存储这个变量的位在哪一个字节中。当然，本程序是针对 DKB-1A 这块工控板设计的，这块板总共只有 12 个输入点，因此在本程序设计时只用 2 个字节，但是道理是一样的。

InDat 数组用来存放 InPut（）函数所读到的输入触点的值，testbit 宏的第一个参数是字节，其含义已在前一程序行的注释中说明，而第二个参数是这一字节的某一位。

如果要操作 x1，则 pCode[0]=1，tmp=0，而 pCode[0]%8=1，因此，该行程序相当于：

```
bTmp=testbit（InDat[0]，1）;
```

在得到了当前元件的状态后，根据 bTmp 的值来置位或者复位一个系统变量，这个变量将一直存在于流程中，因为其后的指令需要用到这一变量的状态。例如在 LD　X0 指令后，紧接着执行 AND X1 指令，那么在执行 AND X1 指令时，就需要知道 LD X0 指令执行后究竟是 0 还是 1，以便决定本条指令执行完以后究竟是 0 还是 1。但是这里不能简单地使用一个位变量作为系统变量，且看下一节的分析。

14．3．4 系统变量设计

在程序行 setbit（bitVar，pBitVar）；中，bitVar 是一个 char 型变量，其各位分别用来保存当前 LD、LDI 等指令所获取的触点状态。一个字节变量有 8 位，每 1 位都被独立地用作状态保存，因此，它一共可以保存 8 个状态。而当前状态究竟保存在哪一个位上，则取决于变量 pBitVar 的值。为何要做这样的处理呢？

如图 14-7 所示是一段 PLC 程序。图的上方是梯形图，下方是对应的指令表。

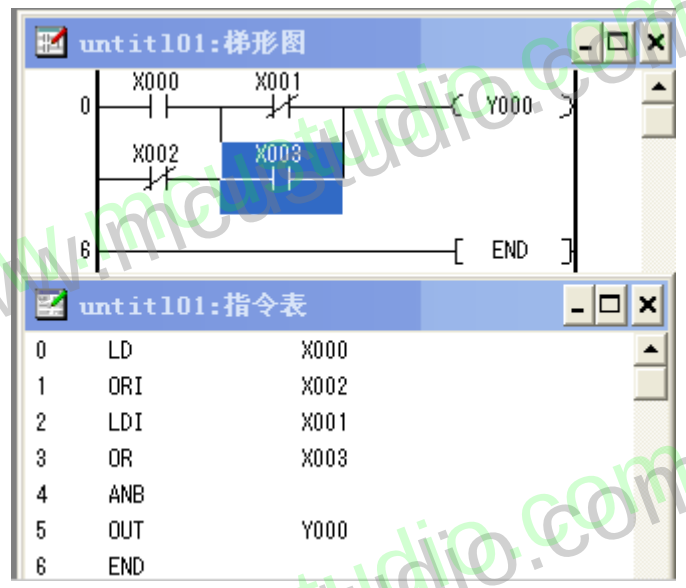

图 14-7　PLC 程序

执行这段程序时，首先取触点 X0 的状态，然后与触点 x2 的“非”状态相或，这是第一个电路块；接着读入 x1 触点的“非”状态，与触点 x3 的状态相或，这是第二个电路块；接着执行 ANB 指令，将这两个电路块的状态相与；最后将结果送到输出触点 Y0 的内存映象中。

分析程序可知，当程序的 OUT、SET 等输出指令出现之前或者在两条输出类指令之间有两条及以上 LD 或 LDI 类指令，那么必然存在着电路块形态。即要求对两个电路块的状态进行“块与”（ANB）、“块或”（ORB）之类的操作，这样，必然要求系统中保留这两个电

路块各自的状态。因此，程序中一旦执行了 ld 或者 ldi 指令以后，立即执行：

```
pBitVar++;
```

程序行，令 pBitVar 的值加 1，以便下次执行 LD 或者 LDI 等指令时，将当前状态保存于 bitVar 的下一位上。变量 bitVar 与指针 pBitVar 的示意图如图 14-8 所示，图中所示为系统初始化时，pBitVar=0 时的状态，每执行完一次 ld 类指令，图中所示指针就左移一格。而一旦当遇到 OUT、SET 等输出类指令时，指针又回到 0。

Bit 7							Bit 0	变量名
pS7	pS6	pS5	pS4	pS3	pS2	pS1	pS0	bitVar
							↑	pBitVar

图 14-8 系统变量保存

对于 and，ani，or，ori 等指令，不必要如此处理，因为它们只是将取得的元件状态与当前系统所保存的状态做逻辑操作，下面是一段处理 AND　X*的程序行：

```
case 0x44：                                    //AND　X*
{
    tmp=pCode[0]/8；
    bTmp=testbit（InDat[tmp]，pCode[0]%8）；
    bTmp&=testbit（bitVar，（pBitVar-1））；
    if（bTmp）
        setbit（bitVar，pBitVar-1）；          //置位相应的状态位
    else
        clrbit（bitVar，pBitVar-1）；          //清零相应状态位
    break；
}
```

程序分析：程序行：　bTmp=testbit（InDat[tmp]，pCode[0]%8）；中 tmp 的含义与 pCode[0]%8 的意义如 12.3.3 节程序分析中所述，这行程序用于读出指定 X 触点的状态，并赋给变量 bTmp。

读出的 bTmp 值与系统的当前状态相与，由于在执行完 ld 或者 ldi 指令后 pBitVar 加 1，所以指针总是指向下一个未用的状态位，因此，这里要将 pBitVar 的值减 1，才是指向当前所用的状态位。

如果相与的结果为 1，那么置位当前状态，否则清零当前状态位，注意程序中没有 pBitVar++这样的程序行，这是与 ld、ldi 类指令不同之处。

上面分析了 LD，AND 的几条指令，LD，LDI，AN，ANI，OR，ORI 类其他指令都与此类似，这里就不再逐一分析了，下面来看一看其他指令的分析。

14．3．5 计数器类指令

PLC 中的计数器需要用到多个参数，如计数器号，计数器设定值等，因此，每条指令不再是 2 个字节，而需要 6 个字节，下面来分析一下这类指令的处理方法。

```
case 0x0e：                    //out　c 类指令，6 字节
{    int　iTmp；
    pCode[2]=Code[i]；
    i++；
```

```
        pCode[3]=Code[i];
        i++;
        pCode[4]=Code[i];
        i++;
        pCode[5]=Code[i];
        i++;
    /*当前指令需要 6 个字节，而在程序开始时只读出了 2 个字节，因此，需要再读出剩余的 4 个字节，同时让指针 i 加 1*/
        if（(pCode[3]==0x80）&&（pCode[5]==0x80））      //确认
    /*对于 out c 类指令来说，第 4 和第 6 个字节均为 0x80*/
        {
            tmp=pCode[0]/8;
            iTmp=pCode[4];
            iTmp*=256;
            iTmp+=pCode[2];
    /*这 4 行程序用来计算计数器的计数值，计数值超过了 1 个字节所表示的范围，需要用 int 型变量 iTmp 来表示*/
            if（testbit（bitVar,（pBitVar-1）））
            {   sCount[pCode[0]]=iTmp;          /*pCode[0]中保存的是*/
                if（Count[pCode[0]]<iTmp）      //如果计数值小于设定值
                {   if（!bTmp1）                //如果一直接通，也不能算作计数
                        Count[pCode[0]]++;
                }
                else                            //已经计到预置值
                    setbit（co[pCode[0]/8]，pCode[0]%8）;    //设置输出
                bTmp1=1;
            }
            else
                bTmp1=0;
        }
        pBitVar=0;          /*执行到 Out 类指令，则将保存当前状态的状态指针回零*/
        break;
    }
```

程序分析：计数器类指令共有 6 个字节，因此，一旦判断是要处理计数器类指令，就要再读出 4 字节的代码；随后根据此类指令的第 4 和第 6 个字节均为 0x80 的特点进行确认；确认完毕，计算本条指令中计数值，这通过变量 iTmp 来实现。紧接着根据当前系统变量的值来确定是否让计数器加 1；在系统中为每个计数器都配备了一个 int 型的计数器，如果当前计数值小于指令中设定的计数值，说明计数要求尚未达到，因此执行 Count[pCode[0]]++; 这样一个程序行。

本段程序执行之前还要判断前一次扫描状态时 bTmp1 是否为 0，也就是计数输入端是否出现有断开的情况。如果没有 bTmp 为 1，说明计数端一直接通的。这并不能视作是一次有效的计数要求，而是前次计数要求的延续符合。因此，这种情况下就不要执行 Count[pCode[0]]++；这条指令了。

14．3．6 定时器类指令

接下来，继续看 OUT　T 类指令的处理方法。这类指令也是 6 个字节，第 1 个字节是定时器的元件号，第 2 个是 0x06，这是该条命令的命令码；第 4 和第 6 字节固定为 0x80，第 3 和第 5 字节是定时时间的长度，最大值超过了 255，需要用 2 个字节来表示。

```
case 0x06:                                          //out       t 类指令，6 字节
{    int iTmp;
     pCode[2]=Code[i]; i++;
     pCode[3]=Code[i]; i++;
     pCode[4]=Code[i]; i++;
     pCode[5]=Code[i]; i++;
     if ((pCode[3]==0x80) && (pCode[5]==0x80))      //确认
     {
          iTmp=pCode[4];
          iTmp*=256;
          iTmp+=pCode[2];
          tmp=pCode[0]/8;
          if (!testbit (T100msi[tmp], pCode[0]%8))
/*如果定时器已在运行，不再初始化数据*/
               sT100ms[pCode[0]]=iTmp;
          if (testbit (bitVar, (pBitVar-1)))
               setbit (T100msi[tmp], pCode[0]%8);
          else
          {    clrbit (T100msi[tmp], pCode[0]%8);
               clrbit (T100mso[tmp], pCode[0]%8);    //将输出接点也关掉
          }
          pBitVar=0;                    //执行 Out 类指令后，将 pBitVar 回 0
     }
```

程序分析：每个定时器至少需要 1 个位变量用来保存当前定时器的触点状态，1 个位变量用来保存其线包状态。这里使用 T100msi[0]和 T100msi[1]两个变量作为定时器的线包状态映象，如图 14-9 所示；使用 T100mso[0]和 T100mso[1]两个变量作为定时器触点的状态映象，如图 14-10 所示。

Bit 7							Bit 0	变量名
T7i	T6i	T5i	T4i	T3i	T2i	T1i	T0i	T100msi[0]
T15i	T14i	T13i	T12i	T11i	T10i	T9i	T8i	T100msi[1]

图 14-9 定时器线包的内存映象

Bit 7							Bit 0	变量名
T7o	T6o	T5o	T4o	T3o	T2o	T1o	T0o	T100mso [0]
T15o	T14o	T13o	T12o	T11o	T10o	T9o	T8o	T100mso [1]

图 14-10 定时器触点的内存映象

定时器除了需要线包和触点以外，还需要定时值计数器，因此，每个定时器还配有一个

int 型的变量作为计数器使用。从这里的分析可以看到，这里所说的定时器是软件定时器，它依赖于硬件定时器来实现。

要实现 PLC 中的定时器，可以有各种方案，下面分析一下本项目中所采用的方法，以下是定时中断的处理程序：

```
/*************************************************************************
函数功能：定时器 T1 中断处理程序
参      数：无
返      回：无
备      注：无
*************************************************************************/
bit    b100ms;
void Timer1（）  interrupt 3
{    static uchar c10ms，c100ms;
     uchar       i;                    //循环变量
     TH1=（65536-5000）/256;
     TL1=（65536-5000）%256;           //重置定时初值 5ms
     if（++c10ms==2）                  //10ms 到
     {   c10ms=0;
         c100ms++;
         b10ms=1;
     }
     if（c100ms==10）                  //100ms 到
     {   c100ms=0;
         b100ms=1;
     }
     if（b100ms）
     {   for（i=0；i<16；i++）
         {   if（testbit（T100msi[i/8]，i%8））
             {   if（sT100ms[i]>0）
                     sT100ms[i]--;
             }
             if（（sT100ms[i]==0）&&（testbit（T100msi[i/8]，i%8）））
/*如果定时器触点接通且此时的计数值为 0*/
                 setbit（T100mso[i/8]，i%8）;
         }
         b100ms=0;
     }
}
```

程序分析：定时器 T1 每 5ms 产生一次中断，在中断处理程序中定义了 c10ms，c100ms 两个变量，用于计数。当 c100 计数到 10 时，说明 100ms 时间到。随即使用一个循环次数为 16 的循环处理程序对 16 个定时器进行判断，通过 testbit（T100msi[i/8]，i%8）这个宏来判断相应的定时器线圈是否得电，如果得到的返回值为 1，说明线圈得电，否则说明线圈未得电，不做任何处理，处理下一个定时器。如果线圈得电，接着就判断该定时器的计数器值是否大于 0？如果大于 0，说明定时时间尚未到，将计数器的值减 1。否则，说明定时时间

已到，通过 setbit（T100mso[i/8]，i%8）；来置位相应定时器的输出触点。这样，就完成了定时器的功能。

由于本项目中的定时器数量有限，所以可以采用这种方法来实现，即在一次中断中对 16 个软件定时器统一进行处理。如果所做的系统较大，如需要用到 100 个定时器，那就要在中断处理程序中同时对这 100 个定时器进行处理，显然这是难以做到的，因此这种方法再也无法使用。为解决这个问题，可以采用其他的方法，这里就不再讨论了，有兴趣的读者可以自行思考或者在网上查找相关资料。

程序编写到这里后，一个能处理 LD、LDI、AND、OR 等若干条基本指令的程序框架搭好，如果输出程序完成，那么就构成了一个小的微型系统，能够先用起来了。所以接下来先写输出程序，而其他指令如 LDF、LDP 则准备在稍后完成。

14．3．7 输出处理

根据前面的分析，PLC 在执行程序阶段，所有的输出指令并不直接进行输出，而是将其送到一个内存映象中。由于 DKB-1A 总共只有 8 位输出，因此这里定义了一个 unsigned char 型变量 OutDat 作为 Y 元件的内存映象。Y 元件与变量 OutDat 的映象关系如图 14-11 所示。

Bit7							Bit 0	变量名
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	OutDat

图 14-11　Y 元件的内存映象图

在程序执行阶段完成后，所有输出被送入变量 OutDat 中，随后执行：

```
OutPut（OutDat[0]）;          //将数据输出
```

函数，以便将保存在输出映像中的输出数据送到输出锁存器中。函数 OutPut 的程序代码如下：

```
/*************************************************************************
函数功能：将待输出数据送到输出端口
参　　数：输出数据
返　　回：无
备　　注：不同的硬件设计，只需改变这个函数即可
*************************************************************************/
void OutPut（uchar OutData）
 {        uchar Tmp1，Tmp2；
          uchar i；
          Cntr=0；
          _nop_ （）；
          _nop_ （）；
          OutPort1=OutData；
          _nop_ （）；
          _nop_ （）；
          Cntr=1；
 }
```

这样，就完成了 PLC 工作的一个完整过程。

程序分析到这里，似乎一切都很好地实现了设计要求。于是开始将其他一些未处理的指

令逐渐完善起来。但是随着程序编写的完善，系统迅速变得庞大，仅将部分指令加入，编译后的目标代码就远大于 4K 了。这有些出乎预料之外，因为当初只是预算使用 4K 的空间来存放操作系统，因此，使用 89S52 一类的单片机就足够了。而按目前的情况来看，这是远远不够的。问题出在哪里？应该如何解决呢？如果读者在阅读前面源程序时早已心生疑惑，那么您的疑惑是对的，这是一个代码效率很低的处理方案，效率低的原因在于编程时将每一类指令中的每一条指令都进行了单独处理。

以下将对指令代码进一步分析，以便找出一种效率较高的处理方案。

14．4 较高代码效率的程序

由于 14．3 节中的程序对每一条指令都进行了单独处理，而每一类指令的操作对象大都有 S、X、Y、T、M、C 等多种，因而使得待处理的指令数据量较大，使得程序量超出了预期要求，这就要求对指令进行分析，进一步找出其共性，从而可以使用公共代码进行处理，减少程序量。以下先对指令的代码进行分析。

.
.
.
.
.
.
.
.