第一章　介绍

这是一本关于 Intel 80C51 以及广大的 51 系列单片机的书。这本书介绍给读者一些新的技术，使你的 8051 工程和开发过程变得简单。请注意，这本书的目的可不是教你各种 8051 嵌入式系统的解决方法。

为使问题讨论更加清晰，在适当的地方给出了程序代码。我们以讨论项目的方法来说明每章碰到的问题。所有的代码都可在附带的光盘上找到。你必须熟系 C 和 8051 汇编，因为本书不是一本 C 和汇编的指导书。你可以买到不少关于 ANSI C 的书，最佳选择当然是 Intel 的数据书（可从你的芯片供应商处免费索取）和随编译工具附送的手册。

附送光盘中有我为这本书编写和收集的程序，这些程序已经通过测试，这并不意味着你可以随时把这些程序加到你的应用系统或工程中，有些地方必须首先经过修改才能结合到你的程序中。

这本书将教你充分使用你的工具，如果你只有 8051 的汇编程序，你也可以学习该书和使用这些例子，但是你必须把 C 语言的程序装入你的汇编程序中，这对懂得 C 语言和 8051 汇编程序指令的人来说并不是一件困难的事。

如果你有 C 编译器的话，那恭喜你。使用 C 语言进行开发是一个好的决定，你会发现使用 C 进行开发将使你的工程开发和维护的时间大大减少。如果你已经拥有 Keil C51，那你已经选择了一个非常好的开发工具，我发现 Keil 软件包能够提供最好的支持，本书支持 Keil C 的扩展。如果你有其它的开发工具像 Archimedes 和 Avocet，这本书也能很好地为你服务，但你必须根据你所用的开发工具改变一些 Keil 的特殊指令。

在书的一些地方有硬件图，实例程序在这些硬件上运行。这些图绘制地不是很详细，主要是方框图，但足以使读者明白软件和硬件之间的接口。

读者应该把这本书看成工具书，而不是用来学习各种系统设计。通过本书，你可以了解给定一定的硬件和软件设计之后 8051 的各种性能，希望你能从本书中获取灵感，并有助于你的设计，使你豁然开朗。当然，我希望你也能够从本书中学到有用的知识，使之能够提升你的设计。

第二章　硬件

1 概述

8051 系列微处理器基于简化的嵌入式控制系统结构，被广泛应用于从军事到自动控制再到 PC 机上的键盘上的各种应用系统上。仅次于 Motorola 68HC11 在 8 位微控制器市场上的销量，很多制造商都可提供 8051 系列单片机，像 Intel，Philips，Siemens 等。这些制造商给 51 系列单片机加入了大量的性能和外部功能，像 I^2C 总线接口，模拟量到数字量的转换，看门狗，PWM 输出等，不少芯片的工作频率达到 40M，工作电压下降到 1.5V。基于一个内核的这些功能使得 8051 单片机很适合作为厂家产品的基本构架，它能够运行各种程序，而且开发者只需要学习这一个平台。

8051 系列的基本结构如下：

1．一个 8 位算术逻辑单元

2．32 个 I/O 口（4 组 8 位端口），可单独寻址

3．两个 16 位定时计数器

4．全双工串行通信

5．6 个中断源，两个中断优先级

6．128 字节内置 RAM

7．独立的 64K 字节可寻址数据和代码区

每个 8051 处理周期包括 12 个振荡周期，每 12 个振荡周期用来完成一项操作，如取指令和？。计算指令执行时间可把时钟频率除以 12，取倒数，然后指令执行所须的周期数，因此，如果你的系统时钟是 11.059MHz，除以 12 后就得到了每秒执行的指令个数，为 921583 条指令，取倒数将得到每条指令所须的时间（1.085ms）。

2 存储区结构

8051 结构提供给用户 3 个不同的存储空间，如图 A-1。每个存储空间包括从 0 到最大存储范围的连续的字节地址空间。通过利用特定地址的寻址指令，解决了地址重叠的问题。三个地址空间的功能如图所示

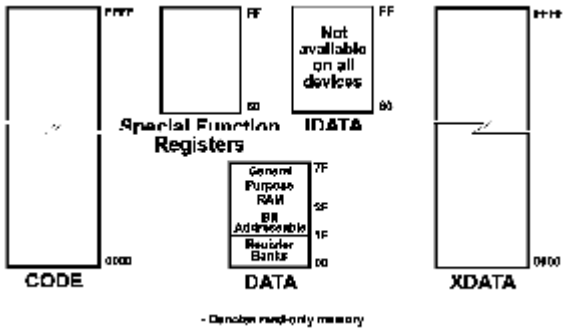

图 A-1-8051 存储结构

2.1 CODE 区

第一个存储空间是代码段，用来存放可执行代码。被 16 位寻址，空间可达 64K。代码段是只读的，当要对外接存储器件如 EPROM 进行寻址时，处理器会产生一个信号。但这并不意味着代码区一定要用一个 EPROM。目前，一般使用 EEPROM 作为外接存储器，可以被外围器件或 8051 进行改写。这使系统更新更加容易，新的软件可以下载到 EEPROM 中，而不用拆开它，然后装入一个新的 EEPROM。另外，带电池的 SRAMs 也可用来代替 EPROM，他可以像 EEPROM 一样进行程序的更新，并且没有像 EEPROM 那样读写周期的限制。但是，当电源耗尽时，存储在 SRAMs 中的程序也随之丢失。使用 SRAMs 来代替 EPROM 时，允许快速下载新程序到目标系统中，这避免了编程/调试/擦写这样一个循环过程，不再需要使用昂贵的在线仿真器。

除了可执行代码，还可在代码段中存储查寻表，为达此目的，8051 提供了通过数据指针 DPTR 或程序计数器加上由累加器提供的偏移量进行寻址的指令，这样就可以把表头地址装入 DPTR 中，把表中要寻址的元素的偏移量装入累加器中，8051 在执行指令时的过程中把这两者相加，由此可节省不少指令周期。在以后的例子中我们会看到这点。

2.2 DATA 区

第二个存储区是 8051 内 128 字节的内部 RAM，或 8052 的前 128 字节内部 RAM。这部分主要是作为数据段，称为 DATA 区。指令用一个或两个周期来访问数据段。访问 DATA 区比访问 XDATA 区要快，因为它采用直接寻址方式，而访问 XDATA 须采用间接寻址，必须先初始化 DPTR。通常我们把使用比较频繁的变量或局部变量存储在 DATA 段中，但是必须节省使用 DATA 段，因为它的空间毕竟有限。

在数据段中也可通过 R0 和 R1 采用间接寻址，R0 和 R1 被作为数据区的指针，将要恢复或改变字节的地址放入 R0 或 R1 中，根据源操作数和目的操作数的不同，执行指令需要一个或两个周期。

数据段中有两个小段，第一个子段包含四组寄存器组，每组寄存器组包含八个寄存器，共 32 个寄存器，可在任何时候通过修改 PSW 寄存器的 RS1 和 RS0 这两位来选择四组寄存器的任意一组作为工作寄存器组，8051 也可默认任意一组作为工作寄存器组。工作寄存器组的快速切换不仅使参数传递更为方便，而且可在 8051 中进行快速任务转换。

另外一个子段叫做位寻址段（BDATA），包括 16 个字节，共 128 位，每一位都可单独寻址。8051 有好几条位操作指令，这使得程序控制非常方便，并且可帮助软件代替外部组合逻辑，这样就减少了系统中的模块数。位寻址段的这 16 个字节也可像数据段中其它字节一样进行字节寻址。

2.3 特殊功能寄存器

中断系统和外部功能控制做特殊功能寄存器，简称 SFR。其中很多寄存器都可位寻址，可通过名字进行引用。如果要对中断使能寄存器中的 EA 位进行寻址，可使用 EA 或 IE.7 或 0AFH。SFRs 控制定时/计数器，串行口，中断源及中断优先级等。这些寄存器的寻址方式和 DATA 取中的其它字节和位一样。可位寻址 SFR 如表 A-1 所示。

+	0	1	2	3	4	5	6	7
F8								
F0	B							
E8								
E0	ACC							
D8								
D0	PSW							
C8	T2CON		RCAP2L	RCAP2H	TL2	TH2		
C0								
B8	IP							
B0	P3							
A8	IE							
A0	P2							
98	SCON	SBUF						
90	P1							
88	TCON	TMOD	TL0	TL1	TH0	TH1		
80	P0	SP	DPL	DPH				PCON

可进行位寻址的 SFR　　表 A-1

2.4 IDATA 区

8051 系列的一些单片机如 8052 有附加的 128 字节的内部 RAM，位于从 80H 开始的地址空间中，被称为 IDATA。因为 IDATA 区的地址和 SFRs 的地址是重叠的，通过区分所访问的存储区来解决地址重叠问题，因为 IDATA 区只能通过间接寻址来访问。

2.5 XDATA 区

8051 的最后一个存储空间为 64K，和 CODE 区一样，采用 16 位地址寻址，称作外部数据区，简称 XDATA 区。这个区通常包括一些 RAM（如 SRAM）或一些需要通过总线接口的外围器件。对 XDATA 的读写操作需要至少两个处理周期，使用 DPTR，R0 或 DPTR，R1。对 DPTR 来说，至少需要两个处理周期来装入地址，而读写又需要两个处理周期。同样， 对于 R0 或 R1 装入需要一个以上的处理周期，而读写又需两个周期，由此可见，处理 XDATA 中的数据至少要花 3 个指令周期，因此，使用频繁的数据应尽量保存在 DATA 区中。

如果不需要和外部器件进行 I/O 操作或者希望在和外部器件进行 I/O 操作时开关 RAM，则 XDATA 可全部使用 64K RAM。关于这方面的应用将在以后介绍。

3 位操作和布尔逻辑

8051 可分别对 BDATA 和 SFRs 中 128 个可寻址位，32 个 I/O 口进行位逻辑操作。可对这些位进行与，或，异或，求补，置位，清零等操作，并可像转移字节那样转移位。

列表 A-1

```
MOV     C，22H          ；把位地址 22H 中的数移入进位位中
ORL     C，23H          ；把位地址 23H 中的数和进位位中的数相或
MOV     24H，C          ；把进位位中的数移入位地址 24H 中
```

可寻址位也可作为条件转移的条件，一条很有用的指令就是 JBC，通过判断可寻址位是否置位来决定是否进行转移，如果该位置位则转移，并清零该位。这条指令能够在两个处理周期中完成，比在两个代码段中分别使用跳转和清零指令要节省一到两个处理周期。比如说，你要编写一个过程，等待 P0.0 置位，然后跳转，但是等待有时间限制。这样就需要设置一个时间，时间到达后跳出查询，检测到 P0.0 置位后跳出，并清零 P0.0。一般的逻辑流程如下。

例 A-2

```
      MOV   timeout，#TO_VALUE   ；设置查询时间
L2：  JB      P0.0，L1             ；P0.0 置位则跳转
      DJNZ    timeout，L2          ；查询时间计数
L1：  CLR     P0.0                 ；P0.0 清零
      RET                          ；退出
```

当使用 JBC 时程序如下

例 A-3

```
      MOV   timeout，#TO_VALUE   ；设置查询时间
L2：  JBC     P0.0，L1             ；P0.0 置位则跳转并清零
      DJNZ    timeout，L2          ；查询时间计数
L1：  RET                          ；退出
```

利用 JBC 不但节省了代码长度，而且使程序更加简洁美观。以后在编制代码时要习惯使用这条指令。

4 寻址方式

8051 可对存储区直接或间接寻址。这些是典型的寻址方式。直接寻址是在指令中直接包含所须寻址的字节地址，直接寻址只能在 DATA 区和 SFR 中进行。如下例：

列表 A-4

```
MOV   A，03H          ；把地址 03H 中的数移入累加器
MOV   43H，22H        ；把地址 22H 中的数移入地址 43H 中
MOV   02H，C          ；把 C 中的数移入位地址 02H 中
MOV   42H，#18         ；把立即数 18 移入地址 42H 中
MOV   09H，SBUF       ；把串行缓冲区中的数移入地址 09H 中
```

间接寻址要使用 DPTR，PC，R0，R1 寄存器，用来存放所要访问数据的地址，指令使用指针寄存器，而不是直接使用地址。用间接寻址方式可访问 CODE，IDATA，XDATA 存储区。对 DATA 存储区也可进行间接寻址。只能用直接寻址方式对位地址进行寻址。

在进行块移动时，用间接寻址十分方便，能用最少的代码完成操作。可以利用循环过程使指针递增。对 CODE 区进行寻址时，将基址存入 DPTR 或 PC 中，把变址存入累加器中，这种方法在查表时十分有用，举例如下：

例 A-5

```
DATA 和 IDATA 区寻址
MOV  R1，#22H        ；设置 R1 为指向 DATA 区内的地址 22H 的指针
MOV  R0，#0A9H       ；设置 R0 为指向 IDATA 区内的地址 0A9H 的指针
MOV  A，@R1          ；读入地址 22H 的数据
MOV  @R0，A          ；将累加器中的数据写入地址 A9H
INC  R0              ；R0 中的地址变为 AAH
INC  R1              ；R1 中的地址变为 23H
MOV  34H，@R0        ；将地址 AAH 中的数据写入 34H
MOV  @R1，#67H       ；把立即数写入地址 23H

XDATA 区寻址
MOV  DPTR，#3048H    ；DPTR 指向外部存储区
MOVX A，@DPTR        ；读入外部存储区地址 3048H 中的数
INC  DPTR            ；指针加一
MOV  A，#26H         ；立即数 26H 写入 A 中
MOVX @DPTR，A        ；将 26H 写入外部存储区地址 3049H 中
MOV  R0，#87H        ；R0 指向外部存储区地址 87H
MOVX A，@R0          ；将外部存储区地址 87H 中的数读入累加器中

代码区寻址
MOV  DPTR，#TABLE_BASE     ；DPTR 指向表首地址
MOV  A，index              ；把偏移量装入累加器中
MOVC A，@A+DPTR            ；从表中读入数据到累加器中
```

5 处理器状态

处理器的状态保存在状态寄存器 PSW 中，状态字中包括进位位，用于 BCD 码处理的辅助进位位，奇偶标志位，溢出标志位，还有前面提到的用于寄存器组选择的 RS0 和 RS1。0 组从地址 00H 开始，1 组从地址 08H 开始，2 组从地址 10H 开始，3 组从地址 18H 开始。这些地址都可通过直接或间接方式进行寻址。PSW 的结构如下：

CY	AC	F0	RS1	RS0	OV	USR	P

CY　　进位标志位
AC　　辅助进位标志位
F0　　通用标志位
RS1　　寄存器组选择位高位
RS0　　寄存器组选择位低位
OV　　溢出标志位
USR　　用户定义标志位
P　　奇偶标志位

6 电源控制

8051 的 CHMOS 版本可通过软件设置两种节电方式，空闲模式和低功耗模式。设置电源控制寄存器 PCON 的相应位来进入节电方式。置位 IDLE 进入空闲模式，空闲模式将停止程序执行，RAM 中的数据仍然保持，晶振继续工作，但与 CPU 断开。定时器和串行口继续工作。发生中断将退出中断模式。执行完中断程序后，将从程序停止的地方继续指令的执行。

通过置位 PDWN 位来进入低功耗模式，低功耗模式中晶振将停止工作，因此，定时器和串行口都将停止工作。至少有两伏的电压加在芯片上，因此，RAM 中的数据仍将保存。退出低功耗模式只有两种方式，上电或复位。

SMOD 位可控制串行通信的波特率，将使由定时器 1 的溢出率或晶振频率产生的波特率翻倍。置位 SMOD 可使工作于方式 1，2，3 定时器产生的波特率翻倍。当使用定时器 2 产生波特率时，SMOD 将不影响波特率。

电源控制寄存器（不可位寻址）

SMOD	-	-	-	GF1	GF0	PDWN	IDLE

SMOD　　串行口通信波特率控制位，置位使波特率翻倍
-　　保留
-　　保留
-　　保留
GF1　　通用标志位
GF0　　通用标志位
PDWN　　低功耗标志位，置位进入低功耗模式
IDLE　　空闲标志位，置位进入空闲模式

表　A-3

6 中断系统

基本的 8051 支持 6 个中断源，两个外部中断，两个定时/计数器中断，一个串行口输入/输出中断。中断发生后，处理器转到将五个中断入口处之一执行中断处理程序。中断向量位于代码段的最低地址出（串行口输入，输出中断共用一个中断向量）。中断服务程序必须在中断入口处或通过跳转，分支转移到别处。8051/8052 的中断向量表 A-4：

8051 支持两个中断优先级，有标准的中断机制，低优先级的中断只能被高优先级的中断所中断，而高优先级的中断不能被中断。

中断源	中断向量
上电复位	0000H
外部中断 0	0003H
定时器 0 溢出	000BH
外部中断 1	0013H
定时器 1 溢出	001BH
串行口中断	0023H
定时器 2 溢出	002BH

6.1 中断优先级寄存器

每个中断源都可通过设置中断优先级寄存器 IP 来单独设置中断优先级。如果每个中断源的相应位被置位，则该中断源的优先级为高。如果相应的位被复位，则该中断源的优先级为低。如果你觉得两个中断源不够用，别急，以后我会教你如何增加中断优先级。表 A-5 示出了 IP 寄存器的各位，此寄存器可位寻址。

IP 寄存器（可位寻址）

-	-	PT2	PS	PT1	PX1	PT0	PX0

- 保留
- 保留
PT2 定时器 2 中断优先级
PS 串行通信中断优先级
PT1 定时器 1 中断优先级
PX1 外部中断 1 优先级
PT0 定时器 0 中断优先级
PX0 外部中断 0 优先级

表 A-5

6.2 中断使能寄存器

通过设置中断使能寄存器 IE 的 EA 位，使能所有中断。每个中断源都有单独的使能位，可通过软件设置 IE 中相应的使能位在任何时候使能或禁能中断。中断使能寄存器 IE 的各位如下所示

中断使能寄存器 IE（可位寻址）

EA	-	ET2	ES	ET1	EX1	ET0	EX0

EA 使能标志位，置位则所有中断使能，复位则禁止所有中断
- 保留
ET2 定时器 2 中断使能
ES 串行通信中断使能
ET1 定时器 1 中断使能
EX1 外部中断 1 使能
ET0 定时器 0 中断使能
EX0 外部中断 0 使能

6.3 中断延迟

8051 在每个处理周期查询中断标志，确定是否有中断请求。当发生中断时，置位相应的标志，处理器将在下个周期查询到中断标志位，这样，从发生中断到确认中断之间有一个指令周期的延时。这时，处理器将用两个周期的时间来调用中断服务程序，总共要花 3 个时钟周期。在理想情况下，处理器将在 3 个指令周期内响应中断，这使得用户能很快响应系统事件。

不可避免地，系统有可能在 3 个处理周期能不能响应中断请求，特别是当有同级或更

高级的中断服务程序正在执行的时候。因此，中断的延迟主要取决于正在执行的程序。

另外一种大于 3 个周期的中断延迟是，程序正在执行一条多周期指令，要等到当前的指令执行完后，处理器才会处理中断事件，这将在原来的基础上至少增加一个周期的延时（假设在执行完多周期指令的第一个周期后发现中断）。除被其它中断所阻的情况，中断不被响应的最长延时为 6 个处理周期（3 个周期的多周期指令执行时间，3 个周期的指令响应时间 4）。

最后一种大于 3 个指令周期的中断延迟是，当检测到中断时，正在执行写 IP，IE 或 RETI 指令。

6.4 外部中断信号

8051 支持两个外部中断信号，这使外部器件能请求中断，从而得到相应的服务。外部中断由外部中断引脚（外部中断 0 为 P3.2，外部中断 1 为 P3.3）电平为低或电平由高到低跳变引起。由电平触发还是跳变触发取决于寄存器 TCON 的 ITx 位，见 A-7。

电平触发时，当检测到中断引脚电平为低时，将产生中断。低电平应至少保持一个指令周期或 12 个时钟周期。因为，处理器每个指令周期检测一次引脚。跳变触发时，当在连续的两个周期中检测到由高到低的电平跳变时，将产生中断，而电平的 0 状态应至少保持一个周期。

7 内置定时/计数器

标准的 8051 有两个定时/计数器，每个定时器有 16 位。定时/计数器既可用来作为定时器（对机器周期计数），也可用来对相应 I/O 口（T0，T1）上从高到低的跳变脉冲计数。 当用作计数器时，脉冲频率不应高于指令的执行频率的 1/2，因为每周期检测一次引脚电平，而判断一次脉冲跳变需要两个指令周期。如果需要的话，当脉冲计数溢出时，可以产生一个中断。

TCON 特殊功能寄存器（timer controller）用来控制定时器的工作起停和溢出标志位。通过改变定时器运行位 TR0 和 TR1 来启动和停止定时器的工作。TCON 中还包括了定时器 T0 和 T1 的溢出中断标志位。当定时器溢出时，相应的标志位被置位，当程序检测到标志位从 0 到 1 的跳变时，如果中断是使能的，将产生一个中断。注意，中断标志位可在任何时候置位和清除，因此，可通过软件产生和阻止定时器中断。

定时器控制寄存器（TCON 可位寻址）

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	定时器 1 溢出中断标志。响应中断后由处理器清零
TR1	定时器 1 控制位，置位时定时器 1 工作，复位时定时器 1 停止工作
TF0	定时器 0 溢出标志位。定时器 0 溢出时置位，处理器响应中断后清除该位
TR0	定时器 0 控制位，置位时定时器 0 工作，复位时定时器 0 停止工作
IE1	外部中断 1 触发标志位，当检测到 P3.3 有从高到低的跳变电平时置位，处理器响应中断后，由硬件清除该位。
IT1	中断 1 触发方式控制位，置位时为跳变触发，复位时为低电平触发
IE0	外部中断 1 触发标志位，当检测到 P3.3 有从高到低的跳变电平时置位，处理器响应中断后，由硬件清除该位。
IT0	中断 1 触发方式控制位，置位时为跳变触发，复位时为低电平触发

表 A-7

定时器的工作方式由特殊功能寄存器 TMOD 来设置。通过改变 TMOD，软件可控制两个定时器的工作方式和时钟源（是 I/O 口的触发电平还是处理器的时钟脉冲）。TMOD 的高四

位控制定时器 1，低四位控制定时器 0。TMOD 的结构如下

定时器控制寄存器 TMOD-不可位寻址

GATE	C/T	M1	M0	GATE	C/T	M1	M0
定时器 1				定时器 0			

GATE　　当 GATE 置位时，定时器仅当 TR=1 并且 INT=1 时才工作，如果 GATE=0，置位 TR 定时器就开始工作

C/T　　定时器方式选择。如果 C/T=1，定时器以计数方式工作，C/T=0 时，以定时方式工作

M1　　模式选择位高位

M0　　模式选择位低位

表 A-8

可通过 C/T 位的设置来选择定时器的时钟源。C/T=1，定时器以计数方式工作（对 I/O 引脚脉冲计数），C/T=0 时，以定时方式工作（对内部时钟脉冲计数）。当定时器用来对内部时钟脉冲计数时，可通过硬件或软件来控制。GATE=0 为软件控制，置位 TR 定时器就开始工作，GATE=1 为硬件控制，当 TR=1 并且 INT=1 时定时器才工作。当 INT 脚给出低电平时，定时器将停止工作。这在测量 INT 脚的脉冲宽度时十分有用，当然，INT 脚不作为外部中断使用。

7.1 定时器工作方式 0 和方式 1

定时器通过软件控制有四种工作方式。方式 0 为十三位定时/计数器方式，定时器溢出时置位 TF0 或 TF1，并产生中断。方式 1 将以十六位定时/计数器方式工作，除此之外和方式 0 一样。

7.2 定时器工作方式 2

方式 2 为 8 位自动重装工作方式。定时器的低 8 位（TL0 或 TL1）用来计数，高 8 位（TH0 或 TH1）用来存放重装数值。当定时器溢出时，TH 中的数值被装入 TL 中。定时器 0 和定时器 1 在方式 2 时是同样的。定时器 1 常用此方式来产生波特率。

7.3 定时器工作方式 3

方式 3 时，定时器 0 成为两个 8 位定时/计数器（TH0 和 TL0）。TH0 对应于 TMOD 中定时器 0 的控制位，而 TL0 占据了 TMOD 中定时器 1 的控制位。这样定时器 1 将不能产生溢出中断了，但可用于其它不需产生中断的场合，如作为波特率发生器或作为定时计数器被软件查询。当系统需要用定时器 1 来产生波特率，而又同时需要两个定时/计数器时，这种工作方式十分有用。当定时器 1 设置为工作方式 3 时，将停止工作。

7.4 定时器 2

51 系列单片机如 8052 第三个定时/计数器，定时器 2。他的控制位在特殊功能寄存器 T2CON 中。结构如下：

定时器 2 控制寄存器（可位寻址）

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

TF2　　定时器 2 溢出标志位。定时器 2 溢出时将置位，当 TCLK 或 RCLK 为 1 时，将不会置位。

EXF2　　定时器 2 外部标志，当 EXEN2=1，并在引脚 T2EX 检测到负跳变时置位。如果定时器 2 中断被允许，将产生中断。

RCLK　　接收时钟标志，当串行口以方式 1 或 3 工作时，将使用定时器 2 的溢出率作为串行口接收时钟频率。

TCLK　　发送时钟标志位，当串行口以方式 1 或 3 工作时，将使用定时器 2 的溢出率作为串行口接收时钟频率。

EXEN2　　定时器 2 外部允许标志，当 EXEN2=1 时，在 T2EX 引脚出现负跳变时将造成定时器 2 捕捉或重装，并置位 EXF2，产生中断。

TR2　　定时器运行控制位，置位时，定时器 2 将开始工作，否则，定时器 2 停止工作。

C/T2　　定时器计数方式选择位，如果 C/T2=1，定时器 2 将作为外部事件计数器，否则对内部时钟脉冲计数。

CP/RL2　　捕捉/重装标志位，当 EXEN2=1 时，如果 CP/RL2=1，T2EX 引脚的负跳变将造成捕捉，如果 CP/RL2=0，T2EX 引脚的负跳变将造成重装。

通过由软件设置 T2CON，可使定时/计数器以三种基本工作方式之一工作。第一种为捕捉方式。设置为捕捉方式时，和定时器 0 或定时器 1 一样以 16 位方式工作。这种方式通过复位 EXEN2 来选择。当置位 EXEN2 时，如果 T2EX 有负跳变电平，将把当前的数，锁存在（RCAP2H 和 RCAP2L）中。这个事件可用来产生中断。

第二种工作方式为自动重装方式，其中包含了两个子功能，由 EXEN2 来选择。当 EXEN2 复位时，16 位定时器溢出将触发一个中断并将 RCAP2H 和 RCAP2L 中的数装入定时器中。当 EXEN2 置位时，除上述功能外，T2EX 引脚的负跳变将产生一次重装操作。

最后一种方式用来产生串行口通讯所需的波特率，这通过同时或分别置位 RCLK 和 TCLK 来实现。在这种方式中，每个机器周期都将使定时器加 1，而不像定时器 0 和 1 那样，需要 12 个机器周期。这使得串行通讯的波特率更高。

8 内置 UART

8051 有一个可通过软件控制的内置，全双工串行通讯接口。由寄存器 SCON 来进行设置，可选择通讯模式，允许接收，检查状态位。SCON 的结构如下：

串行控制寄存器（SCON）-可位寻址

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0　　串行模式选择

SM1　　串行模式选择

SM2　　多机通讯允许位，当模式 0 时，此位应该为 0。模式 1 时，当接收到停止位时，该位将置位。模式 2 或模式 3 时，当接收的第 9 位数据为 1 时，将置位。

REN　　串行接收允许位

TB8　　在模式 2 和模式 3 中，将被发送数据的第 9 位

RB8　　在模式 0 中，该位不起作用，在模式 1 中，该位为接收数据的停止位。在模式 2 和模式 3 中，为接收数据的第 9 位

TI　　串行中断标志位，由软件清零

RI　　接收中断标志位，有软件清零

表 A-10

UART 有一个接收数据缓冲区，当上一个字节还没被处理，下一个数据仍然可以缓冲区接收进来，但如果接收完这个字节如果上个字节还没被处理，上个字节将被覆盖。因此，软件必须在此之前处理数据。当连续发送字节时也是如此。

8051 支持 10 位和 11 位数据模式，11 数据模式用来进行多机通讯。并支持高速 8 位移位寄存器模式。模式 1 和模式 3 中波特率可变。

8.1 UART 模式 0

模式 0 时，UART 作为一个 8 位的移位寄存器使用，波特率为 fosc/12。数据由 RXD 从低位开始收发。TXD 用来发送同步移位脉冲，因此，方式 0 不支持全双工。这种方式可用来和像某些具有 8 位串行口的 EEPROM 之类的器件通讯。

当向 SBUF 写入字节时，开始发送数据。数据发送完毕时，TI 位将置位。置位 REN 时，将开始接收数据，接收完 8 位数据时，RI 位将置位。

8.2 UART 模式 1

工作于模式 1 时，传输的是 10 位：1 个起始位，8 个数据位，1 个停止位。这种方式可和包括 PC 机在内的很多器件进行通讯。这种方式中波特率是可调的。而用来产生波特率的定时器的中断应该被禁止。PCON 的 SMOD 位为 1 时，可使波特率翻倍。

TI 和 RI 在发送和接收停止位的中间时刻被置位。这使软件可以响应中断并装入新的数据。数据处理时间取决于波特率和晶振频率。

如果用定时器 1 来产生波特率，应通过下式来计算 TH1 的装入值

TH1=256-（K*OscFreq）/（384*BaudRate）

K=1 if SMOD=0

K=2 if SMOD=1

重装值要小于 256，非整数的重装值必须和下一个整数非常接近。通常产生的波特率都能使系统正常的工作，这点需要开发者把握。

这样,如果你使用 9.216M 晶振,想产生 9600 的波特率,第一步,设 K=1,分子为 9216000，分母为 3686400，相除结果为 2.5，不是整数。设 K=2，分子为 18432000，分母为 3686400，相除结果为 5，可得 TH1=251 或 0FBH。

如果用 8052 的定时器 2 产生波特率，RCAP2H 和 RCAP2L 的重装值也需要经过计算，根据需要的波特率，用下式计算

[RCAP2H，RCAP2L]=65536-OsFreq/（32*BaudRate）

假设你的系统使用 9.216M 晶振，你想产生 9600 的波特率。用上式产生的结果必须是正的，而且接近整数。最后得到结果 30，重装值为 65506 或 FFE2H。

8.3 UART 模式 2

模式 2 的数据以 11 位方式发送：1 位起始位，8 位数据位，第九位，1 位停止位。发送数据时，第九位为 SCON 中的 TB8，接收数据的第九位保存在 RB8 中。第九位一般用来多机通信，仅在第九位为 1 时，单片机才接收数据。多机通信用 SCON 的 SM2 来控制。当 SM2 置位时，仅当数据的第九位为 1 时才引发通讯中断，当 SM2 为 0 时，只要接收完 11 位就产生一次中断。

第九位可在多机通讯中避免不必要的中断，在传送地址和命令时，第九位置位，串行总线上的所有处理器都产生一个中断，处理器将决定是否继续接收下面的数据，如果继续接收数据就清零 SM2。否则，SM2 置位，以后的数据流将不会使他产生中断。

SMOD=0 时，模式 2 的波特率为 1/64Osc，SMOD=1 时，波特率为 1/32Osc。因此，使用模式 2，当晶振频率为 11.059M 时，将有高达 345K 的波特率。模式 3 和模式 2 的差别在于可变的波特率。

9 其它功能

很多 51 系列的单片机有了许多新增加的功能，使之更适合于嵌入式应用。51 系列的其它功能如下。

9.1 I²C

I²C 是一种新的芯片间的通讯方式，由 PHILIPS 开发和推广。I²C 通讯采用两条线进行通讯，一条数据线，一条时钟线，可进行多器件通讯。总线上的每个器件都有自己的地址。数据传送是双向的，总线支持多主机。8051 上 I²C 总线的接口为 P0 端口的两根线，有专门的特殊功能寄存器来控制总线的工作和执行传输协议。

9.2 A/D 转换

并不是所有 51 系列单片机都带 A/D 转换，但 A/D 转换的使用非常普遍。A/D 转换一般由寄存器 ADCON 来控制。用户通过 ADCON 来选择 A/D 转换的通道，开始转换，检查转换状态。一般 A/D 转换的过程不多于 40 个指令周期，转换完成后产生中断，中断程序将处理转换结果。A/D 转换需要处理器一直处于工作状态。转换结果保存于特殊功能寄存器中。

9.3 看门狗

大多数 51 系列单片机都有看门狗。当看门狗没有被定时清零时，将引起复位。这可防止程序跑飞。设计者必须清楚看门狗的溢出时间，以决定在合适的时候清看门狗。清看门狗也不能太过频繁，否则会造成资源浪费。

51 系列有专门的看门狗定时器，对系统频率进行分频计数，定时器溢出时，将引起复位。看门狗可设定溢出率，也可单独用来作为定时器使用。

10 设计

51 系列单片机有着各种具有不同的外设功能的成员，可适用于各方面的应用。选择一款合适的单片机是十分重要的。考虑到电路板空间和成本，应使外围部件尽可能少。51 系列最多 512 字节的 RAM 和 32K 字节的 EPROM。有时，只要使用系统内置的 RAM 和 EPROM 就可以了，应充分利用这些部件，不再需要外接 EPROM 和 RAM，这样就省下了 I/O 口，可用来和其它器件相连。当不需要扩展 I/O 口并且程序代码较短时，使用 28 脚的 51 单片机可节省不少空间。但很多应用需要更多的 RAM 和 EPROM 空间，这时就要用外围器件(SRAM,EPROM 等)。许多外围器件能被 51 系列的内部功能和相应的软件代替，这将在以后讨论。

经常要考虑系统的功耗问题。如果处理器有很多工作要做，而不能进入低功耗和空闲模式，应选择 3.6V 的工作电压以降低功耗，如果有足够的空闲时间的话，可以考虑关闭晶振，降低功耗。

设计者必须仔细选择晶振频率，确保标准的通讯波特率（1200，4800，9600，19.2K 等)。你不妨先列出可供选择的晶振所能产生的波特率，然后根据需要的波特率和系统要求选择晶振。有时也不必过分考虑晶振问题，因为可以定制晶振。当晶振频率超过 20M 时，必须确保总线上的其它器件能够在这种频率下工作。一般，EPROM，SRAM，高速 CMOS 版的锁存器都支持 51 的工作频率。当工作频率增加时，功耗也会增加，这点在使用电池作为电源的系统中应充分考虑。

11 实现

当选择好单片机和外围器件后，下一步就是设计和分配系统 I/O 地址。代码段在从地址零开始的连续空间内。外部数据存储空间地址一般和 RAM 和器件地址相连。RAM 一般在从地址 0000H 或 8000H 开始的连续空间内。一种比较有用的处理方法是 SRAM 的地址也从 0000H 开始，用 A15 使能 RAM，RAM 的 OE 和 WE 线分别和单片机的 RD 和 WR 线相连。这种方法可使 RAM 区超过 32K，这足够嵌入式系统使用。此外，32K 的地址也可分配给 I/O 器件。大多数情况下，I/O 器件是比较少的，所以，地址线的高位可接解码器工作给外围器件提

供使能信号。一个为系统 I/O 分配地址的例子如 A-2-8051 总线 I/O 所示。可以看到，通过减少地址解码器的数量简化了硬件设计。因为在 I/O 操作中不用装载 DPTR 的低 8 位，使软件设计也得到简化。

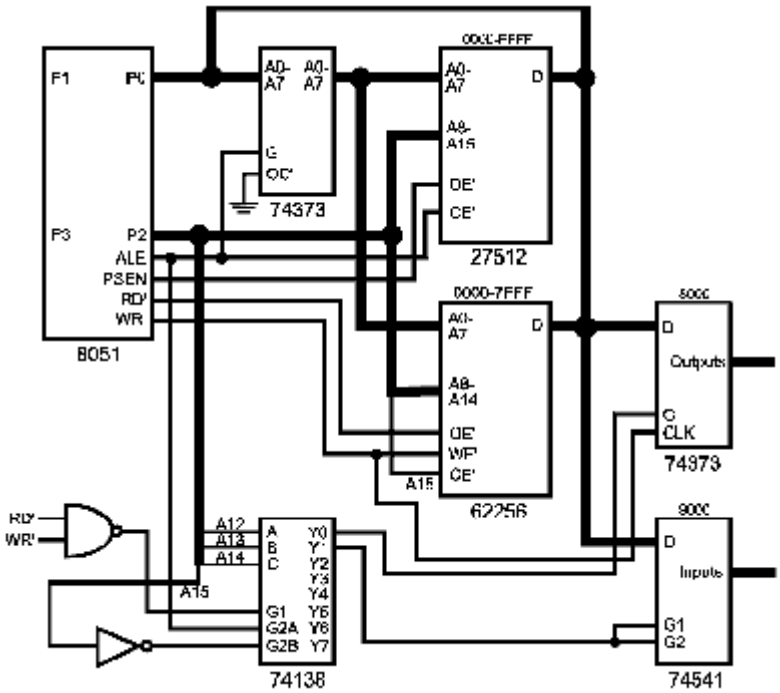

图 A-2-8051 总线 I/O

对输入输出锁存器的寻址如下例

列表 A-6

```
MOV    DPTR，#09000H      ；设置指针
MOVX   A，@DPTR
MOV    DPH，#080H
MOVX   @DPTR，A
```

可以看到，因为电路设计，连续的 I/O 操作将被简化，软件不需要考虑数据指针的低字节。第一条指令也可用 MOV DPH，#090H 代替。

12 结论

我希望上面所讲的关于 8051 的基本知识能给你一些启发。但这不能代替 8051 厂商提供的数据书，因为每款芯片都有其自身的特点。下面，我们将开始讨论 8051 的软件设计，包括用 C 进行软件开发。

第二章　用 C 对 8051 编程

1 为什么要用高级语言？

当设计一个小的嵌入式系统时，一般我们都用汇编语言。在很多工程中，这是一个很好的方法，因为，代码一般都不超过 8K，而且都比较简单。如果硬件工程师要同时设计软件和硬件，经常会采用汇编语言来做程序。我的经验告述我，硬件工程师一般不熟系像 C 一类的高级语言。

使用汇编的麻烦在于它的可读性和可维护性，特别当程序没有很好的标注的时候。代码的可重用性也比较低。如果使用 C 的话，可以很好的解决这些问题。

用 C 编写的程序，因为 C 语言很好的结构性和模块化，更容易阅读和维护，而且由于模块化，用 C 语言编写的程序有很好的可移植性。功能化的代码能够很方便的从一个工程移植到另一个工程，从而减少了开发时间。

用 C 编写程序比汇编更符合人们的思考习惯。开发者可以更专心的考虑算法而不是考虑一些细节问题。这样就减少了开发和调试的时间。

使用像 C 这样的语言，程序员不必十分熟系处理器的运算过程。这意味着对新的处理器也能很快上手，不必知道处理器的具体内部结构，使得用 C 编写的程序比汇编程序有更好的可移植性。很多处理器支持 C 编译器。

所有这些并不说明汇编语言就没了立足之地，很多系统，特别是实时时钟系统都是用 C 和汇编语言联合编程。对时钟要求很严格时，使用汇编语言成了唯一的方法，除此之外，根据我的经验，包括硬件接口的操作都应该用 C 来编程。C 的特点就是，可以使你尽量少地对硬件进行操作，是一种功能性和结构性很强的语言。

2 C 语言的一些要点

这里不是教你如何使用 C 语言。关于 C 语言的书有很多，像 Kernighan 和 Ritchie 所著的 C 编程语言等，这本书被认为是 C 语言的权威著作。Keil 的 C51 完全支持 C 的标准指令和很多用来优化 8051 指令结构的 C 的扩展指令。

我们将复习关于 C 的一些概念，如结构，联合和类型定义。可能会使一些人伤脑筋。

2.1 结构

结构是一种定义类型，它允许程序员把一系列变量集中到一个单元中。当某些变量相关的时候使用这种类型是很方便的。例如，你用一系列变量来描述一天的时间，你需要定义时，分，秒三个变量

```
unsighed char hour,min,sec;
```

还要定义一个天的变量

```
unsighed int  days;
```

通过使用结构，你可以把这四个变量定义在一起，给他们一个共同的名字。声明结构的语法如下

```
struct time_str{
  unsigned char hour,min,sec;
  unsigned int days;
}time_of_day;
```

这告述编译器定义一个类型名为 time_str 的结构，并定义一个名为 time_of_day 的结构变量。变量成员的引用为结构 变量名.结构成员

```
time_of_day.hour=XBYTE[HOURS];
time_of_day.days=XBYTE[DAYS];
```

```
time_of_day.min=time_of_day.sec
curdays=time_of_day.days;
```

成员变量和其它变量是一样的，但前面必须有结构名。你可以定义很多结构变量，编译器把他们看成新的变量。例如

```
struct time_str oldtime,newtime;
```

这样就产生了两个新的结构变量，这些变量都是相互独立的，就像定义了很多 int 类型的变量一样。结构变量可以很容易的复制：

```
oldtime=time_of_day;
```

这使代码很容易阅读，也减少了打字的工作量。当然，你也可以一句一句的复制：

```
oldtime.hour=newtime.hour;
oldtime.days=newtime.days-1;
```

在 Keil C 和大多数 C 编译器中，结构被提供了连续的存储空间，成员名被用来对结构内部进行寻址。这样，结构 time_str 被提供了连续 5 个字节的空间。空间内的变量顺序和定义时的变量顺序一样。如表 0-1:

Offset	Member	Bytes
0	hour	1
1	min	1
2	sec	1
3	days	2

表 0-1

如果你定义了一个结构类型，它就像一个变量新的变量类型。你可建立一个结构数组，包含结构的结构，和指向结构的指针。

2.2 联合

联合和结构很相似，它由相关的变量组成，这些变量构成了联合的成员。但是这些成员只能有一个起作用。联合的成员变量可以是任何有效类型，包括 C 语言本身拥有的类型和用户定义的类型，如结构和联合。一个定义联合的类型如下：

```
union time_type {
  unsigned long secs_in_year;
  struct time_str time;
}mytime;
```

用一个长整形来存放从这年开始到现在的秒数，另一个可选项是用 time_str 结构来存储从这年开始到现在的时间。

不管联合包含什么，可在任何时候引用他的成员，如下例：

```
mytime.secs_in_year=JUNEIST;
mytime.time.hour=5;
curdays=mytime.time.days;
```

像结构一样，联合也以连续的空间存储，空间大小等于联合中最大的成员所需的空间

Offset	Member	Bytes
0	Secs_in_year	4
0	Mytime	5

表 0-2

因为最大的成员需要 5 个字节，联合的存储大小为 5 个字节。当联合的成员为 secs_in_year 时，第 5 个字节没有使用。

联合经常被用来提供同一个数据的不同的表达方式。例如，假设你有一个长整型变量用来存放四个寄存器的值。如果希望对这些数据有两种表达方法，可以在联合中定义一个长整型变量，同时再定义一个字节数组，如下例：

```
union status_type{
   unsigned char status[4];
```

```
    unsigned long status_val;
}io_status;
io_status.status_val=0x12345678;
if(i0_status.status[2]&0x10){
  ...
}
```

2.3 指针

指针是一个包含存储区地址的变量。因为指针中包含了变量的地址，它可以对它所指向的变量进行寻址，就像在 8051 DATA 区中进行寄存器间接寻址和在 XDATA 区中用 DPTR 进行寻址一样。使用指针是非常方便的，因为它很容易从一个变量移到下一个变量，所以可以写出对大量变量进行操作的通用程序。

指针要定义类型，说明指向何种类型的变量。假设你用关键字 long 定义一个指针，C 就把指针所指的地址看成一个长整型变量的基址。这并不说明这个指针被强迫指向长整型的变量，而是说明 C 把该指针所指的变量看成长整型的。下面是一些指针定义的例子：

```
unsigned char *my_ptr,*anther_ptr;
unsigned int *int_ptr;
float *float_ptr;
time_str *time_ptr;
```

指针可被赋予任何已经定义的变量或存储器的地址。

```
My_ptr=&char_val;
Int_ptr=&int_array[10];
Time_str=&oldtime;
```

可通过加减来移动指针，指向不同的存储区地址。在处理数组的时候，这一点特别有用。当指针加 1 的时候，它加上指针所指数据类型的长度。

```
Time_ptr=(time str *) (0x10000L);      //指向地址 0
Time_ptr++;                            //指向地址 5
```

指针间可像其它变量那样互相赋值。指针所指向的数据也可通过引用指针来赋值

```
time_ptr=oldtime_ptr          //两个指针指向同一地址
*int_ptr=0x4500               //把 0X4500 赋给 int_ptr 所指的变量
```

当用指针来引用结构或联合的成员时，可用如下方法：

```
time_ptr->days=234;
*time_ptr.hour=12;
```

还有一个指针用得比较多的场合是链表和树结构。假设你想产生一个数据结构，可以进行插入和查询操作，一种最简单的方法就是建立一个双向查询树，你可以像下面那样定义树的节点：

```
struct bst_node{
    unsigned char name[20];          //存储姓名
    struct bst_node *left, right;    //分别指向左，右子树的指针
};
```

可通过定位新的变量，并把他的地址赋给查询树的左指针或右指针来使双向查询树变长或缩短。有了指针后，对树的处理变得简单。

2.4 类型定义

在 C 中进行类型定义就是对给定的类型一个新的类型名。换句话说就是给类型一个新的名字。例如，你想给结构 time_str 一个新的名字

```
typedef struct time_str{
  unsigned char hour,min,sec;
  unsigned int days;
}time_type;
```

这样，就可以像使用其它变量那样使用 time_type 的类型变量

```
time_type time,*time_ptr,time_array[10];
```

类型定义也可用来重新命名 C 的标准类型

```
typedef unsigned char UBYTE;
typedef char *strptr;
strptr name;
```

使用类型定义可使你的代码的可读性加强，节省了一些打字的时间。但是很多程序员大量的使用类型定义，别人再看你的程序时就十分困难了。

3 Keil C 和 ANSI C

下面将介绍 Keil C 的主要特点和它与 ANSI C 的不同之处，并给你一些对 8051 使用 C 的启发。

Keil 编译器除了少数一些关键地方外，基本类似于 ANSI C。差异主要是 Keil 可以让户针对 8051 的结构进行程序设计，其它差异主要是 8051 的一些局限引起的。

3.1 数据类型

Keil C 有 ANSI C 的所有标准数据类型，除此之外，为了更加有利的利用 8051 的结构，还加入了一些特殊的数据类型。下表显示了标准数据类型在 8051 中占据的字节数。注意，整型和长整型的符号位字节在最低的地址中。

除了这些标准数据类型外，编译器还支持一种位数据类型。一个位变量存在于内部 RAM 的可位寻址区中。可像操作其它变量那样对位变量进行操作，而位数组和位指针是违法的。

数据类型	大小
char/unsigned char	8 bit
int/unsigned char	16 bit
long/unsigned long	32 bit
float/double	32 bit
generic pointer	24 bit

表 0-3

3.2 特殊功能寄存器

特殊功能寄存器用 sfr 来定义，而 sfr16 用来定义 16 位的特殊功能寄存器如 DPTR。通过名字或地址来引用特殊功能寄存器。地址必须高于 80H。可位寻址的特殊功能寄存器的位变量定义用关键字 sbit。SFR 的定义如列表 0-1 所示。对于大多数 8051 成员，Keil 提供了一个包含了所有特殊功能寄存器和他们的位的定义的头文件。通过包含头文件可以很容易的进行新的扩展。

列表 0-1

```
sfr SCON=0X98;          //定义 SCON
sbit SM0=0X9F;          //定义 SCON 的各位
sbit SM1=0X9E;
sbit SM2=0X9D;
sbit REN=0x9C;
sbit TB8=0X9B;
```

```
sbit RB8=0X9A;
sbit TI=0X99;
sbit RI=0X98;
```

4 存储类型

Keil 允许使用者指定程序变量的存储区。这使使用者可以控制存储区的使用。编译器可识别以下存储区

存储区	描述
DATA	RAM 的低 128 个字节，可在一个周期内直接寻址
BDATA	DATA 区的 16 个字节的可位寻址区
IDATA	RAM 区的高 128 个字节，必须采用间接寻址
PDATA	外部存储区的 256 个字节，通过 P0 口的地址对其寻址，使用指令 MOVX @Rn,需要两个指令周期。
XDATA	外部存储区，使用 DPTR 寻址
CODE	程序存储区，使用 DPTR 寻址

4.1 DATA 区

对 DATA 区的寻址是最快的，所以应该把使用频率高的变量放在 DATA 区。由于空间有限，必须注意使用。DATA 区除了包含程序变量外，还包含了堆栈和寄存器组，DATA 区的声明如列表 0-2：

列表 0-2

```
unsigned char data system_status=0;
unsigned int data unit_id[2];
char data inp_string[16];
float data outp_value;
mytype data new_var;
```

标准变量和用户自定义变量都可存储在 DATA 区中，只要不超过 DATA 区的范围。因为 C51 使用默认的寄存器组来传递参数，你至少失去了 8 个字节。另外，要定义足够大的堆栈空间。当你的内部堆栈溢出的时候，你的程序会莫名其妙的复位。实际原因是 8051 系列微处理器没有硬件报错机制，堆栈溢出只能以这种方式表示出来。

4.2 BDATA 区

你可以在 DATA 区的位寻址区定义变量，这个变量就可进行位寻址，并且声明位变量。这对状态寄存器来说是十分有用的，因为它需要单独的使用变量的每一位。不一定要用位变量名来引用位变量。下面是一些在 BDATA 段中声明变量和使用位变量的例子。

列表 0-3

```
unsigned char bdata status_byte;
unsigned int bdata status_word;
unsigned long bdata status_dword;
sbit stat_flag=status_byte^4;
if(status_word^15){
...
}
stat_flag=1;
```

编译器不允许在 BDATA 段中定义 float 和 double 类型的变量。如果你想对浮点数的每

位寻址，可以通过包含 float 和 long 的联合来实现。

列表 0-4

```
typedef union{                  //定义联合类型
    unsigned long lvalue;       //长整型 32 位
    float fvalue;               //浮点数 32 位
}bit_float;                     //联合名
bit_float bdata myfloat;        //在 BDATA 段中声名联合
sbit float_ld=myfloat^31        //定义位变量名
```

下面的代码访问状态寄存器的特定位。把访问定义在 DATA 段中的一个字节和通过位名和位号访问同样的可位寻址字节的位的代码对比。注意，对变量位进行寻址产生的汇编代码比检测定义在 DATA 段的状态字节位所产生的汇编代码要好。如果你对定义在 BDATA 段中的状态字节中的位采用偏移量进行寻址，而不是用先前定义的位变量名时，编译后的代码是错误的。下面的例子中 use_bitnum_status 的汇编代码比 use_byte_status 的代码要大。

列表 0-5

```
1       //定义一个字节宽状态寄存器
2       unsigned char data byte_status=0x43;
3
4       //定义一个可位寻址状态寄存器
5       unsigned char bdata bit_status=0x43;
6       //把 bit_status 的第 3 位设为位变量
7       sbit status_3=bit_status^3;
8
9       bit use_bit_status(void);
10
11      bit use_bitnum_status(void);
12
13      bit use_byte_status(void);
14
15      void main(void){
16        unsigned  char temp=0;
17        if (use_bit_status()){      //如果第 3 位置位 temp 加 1
18          temp++;
19         }
20        if (use_byte_status()){     //如果第 3 位置位 temp 再加 1
21           temp++;
22         }
23        if (use_bitnum_status()){    //如果第 3 位置位 temp 再加 1
24           temp++;
25         }
26      }
27
28      bit use_bit_status(void){
29        return(bit)(status_3);
```

```
30      }
31
32      bit use_bitnum_status(void){
33        return(bit)(bit_status^3);
34      }
35
36      bit use_byte_status(void){
37        return byte _status&0x04;
38      }
```

目标代码列表

```
         ; FUNCTION main (BEGIN)
                                          ; SOURCE LINE # 15
                                          ; SOURCE LINE # 16
0000 E4               CLR    A
0001 F500   R         MOV    temp,A
                                          ; SOURCE LINE # 17
0003 120000 R         LCALL  use_bit_status
0006 5002             JNC    ?C0001
                                          ; SOURCE LINE # 18
0008 0500   R         INC    temp
                                          ; SOURCE LINE # 19
000A        ?C0001:
                                          ; SOURCE LINE # 20
000A 120000 R         LCALL  use_byte_status
000D 5002             JNC    ?C0002
                                          ; SOURCE LINE # 21
000F 0500   R         INC    temp
                                          ; SOURCE LINE # 22
0011        ?C0002:
                                          ; SOURCE LINE # 23
0011 120000 R         LCALL  use_bitnum_status
0014 5002             JNC    ?C0004
                                          ; SOURCE LINE # 24
0016 0500   R         INC    temp
                                          ; SOURCE LINE # 25
                                          ; SOURCE LINE # 26
0018        ?C0004:
0018 22               RET
          ; FUNCTION main (END)
          ; FUNCTION use_bit_status (BEGIN)
                                          ; SOURCE LINE # 28
                                          ; SOURCE LINE # 29
0000 A200   R         MOV C,status_3
                                          ; SOURCE LINE # 30
```

```
0002        ?C0005:
0002 22                 RET
                    ; FUNCTION use_bit_status (END)
                    ; FUNCTION use_bitnum_status (BEGIN)
```

The compiler obtains the desired bit by using the entire byte instead of using a bit address.

```
                                            ; SOURCE LINE # 32
                                            ; SOURCE LINE # 33
0000 E500  R            MOV   A,bit_status
0002 6403               XRL   A,#03H
0004 24FF               ADD   A,#0FFH
                                            ; SOURCE LINE # 34
0006        ?C0006:
0006 22                 RET
                    ; FUNCTION use_bitnum_status (END)
                    ; FUNCTION use_byte_status (BEGIN)
                                            ; SOURCE LINE # 36
                                            ; SOURCE LINE # 37
0000 E500  R            MOV   A,byte_status
0002 A2E2               MOV   C,ACC.2
                                            ; SOURCE LINE # 38
0004        ?C0007:
0004 22                 RET
                    ; FUNCTION use_byte_status (END)
```

记住在处理位变量时，要使用声明的位变量名，而不要使用偏移量。

4.3 IDATA 段

IDATA 段也可存放使用比较频繁的变量，使用寄存器作为指针进行寻址。在寄存器中设置 8 位地址，进行间接寻址。和外部存储器寻址比较，它的指令执行周期和代码长度都比较短。

```
unsigned char idata system_status=0;
unsigned int idata unit_id[2];
char idata inp_string[16];
float idata outp_value;
```

4.4 PDATA 和 XDATA 段

在这两个段声明变量和在其它段的语法是一样的。PDATA 段只有 256 个字节，而 XDATA 段可达 65536 个字节。下面是一些例子。

```
unsigned char xdata system_status=0;
unsigned int pdata unit_id[2];
char xdata inp_string[16];
float pdata outp_value;
```

对 PDATA 和 XDATA 的操作是相似的。对 PDATA 段寻址比对 XDATA 段寻址要快，因为，对 PDATA 段寻址只需要装入 8 位地址，而对 XDATA 段寻址需装入 16 位地址。所以尽量把外

部数据存储在 PDATA 段中。对 PDATA 和 XDATA 寻址要使用 MOVX 指令，需要两个处理周期。
列表 0-6

```
1    #include <reg51.h>
2
3    unisgned char pdata inp_reg1;
4
5    unsigned char xdata inp_reg2;
6
7    void main(void){
8       inp_reg1=P1;
9       inp_reg2=P3;
10   }
```

产生的目标代码列表

```
             ; FUNCTION main (BEGIN)
                                        ; SOURCE LINE # 7
                                        ; SOURCE LINE # 8

注意 'inp_reg1=P1' 需要4个指令周期

0000 7800    R      MOV       R0,#inp_reg1
0002 E590           MOV       A,P1
0004 F2             MOVX      @R0,A
                                        ; SOURCE LINE # 9

注意 'inp_reg2=P3' 需要5个指令周期
0005 900000  R      MOV       DPTR,#inp_reg2
0008 E5B0           MOV       A,P3
000A F0             MOVX      @DPTR,A
                                        ; SOURCE LINE # 10
000B 22             RET
             ; FUNCTION main (END)
```

经常，外部地址段中除了包含存储器地址外还包含 I/O 器件的地址。对外部器件寻址可通过指针或 C51 提供的宏。我建议使用宏对外部器件进行寻址，因为这样更有可读性。宏定义使得存储段看上去像 char 和 int 类型的数组。下面是一些绝对寄存器寻址的例子。
列表 0-7

```
inp_byte=XBYTE[0x8500];      // 从地址8500H读一个字节
inp_word=XWORD[0x4000];      // 从地址4000H读一个字和2001H
c=*((char xdata *) 0x0000); // 从地址0000读一个字节
XBYTE[0x7500]=out_val;       // 写一个字节到 7500H
```

可对除BDATA和BIT段之外的其它数据段采用以上方法寻址。通过包含头文件 absacc.h 来进行绝对地址访问。

4.5 CODE 段

代码段的数据是不可改变的，8051 的代码段不可重写。一般，代码段中可存放数据表，

跳转向量和状态表。对 CODE 段的访问和对 XDATA 段的访问的时间是一样的。代码段中的对象在编译的时候初始化，否则，你就得不到你想要的值。下面是代码段的声明例子。

```
unsigned int code unit_id[2]=1234;
unsigned char
         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
         0x08, 0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15
};
```

5 指针

C51 提供一个 3 字节的通用存储器指针。通用指针的头一个字节表明指针所指的存储区空间。另外两个字节存储 16 位偏移量。对于 DATA，IDATA 和 PDATA 段，只需要 8 位偏移量。

指针类型	大小
通用指针	3 字节
XDATA 指针	2 字节
CODE 指针	2 字节
IDATA 指针	1 字节
DATA 指针	1 字节
PDATA 指针	1 字节

表 0-5

Keil 允许使用者规定指针指向的存储段。这种指针叫具体指针。使用具体指针的好处是节省了存储空间。编译器不用为存储器选择和决定正确的存储器操作指令产生代码，这样就使代码更加简短，但你必须保证指针不指向你所声明的存储区以外的地方，否则会产生错误，而且很难调试。

下面的例子反映出使用具体指针比使用通用指针更加高效。使用通用指针的第一个循环需要 378 个处理周期，使用具体指针只需要 151 个处理周期。

列表 0-8

```
1       #include <absacc.h>
2
3       char *generic_ptr;
4
5       char data *xd_ptr;
6
7       char mystring[]="Test output";
8
9       main() {
10 1      generic_ptr=mystring;
11 1      while (*generic_ptr) {
12 2        XBYTE[0x0000]=*generic_ptr;
13 2        generic_ptr++;
14 2      }
15 1
16 1      xd_ptr=mystring;
17 1      while (*xd_ptr) {
18 2        XBYTE[0x0000]=*xd_ptr;
19 2        xd_ptr++;
20 2      }
21 1    }
```

编译产生的汇编代码

```
                ; FUNCTION main (BEGIN)
                                             ; SOURCE LINE # 9
                                             ; SOURCE LINE # 10
0000 750004     R       MOV     generic_ptr,#04H
0003 750000     R       MOV     generic_ptr+01H,#HIGH mystring
0006 750000     R       MOV     generic_ptr+02H,#LOW mystring
0009            ?C0001:
                                             ; SOURCE LINE # 11
0009 AB00       R       MOV     R3,generic_ptr
000B AA00       R       MOV     R2,generic_ptr+01H
000D A900       R       MOV     R1,generic_ptr+02H
000F 120000     E       LCALL   ?C_CLDPTR
0012 FF                 MOV     R7,A
0013 6011               JZ      ?C0002
                                             ; SOURCE LINE # 12
0015 900000             MOV     DPTR,#00H
0018 F0                 MOVX    @DPTR,A
                                             ; SOURCE LINE # 13
0019 7401               MOV     A,#01H
001B 2500       R       ADD     A,generic_ptr+02H
001D F500       R       MOV     generic_ptr+02H,A
001F E4                 CLR     A
0020 3500       R       ADDC    A,generic_ptr+01H
0022 F500       R       MOV     generic_ptr+01H,A
                                             ; SOURCE LINE # 14
0024 80E3               SJMP    ?C0001
0026            ?C0002:
                                             ; SOURCE LINE # 16
0026 750000     R       MOV     xd_ptr,#LOW mystring
0029            ?C0003:
                                             ; SOURCE LINE # 17
0029 A800       R       MOV     R0,xd_ptr
002B E6                 MOV     A,@R0
002C FF                 MOV     R7,A
002D 6008               JZ      ?C0005
                                             ; SOURCE LINE # 18
002F 900000             MOV     DPTR,#00H
0032 F0                 MOVX    @DPTR,A
                                             ; SOURCE LINE # 19
0033 0500       R       INC     xd_ptr
                                             ; SOURCE LINE # 20
0035 80F2               SJMP    ?C0003
                                             ; SOURCE LINE # 21
0037            ?C0005:
```

```
0037 22                 RET
                ; FUNCTION main (END)
```

由于使用具体指针能够节省不少时间，所以我们一般都不使用通用指针。

6 中断服务

8051 的中断系统十分重要,C51 使你能够用 C 来声明中断和编写中断服务程序(当然你也可以用汇编来写)。中断过程通过使用 interrupt 关键字和中断号(0 到 31)来实现.中断号告述编译器中断程序的入口地址。中断号对应着 IE 寄存器中的使能位，换句话说，IE 寄存器中的 0 位对应着外部中断 0，相应的外部中断 0 的中断号是 0。表 0-6 反映了这种关系。

IE 寄存器中的使能位和 C 中的中断号	中断源
0	外部中断 0
1	定时器 0 溢出
2	外部中断 1
3	定时器 1 溢出
4	串行口中断
5	定时器 2 溢出

表 0-6

一个中断过程并不一定带上所有参数，可以没有返回值。有了这些限制，编译器不须要担心寄存器组参数的使用和对累加器，状态寄存器，B 寄存器，数据指针和默认的寄存器的保护，只要他们在中断程序中被用到，编译的时候会把他们入栈，在中断程序结束时将他们恢复。中断程序的入口地址被编译器放在中断向量中。C51 支持所有 5 个 8051/8052 标准中断（从 0 到 4）和在 8051 系列中多达 27 个中断源。一个中断服务程序的例子如下：

列表 0-9

```
1     #include <reg51.h>
2     #include <stdio.h>
3
4     #define RELOADVALH 0x3C
5     #define RELOADVALL 0xB0
6
7     extern unsigned int tick_count;
8
9     void timer0(void) interrupt 1 {
10 1    TR0=0;                  // 停止定时器0
11 1    TH0=RELOADVALH;         // 50ms后溢出
12 1    TL0=RELOADVALL;
13 1    TR0=1;                  // 启动 T0
14 1    tick_count++;           // 时间计数器加1
15 1    printf("tick_count=%05u\n", tick_count);
16 1  }
```

编译后产生的汇编代码

```
        ; FUNCTION timer0 (BEGIN)
0000 C0E0     PUSH ACC
0002 C0F0     PUSH B
0004 C083     PUSH DPH
0006 C082     PUSH DPL
```

```
0008 C0D0     PUSH PSW
000A C000     PUSH AR0
000C C001     PUSH AR1
000E C002     PUSH AR2
0010 C003     PUSH AR3
0012 C004     PUSH AR4
0014 C005     PUSH AR5
0016 C006     PUSH AR6
0018 C007     PUSH AR7
                                        ; SOURCE LINE # 9
                                        ; SOURCE LINE # 10
001A C28C     CLR  TR0
                                        ; SOURCE LINE # 11
001C 758C3C   MOV  TH0,#03CH
                                        ; SOURCE LINE # 12
001F 758AB0   MOV  TL0,#0B0H
                                        ; SOURCE LINE # 13
0022 D28C     SETB TR0
                                        ; SOURCE LINE # 14
0024 900000 E MOV DPTR,#tick_count+01H
0027 E0       MOVX  A,@DPTR
0028 04       INC   A
0029 F0       MOVX  @DPTR,A
002A 7006     JNZ   ?C0002
002C 900000 E MOV   DPTR,#tick_count
002F E0       MOVX  A,@DPTR
0030 04       INC   A
0031 F0       MOVX  @DPTR,A
0032 ?C0002:
                                        ; SOURCE LINE # 15
0032 7B05     MOV   R3,#05H
0034 7A00 R   MOV   R2,#HIGH ?SC_0
0036 7900 R   MOV   R1,#LOW ?SC_0
0038 900000 E MOV   DPTR,#tick_count
003B E0       MOVX  A,@DPTR
003C FF       MOV   R7,A
003D A3       INC   DPTR
003E E0       MOVX  A,@DPTR
003F 900000 E MOV   DPTR,#?_printf?BYTE+03H
0042 CF       XCH   A,R7
0043 F0       MOVX  @DPTR,A
0044 A3       INC   DPTR
0045 EF       MOV   A,R7
0046 F0       MOVX  @DPTR,A
```

```
0047 120000 E LCALL  _printf
                                        ; SOURCE LINE # 16
004A D007     POP    AR7
004C D006     POP    AR6
004E D005     POP    AR5
0050 D004     POP    AR4
0052 D003     POP    AR3
0054 D002     POP    AR2
0056 D001     POP    AR1
0058 D000     POP    AR0
005A D0D0     POP    PSW
005C D082     POP    DPL
005E D083     POP    DPH
0060 D0F0     POP    B
0062 D0E0     POP    ACC
0064 32       RETI
        ; FUNCTION timer0 (END)
```

在上面的例子中，调用 printf 函数使得编译器把所有的工作寄存器入栈，因为调用本身和非再入函数 printf 的处理过程中要使用到这些寄存器。如果在 C 源程序中把调用语句去掉的话，编译出来的代码就小得多了。

列表 0-10

```
1        #include <reg51.h>
2
3        #define RELOADVALH 0x3C
4        #define RELOADVALL 0xB0
5
6        extern unsigned int tick_count;
7
8        void timer0(void) interrupt 1 using 0 {
9  1       TR0=0;                // 停止定时器0
10 1       TH0=RELOADVALH;       // 设定溢出时间50ms
11 1       TL0=RELOADVALL;
12 1       TR0=1;                // 启动T0
13 1       tick_count++;         // 时间计数器加1
14 1    }
```

编译后产生的汇编代码

```
         ; FUNCTION timer0 (BEGIN)
0000 C0E0      PUSH    ACC
Push and pop of register bank 0 and the B register is eliminated because printf was
usingthe registers for parameters and using B internally.
0002 C083      PUSH    DPH
0004 C082      PUSH    DPL
```

```
                                        ; SOURCE LINE # 8
                                        ; SOURCE LINE # 9
0006 C28C       CLR     TR0
                                        ; SOURCE LINE # 10
0008 758C3C     MOV     TH0,#03CH
                                        ; SOURCE LINE # 11
000B 758AB0     MOV     TL0,#0B0H
                                        ; SOURCE LINE # 12
000E D28C       SETB    TR0
                                        ; SOURCE LINE # 13
0010 900000 E   MOV     DPTR,#tick_count+01H
0013 E0         MOVX    A,@DPTR
0014 04         INC     A
0015 F0         MOVX    @DPTR,A
0016 7006       JNZ     ?C0002
0018 900000 E   MOV     DPTR,#tick_count
001B E0         MOVX    A,@DPTR
001C 04         INC     A
001D F0         MOVX    @DPTR,A
001E       ?C0002:
                                        ; SOURCE LINE # 14
001E D082       POP     DPL
0020 D083       POP     DPH
0022 D0E0       POP     ACC
0024 32         RETI
         ; FUNCTION timer0 (END)
```

6.1 指定中断服务程序使用的寄存器组

当指定中断程序的工作寄存器组时，保护工作寄存器的工作就可以被省略。使用关键字 using，后跟一个 0 到 3 的数对应着 4 组工作寄存器。当指定工作寄存器组的时候，默认的工作寄存器组就不会被推入堆栈，这将节省 32 个处理周期，因为入栈和出栈都需要 2 个处理周期。为中断程序指定工作寄存器组的缺点是，所有被中断调用的过程都必须使用同一个寄存器组，否则参数传递会发生错误。下面的例子给出了定时器 0 的中断服务程序，但我已经告述编译器使用寄存器组 0。

列表 0-11

```
1       #include <reg51.h>
2       #include <stdio.h>
3
4       #define RELOADVALH 0x3C
5       #define RELOADVALL 0xB0
6
7       extern unsigned int tick_count;
8
9       void timer0(void) interrupt 1 using 0 {
```

```
10 1      TR0=0;              // 停止定时器0
11 1      TH0=RELOADVALH;     // 设置溢出时间为50ms
12 1      TL0=RELOADVALL;
13 1      TR0=1;              // 启动T0
14 1      tick_count++;       // 时间计数器加1
15 1      printf("tick_count=%05u\n", tick_count);
16 1    }
```

编译后产生的汇编代码

```
             ; FUNCTION timer0 (BEGIN)
0000 C0E0            PUSH    ACC
0002 C0F0            PUSH    B
Push and pop of register bank 0 has been eliminated because the compiler assumes
that thisISR 'owns' RB0.
0004 C083            PUSH    DPH
0006 C082            PUSH    DPL
0008 C0D0            PUSH    PSW
000A 75D000          MOV     PSW,#00H
                                          ; SOURCE LINE # 9
                                          ; SOURCE LINE # 10
000D C28C            CLR     TR0
                                          ; SOURCE LINE # 11
000F 758C3C          MOV     TH0,#03CH
                                          ; SOURCE LINE # 12
0012 758AB0          MOV     TL0,#0B0H
                                          ; SOURCE LINE # 13
0015 D28C            SETB    TR0
                                          ; SOURCE LINE # 14
0017 900000 E        MOV     DPTR,#tick_count+01H
001A E0              MOVX    A,@DPTR
001B 04              INC     A
001C F0              MOVX    @DPTR,A
001D 7006            JNZ     ?C0002
001F 900000 E        MOV     DPTR,#tick_count
0022 E0              MOVX    A,@DPTR
0023 04              INC     A
0024 F0              MOVX    @DPTR,A
0025       ?C0002:
                                          ; SOURCE LINE # 15
0025 7B05            MOV     R3,#05H
0027 7A00    R       MOV     R2,#HIGH ?SC_0
0029 7900    R       MOV     R1,#LOW ?SC_0
002B 900000 E        MOV     DPTR,#tick_count
```

```
002E E0             MOVX    A,@DPTR
002F FF             MOV     R7,A
0030 A3             INC     DPTR
0031 E0             MOVX    A,@DPTR
0032 900000 E       MOV     DPTR,#?_printf?BYTE+03H
0035 CF             XCH     A,R7
0036 F0             MOVX    @DPTR,A
0037 A3             INC     DPTR
0038 EF             MOV     A,R7
0039 F0             MOVX    @DPTR,A
003A 120000 E       LCALL   _printf
                                        ; SOURCE LINE # 16
003D D0D0           POP     PSW
003F D082           POP     DPL
0041 D083           POP     DPH
0043 D0F0           POP     B
0045 D0E0           POP     ACC
0047 32             RETI
             ; FUNCTION timer0 (END)
```

7 再入函数

因为 8051 内部堆栈空间的限制，C51 没有像大系统那样使用调用堆栈。一般，C 语言中调用过程时，会把过程的参数和过程中使用的局部变量入栈。为了提高效率，C51 没有提供这种堆栈，而是提供一种压缩栈，每个过程被给定一个空间用于存放局部变量，过程中的每个变量都存放在这个空间的固定位置，当递归调用这个过程时，会导致变量被覆盖。

在某些实时应用中，非再入函数是不可取的，因为函数调用时，可能会被中断程序中断，而在中断程序中可能再次调用这个函数。所以，C51 允许将函数定义成再入函数。再入函数可被递归调用和多重调用而不用担心变量被覆盖，因为每次函数调用时的局部变量都会被单独保存。因为这些堆栈是模拟的，再入函数一般都比较大，运行起来也比较慢。模拟栈不允许传递 bit 类型的变量，也不能定义局部位标量。

8 使用 Keil C 时应做的和应该避免的

Keil 编译器能从你的 C 程序源代码中产生高度优化的代码。但你可以帮助编译器产生更好的代码。下面将讨论这方面的一些问题。

8.1 采用短变量

一个提高代码效率的最基本的方式就是减小变量的长度。使用 C 编程时，我们都习惯于对循环控制变量使用 int 类型，这对 8 位的单片机来说，是一种极大的浪费。你应该仔细考虑你所声明的变量值可能的范围，然后选择合适的变量类型。很明显，经常使用的变量应该是 unsigned char，只占用一个字节。

8.2 使用无符号类型

为什么要使用无符号类型呢。原因是 8051 不支持符号运算，程序中也不要使用含有带符号变量的外部代码。除了根据变量长度来选择变量类型自外，你还要考虑是否变量是否

会用于负数的场合。如果你的程序中可以不需要负数，那么把变量都定义成无符号类型的。

8.3 避免使用浮点指针

在 8 位操作系统上使用 32 位浮点数是得不偿失的，你可以这样做，但会浪费大量的时间。所以当你要在系统中使用浮点数的时候，你要问问自己这是否一定需要。可以通过提高数值数量级和使用整型运算来消除浮点指针。处理 ints 和 longs 比处理 doubles 和 floats 要方便得多。你的代码执行起来会更快，也不用连接处理浮点指针的模块。如果你一定要采用浮点指针的话，你应该采用西门子 80517 和达拉斯半导体公司的 80320 这些已经对数处理进行过优化的单片机。

如果你不得不在你的代码中加入浮点指针，那么，你的代码长度会增加，程序执行速度也会比较慢。如果浮点指针运算能被中断的话，你必须确保要么中断中不会使用浮点指针运算，要么在中断程序前使用 fpsave 指令把中断指针推入堆栈，在中断程序执行后使用 fprestore 指令把指针恢复。还有一种方法是，当你要使用像 sin()这样的浮点运算程序时,禁止使用中断，在运算程序执行完之后再使能它。

列表 0-12

```
#include <math.h>

void timer0_isr(void) interrupt 1 {
  struct FPBUF fpstate;
  ...                           // 初始化代码或
                                // 非浮点指针代码
  fpsave(&fpstate);             // 保留浮点指针系统
  ...                           // 中断服务程序代码, 包括所有
                                // 浮点指针代码
  fprestore(&fpstate);          // 复位浮点指针
                                // 系统状态
  ...                           // 非浮点指针中断
                                // 服务程序代码
}

float my_sin(float arg) {
  float retval;
  bit old_ea;
  old_ea=EA;                   // 保留当前中断状态
  EA=0;                        // 关闭中断
  retval=sin(arg);             // 调用浮点指针运算程序
  EA=old_ea;                   // 恢复中断状态
  return retval;
}
```

你还要决定所需要的最大精度，一旦你计算出你所需要的浮点运算的最多的位数，应该通知编译器知道，它将把处理的复杂度控制在最低的范围内。

8.4 使用位变量

对于某些标志位，应使用位变量而不是 unsigned char。这将节省你的内存，你不用

多浪费 7 位存储区。而且位变量在 RAM 中，访问他们只需要一个处理周期。

8.5 用局部变量代替全局变量

把变量定义成局部变量比全局变量更有效率。编译器为局部变量在内部存储区中分配存储空间，而为全局变量在外部存储区中分配存储空间，这会降低你的访问速度。另一个避免使用全局变量的原因是你必须在你系统的处理过程中调节使用全局变量，因为在中断系统和多任务系统中，不止一个过程会使用全局变量。

8.6 为变量分配内部存储区

局部变量和全局变量可被定义在你想要的存储区中。根据先前的讨论,当你把经常使用的变量放在内部 RAM 中时,可使你的程序的速度得到提高,除此之外,你还缩短了你的代码,因为外部存储区寻址的指令相对要麻烦一些。考虑到存储速度，按下面的顺序使用存储器：DATA，IDATA，PDATA，XDATA，当然你要记得留出足够的堆栈空间。

8.7 使用特定指针

当你在程序中使用指针时，你应指定指针的类型，确定它们指向哪个区域如 XDATA 或 CODE 区。这样你的代码会更加紧凑，因为编译器不必去确定指针所指向的存储区，因为你已经进行了说明。

8.8 使用调令

对于一些简单的操作，如变量循环位移，编译器提供了一些调令供用户使用，许多调令直接对应着汇编指令，而另外一些比较复杂并兼容 ANSI。所有这些调令都是再入函数，你可在任何地方安全的调用他们。

和单字节循环位移指令 RL A 和 RR A 相对应的调令是_crol_（循环左移）和_cror_(循环右移)，如果你想对 int 或 long 类型的变量进行循环位移，调令将更加复杂而且执行的时间会更长。对于 int 类型调令为_irol_,_iror_ ,对于 long 类型调令为_lrol_,_lror_。

在 C 中也提供了像汇编中 JBC 指令那样的调令_testbit_。如果参数位置位他将返回 1，否则将返回 0。这条调令在检查标志位时十分有用，而且使 C 的代码更具有可读性。调令将直接转换成 JBC 指令。

列表 0-13

```
#include <instrins.h>
void serial_intr(void) interrupt 4 {
  if (!_testbit_(TI)) {    // 是否是发送中断
    P0=1;                  // 翻转 P0.0
   _nop_();                // 等待一个指令周期
   P0=0;
   ...
  }
  if (!_testbit_(RI)) {
    test=_cror_(SBUF, 1); // 将SBUF中的数据循环
                          // 右移一位
    ...
  }
}
```

8.8 使用宏替代函数

对于小段代码，像使能某些电路或从锁存器中读取数据，你可通过使用宏来替代函数，使得程序有更好的可读性。你可把代码定义在宏中，这样看上去更像函数。编译器在碰到宏时，按照事先定义的代码去替代宏。宏的名字应能够描述宏的操作。当需要改变宏时，你只要修该宏定义处。

列表 0-14

```
#define led_on() {\
     led_state=LED_ON; \
     XBYTE[LED_CNTRL] = 0x01;}

#define led_off() {\
     led_state=LED_OFF; \
     XBYTE[LED_CNTRL] = 0x00;}
#define checkvalue(val) \
     ( (val < MINVAL || val > MAXVAL) ? 0 : 1 )
```

宏能够使得访问多层结构和数组更加容易。可以用宏来替代程序中经常使用的复杂语句以减少你打字的工作量，且有更好的可读性和可维护性。

9 存储器模式

C51 提供了 3 种存储器模式来存储变量，过程参数和分配再入函数堆栈。你应该尽量使用小存储器模式。很少应用系统需要使用其它两种模式，像有大的再入函数堆栈系统那样。一般来说如果系统所需要的内存数小于内部 RAM 数时，都应以小存储模式进行编译。在这种模式下，DATA 段是所有内部变量和全局变量的默认存储段，所有参数传递都发生在 DATA 段中。如果有函数被声明为再入函数，编译器会在内部 RAM 中为他们分配空间。这种模式的优势就是数据的存取速度很快，但只有 120 个字节的存储空间供你使用（总共有 128 个字节，但至少有 8 个字节被寄存器组使用）你还要为程序调用开辟足够的堆栈。

如果你的系统有 256 字节或更少的外部 RAM，你可以使用压缩存储模式，这样一来，如果不加说明，变量将被分配在 PDATA 段中。这种模式将扩充你能够使用的 RAM 数量。对 XDATA 段以外的数据存储仍然是很快的。变量的参数传递将在内部 RAM 中进行，这样存储速度会比较快。对 PDATA 段的数据的寻址是通过 R0 和 R1 进行间接寻址，比使用 DPTR 要快一些。

在大存储模式中，所有变量的默认存储区是 XDATA 段。Keil C 尽量使用内部寄存器组进行参数传递。在寄存器组中可以传递参数的数量和和压缩存储模式一样。再入函数的模拟栈将在 XDATA 中。对 XDATA 段数据的访问是最慢的，所以要仔细考虑变量应存储的位置，使数据的存储速度得到优化。

10 混合存储模式

Keil 允许使用混合的存储模式。这点在大存储模式中是非常有用的。在大存储器模式下，有些过程对数据传递的速度要求很高，我就把过程定义在小存储模式寄存器中，这使得编译器为该过程的局部变量在内部 RAM 中分配存储空间，并保证所有参数都通过内部 RAM 进行传递。尽管采用混合模式后编译的代码长度不会有很大的改变，但这种努力是值得的。

就像能在大模式下把过程声明为小模式一样，你像能在小模式下把过程声明为压缩模式或大模式。这一般使用在需要大量存储空间的过程上。这样过程中的局部变量将被存储在外部存储区中。你也可以通过过程中的变量声明，把变量分配在 XDATA 段中。

11 运行库

运行库中提供了很多短小精悍的函数。你可以很方便的使用他们，你自己很难写出更好的代码了。值得注意的是库中有些函数不是再入函数，如果在执行这些函数的时候被中断，而在中断程序中又调用了该函数，将得到意想不到的结果，而且这种错误很难找出来。表 0-7 列出了非再入型的库函数，使用这些函数时，最好禁止使用这些函数的中断。

gets	atof	atan2
printf	atol	cosh
sprinf	atoi	sinh
scanf	exp	tanh
sscanf	log	calloc
memccpy	log10	free
strcat	sqrt	Init_mempool
strncat	srand	malloc
strncmp	cos	realloc
strncpy	sin	ceil
strspn	tan	floor
strcspn	acos	modf
strpbrk	asin	pow
strrpbrk	atan	

表 0-7

12 动态存储分配

通过标准 C 的功能函数 malloc 和 free Keil C 提供了动态存储分配功能。对大多数应用来说，应尽可能在编译的时候确定所需要的内存空间并进行分配，但是，对于有些需要使用动态结构如树和链表的应用来说，这种方式就不再适用了。Keil C 对这种应用提供了有力的支持。

动态分配函数要求用户声明一个字节数组作为堆。根据所需要动态内存的大小来决定数组的长度。作为堆被声明的数组在 XDATA 区中，因为库函数使用特定指针来进行寻址，此外，也没有必要在 DATA 区中动态分配内存，因为 DATA 区的空间本身就很小。

一旦在 XDATA 区中声明了这个块，指向块的指针和块的大小要传递给初始化函数（init_mempool）,他将设置一些内部变量和进行一些准备工作并对动态存储空间进行初始化。一旦初始化工作完成，可在任何系统中调用动态分配函数。动态分配的函数包括 malloc(接受一个描述空间大小的 unsigned int 参数,返回一个指针),calloc(接受一个描述数量和一个描述大小的 unsigned int 参数,返回一个指针),realloc(接受一个指向块的指针和一个描述空间大小的 unsigned int 参数,返回一个指向按给出参数分配的空间的指针),free(接受一个指向块的指针,使这个空间可以再次被分配)。所有这些函数都将返回指向堆的指针，如果失败的话将返回 NULL。下面是一个动态分配存储区的例子

列表 0-15

```
#include <stdio.h>
#include <stdlib.h>

// 代码中利用特定指针来提高效率

typedef struct entry_str {          // 定义队列元素结构
  struct entry_str xdata *next;     // 指向下一个元素
  char text[33];                    // 结构中的字符串
} entry;

void init_queue(void);
void insert_queue(entry xdata *);
void display_queue(entry xdata *);
void free_queue(void);
entry xdata *pop_queue(void);
```

```
entry xdata *root=NULL;              // 设置队列为空

void main(void) {
  entry xdata *newptr;
  init_queue();                      // 设置队列
  ...
  newptr=malloc(sizeof(entry));      // 分配一个队列元素
  sprintf(newptr->text, "entry number one");
  insert_queue(newptr);              // 放入队列
  ...
  newptr=malloc(sizeof(entry));
  sprintf(newptr->text, "entry number two");
  insert_queue(newptr);              // 插入另一个元素

  ...
  display_queue(root);               // 显示队列
  ...
  newptr=pop_queue();                // 弹出头元素
  printf("%s\n", newptr->text);
  free(newptr);                      // 删除它
  ...
  free_queue();                      // 释放整个队列空间
}

void init_queue(void) {
  static unsigned char memblk[1000];     // 这部分空间将作为堆
  init_mempool(memblk, sizeof(memblk));  // 建立堆
}

void insert_queue(entry xdata *ptr) {    // 把元素插入队尾
entry xdata *fptr, *tptr;
  if (root==NULL){
    root=ptr;
} else {
  fptr=tptr=root;
  while (fptr!=NULL) {
    tptr=fptr;
    fptr=fptr->next;
  }
  tptr->next=ptr;
 }
ptr->next=NULL;
}
```

```
void display_queue(entry xdata *ptr) {// 显示队列
  entry xdata *fptr;
  fptr=ptr;
  while (fptr!=NULL) {
    printf("%s\n", fptr->text);
    fptr=fptr->next;
  }
}

void free_queue(void) {              // 释放队列空间
  entry xdata *temp;
  while (root!=NULL) {
  temp=root;
  root=root->next;
  free(temp);
 }
}

entry xdata *pop_queue(void) {     // 删除队列
  entry xdata *temp;
  if (root==NULL) {
    return NULL;
  }
  temp=root;
  root=root->next;
  temp->next=NULL;
  return temp;
}
```

可见使用动态分配函数就像 ANSI C 一样，十分方便。

13 结论

使用 C 来开发你的系统将更加方便快捷，他既不会降低你对硬件的控制能力也不会使你的代码长度增加多少，如果你运用得好的话，你能够开发出非常高效的系统，并且非常利于维护。

第三章 使用软件补充硬件

1 介绍

这章将展示用软件来提升你系统整体性能的方法。通过这些软件方法，将提供用户接口，时钟系统，并能够减少不必要的硬件。下面将举一个使用 8051 作时钟的例子。系统用一个接在单片机端口上的标准 2x16 的 LCD 来显示时间。按第一个按钮将进入模式设置状态，并在相应的地方显示光标，按第二个按钮将增加数值。15 秒之后如果无键按下，将回到正常状态。

为了降低成本,用微处理器来仿真实时时钟芯,并且液晶片将接在微处理器的一个口上.用软件仿真实时时钟并直接控制液晶片的接口，这样就不再需要使用译码芯片和实时时钟芯片了。为了进一步减少元器件，将采用内部 RAM，程序能够使用的 RAM 就被控制在 128 个字节以内。

做软件的时候，要认真考虑 RAM 1。系统使用了带内部 EPROM 的 8051，口 0 和口 2 保留用做系统扩展之需。为了有一个比较，图 0-2 给了传统设计方法的接线图。处理器对实时时钟芯片和 LCD 驱动芯片进行寻址，这需要一个地址译码器和一个与非门。这个设计还使用了外部 SRAM。注意两种设计的不同。

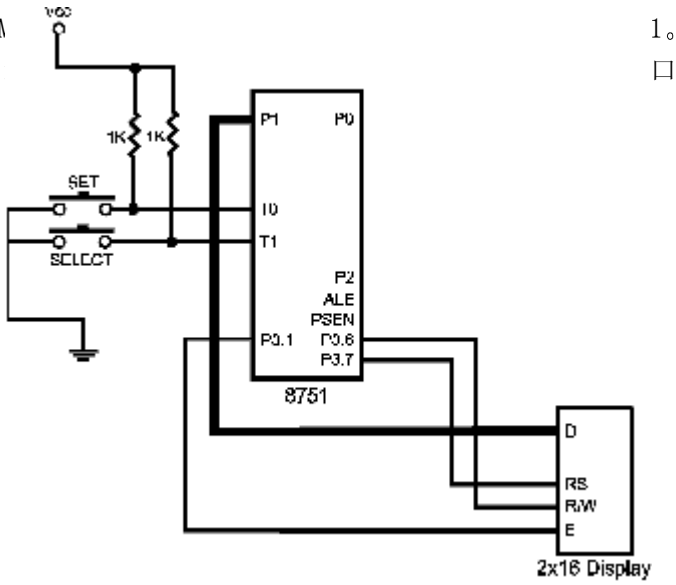

图 0-1 时钟电路

2 使用小存储模式

为了不使用 SRAM，就要使用小存储模式。这把能够使用的 RAM 数量限制在 128 个字节内。处理器内部堆栈，压缩栈，所有程序变量和所有包含进来的库函数都将使用这些数量有限的 RAM。

编译器可以通过覆盖技术来优化 RAM 的使用，所以应尽量使用局部变量。通过覆盖分析，编译器决定哪些变量被分配在一起，哪些不能在同一时间存在。这些分析告诉 L51 如何使用局部存储区。很多时候，根据调用结构，一个存储地址将存储不同的局部变量，所以要多使用局部变量，当然，不可避免的有一些全局变量，像标志位，保存每日时间的变量。也有可能在指定的函数中定义静态变量，编译器会把他们当成全局变量一样处理。

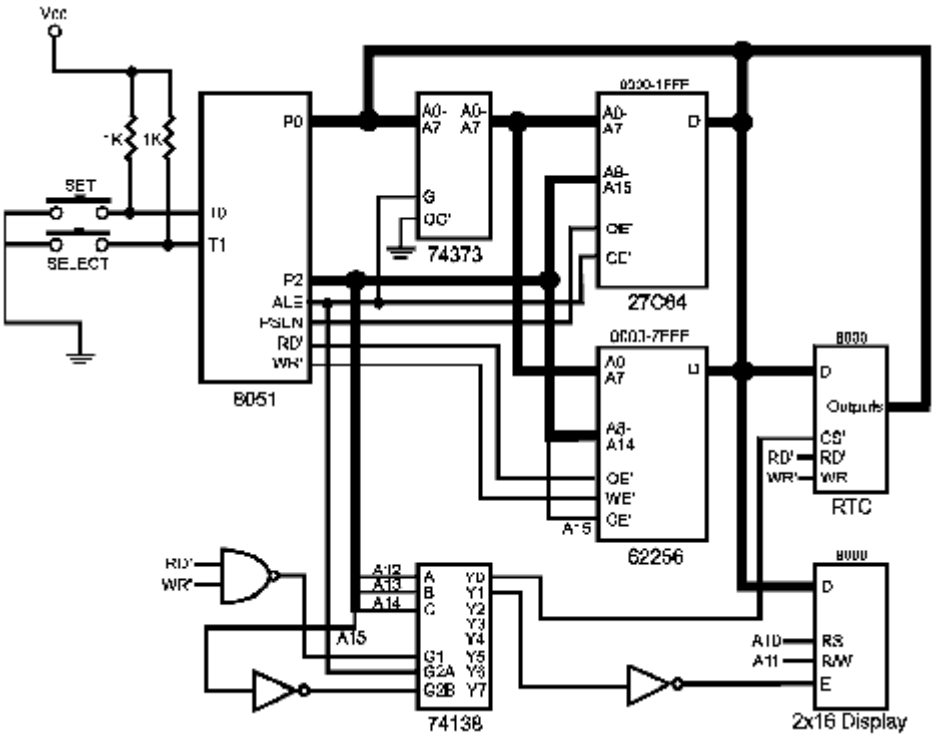

图 0-2 扩展电路

为了节省 RAM，要尽可能少的调用库函数，一些库函数要占用大量的 RAM，并且这些函数的范围和功能都超出了所需。比如 printf 函数，包含了时钟不需要的很多初始化功能。应考虑是否要专门写一个程序来替代标准的 printf 函数，这样会占用更少的资源。

Instruction	Description	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
Clear display	Clears entire display and sets DD RAM address 0 in address counter	0	0	0	0	0	0	0	0	0	1
Return home	Sets DD RAM address 0 in address counter. Returns display from shifted to original position. DD RAM contents remain unchanged.	0	0	0	0	0	0	0	0	1	*
Entry mode set	Sets cursor move direction (I/D) and specifies shift of display (S). These operations are performed during data write and data read.	0	0	0	0	0	0	0	1	I/D	S
Display on/off	Sets on/off of entire display (D), cursor on/off (C), and blink of cursor position character (B).	0	0	0	0	0	0	1	D	C	B
Cursor/display shift	Moves cursor and shifts display without changing DD RAM contents.	0	0	0	0	0	1	S/C	R/L	*	*
Function set	Sets interface data length (DL), number of display lines (N), and character font (F).	0	0	0	0	1	DL	N	F	*	*
Set CG RAM address	Sets CG RAM address. CG RAM data are sent and received after this setting.	0	0	0	1	ACG					
Set DD RAM address	Sets DD RAM address. DD RAM data are sent and received after this setting.	0	0	1	ADD						
Read busy flag and address	Reads busy flag (BF) indicating internal operation and reads address counter contents.	0	1	BF	AC						
Write data to CG of DD RAM	Writes data into DD RAM or CG RAM.	1	0	Write data							
Read data from CG or DD RAM	Reads data from DD RAM or CG RAM.	1	1	Read data							
I/D: 1 = Increment, 0 = Decrement S: 1 = Accompanies display shift S/C: 1 = Display shift, 0 = Cursor move R/L: 1 = Shift to the right, 0 = Shift to the left DL: 1 = 8 bits, 0 =4 bits N: 1 = 2 lines, 0 = 1 line F: 1 = 5 x 10 dots, 0 = 5 x 7 dots BF: 1 = Internally operating, 0 = Ready for next instruction											

表 0-1

3 使用液晶驱动

这个项目所选择的液晶驱动芯片为 GMD16202，有 2x16 段。它的接口十分简单；表 0-1 中列出了对芯片操作的简单的指令。上电后，必须初始化显示，包括总线的宽度，线的数量，输入模式等。每个命令之间要查询显示是否准备好接收下一个数据。执行每条指令一般需要 40ms 时间，有些只需要 1.64ms。

3.1 LCD 驱动接口

我们通过减少元件来降低成本,从液晶驱动接口可以很容易的看出这点,驱动芯片的 8 位数据线和 P1 口相连，用软件来控制显示和产生正确的使能信号脉冲序列，锁住输入输出的数据。而典型的系统，驱动芯片和 8051 的总线相连，软件只需要用 XBYTE[]对芯片寻址就可以了。当把工作交由软件来完成之后，就不再需要解码器和一些支持芯片，这就降低了速度。因为软件要完成 8051 和 LCD 驱动芯片之间的数据传输工作，代码的长度和执行时间都会比较长，对时钟系统来说有大量的 EPROM 空间剩余，代码的长度不是问题，而由以后的分析我们会发现执行的时间长短也不是问题。一旦理解了 LCD 驱动芯片所需的信号和时序之后，显示的接口函数就很容易写了。软件只须要 3 个基本功能。写入一个命令，写入下一个字符，读显示状态寄存器，这些操作的时序关系见图 0-3 和 0-4。在每个信号之间允许有很长的时间间隔，信号有效或无效的时间可以毫秒来计算，而不像系统总线那样以钠秒来计算。I/O 函数只需要按照时序图来操作就可以了。

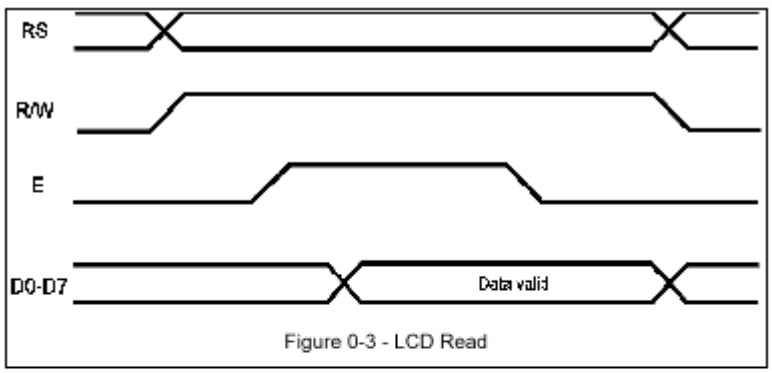

Figure 0-3 - LCD Read

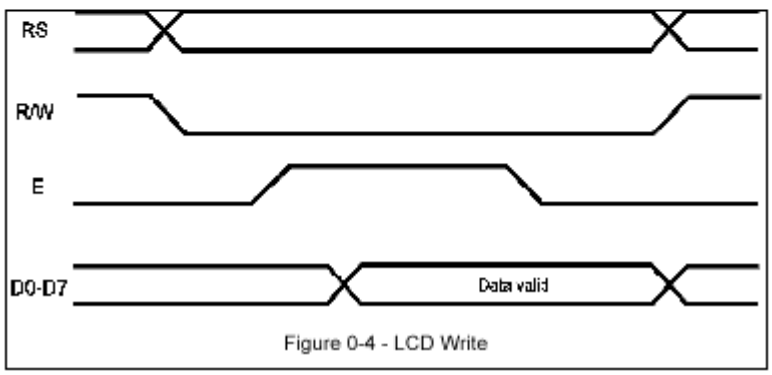

Figure 0-4 - LCD Write

列表0-1

```
void disp_write(unsigned char value) {
  DISPDATA=value; // 发送数据
  REGSEL=1;       // 选择数据寄存器
  RDWR=0;         // 选择写模式
  ENABLE=1;       // 发送数据给LCD
  ENABLE=0;
}
```

disp_write 的功能是送一个字符给 LCD 显示。在送数之前应查看 LCD 驱动芯片是否已经准备好接收数据。

列表 0-2

```
void disp_cmd(unsigned char cmd) {
```

```
  DISPDATA=cmd;       // 发送命令
  REGSEL=0;           // 选择命令寄存器
  RDWR=0;             // 选择写模式
  ENABLE=1;           // 发送命令给LCD
  ENABLE=0;
  TH1=0;              // 定时85ms
  TL1=0;
  TF1=0;
  TR1=1;
  while (!TF1 && disp_read() & DISP_BUSY);// 等待显示
                                          // 结束命令
  TR1=0;
}
```

disp_cmd 函数的时序和 disp_write 一样，但只有到 LCD 驱动芯片准备好接收下一个数据时，才结束函数。

列表 0-3

```
unsigned char disp_read(void) {
  unsigned char value;
  DISPDATA=0xFF;        // 为所有输入设置端口
  REGSEL=0;             // 选择命令寄存器
  RDWR=1;               // 选择读模式
  ENABLE=1;             // 使能LCD输出
  value=DISPDATA;       // 读入数据
  ENABLE=0;             // 禁止LCD输出
  return(value);
}
```

disp_read 函数的功能是锁住显示状态寄存器中的数。根据上面的时序进行操作，同时读出 P1 中的数据。数据被保存，并作为调用结果返回。

如你所见，从控制器的端口控制显示是十分简单的。缺点是所花的时间要长一些，另外，代码也比较长。但是系统的成本却降低了。

4 显示数据

当初始化完成之后，就可以进行显示了。写入字符十分简单。要告诉驱动芯片所接收到字符的显示地址，然后发送所要显示的字符。当接收下一个显示字符时，芯片的内部显示地址将自动加一。

为了正确显示信息和与用户之间相互作用，系统需要一个函数能够完成上述功能,并能清除显示.我们重新定义 putchar 函数来向 LCD 输出显示字符，因此我们必须知道如何使用前面所写的函数来完成字符的输出过程，除此之外还在其它一些地方作了改动，当过程检测到 255 时，将发出命令清除显示并返回。putchar 函数从清除显示开始对写入的数据进行计数，从而决定是否开始在显示的第二行写入。函数如下：

列表 0-4

```
char putchar(char c) {
  static unsigned char flag=0;
  if (!flag || c==255) {      // 显示是否应该回到原位
```

```
    disp_cmd(DISP_HOME);
    flag=0;
    if (c==255) {
      return c;
    }
  }
  if (flag==16) {                    // 是否使用下一个显示行
    disp_cmd(DISP_POS | DISP_LINE2); // 显示移到第二行
  }
  disp_write(c);                     // 送一个字符显示
  while (disp_read() & DISP_BUSY);   // 等待显示
  flag++; // increment the line flag
  if (flag>=32) { flag=0; }          // 显示完之后，清除
  return(c);
}
```

如你所见，函数十分简单，它调用一些低层的 I/O 过程向显示写入数据。如果写入成功的话，返回所传送的字符，它假设显示工作正常，所以总是返回所写入的字符。

4.1 定制 printf 函数

C51 的库函数中包含了 printf 函数。该函数格式化字符串，并把他们输出到标准输出设备，对 PC 来说标准输出设备就是你的显示设备，对 8051 来说是串行口。在这里只有一个显示。就本质来说 printf 函数是通过不断的调用 putchar 函数来输出字符串的。这样通过重新定义 putchar 函数就可以改变 printf 函数，连接器在连接的时候，将使用源代码中的 putchar 函数，而不是运行函数库中的函数。下面的功能将调用 printf 函数来格式化时间串并发送显示。

列表 0-5

```
void disp_time(void) {
  // 显示保存的当前时间
  // 当时间数据使用完毕后才清除使用标志位
  // 这避免了数据在使用中被修改
  printf("\xFFTIME OF DAY IS: %BO2u:%BO2u:%BO2u ",
  timeholder.hour, timeholder.min, timeholder.sec);
  disp_update=0;                    // 清除显示更新标志位
}
```

5 使用定时计数器来计时

不少嵌入式系统，特别是那些低成本的系统没有实时时钟来提供时间信号。然而这些系统一般都要在某个时间或在系统事件的某段时间之后执行某段任务。这些任务包括以一定的时间间隔显示数据和以一定的频率接收数据。一般，设计者会通过循环来延时，这种做法的缺点是，对不同的延时时间要做不同的延时程序，很多延时程序是通过 NOP 和 DJNZ 指令来进行延时的，这对于使用电池的系统来说是一种消耗。

一种好得多的方法是用内置定时器来产生系统时钟。定时器不断的溢出，重装，并在指定的时间产生中断。中断程序重装定时器，分配定时时间，并执行指定的过程。这种方法的好处是很多的。首先，处理器不必一直执行计时循环。他可在各个中断之间处于 idle

模式或执行其它指令。其次，所有控制都在 ISR 中进行，如果系统频率改变了或定时时间需要改变，软件只需要更改一个地方。第三，所有的代码都可用 C 来编写。你可以通过观察汇编后的代码来计算定时器溢出到定时器重装并开始运行所需的时间，进一步根据重装值来计算定时的时间。

我所作过的没有外部时间输入却要有系统时间的嵌入式系统都采用了这种方法。下面将介绍如何每隔 50ms 产生一个时钟信号。在编写软件之前你首先要明确你的要求。如果你最快的任务执行速度是 3ms 一次，那么就以这个时间为准。发生频率比较慢的事件可以很好的被驱动。如果你的系统时间不能很好的兼容，你可以考虑使用两个定时器。

决定了系统的时间标志后，就需要算出按所需频率产生时标的定时器重装值。为此，你要知道你的晶振频率，用它来得到指令周期的执行时间。如果你要产生一个 50ms 的时标，你的系统频率是 12MHz，你的指令执行频率就是 1MHz,每条指令的执行时间就是 1us。

有了指令的执行时间就可以计算出每个系统时间标志所需要的指令周期数。根据前面的条件，需要 50000 个指令周期来获得 50ms 一次的系统频率标志。65536 减去 50000 得到 15536（3CB0）的重装值。如果你的要求不是那么精确的话，可把这个值直接装入定时器中。

下面的例子用定时器 0 产生系统时标，定时器 1 用来产生波特率或其它定时功能。

列表 0-6

```
#define RELOAD_HIGH 0x3C
#define RELOAD_LOW 0xB0
void system_tick(void) interrupt 1 {
  TR0=0;           // 停止定时器
  TH0=RELOAD_HIGH; // 设置重装值
  TL0=RELOAD_LOW;
  TR0=1;           // 重新启动定时器
  // 执行中断操作
}
```

以上为过程的一个基本结构。一旦定时器重装并开始工作之后，你就可以进行一些操作，如保存时标数，事件操作，置位标志位。你必须保证这些操作的时间不超过定时器的溢出的时间，否则将丢失时标数。

可以很容易的让系统在一定的时标数之后执行某些操作，这通过设置一个时标计数变量来完成，这个全球变量在每个时标过程中减一，当它为 0 时将执行操作。例如你有一个和引脚相连的 LED，希望它亮 2 秒钟，然后关掉，代码如下：

```
if (led_timer) {                // 时间计数器不为0
led_timer--;                    // 减时间计数器
  if (!led_timer) {             // 显示时间到...
    LED=OFF; // turn off the LED
}
}
```

虽然上面一段代码很简单，却可以用在大多数嵌入式系统中。当有更复杂的功能需要执行时，这段代码可放置在定时器中断程序中。这样在检查完一个定时时间之后，可以接着检查下一个定时时间，并决定是否执行相应的操作。共用一个时标的定时操作可被放入一个只有时标被某个特定数整除才有效的空间中。

假设你需要以不少于 1 秒的间隔时间执行一些功能，使用上面的时标过程。你只要保存一个计数器，仅当计数器变为 0 的时候，查询那些基于秒的定时操作。而不需要系统每隔 50ms 就查询一次。

```
second_cnt--;              // 减时标计数器
if (!second_cnt) {         // 一秒钟过去了...
  ...                       // 进行相应的操作
  second_cnt=20;            // 重新定时1秒
}
```

注意你的中断服务程序所需的执行时间，如果执行时间超过 50ms 你会发现将丢失时标。在这种情况下，你把一些操作移出中断程序，放到主程序中。通过设置标志位来告诉主程序是否要执行相应的功能。但操作的时间精度就不够高了，因此对时间精度要求很高的操作还是要放在中断程序中。

可用上面的时标过程来做成时钟。它将记录每天的时间，并在需要显示时间的时候置位标志位。主程序将监视标志位，在时间更新的时候显示新的时间。定时器 0 中断程序还将对按键延迟计时。

6 使用系统时标做用户接口

用户接口相对来说比较简单，但并不说明这里讲到的不能用到大系统中。设置键用来击活设置模式，更改时间。当进入设置模式后，设置键将用来增加光标处的数值。选择键将使光标移到下一个位置，当光标移过最后一个位置时，设置模式结束。每次设置键或选择键被击活后，设置模式计数器被装入最大值，每个时标来临时减 1，当减到 0 时，结束设置模式。

每隔 50ms 在中断中查询按键。这种查询速度对人来说已经足够了。有时侯甚至 0.2 秒都可以。对 8051 来说人是一个慢速的 I/O 器件。当检测到有键按下时，将设置一个计数器以防按键抖动。这个计数器在每次中断到来时减 1，直到计数器为 0 时，才再次查询按键。

当设置模式被击活时，软件必须控制光标在显示器上的位置，让操作者知道要设置哪个位置。cur_field 变量指向当前的位置。set_cursor 函数将打开，关闭光标或把它移到所选择的位置。为了简化用户设置的工作和同步时钟，当进行设置时，计时被挂起。这也避免了在设置时，程序用 printf 更新时间。在进行时间更新时，也不允许进入设置模式。这也将避免 pirntf 函数在同一时间被多个中断调用。

下面是系统时标程序，对许多系统来说这个程序已经足够，可把它作为你应用程序的模块。

列表 0-7

```
void system_tick(void) interrupt 1 {
  static unsigned char second_cnt=20;     // 时间计数器顶事为1秒
  TR0=0;                                  // 停止定时器
  TH0=RELOAD_HIGH;                        // 设定重装值
  TL0=RELOAD_LOW;
  TR0=1;                                  // 启动定时器
  if (switch_debounce)
    switch_debounce--;
  }
  if (!switch_debounce) {
  if (!SET) {                             // 如果设置键被按下...
    switch_debounce=DB_VAL;
      if (!set_mode && !disp_update) {    // 如果时钟不在设置模式
        set_mode=1;                       // 进入设置模式
```

```
        set_mode_to=TIMEOUT;              // 设置间隔时间
        cur_field=HOUR;                   // 选择第一个位置
        set_cursor(ON, HOUR);             // 使能光标
      }else {
         cur_field++;                     // 移到下一个位置
         if (cur_field>SEC) {             // 如果移过最后一个位置
                                          // 结束设置模式
           set_mode=0;                    // 离开设置模式
           set_mode_to=0;
           set_cursor(OFF, HOME);         // 禁能光标
         }else {
            set_cursor(ON, cur_field);  // 光标移到下一个位置
            set_mode_to=TIMEOUT;
          }
       }
    }
    if (set_mode && !SELECT) {            // 如果按下选择键
      set_mode_to=TIMEOUT;
      incr_field();                       // 选择下一个位置
      disp_time();                        // 显示更新的时间
    }
  }
  if (!set_mode) {                        // 当处于设置模式时，停止时钟
    second_cnt--;                         // 时间计数器减1
    if (!second_cnt) {                    // 如果过了1秒种...
      second_cnt=20;                      // 重置计数器
      second_tick();
    }
  }
}
```

7 改进时钟软件

在这里你可以开始消除系统时标中的误差。你应该记得误差是由从定时器溢出到定时器重装，并开始运行之间的代码延时引起的。为了消除误差，先用 C51 代码选项汇编这段函数，然后计算启动定时器所需要的时钟周期数，最后再加上进入中断所需的 2 个周期数。你可能会决得当处理器在进行 DIV 或 MUL 操作时检测到中断要花 3 个或更多的周期，但是毕竟没有快速而可靠的方法来确定处理器检测到中断的准确时间。下面是汇编后的指令列表，我已经加入了指令计数。

列表 0-8

```
                ; FUNCTION system_tick (BEGIN)
0000 C0E0             PUSH ACC          2, 2
0002 C0F0             PUSH B            2, 4
0004 C083             PUSH DPH          2, 6
0006 C082             PUSH DPL          2, 8
```

```
0008 C0D0            PUSH PSW          2, 10
000A C000            PUSH AR0          2, 12
000C C001            PUSH AR1          2, 14
000E C002            PUSH AR2          2, 16
0010 C003            PUSH AR3          2, 18
0012 C004            PUSH AR4          2, 20
0014 C005            PUSH AR5          2, 22
0016 C006            PUSH AR6          2, 24
0018 C007            PUSH AR7          2, 26
                                       ; SOURCE LINE # 332
                                       ; SOURCE LINE # 335
001A C28C            CLR TR0           1, 27
                                       ; SOURCE LINE # 336
001C 758C3C          MOV TH0,#03CH     2, 29
                                       ; SOURCE LINE # 337
001F 758AAF          MOV TL0,#0AFH     2, 31
                                       ; SOURCE LINE # 338
0022 D28C            SETB TR0          1, 32
                                       ; SOURCE LINE # 340
```

从指令计数可以知道一共损失了 34（32+2）个指令周期。我们注意到大部分损失的时间是由于把寄存器入栈。因为每个入栈指令又要对应一条出栈指令，这样就要花去 52 个指令周期。这使编译器所做的一种数据保护措施，我们可通过指定寄存器组来消除这种保护措施。

另一个耗时的功能是 printf 函数。仿真显示当准备好接收显示字符时，传送字符串进行显示，需要消耗 6039 个指令周期。我们因此认为 printf 和 putchar 函数的执行时间是 6039 个指令周期，相当于 6.093ms，在每次中断之间执行这个过程并不会导致系统的不稳定。为了确认这点，我们对中断程序进行仿真，当时间从 23：59：59 变为 00：00：00 时（这代表了非设置模式的中断最长执行时间），中断的执行时间是 207 个处理周期，相当于.207ms,当没有时间改变时中断的时间为.076ms。

因为是每 50ms 进行一次中断，那么进行时间更新和显示的时间加起来不过是 6.246ms，在下一次进行中断之前，有 43.754ms 是在空闲模式。如果你的功能只有这些，或许你的系统是用电池供电，减少处理器工作时间的最佳方法是用一个更加精简的函数替代 printf 函数。

因为系统除了显示时间外不需显示其它信息，你可以大大的简化 printf 函数。它不需要处理串行格式化，字符格式化，整型，长整型或浮点数。你可假定只有某一部分的数值需要改变。printf 的替代函数对一个缓冲区进行处理，这个缓冲区包括已经格式化过的字符串，只要把更新的字符插入真确的位置就可以了，为了加快执行的时间，通过查表来得到要显示的字符，这是执行时间和存储空间的交换。因为这个程序比较小（2000 字节以内），有充足的空间。如果不是这样的话，你就需要在中断中进行 BCD 码和 ASCII 码的转化，这样中断程序将占用超过 76 个指令周期的时间。

disp_time 函数将代替 printf 函数，我们不再需要进行字符串初始化和 3 个参数的传递，只需在缓冲区中修改显示字符，并把一个字节传送给 putchar 函数。编程的复杂程度增加了，但即使在增加了 120 个字节的字符表后，代码的长度仍然从 1951 个字节减少到 1189 字节。printf 函数占用了 811 个字节，而 disp_time 函数占用了 105 个字节。下面是 disp_time

函数。

列表 0-9

```
void disp_time(void) {
   // 显示保存的当前时间
   // 当时间数据使用完毕后才清除使用标志位
   // 这避免了数据在使用中被修改
   static char time_str[32]="TIME OF DAY IS: XX:XX:XX ";
   unsigned char I;
   time_str[T_HOURT]=bcdmap[timeholder.hour][0];
   time_str[T_HOUR]=bcdmap[timeholder.hour][1];
   time_str[T_MINT]=bcdmap[timeholder.min][0];
   time_str[T_MIN]=bcdmap[timeholder.min][1];
   time_str[T_SECT]=bcdmap[timeholder.sec][0];
   time_str[T_SEC]=bcdmap[timeholder.sec][1];
   putchar(0xFF);
   for (i=0; i<32; i++) {
     putchar(time_str[i]);
   }
   disp_update=0;       // 清除显示更新标志位
}
```

disp_time 的处理时间为 2238 个指令周期，对 12MHz 系统来说就是 2.238ms。清除显示要花 1.64ms，把 32 个字符送显示要花 1.28ms，每次更新的显示的延时是 2.92ms。如果每秒刷新一次显示的话，则每秒的中断处理时间为 6.866ms。其中包括 76x19 周期(每秒中有 19 次中断)的中断执行时间，207 周期的时间数据更新时间，2238 周期的显示时间再加上 2.92ms 的显示延迟时间。可以看出系统在大部分时间处于空闲模式。

8 优化内部 RAM 的使用

这个系统还没有考虑的另一个缺点是它还没有优化内部 RAM 的使用。通过 M51 得到的数据段存储区列表文件如下：

```
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"
         000AH    0016H                *** GAP ***
BIT      0020H.0  0000H.2 UNIT         ?BI?CH4CLOCK
BIT      0020H.2  0000H.1 UNIT         "BIT_GROUP"
         0020H.3  0000H.5              *** GAP ***
DATA     0021H    002BH   UNIT         ?DT?CH4CLOCK
IDATA    004CH    0001H   UNIT         ?STACK
```

似乎不是很明显,数据段的 0AH 到位寻址段的开始位置 20H 的 22 个字节寄存器没有被利用，这是因为连接器不能把 CH4CLOCK 模块的变量放在这个这个数据段中。这样就使得堆栈变小了，系统更容易发生溢出错误。

之所以发生这种情况是因为你把所有变量都定义在一个文件当中（ch4clock.c）。一种

解决办法是在连接的时候使用指令指定哪些变量将存储在数据段的底部。直接的方法是告诉连接器这个文件的所有变量都存储在数据段的底部。我们在连接选择对话框中选择 tab 项，然后在预编译控制中写入下面的数据

? DT? ch4clock

当你所指定的存储变量不超过寄存器组的最高地址的寄存器和位寻址区的最低地址时，这样做是最好的。但是，假设你定义了位变量，而变量的存储区又超出了上面所说的范围，那么你的位变量就将被覆盖，并发生连接错误。为此你必须把一部分变量移到另外一个文件中。这将产生两个小的数据段，然后你可以使用连接指令定义他们的存储位置。通过这种方法，你可以完全消除数据沟。

另一个文件单独编译并和主文件一起连接。在这里 22 个字节的变量被移到另一个文件中。在时钟系统这个小程序中，只有 9 个字节的变量被移到另一个文件中。结果如下：

```
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   0022H   UNIT       ?DT?CH4NEW
BIT    002AH.0 0000H.2 UNIT       ?BI?CH4NEW
BIT    002AH.2 0000H.1 UNIT       _BIT_GROUP_
       002AH.3 0000H.5            *** GAP ***
DATA   002BH   0009H   UNIT       ?DT?VARS
DATA   0034H   0004H   UNIT       _DATA_GROUP_
IDATA  0038H   0001H   UNIT       ?STACK
```

从上面可以看出，编译器流下了 72 个字节的堆栈空间（80H-28H），数据沟也不见了。你现在必须确认 72 个字节的空间对你的系统已经足够。我们可以算一下，从前面可知 disp_time 调用需要花去 13 个字节,把 PC 入栈要 2 个字节,中断调用花去 2 个字节,disp_time 调用 putchar,而 putchar 又调用 disp_cmd，disp_cmd 再调用 disp_read，这又需要 4 个字节，总共花去 25 个字节。仍然有 47 字节的空间剩余，这说明连接器给出的堆栈空间是足够的。

9 完整的程序

到此为止,这个时钟程序算是完成了。实现了对硬件的简化。整个程序如下所示：

列表 0-10

```
#include<reg51.h>
#include<stdio.h>
//定义定时器 0 的重装值
#define RELOAD_HIGH   0x3C
#define RELOAD_LOW    0xD2

//定义按键弹跳时间
#define DB_VAL
//定义设置模式的最大时间间隔
#define TIMEOUT        200

//定义光标位置常数
#define HOME             0
```

```
#define HOUR            1
#define MIN             2
#define SEC             3

//定义光标状态常数
#define OFF           0
#define ON            1

//定义显示命令常数
#define DISP_BUSY    0x80
#define DISP_FUNC    0x38
#define DISP_ENTRY   0x06
#define DISP_CNTL    0x08
#define DISP_ON      0x04
#define DISP_CURSOR  0x02
#define DISP_CLEAR    0x01
#define DISP_HOME     0x02
#define DISP_POS       0x80
#define DISP_LINE2     0x40

sbit SET=P3^4;              //设置按键输入
sbit SELECT=P3^5;           //选择按键输入
sbit ENABLE=P3^1;           //显示使能输出
sbit REGSEL=P3^7;           //显示寄存器选择输出
sbit RDWR=P3^6;             //显示模式输出

sfr DISPDATA=0x90;         //显示 8 位数据总线

typedef struct {               //定义存储每日时间的结构
  unsigned char hour,min,sec;
}timestruct;

bit set_mode=0;               //进入设置模式时置位
   disp_updata=0;             //需要刷新显示时置位


unsigned char set_mode_to=0;     //为每次按键操作的时间间隔计时
            switch_debounce=0;   //按键跳动计时
            cur_field=HOME;      //设置模式的当前位置选择

timestruct    curtime;             //存放当前的时间
            timeholder;            //存放显示时间

unsigned char code fieldpos[3]={     //
```

```
        DISP_LINE2|0x01;
        DISP_LINE2|0x04;
        DISP_LINE2|0x07;
};

#define T_HOURT      16
#define T_HOUR       17
#define T_MINT       19
#define T_MIN        20
#define T_SECT       22
#define T_SEC        23

char code bcdmap[60][2]={
“00”,”01”,”02”,”03”,”04”,”05”,”06”,”07”,”08”,”09”,
“10”,”11”,”12”,”13”,”14”,”15”,”16”,”17”,”18”,”19”,
“20”,”21”,”22”,”23”,”24”,”25”,”26”,”27”,”28”,”29”,
“30”,”31”,”32”,”33”,”34”,”35”,”36”,”37”,”38”,”39”,
“40”,“41”,”42”,”43”,”44”,”45”,”46”,”47”,”48”,”49”,
“50”,”51”,”52”,”53”,”54”,”55”,”56”,”57”,”58”,”59”,
};
//函数声明
void disp_cmd(unsigned char);
void disp_init(void);
unsigned char disp_read(void);
void disp_time(void);
void disp_write(unsigned char);
void incr_field(void);
void second_tick(void);
void set_cursor(bit,unsigned char);

/*******************************************************
功能:主函数
描述:程序入口函数,初始化 8051,开中断,进入空闲模式。每次中断之后查询标
志位，是否刷新显示
参数：无
返回：无
*******************************************************/
void main(void){
  disp_init();              //显示初始化
  TMOD=0x11;           //设置定时器模式
  TCON=0x15;
  IE=0x82;
  For(;;)
  {
```

```
      if (disp_updata){
         disp_time( );        //显示新时间
       }
       PCON=0x01;
   }
}
/*************************************************
功能：disp_cmd
描述：向 lcd 驱动器写入命令，并等待命令被执行
参数：命令，
返回：无
*************************************************/
void disp_cmd(unsigned char cmd){
  DISPDATA=cmd;        //锁住命令
  REGSEL=0;            //选择命令寄存器
  RDWR=0;              //选择写模式
  ENABLE=1;
  ENABLE=0;
  TH1=0;                //定时 85ms
  TL1=0;
  TF1=0;
  TR1=1;
  while(!TF1&&disp_read()&DISP_BUSY); //等待命令被执行
  TR1=0;
}


/*************************************************
功能:disp_init
描述:初始化显示
参数:无
返回:无
*************************************************/
void disp_init(void){
  TH1=0;
  TL1=0;
  TF1=0;
  TR1=1;
  while (!TF1&&disp_read()&DISP_BUSY);
  TR1=0;
  disp_cmd(DISP_FUNC);              //设置显示格式
  disp_cmd(DISP_ENTRY);             //每输入一个字符,显示地址加 1
  disp_cmd(DISP_CNTL|DISP_ON);      //打开显示,关闭光标
  disp_cmd(DISP_CLEAR);             //清除显示
```

```
}


/*****************************************************
功能:disp_read
描述:读显示状态寄存器
参数:无
返回:从状态寄存器中读回的数据
*****************************************************/
unsigned char disp_read(void){
  unsigned char value;
  DISPDATA=0XFF;
  REGSEL=0;                  //选择命令寄存器
  RDWR=1;                   //选择读模式
  ENABLE=1;                //使能 LCD 输出
  value=DISPDATA;           //读数据
  ENABLE=0;
  retrun(value);
}

/*****************************************************
功能:disp_time
描述:取显示数据进行格式化
参数:无
返回:无
*****************************************************/
void disp_time(void){
  static char time_str[32]= “TIME OF DAY IS:XX:XX:XX        ”;
  unsigned char I;
  time_str[T_HOURT]=bcdmap[timeholder.hour][0];
  time_str[T_HOUR]=bcdmap[timeholder.hour][1];
  time_str[T_MINT]=bcdmap[timeholder.min][0];
  time_str[T_MIN]=bcdmap[timeholder.min][1];
  time_str[T_SECT]=bcdmap[timeholder.sec][0];
  time_str[T_SEC]=bcdmap[timeholder.sec][1];
  putchar(0xFF);
  for(i=0;i<32;i++){
    putchar(time_str[i]);
  }
  disp_updata=0;
}

/*****************************************************
功能:disp_write
```

```
描述:写入一个字节数据
参数:要写入的字节
返回:无
*************************************************/
void disp_write(unsigned char value){
  DISPDATA=value;
  REGSEL=1;
  RDWR=0;
  ENABLE=1;
  ENABLE=0;
}
/************************************************
功能:incr_field
描述:增加数值
参数:无
返回:无
*************************************************/
void incr_field(void){
  if (cur_field= =SEC){
    curtime.sec++;
    if(curtime.sec>59){
     curtime.sec=0;
    }
  }
  if (cur_field= =MIN){
    curtime.min++;
    if(curtime.min>59){
     curtime.min=0;
    }
  }
  if (cur_field= =HOUR){
    curtime.hour++;
    if(curtime.hour>23){
     curtime.hour=0;
    }
  }
}

/************************************************************
功能:putchar
描述:替代标准 putchar 函数,输出字符
参数:要显示的字符
返回:刚刚被写的字符
************************************************************/
```

```
char putchar(char c){
  static unsigned char flag=0;
  if(!flag||c= =255){
    disp_cmd(DISP_HOME);
    flag=0;
    if(c= =255){
       return c;
    }
  }
  if(flag= =16){
    disp_cmd(DISP_POS|DISP_LINE2);
  }
  disp_write(c);
  while(disp_read( )&DISP_BUSY);
  flag++;
  if (flag>=32){flag=0};
  return(c);
}

/************************************************************
功能:second_tick
描述:每秒钟执行一次函数功能,时间更新
参数:无
返回:无
************************************************************/
void second_tick(void){
   curtime.sec++;                 //秒种加 1
   if (curtime.sec>59){           //检测是否超出范围
    curtime.sec=0;
    crutime.min++;                //分钟加 1
    if (curtime.min>59){          //检测是否超出范围
      curtime.min=0;
      curtime.hour++;             //小时数加 1
      if(curtime.hour>23){        //检测是否超出范围
        curtime.hour=0;
      }
     }
    }
    if(!disp_updata){             //确信 timeholder 没有被显示
     timeholder=curtime;          //装入新时间
     disp_updata=1;               //更新显示
    }
   }
```

```
/*************************************************
功能:set_cursor
描述:显示或关闭光标,并把光标移到特定的位置
参数:new_mode 位,隐藏光标时置位
     field    显示光标的位置
返回:无
*************************************************/
void set_cursor(bit new_mode,unsigned char field){
    unsigned char mask;
    mask=DISP_CNTL|DISP_ON;
    if(new_mode){
      mask|=DISP_CURSOR;
    }
   disp_cmd(mask);
   if (field= =HOME){
     mask=DISP_HOME;
   }else{
     mask=DISP_POS|fieldpos[field-1];
   }
   disp_cmd(mask);
}

/*****************************************************
功能: system_tick
描述:定时器 0 的中断服务程序,每 50ms 重装一次定时器
参数:无
返回:无
*****************************************************/
void system_tick(void) interrupt1{
  static unsigned char second_cnt=20;
  TR0=0;
  TH0=RELOAD_HIGH;                //设定重装值
  TL0=RELOAD_LOW;
  TR0=1;                          //开始定时
  if(switch_debounce){            //按键抖动
    switch_debounce--;
  }
  if (!switch_debounce){
    if(!SET){                      //如果设置按钮被按下
     switch_debounce=DB_VAL;       //设置消抖时间
     if(!set_mode&&!disp_updata){  //如果不是设置模式
       set_mode=1;                 //进入设置模式
       set_mode_to=TIMEOUT;        //设置空闲时间
       cur_field=HOUR;             //选择光标起始位置
```

```
            set_cursor(ON,HOUR);          //使能光标
        }else{
            cur_field++;                  //光标位置前进
            if(cur_field>SEC){            //光标是否超出范围
              set_mode=0;                 //退出设置模式
              set_mode_to=0;
              set_cursor(OFF,HOME);       //禁能光标
             }else{
              set_cursor(ON,cur_field);   //光标移到下一个位置
              set_mode_to=TIMEOUT;
             }
            }
           }
        if(set_mode&&!SELECT){          //如果按下选择按钮
          set_mode_to=TIMEOUT;
          incr_field( );                 //所选择处数值增加
          disp_time( );                 //显示时间
          }
         }
        if(!set_mode){                  //设置模式停止时钟
          second_cnt- -;                 //计数值减 1
          if(!second_cnt){               //如果经过 1 秒
            second_cnt=20;               //设置计数值
            second_tick( );
            }
         }
    }
```

10 使用看门狗定时器

很多嵌入式系统利用查询,等待的方法和外部设备进行通信或花大量的时间在循环中处理数据。一直在这种状态下运行对系统来说是很苛刻的。嵌入式系统不应该陷入死循环中，否则将影响系统的正常工作。引起死循环的原因有很多，如 I/O 设备的错误，接收了错误的输入或软件设计中的 bug。不管原因是什么，它都将使你的系统不稳定。

作为一种保护，很多设计者都使用看门狗定时器。看门狗定时器从某一个值开始计时，在它溢出前，必须由软件重装，否则将认为软件运行已经进入死循环或其它一些意想不到的情况，系统将自动复位。设计者编写软件来处理看门狗，并在看门狗定时器溢出之前调用它。这些软件相对来说是比较容易编写的。但必须按照特定的规则。下面是一个初始化和重装 Pilips80C550 看门狗定时器的例子。

```
void wd_init(unsigned prescale){
  WDL=0xFF;                          //把重装值设置为最大
  WDCON=(prescale&0xE0)|0x05;        //定时器预分频为最慢并启动看门狗
  wd_reload();
}
void wd_reload(void){
```

```
  EA=0;                          //关闭所有中断
  WPEED1=0xA5;                   //喂看门狗第一步
  WFEED2=0x5A;                   //喂看门狗第二步
  EA=1;                          //开中断
}
```

一般来说，你可以把这段程序放在有周期性中断的系统的主循环中。如果你的系统能被其它的中断打断，并影响到你主循环的执行，从而不能够定时的重装你的看门狗定时器，这时，你应该在中断程序中也放置清看门狗程序。然而不是每次执行中断时都执行看门狗程序，而是应该对中断执行的次数计数，当达到一定的数值时，再执行看门狗程序。如下面的例子。

列表 0-12

```
void main(void){
  …                              //初始化
  for(;;){                       //主循环等待中断
    …
    wd_reload( );                //重装看门狗
    PCON=0x80;                   //进入空闲模式等待中断
  }
}
void my_ISR(void)interrupt 0{
  static unsigned char int_count=0; //中断次数计数
  int_count++;
  if(int_count>MAXINTS){         //中断次数到了
    int_count=0;
    wd_reload( );                //重装看门狗
  }
  …
}
```

看门狗定时器的复位和正常的上电复位时是不同的。如果你的程序执行过程中产生了数据，你应该在外部 RAM 中倍份它们，除非你确定每次程序开始执行时不需要初始化它们。系统应该知道在何时保存正常运行时产生的数据。

12 保存系统数据

系统应根据先前的状态决定不同的复位方式。例如，你的程序运行正常，但是被看门狗或外部复位键复位，你应该采取和上电复位不同的初始化过程。一般来说，看门狗复位和用户复位是是热启动。在 8051 系统中没有任何 RAM 的备用电池，这种复位很容易通过检测标志位来区分。

当系统首次执行代码时，标志位检测为特定值。如果值不对的话，就将进行上电初始化，如果值是对的，就将只进行所需要的初始化。一旦系统被初始化，热启动标志被设置成特定值。值的选择应避免使用 00 或 FF，否则就难以区分冷启动和热启动。我们应选择像 AA 或 CC 这样的值。对必须在内部 RAM 中保存数据的系统，必须在编译的启动代码中检测标志，这意味着你必须修改 startup.a51。对可以在外部 RAM 中保存数据的系统来说，如果你的标志位保存在外部 RAM 中，你就不需要改动 startup.a51 了。因为默认时由 startup.a51 编译过来的代码只会初始化内部 RAM 中的数据，而不会置 0 外部 RAM 中的数

据。如果你把标志位保存在内部 RAM 中，而没有在内部 RAM 被置 0 前检测它，将导致系统冷启动。

下面是一个启动时对标志位作检测的例子。

列表 0-13：

```
unsigned char xdata bootflag;
void main(void){
  ...
  if (bootflag!=0xAA){              //系统是否冷启动
    init_lcd();                     //初始化显示
    init_rtc();                     //初始化时钟
    init_hw();                      //设置 I/O 端口
    reset_queue();                  //复位数据结构
    bootflag=0xAA;                  //设置热启动标志
  }else{
     clear_lcd();                   //清除显示
  }
  ...
}
```

对只能在内部 RAM 中保存数据的系统来说，必须修改 startup.a51 文件以确保程序只清除被编译器使用的和不需要被系统记住的区域。被修改的 startup.a51 如下所示：

列表 0-14

```
;------------------------------------------------------------------
; This file is part of the C-51 Compiler package
; Copyright (c) KEIL ELEKTRONIK GmbH and Keil Software, Inc.,
; 1990-1992
;------------------------------------------------------------------;STARTUP.A51:
This code is executed after processor reset.
;
; To translate this file use A51 with the following invocation:
;
; A51 STARTUP.A51
;
; To link the modified STARTUP.OBJ file to your application use
    ; the following L51 invocation:
;
; L51 <your object file list>, STARTUP.OBJ <controls>
;
;------------------------------------------------------------------
;
; User-defined Power-On Initialization of Memory
;
; With the following EQU statements the initialization of memory
; at processor reset can be defined:
;
```

```
EXTRN DATA (bootflag)
;
; the absolute start-address of IDATA memory is always 0
IDATALEN    EQU    80H     ; the length of IDATA memory in bytes.
;
XDATASTART  EQU    0H      ; the absolute start-address of XDATA
                           ; memory
XDATALEN    EQU    0H      ; the length of XDATA memory in bytes.
;
PDATASTART  EQU    0H      ; the absolute start-address of PDATA
                           ; memory
PDATALEN    EQU    0H      ; the length of PDATA memory in bytes.
;
; Notes: The IDATA space overlaps physically the DATA and BIT
;        areas of the 8051 CPU. At minimum the memory space
;        occupied from the C-51 run-time routines must be set
;        to zero.
;------------------------------------------------------------------
;
; Reentrant Stack Initilization
;
; The following EQU statements define the stack pointer for
; reentrant functions and initialized it:
;
; Stack Space for reentrant functions in the SMALL model.
IBPSTACK      EQU    0        ; set to 1 if small reentrant is used.
IBPSTACKTOP   EQU    0FFH+1   ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the LARGE model.
XBPSTACK      EQU    0        ; set to 1 if large reentrant is used.
XBPSTACKTOP   EQU    0FFFFH+1 ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the COMPACT model.
PBPSTACK      EQU    0        ; set to 1 if compact reentrant is used.
PBPSTACKTOP   EQU    0FFFFH+1 ; set top of stack to highest location+1.
;
;------------------------------------------------------------------
;
; Page Definition for Using the Compact Model with 64 KByte xdata
; RAM
;
; The following EQU statements define the xdata page used for
; pdata variables. The EQU PPAGE must conform with the PPAGE
; control used in the linker invocation.
```

```
;
PPAGEENABLE     EQU   0       ; set to 1 if pdata object are used.
PPAGE           EQU   0       ; define PPAGE number.
;
;------------------------------------------------------------------
                NAME        ?C_STARTUP
?C_C51STARTUP   SEGMENT     CODE
?STACK          SEGMENT     IDATA
RSEG ?STACK
                DS 1

                EXTRN       CODE (?C_START)
                PUBLIC      ?C_STARTUP
                CSEG        AT 0
?C_STARTUP:     LJMP        STARTUP1

                RSEG        ?C_C51STARTUP

STARTUP1:
                MOV         A, bootflag     ; check if RAM is good
                CJNE        A, #0AAH, CLRMEM
                SJMP        CLRCOMP         ; RAM is good, clear only
                                            ; compiler owned locations
CLRMEM:                                     ; RAM was not good,
                                            ; zero it all
IF IDATALEN <> 0
                MOV         R0,#IDATALEN - 1
                CLR         A
IDATALOOP:      MOV         @R0,A
                DJNZ        R0,IDATALOOP
                JMP         CLRXDATA
ENDIF

CLRCOMP:        CLR         A               ; zero out compiler owned
                                            ; areas
                MOV         20H, A
                MOV         R0, #3EH
L1:             MOV         @R0, A
                INC         R0
                CJNE        R0, #76H, L1
CLRXDATA:

IF XDATALEN <> 0
                MOV         DPTR,#XDATASTART
```

```
                MOV         R7,#LOW (XDATALEN)
   IF (LOW (XDATALEN)) <> 0
                MOV         R6,#(HIGH XDATALEN) +1
   ELSE
                MOV         R6,#HIGH (XDATALEN)
   ENDIF

                CLR         A
XDATALOOP:      MOVX        @DPTR,A
                INC         DPTR
                DJNZ        R7,XDATALOOP
                DJNZ        R6,XDATALOOP
ENDIF

IF PDATALEN <> 0
                MOV         R0,#PDATASTART
                MOV         R7,LOW (PDATALEN)
                CLR         A
PDATALOOP:      MOVX        @R0,A
                INC         R0
                DJNZ        R7,PDATALOOP
ENDIF

IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)
                MOV         ?C_IBP,#LOW IBPSTACKTOP
ENDIF

IF XBPSTACK <> 0
EXTRN DATA (?C_XBP)
                MOV         ?C_XBP,#HIGH XBPSTACKTOP
                MOV         ?C_XBP+1,#LOW XBPSTACKTOP
ENDIF

IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)
                MOV         ?C_PBP,#LOW PBPSTACKTOP
ENDIF

IF PPAGEENABLE <> 0
                MOV         P2,#PPAGE
ENDIF

                MOV         SP,#?STACK-1
```

```
        LJMP    ?C_START
        END
```

检测启动标志,如果标志符合，那么只清除编译器使用到的那部分存储区。程序的所有局部变量必须在用户产生代码中清晰的处理。库函数的地址通过检察连接输出文件和清除那些存储段来决定。正如你所见，地址 20H 的位变量，3EH 到 75H 的存储区必须被清零。上面 startup.a51 的连接输出文件如下所示：

```
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   0012H   UNIT       ?DT?VARS
DATA   001AH   0001H   UNIT       ?DT?PUTCHAR
001BH  0005H                      *** GAP ***
DATA   0020H   0001H   BIT_ADDR   ?C_LIB_DBIT
BIT    0021H.0 0000H.5 UNIT       ?BI?COINOP
BIT    0021H.5 0001H.2 UNIT       "BIT_GROUP"
       0022H.7 0000H.1            *** GAP ***
DATA   0023H   001BH   UNIT       ?DT?COINOP
DATA   003EH   000FH   UNIT       ?C_LIB_DATA
DATA   004DH   0029H   UNIT      "DATA_GROUP"
IDATA  0076H   001EH   UNIT       ?ID?COINOP
IDATA  0094H   0001H   UNIT       ?STACK
```

另外一种存储你的内部变量而不用去考虑哪里是安全的，哪里会被清零是把变量存储在外部 RAM 中。这当然是指你有外部 RAM 的情况下，如果没有也可以用 EEPROM 或 flash 存储器代替，这样会更加可靠。但一般都会使用 RAM，因为 RAM 比 EEPROM 要快，当处理器接收到关闭中断时，系统要把所有有效的变量都存储到外部 RAM 中。中断被击活时，系统有足够的时间把变量存入 SRAM 中，并进入低功耗模式。而 EEPROM 则是一个很慢的器件，不能满足这个要求。如果你需要保存的数据不会经常改变,那么可在存储区中倍份这个数据，当源数据改变时，倍份数据也要改变。如果数据经常被改变的话，这种方法就不可行了。

不管采用何种方法，当系统重新上电后，检测一个数据字节（像前面所讨论的启动标志），如果数据正确就恢复内部变量。这些都在系统初始化时的条件循环中完成。

13 结论

这一章展示了一些如何减少硬件并减轻硬件工作压力的方法，当然方法远远不止这些，这里只是告诉你一些技巧。在不少情况下，可以用软件来代替硬件的工作，因此可以简化硬件的设计。要完全掌握这些方法要花大量的时间，你应该不断的学习以提高自己的水平。

第四章 在 8051 上使用汇编和 C

1 介绍

在一些时候你会发现不得不使用汇编来编写程序，而不是使用高级语言。而大多数情况下，汇编程序能和用 C 编写的程序很好的结合在一起。这章将告诉你如何进行汇编和 C 的混合编程，并且如何修改由 C 程序编译后的汇编代码，从而精确的控制时间。

2 增加段和局部变量

要把汇编程序加入到 C 程序中，你必须使你的汇编程序像 C 程序，这就是说要和 C 程序一样有明确的边界，参数，返回值和局部变量。

一般来说用汇编编写的程序变量的传递参数所使用的寄存器是无规律的。这使得在用汇编语言编写的函数之间传递参数变得混乱，难以维护。使的汇编功能函数看上去像 C 函数，并按照 C51 的参数传递标准，可让你的程序有很好的可读性并有利于维护。而且你会发现这样编写出来的函数很容易和 C 编写的函数进行连接。如果你用汇编编写的函数和 C 编译器编译出来的代码风格一样的话，连接器将能够对你的数据段进行覆盖分析。

汇编程序中，你的每一个功能函数都有自己的代码段。如果有局部变量的话，他们也都有相应的存储空间（DATA，XDATA 等）。例如，你有一个需要快速寻址的变量，你可把它声明在 DATA 段中，如果你有函数查寻表格的话，你可把它们声明在 CODE 段中。关键是局部变量只对当前使用他们的程序段是可见的。下面的例子中，一个功能段在 DATA 区中定义了几个局部变量。

列表 0-1

```
; declare the code segment for the function
?PR?IDCNTL?IDCNTL SEGMENT CODE

; declare the data segment for local storage
; this segment is overlayable for linker optimization
; of memory usage
?DT?IDCNTL?IDCNTL SEGMENT DATA OVERLAYABLE
                  PUBLIC idcntl
                  PUBLIC ?idcntl?BYTE
; define the layout of the local data segment
                  RSEG ?DT?IDCNTL?IDCNTL
?idcntl?BYTE:
TEMP:             DS     1
COUTNT:           DS     1
VAL1:             DS     2
VAL2:             DS     2

                  RSEG ?PR?IDCNTL?IDCNTL
idcntl:           ... ; function code begins here

                  RET
```

DATA数据段中的标号就像汇编程序中的变量一样,连接器在连接的时候会赋予它们物理地址。段的覆盖属性将允许连接器进行覆盖分析。没有这个属性，?idcntl?BYTE段中的变量将一直占用这些空间，就像C中的静态变量一样，这样将使内存的效率降低。

3 设置变量地址

有时候我们希望把变量存储在指定的地点，特别是在主控制器初始化SRAM之后，从8051系统才开始工作的情况。在这种情况下，两个系统必须在存储器分配上达成一致，否则当8051使用不正确地使用初始化过的数据时将导致数据丢失。因此8051必须确保变量被存储在正确的区域。如果你不想在编译时才给变量分配地址，Keil C可以让你指定变量的存储地址。例如，你想定义一个整型变量，并把它初始化为0x4050，用C是不能够把变量指定在某个地址的，另外你也不能指定位变量的地址。但是，对于不需要初始化的变量，你可以使用关键字_at_来指定地址。

```
type [memory_space] variable_name _at_ constant;
```

如果不指定地址的话,将由选择编译的模式来指定默认的地址。假设你以小模式编译，你的变量将分配在DATA段中。下面是一个指定地址的例子：

```
unsigned char data byteval _at_ 0x32;
```

关键字_at_的另一个有趣的功能是能通过给I/O器件指定变量名为你的输入输出器件指定变量名。例如你在XDATA段的地址0x4500处有一个输入寄存器，你可以通过下面的代码为它指定变量名：

```
unsigned char xdata inpreg _at_ 0x4500;
```

以后在读该输入寄存器的时候只要使用变量名inpreg就可以了，当然，你也可以用Keil C提供的宏来完成。如列表0-2的例子。

当你想为指定地址的变量初始化时，你可使用传统汇编的方法。有时候需要查表，如果把表的基址定义在某个地址的话可以简化你的寻址过程，但由于在代码段中，它的地址在编译的时候决定。假设你有一个256字节的表，想对它进行快速寻址，你可以使用列表0-3的方法：

列表 0-2

```
void myfunc(void) {
  unsigned char inpval;
  inpval=inpreg;        // 这行和下行是一样的
  inpval=XBYTE[0x4500];
  ...
  if (inpreg & 0x40) { 根据输入的值做决定
  ...
  }
}
```

列表 0-3

```
; 取得表地址的高字节
MOV DPH, #HIGH mytable
MOV DPL, index
CLR A
MOVC A, @A+DPTR        ;读数
```

把变量地址放在给定段中是一种简单的方法来定义那些不能被连接器重定位的段，并且指定它的起始地址。上例中的表头地址可被定义在8000H中。另外还可在DATA段中放置变量。

列表 0-4

```
            ;定义代码段
            CSEG AT 8000H
    mytable: DB 1, 2, 3, 4, 5, 6, 7, 8
            ... ;剩下的表格定义
            DSEG AT 70H
            cur_field: DS 1
            ...
            END ;
```

用这种方法,一个变量可被安置在任何地方。如果你在你的C代码中用extern申明了这些变量，那么你的C代码可对他们进行寻址。有了这些信息之后，连接器将能够定位你的变量。

还有一种方法用来定位中断服务程序，可以把中断入口向量或中断程序放在一个绝对段中。如果你所有的中断服务程序都是用C写的，但是有一个中断程序必须用汇编来写，最好的方法就是把中断程序定位在正确的位置上,这和给变量设置地址很相似。

列表0-5

```
    CSEG AT 023H
    LJMP serial_intr
```

由中断向量调用的中断服务程序就像其它过程一样在代码段中

列表 0-6

```
                ; 定义可重定位段
                RSEG ?PR?serial_intr?SERIAL

                USING 0             ; 使用寄存器组0
    serial_intr: PUSH ACC           ; 中断服务程序
                ...
                RETI
```

4 结合C和汇编

假设你要执行的操作很难用C代码来完成，如使用BCD码，你会觉得用汇编来编写代码比用C更加有效率，还有就是对时间要求很严格的功能，用C来编程不是很保险，你希望用汇编来做，但是又不愿意仅仅因为这么一小部分就把整个程序都用汇编来做，这样你就必须学会把汇编编写的程序和C编写的程序连接起来。

给用汇编编写的程序段指定段名和进行定义，这将使汇编程序段和C程序兼容。如果你希望在它们之间传递函数，那你必须保证汇编程序用来传递函数的存储区和C函数使用的存储区是一样的。下面是一个典型的可被C程序调用的汇编函数，该函数不传递参数。

列表 0-7

```
;申明代码段
?PR?clrmem?LOWLVL SEGMENT CODE
;输出函数名
PUBLIC clrmem
;这个函数可被连接器放置在任何地方
RSEG ?PR?clrmem?LOWLVL
;********************************************************************
; Function: CLRMEM
```

```
; Description: 清除内部RAM区
; Parameters: 无
; Returns: 无.
; Side Effects: 无.
;*********************************************************************
clrmem:     MOV R0,#7FH
            CLR A
IDATALOOP: MOV @R0,A
            DJNZ R0,IDATALOOP
            RET
            END
```

汇编文件的格式化是很简单的。给存放功能函数的段一个段名。因为是在代码区内，所以段名的开头为？PR。这头两个字符是为了和C51的内部命名转换兼容。见表0-1：

段名被赋予了RSEG的属性。这意味着连接器可把该段放置在代码区的任意位置。一旦段名被确定，文件必须申明公共符号然后编写代码。对于传递参数的功能函数必须符合参数的传递规则。Keil C在内部RAM中传递参数时一般都是用当前的寄存器组。当你的功能函数接收3个以上参数时，存储区中的一个默认段将用来传递剩余的参数。用做接收参数的寄存器如下表。

Memory Space	Naming Convention
CODE	?PR, ?CO
XDATA	?XD
DATA	?DT
BIT	?BI
PDATA	?PD

表0-1

	Parameter Type			
Parameter #	char, 1 byte ptr	int, 2 byte ptr	long, float	generic ptr
1	R7	R6, R7	R4...R7	R1, R2, R3
2	R5	R4, R5	R4...R7	R1, R2, R3
3	R3	R2, R3	N/A	R1, R2, R3

表0-2

汇编功能函数要得到参数值时就访问这些寄存器。如果这些值被使用并保存在其它地方或已经不再需要了，那么这些寄存器可被用做其它用途。下面是一个C程序和汇编程序的接口例子。你应该注意到通过内部RAM传递参数的功能函数将使用规定的寄存器，汇编功能函数将使用这些寄存器接收参数。对于要传递多于3个参数的函数，剩余的参数将在默认的存储器段中进行。

列表 0-8

```
; C code
// C 程序中汇编函数的申明
bit devwait(unsigned char ticks, unsigned char xdata *buf);

// invocation of assembly function
if (devwait(5, &outbuf)) {
   bytes_out++;
```

列表 0-9

```
; 汇编代码
; 在代码段中定义段
?PR?_devwait?LOWLVL SEGMENT CODE
```

```
    ; 输出函数名
            PUBLIC _devwait
    ;这个函数可被连接器放置在任何地方
            RSEG    ?PR?_devwait?LOWLVL
;*********************************************************************
; Function: _devwait
; Description: 等待定时器0溢出，向外部器件表明P1中的数据是有效的。如果定时器尚
;              未溢出，被写入XDATA的指定地址中。
; Parameters: R7 – 存放要等待的时标数
; R4|R5 – 存放要写入的XDATA区地址
; Returns: 读数成功返回1,时间到返回0
; Side Effects: none.
;*********************************************************************
_devwait:   CLR TR0         ;设置定时器0
            CLR TF0
            MOV TH0, #00
            MOV TL0, #00
            SETB TR0
            JBC TF0, L1         ; 检测时标
            JB T1, L2           ; 检测数据是否准备就绪
    L1:     DJNZ R7, _devwait ; 时标数减1
            CLR C
            CLR TR0             ; 停止定时器0
            RET
    L2:     MOV DPH, R4         ; 取地址并放入DPTR
            MOV DPL, R5
            PUSH ACC
            MOV A, P1           ; 得到输入数据
            MOVX @DPTR, A
            POP ACC
            CLR TR0             ; 停止定时器0
            SETB C              ; 设置返回位
            RET
            END
```

上面的代码中有些我们没有讨论的问题——返回值。在这里函数返回一个位变量。如果时间到将返回0，如果输入字节被写入指定的地址中将返回1。

当从功能函数中返回值时，C51通过转换使用内部存储区，编译器将使用当前寄存器组来传递返回参数。返回参数所使用的寄存器见表0-3。

Return Type	Register(s) Used
bit	carry flag
(unsigned) char	R7
(unsigned) int	R6...R7
(unsigned) long	R4...R7
float	R4...R7
pointer	R1...R3

表0-3

返回这些类型的功能函数可使用这些寄存器来存储局部变量，直到这些寄存器被用来返回参数。假使你有一个函数要返回一个长整型，你可以使用R4到R7这4个寄存器。这样你就不需要声明一个段来存放局部变量，存储区就更加优

化了。功能函数不应随意使用没有被用来传递参数的寄存器。

5 内联汇编代码

有时候,你的程序需要使用汇编语言来编写,像对硬件进行操作或一些对时钟要求很严格的场合,但你又不希望用汇编语言来编写全部程序或调用用汇编语言编写的函数。那么你可以通过预编译指令"asm"在C代码中插入汇编代码。

列表 0-10

```
#include <reg51.h>
extern unsigned char code newval[256];
void func1(unsigned char param) {
    unsigned char temp;
    temp=newval[param];
    temp*=2;
    temp/=3;
#pragma asm
    MOV P1, R7 ; 输出temp中的数
    NOP        ;
    NOP
    NOP
    MOV P1, #0
#pragma endasm
}
```

当编译器在命令行加入"src"选项时,在"asm"和"endasm"中的代码将被复制到输出的SRC文件中。如果你不指定"src"选项，编译器将忽略在"asm"和"endasm"中的代码。很重要的一点是编译器不会编译你的代码并把它放入它所产生的目标文件中，必须用得到的.src文件，经过编译后再得到.obj文件。从上面的文件将得到下面的.src文件。

列表 0-11

```
; ASMEXAM.SRC generated from: ASMEXAM.C

$NOMOD51

NAME ASMEXAM

P0    DATA    080H
P1    DATA    090H
P2    DATA    0A0H
P3    DATA    0B0H
T0    BIT     0B0H.4
AC    BIT     0D0H.6
T1    BIT     0B0H.5
EA    BIT     0A8H.7
IE    DATA    0A8H
RD    BIT     0B0H.7
ES    BIT     0A8H.4
```

```
IP    DATA    0B8H
RI    BIT     098H.0
INT0  BIT     0B0H.2
CY    BIT     0D0H.7
TI    BIT     098H.1
INT1  BIT     0B0H.3
PS    BIT     0B8H.4
SP    DATA    081H
OV    BIT     0D0H.2
WR    BIT     0B0H.6
SBUF  DATA    099H
PCON  DATA    087H
SCON  DATA    098H
TMOD  DATA    089H
TCON  DATA    088H
IE0   BIT     088H.1
IE1   BIT     088H.3
B     DATA    0F0H
ACC   DATA    0E0H
ET0   BIT     0A8H.1
ET1   BIT     0A8H.3
TF0   BIT     088H.5
TF1   BIT     088H.7
RB8   BIT     098H.2
TH0   DATA    08CH
EX0   BIT     0A8H.0
IT0   BIT     088H.0
TH1   DATA    08DH
TB8   BIT     098H.3
EX1   BIT     0A8H.2
IT1   BIT     088H.2
P     BIT     0D0H.0
SM0   BIT     098H.7
TL0   DATA    08AH
SM1   BIT     098H.6
TL1   DATA    08BH
SM2   BIT     098H.5
PT0   BIT     0B8H.1
PT1   BIT     0B8H.3
RS0   BIT     0D0H.3
TR0   BIT     088H.4
RS1   BIT     0D0H.4
TR1   BIT     088H.6
PX0   BIT     0B8H.0
```

```
PX1  BIT    0B8H.2
DPH  DATA   083H
DPL  DATA   082H
REN  BIT    098H.4
RXD  BIT    0B0H.0
TXD  BIT    0B0H.1
F0   BIT    0D0H.5
PSW  DATA   0D0H
?PR?_func1?ASMEXAM  SEGMENT CODE
EXTRN CODE (newval)
PUBLIC _func1
;
; #include <reg51.h>
;
; extern unsigned char code newval[256];
;
; void func1(unsigned char param) {
       RSEG   ?PR?_func1?ASMEXAM
       USING 0
_func1:
;---- Variable 'param?00' assigned to Register 'R7' ----
                    ; SOURCE LINE # 6
; unsigned char temp;
;
; temp=newval[param];
                    ; SOURCE LINE # 9
    MOV  A,R7
    MOV  DPTR,#newval
    MOVC A,@A+DPTR
    MOV  R7,A
;---- Variable 'temp?01' assigned to Register 'R7' ----
; temp*=2;
                   ; SOURCE LINE # 10
    ADD A,ACC
    MOV R7,A
; temp/=3;
                   ; SOURCE LINE # 11
    MOV B,#03H
    DIV AB
    MOV R7,A
;
; #pragma asm
    MOV P1, R7 ; write the value of temp out
    NOP ; allow for hardware delay
```

```
    NOP
    NOP
    MOV P1, #0 ; clear P1
; #pragma endasm
; }
                    ; SOURCE LINE # 20
    RET
; END OF _func1
    END
```

正如你所见，在"asm"和"endasm"中的代码被复制到输出的SRC文件中。然后，这个文件被编译，并和其它的目标文件连接后产生最后的可执行文件。

6 提高编译器的汇编能力

很多软件设计者都相信他们所编写的汇编代码比编译器所产生的代码效率更高，因此他们认为用汇编语言所做的项目比用高级语言所做的项目要好。对这些工程师来说，汇编语言所带来的高效比前面所讨论的C语言的优点重要得多。我相信如果这些工程师把他们所编写的汇编代码和用C语言编写的程序通过编译后产生的代码比较一下，他们肯定会非常吃惊。用高级语言来开发项目的速度和效率都比用汇编好。

对于那些现在还难以决定用汇编还是C的开发者来说，让我给你提供一个选择。Keil C编译器提供一个参数使生成的文件为汇编代码，把这些汇编代码可用A51编译并和其它模块连接。这和直接用编译器产生目标文件是一样的。这种做法的优点是可对产生的汇编代码进行编辑，这样可对你的代码进行优化，然后再把修改后的代码进行编译和连接。

决大多数情况下，你不必对汇编代码进行修改，因为这些代码都是经过了优化的。但有时候还是要修改的。前面的一个例子告诉你如何在代码段定位表格，当需要查表时，只需要计算DPTR的低字节。我们再引用以前时钟系统的例子：

列表 0-12

```
char code bcdmap[60][2]={
"00", "01", "02", "03", "04", "05", "06", "07", "08", "09",
"10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
"20", "21", "22", "23", "24", "25", "26", "27", "28", "29",
"30", "31", "32", "33", "34", "35", "36", "37", "38", "39",
"40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
"50", "51", "52", "53", "54", "55", "56", "57", "58", "59"
};
void disp_time(void) {
  static char time_str[32]="TIME OF DAY IS: XX:XX:XX ";
  unsigned char i;
  time_str[T_HOURT]=bcdmap[timeholder.hour][0];
  time_str[T_HOUR]=bcdmap[timeholder.hour][1];
  time_str[T_MINT]=bcdmap[timeholder.min][0];
  time_str[T_MIN]=bcdmap[timeholder.min][1];
  time_str[T_SECT]=bcdmap[timeholder.sec][0];
  time_str[T_SEC]=bcdmap[timeholder.sec][1];
  putchar(0xFF);
```

```
  for (i=0; i<32; i++) {
    putchar(time_str[i]);
  }
  disp_update=0;   // 清除显示更新标志位
}
```

正如你所看到的bcdmap包括120个字节，因此只用一个字节就可以包含偏移量。时钟系统中，表的存放地址并不在256个字节之内，我们必须得到这个表的基址，再加上表内数据的偏移量。下面是编译器得到的寻址汇编代码。

列表0-13

```
; time_str[T_HOURT]=bcdmap[timeholder.hour][0];
; SOURCE LINE # 214
    MOV A,timeholder
    ADD A,ACC
    ADD A,#LOW bcdmap
    MOV DPL,A
    CLR A
    ADDC A,#HIGH bcdmap
    MOV DPH,A
    CLR A
    MOVC A,@A+DPTR
    MOV time_str?42+010H,A
```

这段代码在程序中重复了6次，你可以看到编译器产生的代码在bcdmap的地址上加上偏移量，在寄存器DPTR中得到新的地址。一种简化寻址过程的方法是把表格放置在代码段的每页的顶端，这样只需要一个寻址字节就可以对表内的数据进行寻址。可通过产生一个小的汇编代码文件（见表0-14），并把它和现存的C程序文件连接来实现。原来C文件中的初始化表格就要去掉了。现在C文件要包含一个外部声明的bcdmap。

列表 0-14

```
        CSEG AT 0400H
bcdmap: DB '0' ,'0'
        DB '0' ,'1'
        DB '0' ,'2'
        ...
        DB '5' ,'7'
        DB '5' ,'8'
        DB '5' ,'9'

        END
```

产生的汇编代码将使用新的寻址方式，见表0-15：

列表 0-15

```
; time_str[T_HOURT]=bcdmap[timeholder.hour][0];
          ; SOURCE LINE # 214
    MOV A,timeholder
    ADD A,ACC
    MOV DPL,A
```

```
    MOV DPH,#HIGH bcdmap
    MOVC A,@A+DPTR
    MOV time_str?42+010H,A
```

表寻址的前一种方法需要11个处理周期,17个代码的存储空间。相比之下，第二种方法只需要8个处理周期和12个字节存储空间。如果你的目的是优化速度，那么你已经作到了。但是，当你的目的是优化代码空间，可以把6个寻址代码段合并成一个功能段，在程序中调用它6次。这可以大大的减少代码长度。功能段代码见列表0-16：

列表 0-16

```
getbcd:     ADD A,ACC
            MOV DPL,A
            MOV DPH,#HIGH bcdmap
            MOVC A,@A+DPTR
            RET
; time_str[T_HOURT]=bcdmap[timeholder.hour][0];
            ; SOURCE LINE # 214
            MOV A,timeholder
            LCALL getbcd
            MOV time_str?42+010H,A
```

“getbcd”功能函数代码在”disp_time”函数代码段中，这样，就只有”disp_time”函数能调用它。

除了进行优化，还可以对编译后的文件进行修改，消除编译器输出文件中不必要的功能调用。我们在看一下前面的时钟例子，其中包括一段更新显示的代码。存放时间的结构定义如下：

```
    typedef struct
        unsigned char hour, min, sec;
    } timestruct;
```

结构中的数据只有3个字节。我们看一看编译后的结构数据的复制代码，

列表 0-17

```
    ; timeholder=curtime;
        ; SOURCE LINE # 327
        MOV R0,#LOW timeholder
        MOV R4,#HIGH timeholder
        MOV R5,#04H
        MOV R3,#04H
        MOV R2,#HIGH curtime
        MOV R1,#LOW curtime
        MOV R6,#00H
        MOV R7,#03H
        LCALL ?C_COPY
```

这段代码需要16个处理周期和11个字节的存储空间，而对C_COPY的调用又要花去70个处理周期，而仅仅只为了复制3个字节，这时我们对代码做如下修改，对这写字节进行手工复制。

列表 0-18

```
  ; timeholder=curtime;
```

```
; SOURCE LINE # 327
MOV timeholder, curtime
MOV timeholder+1, curtime+1
MOV timeholder+2, curtime+2
```

这段代码同样可以完成上面的工作，只需要6个处理周期和6个字节存储空间。

编辑产生的汇编代码使你得到很好的速度和代码空间，让你使用C来进行产品开发更加得心应手，使你最终得到的代码像汇编高手编出的代码那样紧凑而高效。

7 仿真多级中断

很多时候,我希望嵌入式系统的中断级别多于两级。因为，一般来说系统都有掉电中断，并且都被置为高优先级。这样的话其它中断都共用一个低优先级。在Intel 8051的数据书中介绍了一种通过软件来扩充3个中断优先级的方法。这种方法要求首先按正常方式设置前两个中断优先级，然后把要设置为最高级的那个中断设置为中断优先级1，并且在原先中断优先级为1的中断服务程序中使能它。下面是一个例子：

列表 0-19

```
        PUSH IE             ; 保存当前IE值
        MOV IE, #LVL2INTS   ; 使能中断优先级为2的中断
        CALL DUMMY_LBL      ; 伪 RETI
        ...                 ; 中断服务程序
        POP IE              ; 恢复 IE
        RET
DUMMY_LBL: RETI
```

原理是很简单的，首先保存IE的状态。然后给IE送数，使得只有中断优先级为2的中断被使能，然后调用伪RETI指令，允许硬件产生中断。

这样就可不必使用硬件如PICs(programmable inter充中断优先级。新增加的代码不会对ISR对中断事件的响应前面多了10个处理周期的时间，这对一般系统来说都是可以使每个中断都有自己的优先级。

Level	Source
0 (lowest)	Timer 0
1	External Interrupt 1
2	Serial Interrupts
3	Timer 1
4 (highest)	External Interrupt 0

如果系统要求每个中断都有自己的优先级。假设你的中断优先级如表0-4所示，那么系统就需要5个中断优先级。

按照前面所讲的方法，你必须仔细选择ISR中IE的屏蔽值，只允许更高优先级的中断，像串行口中断服务程序中只能允许定时器1中断和外部中断0。而外部中断0的中断优先级最高，定时器0的中断优先级最低，它们的中断服务程序无须做变动。

表0-4

在初始化程序中必须将定时器0的中断优先级设置为0，而其它所有中断的优先级被设置为1。对中断优先级1到3，在它们的中断服务程序设置如下屏蔽位：

列表 0-20

```
EX1_MASK EQU 99H ; 允许串行口中断，定时器1中断，外部中断0
SER_MASK EQU 89H ; 允许定时器1中断，外部中断0
T1_MASK  EQU 81H ; 允许外部中断0
```

现在,在中断服务程序中加入仿真代码。

列表 0-21

```
?PR?EXT1?LOWLVL SEGMENT CODE
```

```
EXT1:       PUSH IE             ; 保存IE值
            MOV IE, #EX1_MASK   ; 使能串行口中断，定时器1中断，外部中断0
            CALL DUMMY_EX1      ; 伪 RETI
            LCALL ex1_isr       ; 用C代码编写的中断服务程序
            POP IE              ; 恢复 IE
            RET
DUMMY_EX1:  RETI

?PR?SINTR?LOWLVL SEGMENT CODE

SINTR:      PUSH IE             ; 保存IE值
            MOV IE, #SER_MASK   ; 使能定时器1中断，外部中断0
            CALL DUMMY_SER      ; 伪 RETI
            LCALL ser_isr       ; 用C代码编写的中断服务程序
            POP IE              ; 恢复 IE
            RET
DUMMY_SER:  RETI

?PR?TMR1?LOWLVL SEGMENT CODE

TMR1:       PUSH IE             ; 保存IE值
            MOV IE, #T1_MASK    ; 使能外部中断0
            CALL DUMMY_T1       ; 伪 RETI
            LCALL tmr1_isr      ; 用C代码编写的中断服务程序
            POP IE              ; 恢复 IE
            RET
DUMMY_T1:   RETI
```

用少量的汇编代码使系统对硬件的功能进行了扩展。系统的主要代码功能还是用C编写的。

8 时序问题

有时,代码要执行的任务有严格的时间要求，这些代码必须用汇编来完成，时间的精确度要达到一两个处理周期。像这种情况，一种最简单的方法就是在注释区中加上指令周期计数。这给代码的编写带来很大的方便，当代码改变时，时序也跟着改变，有了指令周期计数后，我们很容易计算时序。

例如你在引脚T1按一定的时序输出数据，另外一个系统监视输出并以100KHz的速率进行采样。每位数据之前都有一个2us的起始信号，然后是宽度为3us的数据位，其它时间T1被置低。时序如图0-1

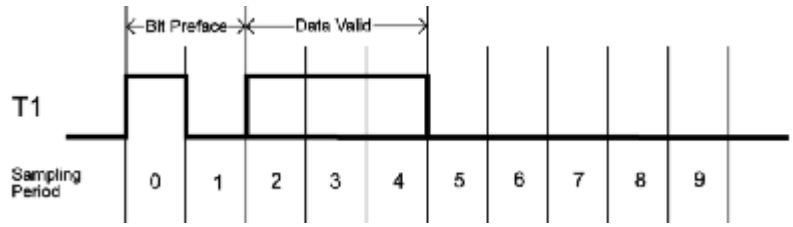

图0-1

系统时钟为12MHz，所以指令周期为1us，使用C语言很难保证时序，所以必须用汇编语言。函数接收作为参数传递过来的字节数据并从高位到低位向外发送。程序见列表0-22：

列表 0-22

```
; 该函数有如下声明
; void sendbyte(unsigned char);

?PR?_sendbyte?SYS_IO    SEGMENT CODE
?DT?_sendbyte?SYS_IO    SEGMENT DATA OVERLAYABLE

            PUBLIC _sendbyte
            PUBLIC ?_sendbyte?BYTE
            RSEG ?DT?_sendbyte?SYS_IO
?_sendbyte?BYTE:
BITCNT: DS   1
            RSEG ?PR?_sendbyte?SYS_IO
_sendbyte:   PUSH ACC        ; 保存累加器
            MOV  BITCNT, #8 ; 发送8位数据
            MOV  A, R7      ; 获取参数

            RLC  A          ; 得到第一位要发送的数据

LOOPSTRT:    JC   SETHIGH    ; 2, 9 确认输出值
            SETB T1         ; 1, 0
            CLR  T1         ; 1, 1
            RLC  A          ; 1, 2 得到下一位数据
            NOP             ; 1, 4
            NOP             ; 1, 5
            NOP             ; 1, 6
            DJNZ BITCNT, LOOPSTRT; 2, 7 是否发送完毕

SETHIGH:     SETB  T1        ; 1, 0
            CLR   T1        ; 1, 1
            SETB  T1        ; 1, 2 数据位置1
            RLC   A         ; 1, 3 得到下一位数据
            NOP             ; 1, 4
            CLR   T1        ; 1, 5 清除输出
            DJNZ  BITCNT, LOOPSTRT; 2, 7 是否发送完毕
            POP   ACC       ; 恢复累加器
```

```
            RET

            END
```

可以看到，每条指令后面都有指令执行所需要周期数和到目前所消耗的指令周期数。每10个指令周期发送1位数据（周期计数从0到9）。你选择从哪条指令开始计数都没关系，在这里我选择了起始位置高的那条只作为参考指令，当你的循环中有两条分支的时候，你应保证这两条分支所需要的时间是一样的，这可通过使用NOP指令来平衡。

当系统晶振频率不变的时候，上面的程序完全可以胜任。但是，假使你并不生产监视T1脚输出的模块，而生产这个模块的厂家做不到以100KHz的频率进行采样。这时你必须改变你数据的输出速率。为了不经常的改动程序，你需要对程序重新做调整，使用户能够指定数据的输出速率。

这样程序会变得复杂一些，我们使用循环来消耗时间，从而改变数据输出速率。

列表 0-23

```
?PR?_sendbyte?SYS_IO SEGMENT CODE
?DT?_sendbyte?SYS_IO SEGMENT DATA OVERLAYABLE
?BI?_sendbyte?SYS_IO SEGMENT BIT OVERLAYABLE

            PUBLIC  _sendbyte
            PUBLIC  ?_sendbyte?BYTE
            PUBLIC  ?_sendbyte?BIT

            RSEG    ?DT?_sendbyte?SYS_IO
?_sendbyte?BYTE:
BITCNT:     DS 1
DELVAL:     DS 1

            RSEG    ?BI?_sendbyte?SYS_IO
?_sendbyte?BIT:
ODD:        DBIT    1

            RSEG    ?PR?_sendbyte?SYS_IO
_sendbyte: PUSH ACC             ; 保存累加器
            MOV BITCNT, #8      ; 发送8位数据
            CLR C
            MOV A, R5           ; 得到延时周期数
            CLR ODD             ; 延时为偶数
            JNB ACC.0, P_EVEN
            SETB ODD            ; 延时为奇数
            DEC ACC             ; 对偶数的延时，减去一个周期

P_EVEN:     SUBB A, #4          ; 减去前面4个周期的延时
            RR A                ; 除2，得到所需执行DJNZs的数
            MOV DELVAL, A
            MOV R5, A
```

```
            JNB ODD, SEND_EVEN

SEND_ODD:   MOV A, R7           ; 要输出的数据
            RLC A               ; 第一位

LOOP_ODD:   JC SETHIGH_0        ; 2, 9 检测数据值
            SETB T1             ; 1, 0
            CLR T1              ; 1, 1
            RLC A               ; 1, 2 下一位
            NOP                 ; 1, 3
            NOP                 ; 1, 4
            MOV R5, DELVAL      ; 2, 6
            DJNZ R5, $          ; 2, 8
            NOP                 ; 1, 9
            DJNZ BITCNT, LOOP_ODD ; 2, 11 是否传输完毕

SETHIGH_0: SETB T1              ; 1, 0
            CLR T1              ; 1, 1
            SETB T1             ; 1, 2
            RLC A               ; 1, 3 下一位
            NOP                 ; 1, 4
            MOV R5, DELVAL      ; 2, 6
            DJNZ R5, $          ; 2, 8
            CLR T1              ; 1, 9 清除输出
            DJNZ BITCNT, LOOP_ODD ; 2, 11 数据是否发送完毕
            POP ACC             ; 恢复累加器
            RET

SEND_EVEN: MOV A, R7            ; 要输出的数据
            RLC A               ; 要发送的第一位

LOOP_EVEN: JC SETHIGH_E         ; 2, 9 检测输出值
            SETB T1             ; 1, 0
            CLR T1              ; 1, 1
            RLC A               ; 1, 2 下一位
            MOV R5, DELVAL      ; 2, 4
            DJNZ R5, $          ; 2, 6
            NOP                 ; 1, 7
            NOP                 ; 1, 8
            DJNZ BITCNT, LOOP_EVEN ; 2, 10 数据是否发送完毕

SETHIGH_E: SETB T1              ; 1, 0
            CLR T1              ; 1, 1
            SETB T1             ; 1, 2
```

```
        RLC A                ; 1, 3 下一位
        MOV R5, DELVAL       ; 2, 5
        DJNZ R5, $           ; 2, 7
        CLR T1               ; 1, 8 清楚输出
        DJNZ BITCNT, LOOP_EVEN ; 2, 10 数据是否发送完毕
        POP ACC              ; 恢复累加器
        RET

        END
```

函数首先确认所要延时的周期数的奇偶性。然后决定DJNZ的执行次数。我们要减去延时循环前面所消耗的指令周期数（偶数减4，奇数减5），剩下的除2就得到了要执行DJNZ的次数（DJNZ要消耗两个指令周期）。这样功能函数的最小延时为6个指令周期。

现在你可通过在C中改变参数，来改变数据的传输速率了，而无须去更改程序。

9 结论

这章向你说明了汇编语言在系统开发中仍然有不可替代的作用。用高级语言可使你产品的开发更加快速而稳定，这并不说明你不可以把C和汇编结合起来使用，汇编的确能够完成一些高级语言不能做到的事情。

第五章 系统调试

1 介绍

调试嵌入式系统没有什么固定可寻的方法，硬件接口所带来的复杂和时钟的限制使嵌入式系统比PC或对应的主机应用更加复杂，这些系统可以很方便的用软件进行调试，能够按使用者的步伐进行单步运行。而嵌入式系统在目标板上全速运行进行调试，这意味着要控制调试，你不得不使用ICE。在你的代码时间要求很严格的部分不能使用单步运行，那样会使你的程序运行出现混乱。对于使用时标或看门狗的系统的调试更加困难。

因为嵌入式系统的复杂性，很多方法都不能使用。这章将探索这些方法，使你对设计和调试你的系统有个初步的了解。

2 通过系统设计来帮助调试

在你系统的设计阶段，如果你的系统规划得好的话，会给你将来的调试带来很大的方便。我们可以通过串行口来输出调试信息。换句话说，就是通过一系列I/O口来反映程序在不同的执行阶段时的程序状态和变量状态。这种方法的缺点是会增加不必要的硬件，但也可以为系统将来的扩充留下余地。也可通过显示板输出调试信息，还可以把调试信息存储在RAM中，当程序执行完成后再下载这些信息。

不管你用什么方法调试程序，当使用I/O作为调试用时好处很多。在你设计系统的时候就应该考虑这些方面，并进行各种整体功能调试，当然在PCB板作成之后还要做各种调试，但这时你应该已经排除了大多数的问题。用PCB板进行调试时，你可能会发现它像逻辑分析仪之类的仪器。你不应该完全依赖于ICE，尽管那是最方便的调试方法，但不是那么容易得到的，所以应该学会如何在没有这种奢侈工具的帮助下进行调试。

在没有ICE的情况下进行调试可使你很快擅长使用数字存储技术，这对你调试系统很有帮助，如果你对系统在什么时候做什么事情很了解的话，就可以知道在什么地方程序运行开始出错。当你发现了出错的地方后，你就可以在这些地方加入调试语句，把调试信息通过显示，串行口或I/O发送出来。

3 使用调试端口

在没有ICE时进行调试的最有效的一种手段是通过调试端口输出数据。一般来说，这些数据包括系统事件，反映程序运行到某一点的调试状态，变量值等。调试端口一般是串行口，串行口要么完全作为调试用，要么在调试端口和数据接口间时分复用。而对8051来说麻烦在于一般只有一个串行口，这意味着要进行时分复用。如果你有两个串行口的话，那就幸运多了，不必担心调试数据会影响正常数据。

当你用10个数据位向PC发送数据的时候，串行调试端口会出错，所以你最好使用其它模式。另外，向外输出数据多出来的这部分调试代码会改变你程序的进程，而且会产生一些莫名其妙的问题。

调试端口适用于那些对时间要求不严格并且用多余串行口的系统。从这些讨论可以看出，第4章所说的实时时钟系统就很适合，它有多余的串行口和大量的空闲时间。如果你要在这个系统上使用调试端口，代码由中断进行驱动并将缓冲区中的调试数据从数据调试端口送出。

列表 0-1

```
#include <reg51.h>
#include <intrins.h>

#ifndef NULL
```

```
#define NULL ((void *) 0L)
#endif

#define DB_MAXSIZE 0x20

unsigned char db_head, db_tail, db_buffer[DB_MAXSIZE];
/*********************************************************************
Function: debug_init
Description: 将串行口设置为调试端口，把缓冲区指针设为0
Parameters: 无
Returns: 无
*********************************************************************/
void debug_init(void) {
  SCON=0x90; 使用串行通讯模式2
  db_head=db_tail=0;
  ES=1;
}

/*********************************************************************
Function: debug_insert
Description: 把所指向的存储区中的数据拷贝到缓冲区中
Parameters: base – 指针，指向要拷贝数据的头地址
            size – 所要拷贝数据的数量
Returns: 无
*********************************************************************/
void debug_insert(unsigned char data *base, unsigned char size) {
    bit sendit=0; // 标志位，表明是否要进行串行传输初始化
    unsigned char i=0;
    if (!size || base==NULL) { return; }//测试参数是否有效
    if (db_tail==db_head)
        sendit=1;
    }
    while (db_tail!=db_head && i<size) {// 当缓冲区有空间且数据区中还有数据时，
                                        // 进行拷贝
        db_buffer[db_tail]=base[i];     // 拷贝当前字节
        i++;
        db_tail++;                      // 移动指针
        if (db_tail==DB_MAXSIZE) {      // 指针是否超出范围
            db_tail=0;
         }
    }
    if (sendit) {                       // 是否要传输一个字节
          SBUF=db_buffer[db_head];
    }
```

```
  }
/*********************************************************************
Function: debug_output
Description: 串行口中断服务程序，增加缓冲区头指针，如果头指针不等与尾
             指针输出下一个字节
Parameters: 无
Returns: 无
*********************************************************************/
void debug_output(void) interrupt 4 {
    RI=0;
    if (_testbit_(TI))
       db_head++;
       if (db_head==DB_MAXSIZE) { // 是否超出范围
          db_head=0;
       }
       if (db_head!=db_tail)
         SBUF=db_buffer[db_head]; // 送下一个字节
       }
    }
}
```

通过调用功能函数把数据块插入缓冲区中,功能函数中包含一个指针指向待传送的数据,还包含一个计数器表明要传送字节的数量。数据被拷贝到缓冲区中，缓冲区中的尾指针被相应的更新。你可以根据RAM的大小和需要传送调试数据的多少调整缓冲区的大小。第4章的实时时钟系统没有外部RAM，你必须把缓冲区设置在内部RAM中。这限制了一次传送数据的数量。

4 使用Monitor-51

有些时候，我们设计系统既希望从代码段读出数据，也希望往代码段写入数据，这使得系统相信自己只有一个存储段而不是两个。这样做的好处是你可以使用一个简单的8051程序把代码下载到存储区中。这使你避免不断的转换，编译，烧写EPROM和测试。

如果你的系统能够向代码区写入数据，你应该考虑使用Keil C51自带的软件包Monitor-51。这个程序允许你在目标板上运行代码和调试功能。它要求你在代码区装入通信和控制模块，通信模块将通过串行口和PC进行通信。你在PC运行另外一个程序MON51.EXE，这个程序作为你目标板和PC之间的接口，这样就相当有了一台仿真器。

监视程序将使你可以看到各个存储段并改变他们的内容。你可以查看SFRs的的值，禁能你的代码，加入新的代码。你还可以加入断点。当你的系统挂起时，你可以单步运行你的程序。所有这些的前提是你的系统必须可以对代码区进行写操作。

监视程序可在没有SRAM的系统上，通过设置在代码区内运行，但这样的话，你就不能设置断点，单步运行，改变代码区的内容。Monitor-51将控制一个串行口和一个定时器，同时还占用了2816字节的代码空间和256字节的XDATA区。

调整你的Monitor-51 install.a51文件来适应你的系统。当你的系统频率为11.059MHz时，它设置串行口的波特率为9600。此外它还将把在地址8000H以上的中断向量入栈。如果你不想这样，可以更改install.a51文件头的常数来进行调整。对于PC上的软件接口可以查询手册，在这里就不详细介绍了。

5 利用I/O端口进行调试

如果你不能使用串行口做为调试端口，你可以利用分立的I/O口进行调试。把I/O口和锁存器相连，锁存器被分配一个外部地址。我想至少可以找到一些空余的脚来显示系统的执行点。最为理想的方法是使用8个引脚在同一时间显示一个字节。把这写引脚连接到LED上，这样易于观察，此外，你也可以使用示波器。

我所做的很多系统都使用输出引脚来显示状态。一般来说，一个引脚来显示系统正在执行，引脚电平以一定的频率进行翻转，你通过检查这个引脚以确定系统工作正常。一个引脚用来显示程序已经运行过某一点或程序正在等待输入等。你也可以把寄存器的内容发送到引脚上，然后程序进入等待状态并观察引脚的数值来确定程序运行是否正常。你必须自己确定每个引脚在程序运行的每个状态的作用。调试程序时要一段一段的进行，这样才便于你进行观察，这和使用串行口进行调试是不同的。

如果你有逻辑分析仪的话就再好不过了，把用作调试的输出口和逻辑分析仪连起来，逻辑分析仪将记下他所见到的数据。对输出的数据进行分析后，你可以知道你程序运行的状况。

6 使用ICE

8051的在线仿真器的种类有很多，这里不对他的使用方法做讨论，关于这方面的书层出不穷。我们将对一些要点做一些说明。

第一点，对你将要进行仿真的代码段使用"debug"选项进行再编译。对于包含了结构或数组的C程序，如果要对这些数据进行访问，更改或检测，须在汇编时加"objectextend"参数。这样将使系统在目标文件和以后产生的可执行文件中加入调试信息。如果不这样做的话，你在调试窗口中看到的将不是C程序，而是一些没有标号的汇编代码。

在系统中安装ICE时，要特别注意设置。这些设置可使仿真器以系统晶振频率运行或以仿真器内部时钟频率运行。如果你的系统有外接晶振，并且系统的运行完全依赖于这个频率，这时如果以仿真器频率运行程序的话，程序将不会按你想象的那样工作了。时钟发生器将以错误的频率振荡，串行通信将发生错误，因为baud率已经改变了。

同样，你还可以使能仿真器的电源和复位功能，这可使你对硬件做更多的测试。

调试有看门狗或运行定时器的系统时，要记住当你检查代码或数据时，或单步运行时，时钟并没有停止运行，它们还是会溢出并产生中断，有时候给程序的调试带来很大的不便。所以测试的时候最好关闭看门狗定时器。而对于时标的产生，最好就是禁能定时器中断。

如果你想购买仿真器，我建议你买带有跟踪缓冲区的那种。很多仿真器都带有从16K到128K的跟踪缓冲区。这些缓冲区存储执行的指令，指令指针的值，引脚，和仿真器相连的引脚的输入输出值。分析这些数据，可以发现系统的问题出自那里，这样跟踪缓冲区的功能相当于逻辑分析仪。

7 结论

使用仿真器是对系统进行测试和集成最有效的方法。但是你在进行系统调试时不要完全依赖仿真器。你有可能碰到没有仿真器或仿真器的作用不大的时候，像一些对外围器件（如EEPROM）进行控制的系统，硬件接口，时序信号等。这使，你应该使用示波器或其它一些仪器进行系统测试。

这章给出了一些对你的工程进行调试的方法。值得重声的一点是对实时时钟之类的系统的调试不能光从书本上学，还要多多积累经验，书本只能给你指明方向。你应根据不同的情况采用不同的解决方法。

第六章 中断系统

1 介绍

这一章将讨论设计实时时钟系统中的一些问题，其中很多都和软件有关，其它一些和硬件设计有关。这一章的硬件设计集中在中断系统上，将讨论如何使用中断事件和查询方法来触发操作。软件的讨论将集中在程序结构上。

2 中断驱动系统和查寻系统

如果你的嵌入式系统是基于对输入信号做处理的基础上的，那你就要决定你的系统是采用中断的方式来接收数据好呢，还是采用查询的方式好呢？这需要你考虑两方面的问题，第一点是你系统对输入信号变化所作出的反应要多快，如果要求在很短的时间内作出反应，那么最好使用中断的方式，不管你系统的查询速度有多快，它总是比中断的反应要慢一些。第二点你要知道你的输入信号的变化有多快，如果它以接近1/10指令周期的频率变化，就要用中断查询方式了。

有些系统有很多输入源，而且看上去每个输入源都需要中断。这就需要考虑如何在他们之中分配中断或者对其中的一些输入源使用查询的方法。换句话说就是必须对这些输入源建立优先级。举个例子，有一个电机控制系统对电机传感器送过来的信号进行监视，同时又要接收主CPU送过来的状态查询请求，我想你一定会把前者的优先级定得比后者高，因为丢失一个查询请求信号不会对系统造成什么很大的影响，而丢失传感器信号可能会对电机造成很大的影响。如何建立优先级要从系统的高度去考虑。

当你决定了输入源的优先级之后，下一步要决定如何把信号引入处理器。对输入信号进行查询时，那些需要快速查询的信号应直接接到端口引脚上，处理器对端口的寻址只需要一个指令周期。对于查询速度要求不高的信号可通过锁存器由系统总线接入处理器。从前面的讨论知道。设置DPTR需要两个指令周期，对信号读写至少需要两个周期，对信号的查询相对就慢一些。决定了信号的布局之后，接下来要决定对信号的查询频率。如果一个信号每秒钟查询10000次，另一个信号每秒钟查询一次，根据总的信号查询时间把第一个信号接到总线上，把第二个信号接到处理器的引脚上是没有意义的。一般说来，如果要对信号状态的改变作出反应，信号的查询频率应是你所估计频率的两倍。

对信号进行查询应根据不同的情况采取不同的方法。如果输入信号是人发出的，那么10Hz的查询频率就够了，人相当于一个很慢的I/O器件，当他们使用按键向系统传送信息时，10Hz的查询速率和更高的查询速率没什么分别。用户接口信号可在定时器中断或主循环中查询。

当查询变化频率很快或十分重要的信号时有两种方法。可以在定时器中断中查询，但你中断的发生频率要很快。此外你可以不断的对信号进行查询而不干任何其它事情。这两者的缺点是系统资源消耗太大。如果你的系统是用电池供电的，那么这两种方法都不是很好。这种情况下，要把信号的重要程度加以区分。重要的，高速的信号接到外部中断口上，剩下的用查询的方法。

这样做的目的是可以建立中断和查询输入触发事件之间的优先级。我把重要的输入接到INT0和INT1，你也可以用其它端口引脚来扩展外部中断或者共用外部中断。8051对中断的响应延时是很短的。另外对那些变化很慢的信号，也应通过中断接入处理器，而不需要用程序对它进行定期的查询。

3 中断的电平和边沿触发

8051的外部中断支持两种触发方式，一种是电平出发一种是边沿触发。应该根据信号的类型选择那种触发方式。

3.1 电平触发中断

电平触发方式比较好理解，处理器每个指令周期查询中断引脚，当发现引脚电平为低时触发中断。如果信号从1边为0，一个周期后又变为1，中断并不会被清除，直到中断执行完毕并用RETI指令返回之后。但是如果输入信号一直为低，那么将一直触发中断，当要求中断服务的器件在中断服务结束一段时间之后才释放信号线时就会发生这种情况。这时你会发现中断被执行了多次，所消耗的时间比预期的要长很多。这时应使用边沿触发方式。

3.2 边沿触发方式

当外部中断引脚电平由高向低变化时将触发中断。处理器每个指令周期查询中断引脚，当前一个指令周期是引脚电平为高，紧接着下一个指令周期检测到引脚电平为低时将触发中断。像前面所提到的那样，这种方法适用于请求中断服务的器件在中断服务结束一段时间之后才释放信号线时的情况，因为这时只有下降沿才会触发中断。如果你还想触发下一个中断就必须把电平先置高。

当设计中断结构时,你要记住边沿触发适用于那些器件发出的中断请求信号不需要软件清除的场合。最为普遍的例子是系统的时标，这种信号一般由实时时钟电路产生，这些器件一般提供一个占空比为50%的信号（即信号的一半是高电平，另一半为低电平）。如果使用电平触发，将将产生很多中断，这样即使不扰乱程序的运行也将浪费系统的资源。

还有一种类似的情况是解码器，系统通过解码器电路采样串行输入信号，并把它转化成并行输出。每当信号达到某个标准的时候，将产生一个中断信号，问题在于达到标准的信号是一个持久的信号，如果设置成电平触发会引发一连串的中断。

在器件要求中断很频繁的时候，电平触发方式就比较好。假使一个器件，周期性的有高频率的中断请求，在你这个中断服务程序还没完成的时候，下一个中断请求又来了，这样就不必把中断请求信号线置高。如果设置为边沿出发方式，你就检测不到中断信号。

电平方式在多个器件共用一个中断入口的情况下比较有用。当正在执行一个中断服务程序的时候，另外一个中断请求又来了，这样信号线一直被置低，边沿触发方式将检测不到这个中断。这时用电平触发方式就比较好，因为信号线一直被置低，当上一个中断服务程序完成之后，将立即执行下一个服务程序。只要有中断请求，这可使程序提供任何中断服务。这个过程将重复直到执行完所有的中断服务。

经常性的，你的系统所要处理的中断信号比现有的中断引脚要多。这种情况在扩充了中断引脚之后仍然存在，这时利用一些方法来共用中断引脚。你需要知道中断源的数量，中断触发的速度和在你的系统中加入什么器件等。

4 共用中断

至少有3种方法来在多个输入之间共用中断信号，每种方法都需要增加相应的组件。

假使有两个输入信号，当它们请求中断服务时把信号线电平置低，当中断服务程序完成之后再把信号线置高。用与门把这两个信号连起来，再把输出接到INT1。为了让处理器分辨中断请求来自哪个信号，分别把这两个信号接到控制器输入端口的引脚上。下面的例子采用的是P1.0和P1.1。

这里设要求中断服务的器件直到中断服务完成之后才将信号线置高,因为在第一个器件要求中断服务之后,第二个器件还可以申请中断。这要求把INT1设置为电平触发，或在中断程序结束前检测P1.1和P1.0口，这样两个中断都将被执行。如果使用边沿触发的话，当一个中断正在执行时，又产生另一个中断，如果在中断程序结束前不检测P1.1和P1.0口，这个中断将不会被执行。

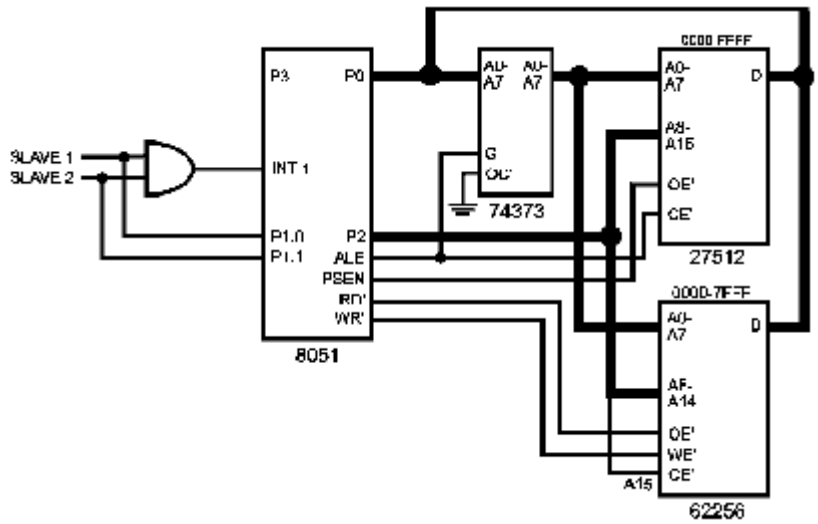

图0-1 INT1被共用

像上面的例子，把中断设置为电平触发，中断服务程序见表0-1。注意如何通过改变检测顺序来建立中断的优先级。另外，一旦完成了第一个中断服务程序之后，将检测低优先级的输入，因为中断为电平触发，如果在高优先级的中断执行期间，又来一次这个中断，那么将始终执行这个中断。

列表 0-1

```
sbit SLAVE1 = P1^0;      // 输入信号命名
sbit SLAVE2 = P1^1;

void int1_isr(void) interrupt 2 {
   if (!SLAVE1) {        // 先检测 slave1
     slave1_service();
   }
   if (!SLAVE2)
     slave2_service();
   }
}
```

可以更改中断服务程序，通过加入 do...while循环语句，只要中断申请存在就不退出中断服务程序。这将导致一直中断，系统将不能进行其它工作。我们在设计系统时，要整体的进行考虑，合理的执行中断，而不应让中断占据所有的系统资源，因为系统还要做其它工作。

前面的共享中断的方法还可以进行扩展。把所有输入信号接到一个与门上，并给每个信号分配一个端口引脚，如果碰到引脚不够用的情况，可把引脚接到数据锁存器上，还是以电平方式触发中断。这将使系统在软件和硬件上都变得复杂。主要不同的是将通过数据锁存器来读取输入信号。

锁存器的优点是可让你加入一些新的硬件，产生中断的信号被硬件记录下来，然后通过软件从锁存器中读取记录。采取什么中断方式决定于接到中断输入口上信号的性质。

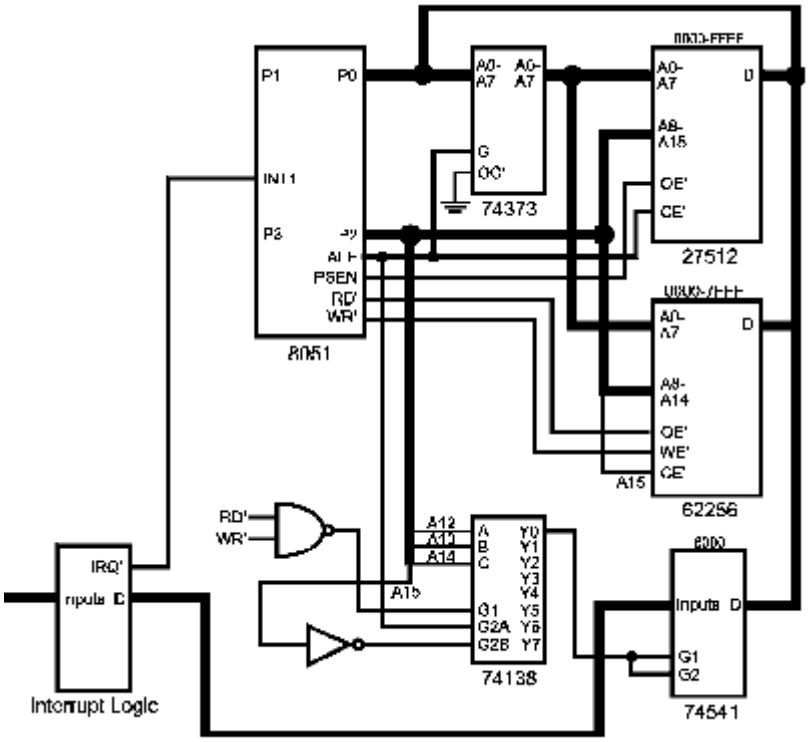

图0-2 共享外部中断

在这个结构中，中断服务程序从地址为8000H的锁存器中读入数据以决定哪个中断源要求中断，并可根据查寻的先后次序决定那个中断被优先执行。下面是中断服务程序。

列表 0-2

```
#define INTREG 0x8000
unsigned char bdata intmask; // 声明一个可位寻址变量存放中断请求记录
sbit signal0 = intmask^0;    // 设置位变量访问记录
sbit signal1 = intmask^1;
sbit signal2 = intmask^2;
sbit signal3 = intmask^3;
sbit signal4 = intmask^4;
sbit signal5 = intmask^5;
sbit signal6 = intmask^6;
sbit signal7 = intmask^7;

void int1_isr(void) interrupt 2 {
     intmask=XBYTE[INTREG];  //读锁存器数据以决定
                             //中断的原因
     if (signal0) {          // 检测所有的中断源

         signal0_isr();
     }
      ...
     if (signal7) {
       signal7_isr();
     }
```

```
    reset_int();              // 执行中断逻辑的复位功能
  }
```

中断逻辑控制的硬件要根据系统而定。有些像上面的例子一样，中断请求直到中断服务程序完成之后才被释放，而有些中断请求信号只是一个脉冲信号。这样硬件必须锁住下降沿脉冲，并产生中断请求信号。，当中断执行完毕之后，软件清除中断逻辑电路中的中断请求信号。只要有中断请求没有被响应，并清除相应的中断信号，中断请求逻辑电路就将一直发出中断请求。这样，中断触发方式应该被设置成电平触发。

上面例子中中断逻辑的实现很像商业性的中断控制，主要的区别是上面的中断逻辑，每个输入信号的中断触发方式将不能改变。如果你要把第3个信号的中断触发方式有电平触发改变成边沿触发，那你的硬件电路就要该变。一个好的中断系统应该允许改变每个中断的触发方式，就像8051可以设置INT0和INT1那样。

中断控制器应该可以通过硬件来使能或禁能中断。当你要关闭某个中断时，不再需要修改软件或增加硬件，这个功能已经包含在中断控制器中。使用这种模块的系统要通过系统总线和它接口，如果一个系统使用了中断控制器，那么系统在决定中断源时会花较长的时间，这样就增加了中断的延时，所以对中断反应速度要求很高的输入应该直接接到处理器上。

6 扩充外部中断数

尽管Intel认为8051的外部中断数不应超过两个，但肯定有方法可以使你的外部中断数超过5个。有两个简单的方法：一是把定时/计数器中断做成外部中断，二是把串行口中断做成外部中断。当然如果你还要使用他们以前的中断功能就不应这样做。如果你需要一个定时器和串行口，那在设计系统时，可把另一个定时器作为外部中断。

扩展外部中断最简单的方法就是把定时器设置为计数模式，然后把信号接到计数器相应的引脚上（T0或T1）。为了使每出现一个从高到低的脉冲的时候产生一个中断，把定时器设置为自动重装模式，令重装值为FFH。当计时器检测到从高到低的脉冲时，定时器将溢出，这时将产生一个中断请求。代码如下：

列表 0-3

```
#include <reg51.h>
void main(void) {
...
    TMOD=0x66; // 两个定时/计数器都设置成8位模式
    TH1=0xFF;  // 设定重装值
    TH0=0xFF;
    TL0=0xFF;
    TL1=0xFF;
    TCON=0x50; // 开始计数
    IE=0x9F;   // 中断使能
...
}
/*********************************************************************
定时器0中断服务程序
*********************************************************************/
void timer0_int(void) interrupt 1 {
```

```
...
}
/*********************************************************************
定时器1中断服务程序
*********************************************************************/
void timer1_int(void) interrupt 3 {
    while (!T1) {       // 确保中断被清除
        ...
    }
}
```

这种方法还是有一定的限制的，第一点，它只能是边沿出发，所以当你需要的是一个电平触发的中断时，就要在中断中不断的对T0或T1进行采样，直到它们变为高。第二点，检测到下降沿和产生中断之间有一个指令周期的延时，这是因为在检测到下降沿一个指令周期之后，计时器才加1。

如果你使用的8052或8051单片机有多个定时器，而且有外部引脚，可以用这种方法来扩充边沿触发的外部中断。值得重申的一点是，当使用定时器作为外部中断时，它以前的功能将不能使用了，除非你用软件对它进行复用。

使用串行口作为外部中断不像使用定时器那样直接。RXD引脚将变成输入信号，检测从高到低的电平跳变。把串行口设置为模式2，当检测到从高到低的电平跳变是，8位数据传输时间过后将产生中断，当中断发生后由软件把RI清零。下面是对UART设置和ISR结构的代码。

列表 0-4

```
#include <reg51.h>
void main(void) {
...
    SCON=0x90; // 模式 2 允许接收
    IE=0x9F;   // 中断使能
...
}

void serial_int(void) interrupt 4 {
    if (!_testbit_(RI)) {
        ...
    }
}
```

像定时器系统一样，用串行口中断作为外部中断也有它的缺点。第一，中断只能是边沿触发。第二，输入信号必须保持5/8位传输时间为低，因为串行口必须确认输入信号是一个起始位。第三，检测到电平跳变之后要等8个位传输时间后UART才请求中断。还有，信号为低的时间不应超过9位数据传输时间。对UART来说，这种方法相当于从RXD脚传送进一个无效字节，这样对时间的要求更高了。这些限制取决于你的系统的的频率，因为传输的波特率取决于系统频率。当UART的模式改变和使用内部定时器时会有不同的时间限制，但延时只会加长不会缩短。

7 中断服务程序

很多新手在设计中断服务程序时不知道该注意些什么。主要问题是哪些功能应该放在中断程序中哪些功能应该放在主程序中。要把握好这一点可不是那么容易。一般来说，中断服务程序应该做最少量的工作。这样作有很多好处，首先，你的系统对中断的反应面更宽了。有些系统如果丢失中断或对中断反应太慢将产生十分严重的后果，这时，有充足的时间等待中断是十分重要的。其次，它可使你的中断服务程序的结构简单，不容易出错。中断程序中放入的东西越多，它们之间越容易起冲突。简化中断服务程序意味着你的软件中将有更多的代码段，但你可把这些都放入主循环中。中断服务程序的设计对你系统的成败有至关重要的作用。你要仔细考虑各中断之间的关系和每个中断执行的时间。特别要注意那些对同一个数据进行操作的ISR。

假设你的系统从UART接收了一系列数据，需要从中得到重要的信息并响应它们。中断服务程序从SBUF中读取数据并把它放到循环队列中。软件的主调用层负责检查队列，取得数据，进行分析。当信息接收完毕，然后进行相应的处理。也可有ISR进行数据分析，再把数据放入队列中由主程序进行处理。但我不主张使用第二种方法，那样将花费很多时间。

有些时候，由于时间的限制或和其它中断的关系的原因，无法将一些操作从ISR中分离出来。例如，有一个系统当外围电路接收到数据之后申请中断，并且每20ms 向处理器发送一个数据单位，我想你应该会接收完所有的数据后才离开中断，否则很容易丢失数据。可以利用8051的中断优先级来解决这个问题。

另外一个留给ISR做的应该是对共享数据的操作。举个例子，如果一个系统有好几个中断，其中有两个中断有同样的优先级，并对同一个数据结构进行操作。当A/D转换单元完成转换之后将引发其中一个中断，每10ms发生一次。系统记录转换结果，并把结果串行输出。另一个中断是系统时标，检查共用的数据结构中是否有新的转换数据。当有新的数据出现时，把数据放入打包，并初始化串行传输。可以看出这两个ISR不应同时使用队列。在这个例子中输入ISR读取数据并完成数据的入队列操作。另一个ISR从队列中取数据并构造消息，初始化串行口。

8 结论

这章主要讨论了如何增强8051的中断功能.把这些技巧和以前的讨论结合起来，如仿真外部中断优先级，可使你拥有比8051设计者想到的更多功能。在设计系统的中断系统的时候，应该注意输入信号和8051中断源的匹配，同时还有软件的设计，像选择中断优先级，中断服务程序的设计，软件和硬件应该结合起来设计。总之，中断系统的设计对实时时钟嵌入式系统来说是十分关键的。

第7章 串行口

1 介绍

8051系统的主要传输方式不是并行口或共享存储区，而是8051的串行传输方式。第二章提到过，内置的UART是十分灵活的，可以和其它系统进行高速的通信，这章将讨论在系统间进行数据传输的软件设计方法。

2 慢速串行口和PC的接口

在很多嵌入式应用中，处理器把时间和数据报告给主机，通常是PC。主机也通过串行连接向处理器传输命令和数据。通常使用像RS-232这样的电压标准在8051系统和通用系统间进行通信。如果通信线路不是很长的话，8051可以不需要RS-232驱动器，而只需要简单的电路就可与PC通信。很多PC系统并不完全遵循RS-232电压标准，这将简化电路接口。从PC输出的12V电压数据通过降压变为5V以下。

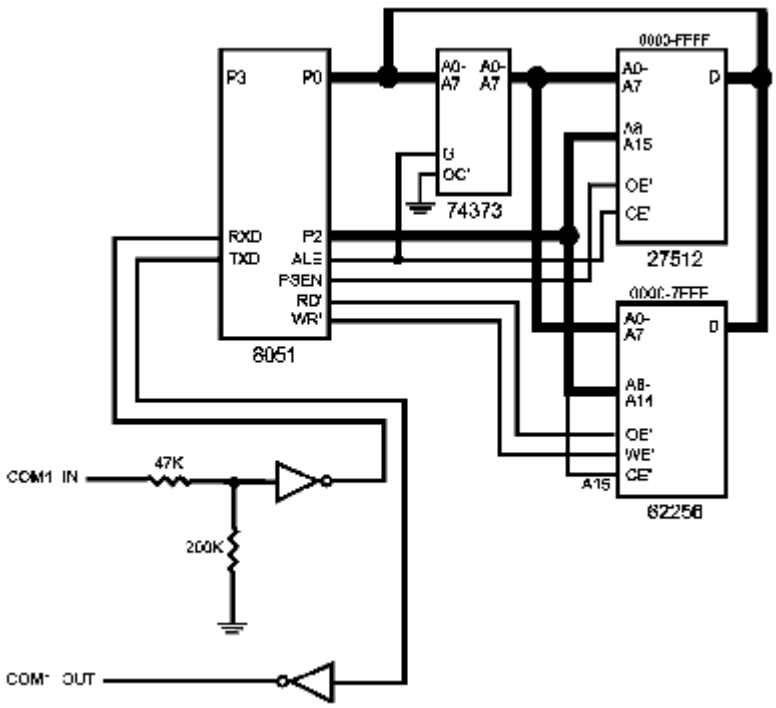

图0-1 PC接口

当简单的接口电路设计好了之后，要设计相应的软件来控制数据的传输。处理输入数据最简单的方法是假设你的传输协议传输的第一个字节是要传输的字节数，接收完第一个字节产生串行传输中断，然后以查询的方式接收输入数据。对输出数据也用相似的方法。当串行传输开始时，向SBUF中写入一个字节数，然后查询SCON看什么时候开始传送下一个数据，当所有字节传送完毕后，结束循环。

上面的软件设计适用于只处理串行通信的系统，这种软件设计结构比较简单，但是对于复杂的系统查询方式就不适用了。下面的设计更好。接收数据时，每个输入字节产生一个串行中断，中断服务程序从SBUF中读取数据，并确认数据的有效性。当数据有效时，把数据放入队列中由主程序去执行。发送数据用类似的方法，把要发送的数据放入队列中。第一个字节发送完后产生中断，只要队列中还有数据，中断服务程序从队列中读出一个字节，写入SBUF。

这个系统允许处理器除了串行传输之外还可处理其它任务，一般来说串行口和其它外围器件比起来是一个很慢的设备，只要你的串行波特率不是特别快（如300K），每个字节间就有足够的时间处理其它任务。

设前面第4章所讲的时钟系统是监控系统的一部分，该监控系统通过PC查询其它设备。如图0-2所示。在这里我们只关心时钟部分。

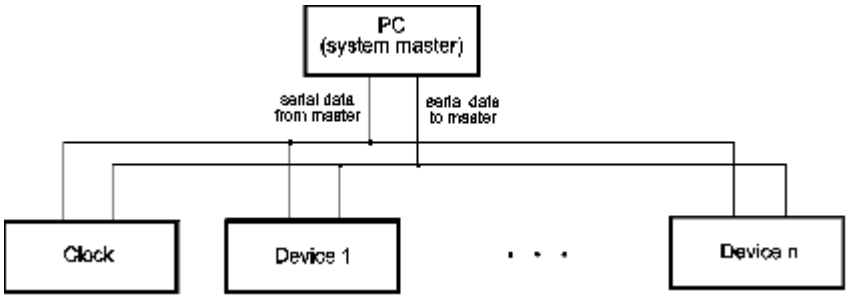

图0-2主/从串行通信

在这个新的监控系统中，时钟通过RS-232和PC进行通信，时钟的设计如图0-3。

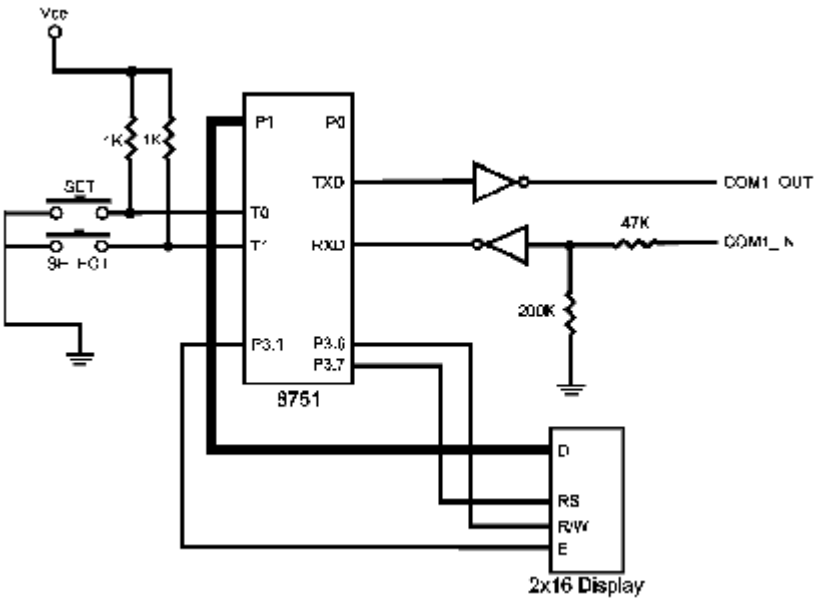

图0-3 时钟作为从设备

PC从时钟处读取数据，设置时钟的时间，复位时间为0，传送32个字符信号进行显示。应该注意，串行通信线路上不止时钟一个设备，要把自己的数据和其它设备的数据区分开。所以被传送数据的结构应该使设备可以鉴别数据是否是自己的。被传送数据的第一个字节是同步信号，包含了被寻址器件的地址，时钟的地址是43H。信息中还包含了命令字节，数据的多少，数据本身和一个校验字节。典型的信息结构如下所示：

Field	Description
Synchronization Byte	Must be set to 33H
Unit Address	Address of destination unit. The clock's address == 43H, the PC's address == 01H.
Command	Indicates the message being sent.
Data Size	Indicates number of bytes in following data block (0 to 255 bytes). If no extra data is required with this message, this byte will be 00H.
Data Block	Extra data required with this message. The number of bytes must equal the value of the Data Size field.
Checksum	Single byte addition of all previous bytes in the message (including the Synchronization Byte) ignoring rollover.

表0-1

对所有从PC传送过来的命令，必须返回一个应答信号。时钟对上面的列出的4个信号负责(时间请求,时间设置,时间复位,时间显示).信息的格式如下:

Message: Reset time to zero. Command 01H. This command resets the time to 00:00:00.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	01H
Data Size	00H
Checksum	77H

表0-2

Message: Time Set. Command 02H. This command sets the time to the values provided.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	02H
Data Size	03H
Hours	Indicates current hours, must be 0 to 23.
Minutes	Indicates current minutes, must be 0 to 59.
Seconds	Indicates current seconds, must be 0 to 59.
Checksum	7BH + Hours + Minutes + Seconds

表0-3

Message: Time Request. Command 03H. This command requests the current time. Direction is from the PC to the clock.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	03H
Data Size	00H
Checksum	79H

表0-4

Message: Time Request. Command 03H. This command reports current time. Direction is from the clock to the PC.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	03H
Data Size	03H
Hours	Indicates current hours, must be 0 to 23.
Minutes	Indicates current minutes, must be 0 to 59.
Seconds	Indicates current seconds, must be 0 to 59.
Checksum	7CH + Hours + Minutes + Seconds

表0-5

Message: Text Display. Command 04H. This command specifies a 32 character text message to display on the LCD panel.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	04H
Data Size	20H
Text	32 ASCII characters for display. Must be space padded.
Checksum	9AH + • (Text field bytes)

表0-6

Message: Acknowledge. Command FFH. This command is sent following receipt of any valid command which does not require a return of data to the PC.	
Field	**Value**
Synchronization Byte	33H
Unit Address	43H
Command	FFH
Data Size	01H
Return	Command number of the message being acknowledged
Checksum	76H + Return

表0-7

明白了数据流和时钟处理器的责任，现在可以设计中断服务程序了。中断服务程序对数据流进行分析，在此可以使用一个简单的有限状态图(FSA)。FSA根据输入从一个状态转移到另一个状态的软件。FSA有一个初始状态，它寻找同步字节和与下部分信号相关的中

间信号，初始状态将读取校验和字节并修改它。如果所接收字节不遵守有效信号的结构，FSA就回到初始状态并开始寻找下一个同步字节。

这种串行接收的原理很容易实现。如果用C来写ISR的话，我们要声明一些变量来保存系统当前的状态。给每个状态一个号码，把号码保存在变量中。当输入字节引发中断后FSA根据保存的系统状态决定系统的下一状态。把状态变量的类型声明为unsigned char。如果用汇编编写程序的话，可以通过更加有效的跳转查表来完成。但你会发现程序大小和执行速度不比用C编写的程序好多少，但如果你处理的是高速的串行通信系统就另当别论。

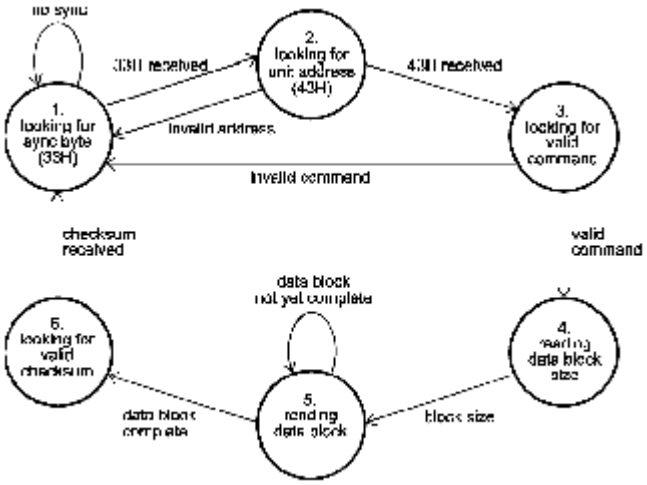

图0-4 接收FSA

在本例中，时钟以波特率9600传送数据，传送一个字节只需要1.042ms。晶振频率为11.059MHz，指令执行周期为1.085us。在每个中断之间有960个指令周期，有足够的时间保存FSA。下面是串行ISR的代码：

列表 0-1

```
// 定义FSA状态常量
#define FSA_INIT     0
#define FSA_ADDRESS  1
#define FSA_COMMAND  2
#define FSA_DATASIZE 3
#define FSA_DATA     4
#define FSA_CHKSUM   5

// 定义信号分析常量
#define SYNC 0x33
#define CLOCK_ADDR 0x43

// 定义输入命令
#define CMD_RESET    0x01
#define CMD_TIMESYNC 0x02
#define CMD_TIMEREQ  0x03
#define CMD_DISPLAY  0x04
#define CMD_ACK      0xFF

#define RECV_TIMEOUT 10     /* define the interbyte timeout */
```

```
unsigned char
recv_state=FSA_INIT,        // 当前状态
recv_timer=0,               // 时间计数
recv_chksum,                // 保存当前输入的校验值
recv_ctr,                   // 接收数据缓冲区的索引
recv_buf[35];               // 保存接受数据

unsigned char code valid_cmd[256]={ // 数组决定当前的命令字节是否有效
                                    // 如果相应的输入是1，那么命令字节
                                    // 有效
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 00 - 0F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 10 - 1F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 20 - 2F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 30 - 3F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 40 - 4F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 50 - 5F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 60 - 6F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 70 - 7F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 80 - 8F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 90 - 9F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // A0 - AF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // B0 - BF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // C0 - CF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};

/*********************************************************************
功能: serial_int
描述: 运行串行口 FSAs.
参数: none.
返回: nothing.
影响: none.
*********************************************************************/
void serial_int(void) interrupt 4 {
unsigned char data c;
  if (_testbit_(TI)) {
                          // 处理发送任务
  }
  if (_testbit_(RI)) {
    c=SBUF;
    switch (recv_state) {
    case FSA_INIT:                // 是否是同步字节
```

```
  if (c==SYNC) {                // 是同步字节
      recv_state=FSA_ADDRESS;   // 进入下一个状态
      recv_timer=RECV_TIMEOUT;  // 最大间隔时间
      recv_chksum=SYNC;         // 设置初始化校验值
  }
  break;
case FSA_ADDRESS:               // 是否是地址
  if (c==CLOCK_ADDR) {          // 是时钟地址
      recv_state=FSA_COMMAND;   // 进入下一个状态
      recv_timer=RECV_TIMEOUT;  // 最大时间间隔
      recv_chksum+=c;           // 保存校验值
  } else {                      // 信息不是给时钟的
    recv_state=FSA_INIT;        // 回到初始状态
    recv_timer=0;               // 清除最大时间间隔
  }
  break;
case FSA_COMMAND:               // 是否是命令
  if (!valid_cmd[c]) {          // 确认命令是否有效
  recv_state=FSA_INIT;          // 复位 FSA
  recv_timer=0;
} else {
  recv_state=FSA_DATASIZE;      // 进入下一个状态
  recv_chksum+=c;               // 更新校验值
  recv_buf[0]=c;                // 保存命令
  recv_timer=RECV_TIMEOUT;      // 设置时间间隔
  }
  break;
case FSA_DATASIZE:              // 发送的字节数
  recv_chksum+=c;               // 更新校验值
  recv_buf[1]=c;                // 保存字节数
  if (c) {                      // 如果有数据段
  recv_ctr=2;                   // 设置查询字节
  recv_state=FSA_DATA;          // 进入下一个状态
  } else {
    recv_state=FSA_CHKSUM;
  }
  recv_timer=RECV_TIMEOUT;
  break;
case FSA_DATA:                      // 读入数据
  recv_chksum+=c;                   // 更新校验值
  recv_buf[recv_ctr]=c;             // 保存数据
  recv_ctr++;                       // 数据计数值
  if ((recv_ctr-2)==recv_buf[1]) { // 接收数据计数器减偏移量
                                    // 是否等于datasize
```

```
      recv_state=FSA_CHECKSUM;          // 数据接收完毕
      }
      recv_timer=RECV_TIMEOUT;          // 设置时间间隔
      break;
    case FSA_CHECKSUM:                  // 读校验字节
      if (recv_chksum==c) {             // 核对校验字
        c=1;                            // 用c表明是否要建立应答信号
        switch (recv_buf[0]) {          // 按指令执行
          case CMD_RESET:               // 复位时钟为0
            break;
          case CMD_TIMESYNC:            // 设置时钟
            break;
          case CMD_TIMEREQ:             // 报告系统
            break;
          case CMD_DISPLAY:             // 显示 ASCII 信息
            break;
        }
        if (c) {
          // 应答
        }
      }
    default:
      recv_timer=0;                     // 复位 FSA
      recv_state=FSA_INIT;
      break;
    }
  }
}
```

所运行的代码充分反应了图0-4中所展示的模型。当然应该还有指令的执行代码和输出数据的代码，这里只是给出你接收数据代码的结构。

向PC回传数据更加简单，由串行中断服务程序完成。时钟假设,同一时刻PC只会传送一个有效命令给它，这样就不必担心维护一大堆输出数据了。这个假设简化了这个例子，当需要发送数据的时候，只需要把数据放入发送缓冲区中，并设置一个变量保存发送的字节数。把第一个字节写入SBUF并设置校验字节。向SBUF写第一个字节就像启动了水泵一样：它将产生第一个中断，当触发了第一个中断之后，串行中断将自动完成数据的发送。下面是串行口中断的代码结构：

列表 0-2

```
// 定义信号分析常量
#define SYNC       0x33
#define CLOCK_ADDR 0x43

unsigned char
trans_buf[7],      // 保存输出数据
trans_ctr,         // 数据缓冲区索引
```

```
trans_size,        // 发送数据的个数
trans_chksum;      // 计算输出校验
/*********************************************************************
功能: serial_int
描述：运行串行口FSAs
参数: 无.
返回: 无.
影响: 无.
*********************************************************************/
void serial_int(void) interrupt 4 {
    unsigned char data c;
        if (_testbit_(TI)) {            // 发送中断
           trans_ctr++;                 // 增加数据索引
           if (trans_ctr<trans_size) {  // 数据是否发送完毕
              if (trans_ctr==(trans_size-1)) { // 输出校验字节
                  SBUF=trans_chksum;
              } else {
              SBUF=trans_buf[trans_ctr];          // 发送当前字节
              trans_chksum+=trans_buf[trans_ctr]; // 更新校验字节
              }
           }
        }
        if (_testbit_(RI)) {
          c=SBUF;
          switch (recv_state) {
          // 接收FSAs
          case FSA_CHECKSUM:            // 读校验字节
            if (recv_chksum==c) {       // 核对校验字节
              c=1;                      // 用c表明是否要建立应答信号
              switch (recv_buf[0]) {    // 执行指令
              case CMD_RESET:           // 复位时钟
                 break;
              case CMD_TIMESYNC:        // 设置时钟
                 break;
              case CMD_TIMEREQ:        // 报告时间
                 c=0;
                 break;
              case CMD_DISPLAY:        // 显示 ASCII 信息
                 break;
              }
             if (c) {                   // 建立应答
              trans_buf[0]=SYNC;        // 信息头
              trans_buf[1]=CLOCK_ADDR;
              trans_buf[2]=CMD_ACK;
```

```
            trans_buf[3]=1;
            trans_buf[4]=recv_buf[1];// 被回应的命令
            trans_ctr=0;              // 设置缓冲区指针到第一个字节
            trans_size=6;             // 总共发送6个字节
            SBUF=SYNC;                // 发送起始字节
            trans_chksum=SYNC;        // 初始化校验值
           }
         }
         default:
           recv_timer=0;
           recv_state=FSA_INIT;
           break;
          }
        }
      }
```

如你所见，输出数据的ISR代码十分简单，代码所占的空间也很小。和接收数据一样，校验字节也在数据的传输过程中建立。

ISR程序在时钟系统的限制下顺利的运行。注意中断中命令的执行可以避免设计上的很多问题。串行中断服务程序和定时器中断服务程序一样都可以修改时间，把这两个中断的优先级设为一样，这样任何一个中断在修该时间的时候都不用担心另一个也在这么做。对数据的处理可以在中断程序中，但是命令的执行应该放到主程序中去执行，因为这些命令的执行会花去太多的时间，这样就很可能丢失其它的中断。假设PC送一系列字符给LCD显示，时钟就有可能丢失一两个时标中断，因为显示要花去很多时间。在一些更复杂的系统中，应该把输入数据放入队列中，然后由主程序去处理。在这里，我们认为，PC直到被告知对方已收到当前信息后才发送下一个信息，。这样队列就仅仅是一个当前信息的缓冲区。

新的中断服务程序和以前的很相似，不同之处在于当接收完毕数据之后，它会把数据拷贝到另一个缓冲区中并置位缓冲区有效标志位，以前在中断服务程序中的命令执行代码现在放到新的功能段中，当主程序检测到第二个缓冲区中有数据的时候调用该功能段。下面是新的ISR代码。

列表 0-3

```
// 定义状态常量
#define FSA_INIT          0
#define FSA_ADDRESS       1
#define FSA_COMMAND       2
#define FSA_DATASIZE      3
#define FSA_DATA          4
#define FSA_CHKSUM        5

// 定义信号分析常量
#define SYNC 0x33
#define CLOCK_ADDR 0x43

// 定义命令常量
```

```
#define CMD_RESET    0x01
#define CMD_TIMESYNC 0x02
#define CMD_TIMEREQ  0x03
#define CMD_DISPLAY  0x04
#define CMD_ACK      0xFF
#define RECV_TIMEOUT 10 /*定义字节间的最大时间间隔 */

unsigned char
      recv_state=FSA_INIT,  // 当前状态
      recv_timer=0,         // 时间间隔计数
      recv_chksum,          // 输入数据的校验字节
      recv_size,            // 输入数据字节数
      recv_ctr              // 数据缓冲区指针
      recv_buf[35];         // 输入数据缓冲区

unsigned char
      trans_buf[7],         // 输出数据缓冲区
      trans_ctr,            // 输出数据指针
      trans_size,           // 输出数据字节数
      trans_chksum;         // 输出数据的校验字节

unsigned char code valid_cmd[256]={ // 如果输入命令有效则为1
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 00 - 0F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 10 - 1F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 20 - 2F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 30 - 3F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 40 - 4F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 50 - 5F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 60 - 6F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 70 - 7F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 80 - 8F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 90 - 9F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // A0 - AF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // B0 - BF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // C0 - CF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};

/*********************************************************************
功能: serial_int
描述: 运行串行口 FSAs.
参数: 无.
```

返回：无.
影响：无.
**********/
```
void serial_int(void) interrupt 4 {
    unsigned char data c;
    if (_testbit_(TI)) {          // 输出中断
      trans_ctr++;                // 输出缓冲区指针加1
      if (trans_ctr<trans_size) { // 数据是否输出完毕
      if (trans_ctr==(trans_size-1)) { // 输出校验字节
        SBUF=trans_chksum;
      } else {
        SBUF=trans_buf[trans_ctr]; // 输出当前字节
        trans_chksum+=trans_buf[trans_ctr]; // u更新校验字节
        }
      }
    }
    if (_testbit_(RI)) {
      c=SBUF;
      switch (recv_state) {
      case FSA_INIT:
        if (c==SYNC) {     // 同步字节
          recv_state=FSA_ADDRESS;  // 下一个状态
          recv_timer=RECV_TIMEOUT;
          recv_chksum=SYNC;
        }
        break;
      case FSA_ADDRESS:
        if (c==CLOCK_ADDR) { // 时钟地址
        recv_state=FSA_COMMAND;
        recv_timer=RECV_TIMEOUT;
        recv_chksum+=c;
        } else { // 不是给时钟的
          recv_state=FSA_INIT; // 返回初始状态
          recv_timer=0;
          }
       break;
     case FSA_COMMAND:
       if (!valid_cmd[c]) { //
         recv_state=FSA_INIT; /
         recv_timer=0;
       } else {
         recv_state=FSA_DATASIZE;
         recv_chksum+=c;
             recv_buf[0]=c;     // s保存命令
```

```
        recv_timer=RECV_TIMEOUT;
        }
       break;
     case FSA_DATASIZE:          // 字节的个数
       recv_chksum+=c;
       recv_buf[1]=c;
       if (c) {            // 是否有数据
         recv_ctr=2;
         recv_state=FSA_DATA;
       } else {
         recv_state=FSA_CHKSUM;
       }
       recv_timer=RECV_TIMEOUT;
       break;
     case FSA_DATA: // 读取数据
       recv_chksum+=c;
       recv_buf[recv_ctr]=c; // 保存数据
       recv_ctr++;
       if ((recv_ctr-2)==recv_buf[1]) { //数据接收完毕
         recv_state=FSA_CHECKSUM;
       }
         recv_timer=RECV_TIMEOUT;
         break;
     case FSA_CHECKSUM: // reading in checksum
       if (recv_chksum==c) { // 校验字节核对正确
        memcpy(msg_buf, recv_buf, recv_buf[1]+2);
        msg_buf_valid=1;
       }
     default: //复位
        recv_timer=0;
        recv_state=FSA_INIT;
        break;
     }
   }
 }
```

现在ISR只负责分析输入数据和从如何缓冲区发送数据。命令的执行代码被放入功能段中由主程序去调用。下面为该功能段和主程序的代码。

列表 0-4

```
/*****************************************************************
Function: execute_cmd
Description: 命令执行
Parameters: 无.
Returns: 无.
Side Effects: 无.
```

```
**********************************************************************/
void execute_cmd(void) {
   bit need_ack=1;
   switch (recv_buf[0])
   case CMD_RESET:       // 复位
     EA=0; // 禁止中断
     curtime.sec=curtime.min=curtime.hour=0;
     timeholder=curtime;
     EA=1; // 开放中断
     break;
   case CMD_TIMESYNC: // 设置时间
     EA=0;
     curtime.hour=recv_buf[3];
     curtime.min=recv_buf[4];
     curtime.sec=recv_buf[5];
     timeholder=curtime;
     EA=1;
     break;
   case CMD_TIMEREQ:   // 报告时间
     trans_buf[0]=SYNC;
     trans_buf[1]=CLOCK_ADDR;
     trans_buf[2]=CMD_TIMEREQ; // 发送当前时间
     trans_buf[3]=3;
     EA=0;
     trans_buf[4]=curtime.hour;
     trans_buf[5]=curtime.min;
     trans_buf[6]=curtime.sec;
     EA=1;
     trans_ctr=0;
     trans_size=8; // 发送8个字节
     need_ack=0;
     break;
   case CMD_DISPLAY: // 显示 ASCII
     recv_buf[34]=0; // 在字符串最后设置一个空字符以结束显示
     printf("\xFF%s", &recv_buf[2]); // 显示字符串
     display_time=100;
     break;
   }
   if (need_ack) {  // 建立应答消息
     trans_buf[0]=SYNC;
     trans_buf[1]=CLOCK_ADDR;
     trans_buf[2]=CMD_ACK;
     trans_buf[3]=1;
     trans_buf[4]=recv_buf[0];
```

```
      trans_ctr=0;    // 发送数据指针
      trans_size=6;
     }
   SBUF=SYNC;    // 开始发送
   trans_chksum=SYNC; // 校验字节
/*********************************************************************
Function: 主程序
Description: 初始化8051，使能相应的中断源，进入空闲模式
             每次进入空闲模式之前是否要运行命令或更新显示
Parameters: 无.
Returns: 无.
*********************************************************************/
void main(void) {
   disp_init();       // 初始化显示
   TMOD=0x21;         // 定时器016位模式，定时器1
                      // 为波特率发生器
   TCON=0x55;         // 开启定时器
   TH1=0xFD;          // 定时器1波特率为9600
   SCON=0x50;         // 串行口模式1
   IE=0x92;           // 使能定时器0中断
   for (;;) {
     if (_testbit_(msg_buf_valid)) { // 有没有新的数据
       execute_cmd();                // 执行命令
     }
     if (disp_update) {
       disp_time();                  // 更新显示
     }
   PCON=0x01;                    // 进入空闲模式
   }
}
```

上面在主程序中处理信息的方法实现起来十分简单，在由输入输出驱动的中断系统中十分重要。因为和PC之间的接口相对比较慢，主循环有足够的时间去执行其它的任务。不可否认，有很多系统的中断比时钟系统要多得多，下面将讨论高速串行数据传输的应用。

3 高速串行I/O

前面所讨论的系统对时钟定期的查询从PC来的串行信号并非每次都是针对时钟的，因为总线上还有其它器件。假设系统的设计者认为PC正在忙于处理用户接口和数据，所以和器件的通信很慢。为了改进它，设计一个基于8051的简单系统。这个电路的工作是代替PC作为主控器。这个新器件称为系统监控器。系统的框图看起来复杂一些，原来的时钟系统不变。

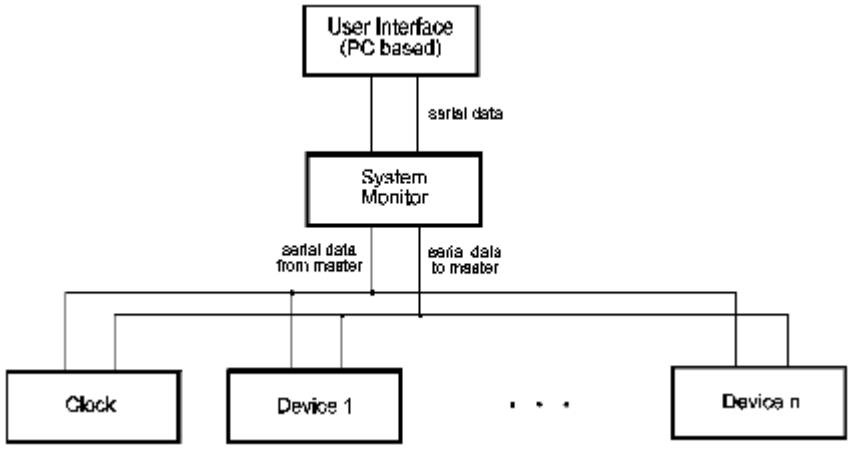

图0-5　新的串行网络

仍然通过串行连接进行查询，但波特率更高一些。系统中的所有器件都使用11.059MHz晶振，串行口可彼此直接连接，这意味着和PC连接时所需要的分压和反向电路不再需要了。电路图如下。

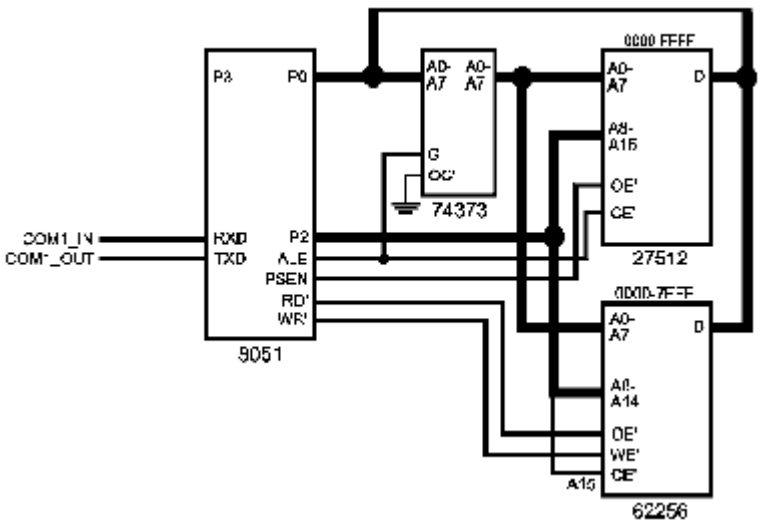

图 0-6 8051作为高速从器件

因为是和其它8051相连，UART可用模式2进行通信，波特率将达到345593.75baud。发送每个字节的时间为31.829us（不到30个指令周期），而波特率为9600时，每发送一个字节需要1.04ms。因为发送时间缩短了，器件之间的接口要稍做改动，使总线上可容纳多个器件。8051的串行口被初始化为模式2，一个起始位，8个数据位，一个校验位，一个停止位。当处理器接收完前面的数据时，校验位用来产生中断。每个器件都要按这种格式输出数据，还要依照以下条件，第一每个信息的同步字节，校验位必须置位。第二信息中的其它字节的校验位必须清零。当收到校验位为1的字节时，所有器件都产生中断。

因为数据传送速度有了很大的提高，时钟的串行接收过程将要有所改变。当校验位置位的字节引发中断时，其余的数据将被保存在数据队列中。

修改时钟的主程序来适应串行通信模式。注意，定时器0不用来产生波特率，可用做其它用途。我们可以用它来对每个字节之间的时间间隔计时，确保时钟不做无限制等待。

列表 0-5

```
// 信号常量
#define SYNC        0x33
#define CLOCK_ADDR 0x43
```

```
// 命令常量
#define CMD_RESET    0x01
#define CMD_TIMESYNC 0x02
#define CMD_TIMEREQ  0x03
#define CMD_DISPLAY  0x04
#define CMD_ACK      0xFF

// 字节间的最大时间间隔128指令周期
#define TO_VAL       0x80

unsigned char data recv_chksum,                // 校验字节
                  recv_buf[35];                // 输入数据
unsigned char trans_buf[7],                 // 输出数据
         trans_ctr,                         // 输出数据缓冲区指针
         trans_size,                        // 输出字节数
         trans_chksum;                      // 输出校验字节
unsigned char code
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 00 - 0F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 10 - 1F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 20 - 2F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 30 - 3F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 40 - 4F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 50 - 5F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 60 - 6F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 70 - 7F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 80 - 8F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 90 - 9F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // A0 - AF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // B0 - BF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // C0 - CF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};
/*********************************************************************
Function: ser_xmit
Description: 处理串行输出中断.
Parameters: none.
Returns: nothing.
Side Effects: none.
*********************************************************************/
void ser_xmit(void) {
     trans_ctr++;                      // 移动输出缓冲区指针
```

```
     if (trans_ctr<trans_size) {     // 数据是否发送完毕
     if (trans_ctr==(trans_size-1)) { // 发送校验字节
     SBUF=trans_chksum;
     } else {
       SBUF=trans_buf[trans_ctr];     // 发送当前字节
       trans_chksum+=trans_buf[trans_ctr]; // 更新校验字节
       }
     }
}
/*********************************************************************
Function: push_msg
Description: 把当前数据入队列
Parameters: none.
Returns: nothing.
Side Effects: none.
*********************************************************************/
void push_msg(void) {
     memcpy(msg_buf, recv_buf, recv_buf[1]+2);
     msg_buf_valid=1;               // 缓冲区数据有效
     recv_chksum=SYNC+CLOCK_ADDR;
}
/*********************************************************************
Function: execute_cmd
Description: 执行发送过来的命令
Parameters: none.
Returns: nothing.
Side Effects: none.
*********************************************************************/
void execute_cmd(void) {
    bit need_ack=1;
    switch (recv_buf[1])
    case CMD_RESET:    // 复位时钟
      EA=0;            // 更改时间是禁止中断
      curtime.sec=curtime.min=curtime.hour=0;
      timeholder=curtime;
      EA=1;
      break;
   case CMD_TIMESYNC:      // 设置时钟
      EA=0;
      curtime.hour=recv_buf[3];
      curtime.min=recv_buf[4];
      curtime.sec=recv_buf[5];
      timeholder=curtime;
      EA=1;
```

```
      break;
   case CMD_TIMEREQ:      // 报告时间
      trans_buf[0]=SYNC;
      trans_buf[1]=CLOCK_ADDR;
      trans_buf[2]=CMD_TIMEREQ;
      trans_buf[3]=3;
      EA=0;
      trans_buf[4]=curtime.hour;
      trans_buf[5]=curtime.min;
      trans_buf[6]=curtime.sec;
      EA=1;
      trans_ctr=0;
      trans_size=8;        // 一共8位
      need_ack=0;
      break;
   case CMD_DISPLAY:       // 显示 ASCII 字符
      recv_buf[34]=0;
      printf("\xFF%s", &recv_buf[2]); // 显示字符串
      display_time=100;               // 设置显示时间为5秒
      break;
   }
   if (need_ack) {                    // 建立应答消息
      trans_buf[0]=SYNC;
      trans_buf[1]=CLOCK_ADDR;
      trans_buf[2]=CMD_ACK;
      trans_buf[3]=1;
      trans_buf[4]=recv_buf[1];
      trans_ctr=0;
      trans_size=6;                // 一共发送6个字节
   }
   SBUF=SYNC;                     // 发送第一个字节
   trans_chksum=SYNC;             // 初始化校验字节
}
/*********************************************************************
Function: main
Description: 主程序的入口，初始化8051，设置中断，
             进入空闲模式，每次中断后查看是否要
             更新显示
Parameters: None.
Returns: Nothing.
*********************************************************************/
void main(void) {
   disp_init();     // 设置显示
   TH1=TO_VAL;      // 设置时间间隔
```

```
    TMOD=0x21;          // 定时器1为8位模式
                        // 定时器0为16位模式
    TCON=0x15;
    SCON=0xB0;          // UART模式2
    IE=0x92;            // 使能串行口和定时器中断
    for (;;) {
      if (_testbit_(msg_buf_valid))
        execute_cmd();
      }
      if (disp_update) {
        disp_time(); // 显示新时间
      }
      PCON=0x01;        // 进入空闲模式
    }
}
```

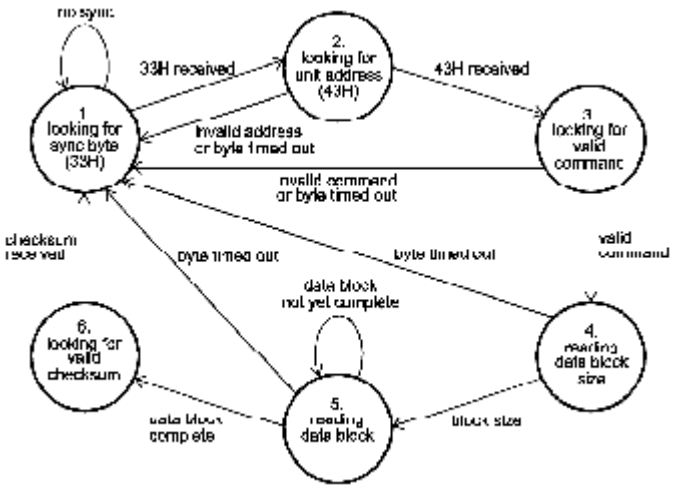

图0-7 新的FSA

主程序的改变相对比较简单。另外，处理串行传输中断的代码在新的串行中断服务程序中。功能函数 push_message 提供给ISR使用，它把目前的数据拷贝到队列中供以后使用。如果说这些改变都比较简单的话，中断程序的改变就比较多了。现在中断函数必须用汇编语言编写，当TI引起中断时，将调用C函数ser_xmit发送下一个字节。当接收数据时，因为每个字节之间相隔的时间很短，所以要用汇编来编写。

下面是用汇编编写的新的ISR的列表，注意定时器0现在用来为每个接收字节间隔时间的定时，如果定时器溢出，当前的接收过程停止，FSA回到初始状态，如图0-7。避免无限制的等待。

用汇编来编写代码保证了不会因为代码过长而丢失数据。UART在模式2下，每个字节之间间隔的时间很短（不到30个指令周期）。如果你在从SBUF中读取数据前用去了超过30个指令周期，那么你就丢失了一个字节，因此每个指令周期都很宝贵。下面是新的代码。

列表 0-6

```
EXTRN DATA (recv_chksum) ; 用来校验输入数据
EXTRN DATA (recv_buf)    ; 存放输入数据
EXTRN CODE (ser_xmit)    ; 处理发送中断
EXTRN CODE (valid_cmd)   ; 执行命令
```

```
EXTRN CODE (push_msg)     ; 把数据拷贝到队列中

SYNC       EQU    33H
CLK_ADDR   EQU    43H
CSEG       AT     23H
           ORG    23H
           LJMP   SER_INTR ; 装载中断服务程序
PUBLIC     SER_INTR
?PR?SER_INTR?SER_INTR   SEGMENT CODE
           RSEG ?PR?SER_INTR?SER_INTR

;*********************************************************************
; Function: readnext
; Description: 从串行口读入数据，如果超时置位进位标志
; Parameters: none.
; Returns: 读取的数据放入R0所指向的地址中
; Side Effects: none.
;*********************************************************************
readnext:      CLR  C              ; 清除返回标志位
               MOV  TL1, TH1       ; 使用T1作为时间间隔计时器
               SETB TR1
RN_WAIT:       JBC  RI, RN_EXIT    ; 如果 RI 置位, 接收到一个字节
               JBC  TF1, RN_TO     ; 等待时间是否溢出
RN_TO:         SETB C              ; 置位时间溢出标志位
               CLR  TR1
               RET
RN_EXIT:       MOV  A, SBUF
               CLR  TR1
               RET
;*********************************************************************
; Function: SER_INTR
; Description: 8051 的串行口中断
; Parameters: none.
; Returns: nothing.
; Side Effects: none.
;*********************************************************************
SER_INTR:  JBC RI, RECV_INT ; 是否是接收中断
CHK_XMIT:  JBC TI, XMIT_INT ; 是否是发送中断
           RETI
XMIT_INT:  LCALL ser_xmit
           RETI
                            ; 根据堆栈中的数据多少
                            ; 跳转到下面代码段相应
                            ; 的位置
```

```
CHK_XMIT3: POP  DPL
           POP  DPH
CHK_XMIT2: POP  00H
CHK_XMIT1: POP  ACC
           SETB SM2
           JMP  CHK_XMIT       ; 恢复堆栈后检测发送中断
RECV_INT:  PUSH ACC
           MOV  A, SBUF        ; 保存接收的数据
           CJNE A, #SYNC, CHK_XMIT1 ; 如果不是同步字节
                                    ; 则退出接收程序
           CLR  SM2                 ; 清除校检位
           PUSH 00H
           MOV  R0, #recv_buf       ; 把接收缓冲区的基址装入R0
           CALL readnext            ; 读下一个字节
           JC   CHK_XMIT2           ; 定时器溢出
           CJNE A, #CLK_ADDR, CHK_XMIT2 ; 一定要是时钟地址
           CALL readnext
           JC CHK_XMIT2
           PUSH DPH
           PUSH DPL
           MOV DPTR, #valid_cmd     ; 确认命令字节的有效性
           MOVC A, @A+DPTR          ; 用命令字节作为偏移量
           JZ   CHK_XMIT3           ; 如果表中的值是1
                                    ; 那么命令有效
           MOV  @R0, A                 ; 保存命令字节
           ADD  A, recv_chksum         ; 更新校验字节
           MOV  recv_chksum, A
           INC  R0                     ; 移动数据指针
           CALL readnext               ; 接收字节数量
           JC   CHK_XMIT3
           MOV  @R0, A                 ; 保存数据
           ADD  A, recv_chksum
           MOV  recv_chksum, A
           MOV  A, @R0                 ; 如果字节数为0，接收
                                       ; 校验位
           JZ   RECV_CHK
           MOV  DPL, A                 ; 保存字节数，作为计数器
RECV_DATA: INC  R0                     ; 移动指针
           CALL readnext
           JC   CHK_XMIT3
           ADD  A, recv_chksum
           MOV  recv_chksum, A
           DJNZ DPL, RECV_DATA         ; 是否还有要接收的字节
RECV_CHK:  CALL readnext
```

```
JC   CHK_XMIT3
CJNE A, recv_chksum, CHK_XMIT3 ; 接收数据有效
CALL push_msg                  ; 数据正确，放入队列
JMP CHK_XMIT3
END
```

上面的代码很小，有两个优点：第一，每个字节之间花费的指令周期很少。第二，代码简洁，好维护。查询和接收字节，判断间隔时间溢出都被放入功能_readnext中。这个功能等待下一个字节，如果间隔时间溢出的话置位无效标志位。接收到的字节被放入累加器中。检查标志位就可知道累加器中的数据是否有效，如果readnext返回无效值，就退出数据接收过程。用累加器来返回接收值，可直接对该值进行很多检测操作，如果要把值存到其它地方，只须直接从累加器中移出。

在这个主从系统中，时钟是作为从设备。对从设备来说，因为总线由主设备控制，所以通信控制比较简单，对主设备来说，控制也不复杂，从设备只有在主设备的命令下才能进行通信，这样就避免了冲突。下一章，我们将讨论在没有主设备的串行通信系统中，如何避免冲突。

4 结论

本章介绍了如何在8051的模块之间，8051和PC之间进行通信的简单串行连接。PC的接口很简单，但可以完成这个工作，如果你想要更加完整的RS-232接口，有很多专门的接口芯片，像National Semiconductor公司的芯片等，可以进行接口电平转换。下一章，我们将讨论比现在更加复杂的，利用UART进行网络通信设计。如果你的系统需要进行大量通信的话，将会对此很感兴趣。

1 复合串行端口

假设第8章的串行系统的主控器需要把从设备的数据传送到监视器中，又因为接口的复杂性，监视器是一台PC。PC将定期的送数据和命令给串行通信控制器（系统监视器），然后系统监视器返回PC所需要的数据。系统监视器和PC之间的连接就和从设备与系统监视器的连接一样，但有两处不同。第一，PC初始化所有的串行通信，系统监视器初始化所有挂接在它上面的从设备的串行通信。第二，PC使用标准的RS-232串行通信协议。这样系统监视器准备在任何时候接收从PC传送过来的波特率为9600的10位结构数据，这对它来说是个挑战，因为它还要不断的查询它的从设备。

对系统监视器来说可以采取两种方法，第一种是增加一个RS-232类型的UART接口，当有数据输入或输出时，它将产生一个中断。第二种方法是复用8051的UART接口，这个接口同时为PC和系统监视器的从设备提供通信服务。一般来说，那些认为越便宜，部件越少就越好的人都会选择第二种方法，毕竟这省去了一个外部UART接口，且不用为它提供12V的电压。

然而，第二种方法却使软件的设计变得十分的复杂。软件虽然能够解决问题，但不是最好的。当系统监视器正在和它的从设备通信时，将不能接收从PC发送过来的消息。为了改进这种情况，PC必须能够识别系统监视器不能接收消息的情况。发送方将再传送一遍数据，同时系统监视器的串行控制也将保证PC有最高的通信优先级，当系统监视器没有在和从设备通信时，它将准备接收从PC发送过来的数据。除了改变串行通信模式和波特率来适应同时和PC，从设备通信外，系统监视器的设计还要有门限制，确保同一时间内只和一个设备传输数据。系统监视器的设计见图0-1。

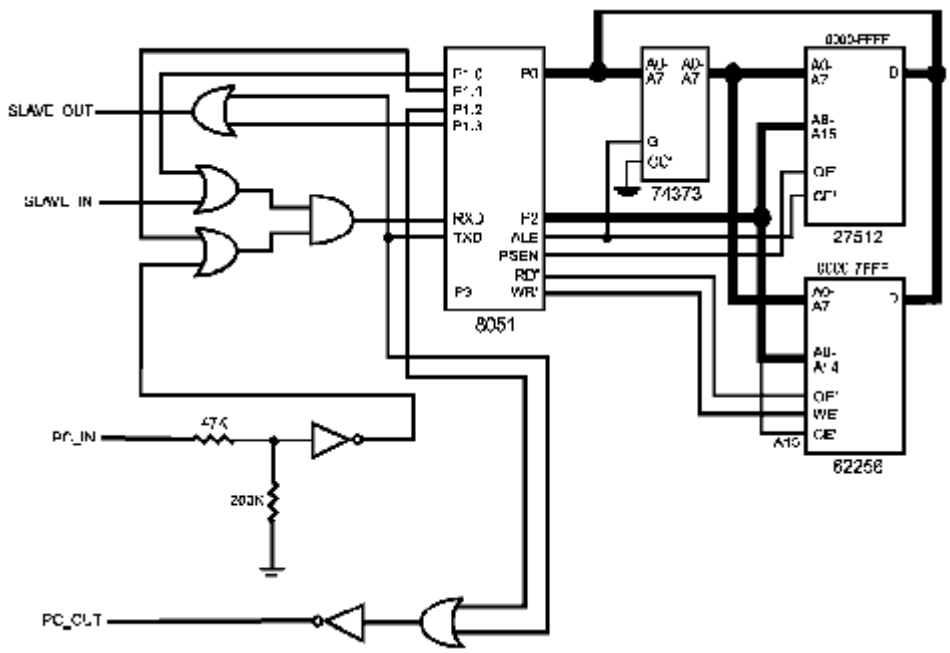

图0-1 复用串行口

系统监视器的软件的串行中断服务程序接收从PC发过来的数据时，采用的波特率为9600。这部分代码可以使用第8章的时钟程序代码。为了简单，我们设PC和系统监视器像第8章那样采用同样的数据格式。

串行中断服务程序的编写一定要高效。第8章所使用的两种串行中断服务程序中接收数据的方法都要使用。当串行口的波特率为9600时，接收过程是由中断驱动的。一小段汇编代码在进入中断后根据标志位决定调用什么处理程序。当标志位表明串行口波特率应使

用9600时，执行用C编写的程序，这时，系统很多时间处于空闲模式等待中断。当波特率应为300K时，则调用汇编编写的处理程序。

你应该记得，汇编代码不断查询输入的数据，因为每个字节间的时间间隔不足30个指令周期。当波特率为9600时，每个字节之间的时间间隔比较长，所以使用中断驱动方式，当数据传输速度很快时，就要采用汇编编写的程序了。

列表 0-1

```
/*********************************************************************
Function: ser_9600
Description: 当串行口波特率为9600时，使用该接收程序
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void ser_9600(void) {
  if (_testbit_(TI)) {      // 调用发送处理程序
    ser_xmit();
  }
  if (_testbit_(RI)) {      // 调用接收处理程序
    ser_recv();
  }
}
/*********************************************************************
Function: ser_recv
Description: 当波特率为9600时，在中断服务程序中接收数据
Parameters: 无.
Returns:  无.
Side Effects: 无.
*********************************************************************/
void ser_recv(void) {
  unsigned char c, temp;
  c=SBUF;
  switch (recv_state) {
  case FSA_INIT:                  // 接收同步字节
    if (c==SYNC) {                // 检查同步字节
      recv_state=FSA_ADDRESS;     // 下一个状态
      recv_timeout=RECV_TIMEOUT;  // 最大间隔时间
      recv_chksum=SYNC;           // 初始化校验值
  }
  break;
  case FSA_ADDRESS:               // 接收地址
    if (c==SM_ADDR) {             // 确认地址
      recv_state=FSA_COMMAND;     // 下一个状态
      recv_timeout=RECV_TIMEOUT;  // 最大时间间隔
      recv_chksum+=c;             // 维护校验值
```

```
  } else {                      // 数据不是给时钟的
    recv_state=FSA_INIT;              // 回到初始状态
    recv_timeout=0;
    }
  break;
case FSA_COMMAND:               // 接收命令字节
  if (!valid_cmd[c]) {          // 命令字节是否有效
    recv_state=FSA_INIT;              // 命令字节无效
    recv_timeout=0;
  } else {
    recv_state=FSA_DATASIZE;    // 下一个
    recv_chksum+=c;             // 更新校验字节
    recv_buf[0]=c;              // 保存命令
    recv_timeout=RECV_TIMEOUT;  // 最大时间间隔
  }
  break;
case FSA_DATASIZE:       // 接收字节的数量
  recv_chksum+=c;
  recv_buf[1]=c;         // 保存字节数
  if (c) {               // 是否有数据
    recv_ctr=2;          // 设置接收缓冲区
    recv_state=FSA_DATA; // 进入接收数据状态
  }else {
   recv_state=FSA_CHKSUM; // 进入校验状态
  }
  recv_timeout=RECV_TIMEOUT;
  break;
case FSA_DATA:            // 读取数据
  recv_chksum+=c;
  recv_buf[recv_ctr]=c;     // 保存数据save data byte
  recv_ctr++;               // 更新缓冲区更新缓冲区指针
  if ((recv_ctr-2)==recv_buf[1]) { // 数据是否接收完毕
    recv_state=FSA_CHKSUM;
  }
  recv_timeout=RECV_TIMEOUT;
  break;
case FSA_CHKSUM:
  if (recv_chksum==c)
    push_msg();
  }
default:
  recv_timeout=0;
  recv_state=FSA_INIT;
  break;
```

```
  }
}
/*********************************************************************
Function: ser_xmit
Description: 处理串行发送中断
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void ser_xmit(void) {
  trans_ctr++;              // 输出数据缓冲区指针加1
                            // 数据是否发送完毕
  if (trans_ctr < ser_queue[XMIT_QUEUE]
                  .entry[ser_queue[XMIT_QUEUE].head].size) {
                 // 最后发送校验字节
    if (trans_ctr == (ser_queue[XMIT_QUEUE]
                          .entry[ser_queue[XMIT_QUEUE].head]
                     .size-1)) {
       SBUF=trans_chksum;
    } else {
                                        // 发送当前字节
      SBUF=ser_queue[XMIT_QUEUE]
             .entry[ser_queue[XMIT_QUEUE].head].buf[trans_ctr];
                                       // 更新校验字节
      trans_chksum+=ser_queue[XMIT_QUEUE]
                    .entry[ser_queue[XMIT_QUEUE].head]
                    .buf[trans_ctr];
    }
   } else {                         // 数据发送完毕
                                    // 如果没有应答
                                    // 再次发送
      if (!ser_queue[XMIT_QUEUE]
            .entry[ser_queue[XMIT_QUEUE].head].retries) {
        if (queue_pop(XMIT_QUEUE)) {      // 输出后面的数据
          check_stat();
        }
      } else {
        xmit_timeout=XMIT_TIMEOUT;        // 设置应答时间间隔计数
      }
    }
  }
```

处理高速串行口中断代码的汇编源程序如下：

列表 0-2

```
; 文件名: SERINTR.A51
```

```
EXTRN DATA (uart_mode)          ; 写入 SCON 的值
EXTRN BIT  (baud_9600)          ; 波特率为 9600时置位
EXTRN DATA (recv_chksum)        ; 计算输入数据的校验值
EXTRN DATA (recv_buf)           ; 存放输入数据
EXTRN CODE (ser_xmit)           ; 处理串行发送中断子程序
EXTRN CODE (valid_cmd)          ; 决定命令是否有效的表格
EXTRN CODE (push_msg)           ; 把完整的消息放入消息队列中
EXTRN CODE (ser_9600)           ; 处理慢速串行通信
SYNC EQU     33H                ; 同步字节
SM_ADDR EQU  40H                ; 时钟地址
CSEG    AT      23H
        ORG     23H
        LJMP    SER_INTR
        PUBLIC  SER_INTR
?PR?SER_INTR?SER_INTR SEGMENT CODE
        RSEG    ?PR?SER_INTR?SER_INTR
;*********************************************************************
; Function: readnext
; Description: 从串行口中读入一个字节
; returns. 时间溢出置位标志位
; Parameters: 无.
; Returns: R0所指向地址内的字节
; Side Effects: 无.
;*********************************************************************
readnext: CLR  C ; clear the return flag
          MOV  TL1, TH1       ; T1为时间间隔定时器
          SETB TR1            ; 开始定时
RN_WAIT:  JBC  RI, RN_EXIT    ; 等待接收的数据
          JBC  TF1, RN_TO     ; 时间间隔溢出
RN_TO:    SETB C              ; 时间溢出，置位标志位
          CLR TR1             ; 停止定时器
          RET
RN_EXIT:  MOV A, SBUF
          CLR TR1
          RET
;*********************************************************************
; Function: SER_INTR
; Description: 8051的串行口中断服务程序
; Parameters: 无.
; Returns: 无.
; Side Effects: 无.
;*********************************************************************
SER_INTR:  JNB   baud_9600, FAST_ISR  ; 选择正确的处理过程
           LCALL ser_9600
```

```
            RETI

FAST_ISR:   JBC    RI, RECV_INT          ; 接收中断
CHK_XMIT:   JBC    TI, XMIT_INT          ; 发送中断
            RETI

XMIT_INT:   LCALL ser_xmit               ; 处理发送中断
            RETI
                                         ; 根据推入堆栈中的
                                         ; 数据多少，跳转到下面的
                                         ; 相应的位置
CHK_XMIT3: POP DPL
            POP DPH
CHK_XMIT2: POP 00H
CHK_XMIT1: POP ACC
            MOV SCON, uart_mode           ; 确保恢复校检位
            JMP CHK_XMIT

RECV_INT:   PUSH ACC                     ; 用累加器来保存数据
            MOV  A, SBUF                 ; 保存输入数据
            CJNE A, #SYNC, CHK_XMIT1     ; 如果不是同步字节
                                         ; 退出接收程序
            PUSH 00H                     ; 保存R0
            MOV R0, #recv_buf            ; 把接收缓冲区地址的基址保存在R0中
            CALL readnext                ; 读下一个数据
            CJNE A, #SM_ADDR, CHK_XMIT2  ; 一定要是时钟的地址
            CLR SM2                      ; 校检位
            CALL readnext                ; 取下一个字节
            PUSH DPH                     ; 保存 DPTR
            PUSH DPL
            MOV  DPTR, #valid_cmd        ; 准备确认命令是否有效
            MOVC A, @A+DPTR
            JZ   CHK_XMIT3               ; 如果表中相应的位为0
                                         ; 这是一个无效的命令
            MOV  @R0, A                  ; 保存命令
            ADD  A, recv_chksum          ; 更新校验字节
            MOV  recv_chksum, A
            INC  R0                      ; 移动缓冲区指针
            CALL readnext                ; 下一个字节
            MOV  @R0, A                  ; 保存要接收的字节的个数
            ADD  A, recv_chksum          ; 更新校验字节
            MOV  recv_chksum, A
            MOV  A, @R0                  ; 如果字节个数是0
                                         ; 进入校验状态
```

```
            JZ  RECV_CHK
            MOV DPL, A                      ; 设置计数器
RECV_DATA: INC R0                           ; 缓冲区指针加1
            CALL readnext
            ADD A, recv_chksum
            MOV recv_chksum, A
            DJNZ DPL, RECV_DATA             ; 数据是否接收完毕
            RECV_CHK: CALL readnext         ; 接收校验字节
            CJNE A, recv_chksum, CHK_XMIT3 ; 接收的数据是否正确
            LCALL push_msg                  ; 接收数据正确
                                            ; 把数据保存到队列
            JMP CHK_XMIT3
            END
```

系统监视器软件将使用时钟的时标过程。当计时到一定的数量后，系统监视器开始查询它的从设备。时间一到，如果总线可用作高速波特率传输时，系统监视器将初始化和第一个串行从设备之间的通信，否则，把数据存入队列等待串行总线空闲。

列表 0-3

```
/*********************************************************************
Function: system_tick
Description: 定时器0的中断服务程序，每50ms中断一次。
             在中断程序中，对那些需要定时的功能函数
             进行计数
Parameters: 无.
Returns: 无.
*********************************************************************/
void system_tick(void) interrupt 1 {
  TR0=0;              // 停止定时器
  TH0=RELOAD_HIGH;    // 重装定时器
  TL0=RELOAD_LOW;
  TR0=1;              // 开始计时
  if (poll_time) {    // 是否开始查询从器件
    poll_time--;
    if (!poll_time) {
      poll_time=POLL_RATE;
      start_poll();   // 开始查询
    }
  }
  if (xmit_timeout) { // 在发送队列中的最前面的消息
                      // 是否还没收到应答
    xmit_timeout--;
     if(!xmit_timeout) { // 重试
       ser_queue[XMIT_QUEUE]
           .entry[ser_queue[XMIT_QUEUE].head].retries--;
      if (ser_queue[XMIT_QUEUE]
```

```
      .entry[ser_queue[XMIT_QUEUE].head].retries) {
                                    // 重新发送一遍消息
      SCON=uart_mode=ser_queue[XMIT_QUEUE]
                        .entry[ser_queue[XMIT_QUEUE].head]
                    .uart_mode;
      ser_queue[XMIT_QUEUE]
       .entry[ser_queue[XMIT_QUEUE].head]
       .status=STAT_SENDING;
      SBUF=ser_queue[XMIT_QUEUE]
          .entry[ser_queue[XMIT_QUEUE].head].buf[0];
      trans_ctr=0;                  // 缓冲区指针指向第一个字节
                                    // 设置发送字节数
      trans_size=ser_queue[XMIT_QUEUE]
            .entry[ser_queue[XMIT_QUEUE].head].size;
                                    // 设置消息校验字节
      trans_chksum=ser_queue[XMIT_QUEUE]
             .entry[ser_queue[XMIT_QUEUE].head]
             .buf[0];
      } else {                      // 可重复次数为0,
                                    //这个消息已经处理完毕
        if (queue_pop(XMIT_QUEUE)) { //清除该消息
                                    // 是否还有待处理的消息
                                    // 确认该消息可以使用串行口
                                    // 如果该消息的UART模式和
                                    // 当前UART的模式一样或接收状态空闲
                                    // 则该消息可使用UART
          check_stat();             // 开始消息发送
        } else {                    // 设置UART模式
                                    // 从PC中接收数据
       TH1=TO9600_VAL;
       TR1=0;
       TL1=TH1;                     // 重装定时器，UART马上有正确
                                    // 的波特率输出
       TF1=0;
       TR1=1;
       baud_9600=1;
       P1=0x09;                     // 允许 PC I/O
      }
    }
  }
}
if (recv_timeout) {                 // 检测接收间隔时间溢出
  recv_timeout--;
  if (!recv_timeout) {              // 超过间隔时间...
```

```
      recv_state=FSA_INIT;              // 重新置位 FSA
      check_stat();
    }
  }
}
/*********************************************************************
Function: start_poll
Description: 把对系统从器件查询所需要的消息装入发送队列中
Parameters: 无.
Returns: 无.
*********************************************************************/
void start_poll(void) {
  unsigned char i, temp;
                                    // 为每个从器件建立一个消息
  for (i=0; i<NUM_SLAVES; i++) {
    temp=queue_push(XMIT_QUEUE);    // 得到队列元素指针
    if (temp!=0xFF) {               // 存储区分配成功
                                    // 建立消息
      memcpy(ser_queue[XMIT_QUEUE].entry[temp].buf,
         slave_buf[i],
         slave_buf[i][3]+4);
                                  // 设置消息大小字节
         ser_queue[XMIT_QUEUE].entry[temp].size=slave_buf[i][3]+5;
                                  // 消息重发次数为3
         ser_queue[XMIT_QUEUE].entry[temp].retries=3;
                                  // 设置串行通信模式
         ser_queue[XMIT_QUEUE].entry[temp].uart_mode=BAUD_300K;
                                  // 设置消息状态
         ser_queue[XMIT_QUEUE].entry[temp].status=STAT_WAITING;
        }
    }
    check_stat();                  // 下一个消息是否可以
                                   // 开始使用UART
}
```

在两个串行通信模式中都使用了定时器1。在波特率为9600的模式下，定时器1作为串行口的波特率发生器。接收字节之间的时间间隔由定时器0产生的时标来计算。在300K波特率模式下，定时器1用作接收字节间的时间间隔计算。定时器1的复用意味着当串行口的模式改变时，必须仔细的改变定时器1的重装值。很重要的一点是，当改变定时器的工作方式，使之作为9600的波特率发生器时，当模式一改变，马上把TH1中的数装入定时器1的低字节中。还有就是，你不会立即得到9600的波特率。如果这个小细节被忽视的话，你的第一个字节可能会丢失。

串行口在这里作为系统资源，每个要发送的数据都在发送队列中排队等待发送。队列中的头消息在下列情况下可以使用串行口：一：当串行口空闲时，如发送和接收都在空闲状态时。二：当发送状态是空闲时，而串行口正在调整接收消息的波特率并且一个消息正

在被接收。

每次，当一个接收消息被推入发送队列时（因为接收状态那时为空闲），每个接收消息被弹出队列时（因为它可能是发送队列中消息的应答，需要清除）和消息超时时，都要检查发送队列的头消息。这样确保了UART及时的被每个消息所使用。执行这项任务的功能见表0-4。

Listing 0-4

```
/*********************************************************************
Function: check_stat
Description: 检测发送缓冲区，头信号是否正在等待串行口
和是否可以开始发送。如果是的话就开始发送
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void check_stat(void) {
  if (ser_queue[XMIT_QUEUE].head!=UNUSED) {// 是否有消息
                                           // 正在等待发送...
                                           // 检测它的状态
    if (ser_queue[XMIT_QUEUE]
     .entry[ser_queue[XMIT_QUEUE].head]
     .status == STAT_WAITING) {
                                        // 正处于等待状态
                                        // 可否占用串行口
      if (recv_state==FSA_INIT ||
         (uart_mode == ser_queue[XMIT_QUEUE]
                         .entry[ser_queue[XMIT_QUEUE].head]
                         .uart_mode)) {
                                          // 开始传送消息
        SCON=uart_mode=ser_queue[XMIT_QUEUE]
                        .entry[ser_queue[XMIT_QUEUE].head]
                        .uart_mode;
        if (uart_mode==BAUD_300K) {       // 确认定时器1正常
                                          // 重装
          TH1=TO300K_VAL;
          baud_9600=0;
          P1=0x06;                        // 使能从器件I/O
        } else {
          TH1=TO9600_VAL;
          TR1=0;
          TL1=TH1;                        // 重装定时器，UART马上有正确
                                          // 的波特率输出
          TF1=0;
          TR1=1;
          baud_9600=1;
```

```
          P1=0x09;                          // 使能PC I/O
        }
        ser_queue[XMIT_QUEUE]
          .entry[ser_queue[XMIT_QUEUE].head]
          .status=STAT_SENDING;
        SBUF=ser_queue[XMIT_QUEUE]
         .entry[ser_queue[XMIT_QUEUE].head].buf[0];
        trans_ctr=0;                    // 缓冲区指针指向第一个字节
        trans_size=ser_queue[XMIT_QUEUE]
                       .entry[ser_queue[XMIT_QUEUE].head].size;
        trans_chksum=ser_queue[XMIT_QUEUE]
                      .entry[ser_queue[XMIT_QUEUE].head].buf[0];
        }
      }
    }
 }
```

当接收成功时，所有接收来的消息都被推入接收队列中。主循环检测接收队列是否还有没被接收完毕的消息。如果有,则调用相关的函数对队列的头消息进行操作。从PC传送过来的命令一般让串行监视器传送数据给从器件，然后发送应答信号给PC。这些应答信号在接收过程中被建立，然后推入传送队列中。如果消息是从PC或从器件传送来的应答或数据，将检测消息是否符合在发送队列头中的消息，如果符合，发送队列头被弹出，开始传输下一个消息。这个功能的基本结构见列表0-5。

Listing 0-5

```
/*********************************************************************
Function: push_msg
Description: 把当前消息放入串行消息队列中确保波特率为9600,
             T1处于自动重装模式
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void push_msg() {
  unsigned char temp;
  temp=queue_push(RECV_QUEUE);      // 在队列中分配存储空间
  if (temp!=0xFF) {                 // 存储空间分配成功
                                    // 拷贝数据
    memcpy(ser_queue[RECV_QUEUE].entry[temp].buf, recv_buf,
               recv_buf[1]+2);
                                    // 消息大小
    ser_queue[RECV_QUEUE].entry[temp].size=recv_buf[1]+2;
                                    // 消息状态
    ser_queue[RECV_QUEUE].entry[temp].status=STAT_IDLE;
                                    // 重发次数为0
    ser_queue[RECV_QUEUE].entry[temp].retries=0;
```

```
                                    // 记录当前串行模式
    ser_queue[RECV_QUEUE].entry[temp].uart_mode=uart_mode;
  }
  recv_chksum=SYNC+SM_ADDR;         // 设置校验字节
  recv_state=FSA_INIT;              // 初始化接收状态
  check_stat();                     // 是否可以开始发送下一个数据
 }
/*********************************************************************
Function: ser_exec
Description: 处理所有的输入消息.
Parameters: 无.
Returns: 无.
*********************************************************************/
void ser_exec() {
#ifdef USEEXEC
do {
  switch (ser_queue[RECV_QUEUE].entry[head].buf[1]) {
  ...
  }
} while (ser_queue(RECV_QUEUE));  // 是否处理完所有消息
#endif
  check_stat();                   // 是否可以开始发送
}
```

2 队列实行

因为串行监视器必须快速的输入输出消息，所以建立一组"entries"的消息队列，每个entry存放一个消息，分别用一个字节表示消息的大小，如果没有应答的话重发送消息的次数，UART的模式及相关的状态。状态变量决定消息什么时候使用UART，在发送模式下什么时候释放UART。在接收模式下，状态字节没什么用。这些消息的数组有一定的大小，并提供了指向消息的头指针和尾指针。通过分配一个字节给头指针和尾指针，避免了所有的指针运算。

列表 0-6

```
typedef struct {                // 结构定义
  unsigned char buf[MSG_SIZE]; // 消息数据
  unsigned char size,           // 消息大小
               retries,         // 重复次数
               uart_mode,       // UART模式
               status;          // 当前消息的状态
} entry_type;

typedef struct {                // 定义队列
  unsigned char head,           // 队列头
               tail;            // 队列尾
  entry_type   entry[QUEUE_SIZE]; // 队列的元素为entry结构
```

```
} queue_type;

extern queue_type ser_queue[2];// 一个发送和一个接收队列
```

正常情况下，像队列这种数据结构应该从堆中动态的分配和释放存储空间。Keil在它的软件包中提供动态分配函数，但是应该尽量避免使用它。

其实队列就是分配了大小的数组，这样推入和弹出操作就很容易实现。使用指针对对列中的元素进行访问，消息从队列头弹出，从队列尾压入。用一些简单的方法防止队列溢出。

列表 0-7

```
/*********************************************************************
Function: queue_push
Description: 在循环队列中分配一个entry结构
Parameters: queue - unsigned char. 可用的队列.
Returns: 如果分配成功，返回指针，否则返回0xFF
Side Effects: none.
*********************************************************************/
unsigned char queue_push(unsigned char queue) {
  unsigned char temp;
  if (ser_queue[queue].head==UNUSED) {  // 如果队列是空的
                                        // 分配头指针为0
    ser_queue[queue].head=ser_queue[queue].tail=0;
    return 0;
  }
  temp=ser_queue[queue].tail;           // 保存尾指针值
                                        // 尾指针值加1
  ser_queue[queue].tail=(ser_queue[queue].tail+1) % QUEUE_SIZE;
                                        // 确保尾指针和头指针不重叠
  if (ser_queue[queue].head == ser_queue[queue].tail) {
    ser_queue[queue].tail=temp;         // 没有可分配的空间
    return 0xFF;
  }
  return ser_queue[queue].tail;        // 返回分配存储空间的地址
}
/*********************************************************************
Function: queue_pop
Description: 把数据从循环队列中弹出
Parameters: queue - unsigned char. 可用队列
Returns: 如果队列已空，返回0。如果队列中还有数据，返回1
Side Effects: 无.
*********************************************************************/
bit queue_pop(unsigned char queue) {
                                        // 头指针加1
  ser_queue[queue].head=(ser_queue[queue].head+1) % QUEUE_SIZE;
                                        // 如果头指针等于尾指针
```

```
                                        // 队列已空
  if (((ser_queue[queue].head-ser_queue[queue].tail)==1) ||
     (!ser_queue[queue].head &&
     (ser_queue[queue].tail==QUEUE_SIZE-1))) {
    ser_queue[queue].head=ser_queue[queue].tail=UNUSED;
    return 0;
  }
  return 1;
}
```

响应输入消息的代码留给读者自己去设计，根据你项目的需要写入相应的代码。每个输入的消息将和发送队列的头消息（该消息正在等待应答）进行比较，如果相符合，就清除发送队列中的头消息。

但接收队列中的消息并非都是发送队列中头消息的应答，它有可能是来自PC的命令。这时可能需要发送一些数据给从器件，和发送应答消息给PC。在发送队列中建立这些消息并排队等待发送。

3 使用内置定时器作TDMA控制

很多通信系统不能简单采用像上面那种使用系统监视器的查询方式。采用查询方式的缺点在于，当网络上的器件很多时，查询方式将浪费很多时间，将有大量的数据从主设备发送到从器件。当从器件之间需要通信时，也要经过主控器。我们要通过一种新的串行网络设计来解决这些问题。

新设计的主要不同点是从器件之间可以直接进行通信，而不再是只和系统监视器进行数据传输。在网络上传输的所有消息都能被从器件监听到。这将影响从器件的设计方式。首先它们的通信能力必须得到增强，因为它们将直接彼此进行通信而不再需要主控器作为过度。此外，每个器件上串行中断的数量将大大增多。连接这些器件的网络拓扑结构如下。

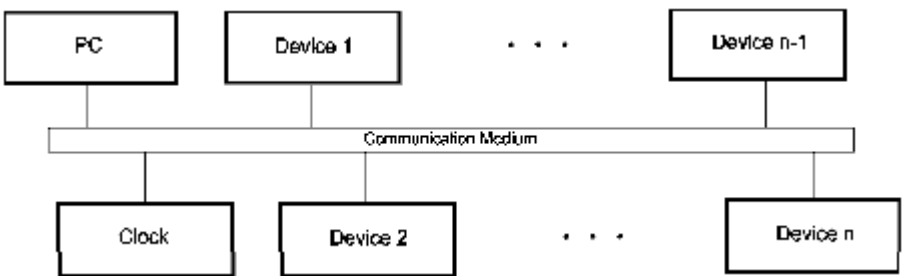

图0-2 网络拓扑结构

和前面的网络结构相比不见了系统监视器，PC也直接连接在网络中，它将从网络中收集它所需要的数据。因为PC连接在网络上，所以网络的通信波特率采用9600。网络上的器件可以和任何其它器件进行通信。

通信权轮流分配。首先是节点1，然后是节点2...节点n，再是节点1，这样无限循环。当轮到某个器件通信时，它能发送数据给其它任何器件，当通信时间结束后，必须释放总线的控制权，这是TDMA（Time Divion-Multiple Access）网络通信的基本原则。

设计中，每个节点都被分配了一个时间段号码用于通信，时间段是一个基本单位时间。基本单位时间的选择根据所要发送数据量的大小来决定。在这里，我们把时间段设置成50ms。知道了时间段号码，总的时间段数量，和时间段的大小，可以很容易用软件跟踪每个时间段，并对时间段计数，当计到自己的号码时就开始发送数据。用51系列单片机来实现这个简单的TDMA网络。现假设所有器件同时启动，这样在上电时使得系统取得同步。在实际系统中可以使用很多方法来取得系统同步。

TDMA网络节点基本硬件的设计十分简单。在这个例子中一排8脚DIP开关和P1端口相连。端口的低位决定器件的时段号，高位决定网络的器件号。这个TDMA网络的时段宽度为50ms。

网络上两个器件之间的交谈是很简单的。假设从器件1想从从器件2处取得一些数据，从器件1在它的的时段中向从器件2发送请求数据消息，从器件2接收并分析从器件2发送过来的消息，当确认消息有效时，把消息放入接收队列中处理，从器件2的代码将产生相应的应答消息，把这个应答消息放入发送队列中，当从器件2的时间段来临时把消息发送出去。与此同时从器件1计时等待从器件2反馈的数据，这样就避免了无限制的等待。

当网络上节点的时间段来临的时候，它会发送尽可能多的数据。例如，如果发送队列中有5个消息，它不会仅仅只发送一个消息，而会充分利用这50ms的时间把尽可能多的消息发送出去。如果这个时间段中只能发送3个消息，另外两个消息就等到下一个时间段再发送。

网络节点的通信程序比较简单，可从系统监视器网络的代码演变过来。下面是系统初始化的主程序。

列表 0-8

```
/*********************************************************************
Function: main
Description: 程序入口.初始化8051,使能中断源
             然后进入空闲模式
Parameters: 无.
Returns: 无.
*********************************************************************/
void main(void) {
  slotnum=P1 & 0x0F;          // 得到节点的时段号码
  slottot=P1 / 16;            // 得到总的节点数
  TH1=TO9600_VAL;             // 设置定时器1的重装值
  TH0=RELHI_50MS;             // 设置定时器0的值
  TL0=RELLO_50MS;
  TMOD=0x21;                  // 定时器0为16位工作方式
                              // 定时器为8位自动重装方式
  TCON=0x55;                  // 定时器开始运行，两个外部中断都为
                              // 边沿触发方式
  SCON=BAUD_9600;             // UART 工作为模式2
  IE=0x92;                    // 使能定时器0中断
                              // 和串行口中断
  init_queue();               // 清空所有队列
  for (;;) {
    if (tick_flag) {          // 检测系统时标
      system_tick();
    }
    if (rcv_queue.head!=UNUSED) { // 如果接收队列中有消息
      ser_exec();                 // 就对它进行处理
    }
    PCON=0x01;                    // 进入空闲模式
```

```
  }
}
```

每当定时器0中断把标志位置位后，从主程序中调用'system_tick'。时标函数不再在定时器0的中断服务程序中执行，因为ISR还要对系统时段计数，而且要非常精确，保证不溢出时段的边界。中断程序用汇编编写，每条指令的执行周期都要精确的计算。

列表 0-9

```
EXTRN   BIT    (tick_flag)      ; 表明一个中断已经产生
EXTRN   CODE   (start_xmitt)    ; 节点时段到了...
EXTRN   XDATA  (curslot)        ; 跟踪当前时段
; 基于系统工作频率11.059MHz的定时器重装值使定时器每50ms溢出一次
; 注意重装要延时9个时钟周期
REL_HI  EQU    04CH
REL_LOW EQU    007H

SEG             AT     0BH
                ORG    0BH
                LJMP   T0_INTR     ; 中断服务程序

                PUBLIC T0_INTR
?PR?T0_INTR?T0INT     SEGMENT CODE
                RSEG   ?PR?T0_INTR?T0INT

;************************************************************************
; Function: T0_INTR
; Description: 定时器0的中断服务程序。定时器每50ms溢出一次
; 重装定时器并检测节点时间段是否到来
; Parameters: 无.
; Returns: 无.
; Side Effects: 无.
;************************************************************************
T0_INTR:        CLR TR0                 ; 1, 3 重装定时器 0
                MOV TH0, #REL_HI        ; 2, 5
                MOV TL0, #REL_LOW       ; 2, 7
                CLR TF0                 ; 1, 8
                SETB TR0                ; 1, 9
                SETB tickflag           ; 置位标志位
                LCALL check_slot        ; 是不是我的时段

                PUSH ACC
                PUSH B
                PUSH DPH
                PUSH DPL

                MOV DPTR, #curslot      ; 把当前时段数读入累加器
```

```
                MOVX A, @DPTR
                INC A                   ; 时段数加1
                MOV B, A
                MOV A, P1               ; 读入总的时段数
                SWAP A
                ANL A, #00FH
                XCH A, B
                CLR C
                SUBB A, B               ; 当前时段数是不是
                                        ; >= 总的时段数
                JC L1
                CLR A                   ; 是，清除 curslot
                MOVX @DPTR, A

L1:             MOVX A, @DPTR           ; 读入当前时段数
                MOV B, A
                MOV A, P1               ; 该器件的时段号
                ANL A, #00FH
                CLR C
                SUBB A, B               ; curslot==slotnum?
                JNZ L2                  ; 不等于
                LCALL start_xmit        ; curslot==slotnum, 开始发送
L2:             POP DPL
                POP DPH
                POP B
                POP ACC
                RETI

                END
```

下面是用C编写的新的定时器溢出服务程序。

列表 0-10

```
/*********************************************************************
Function: system_tick
Description: 定时器溢出的服务程序. 在该程序中对那些需要
             时间限制的函数计数
Parameters: 无.
Returns: 无.
*********************************************************************/
void system_tick(void) {
  unsigned char i;
  tick_flag=0;                      // 清除标志位

  for (i=0; i<MAX_MSG; i++)
    if (xmit_timeout[i][0]) {       // see if the msg timed out
```

```
      xmit_timeout[i][0]--;
      if (!xmit_timeout[i][0]) {    // if so, check retries
        check_msg(xmit_timeout[i][1]);
      }
    }
  }

  if (recv_timeout) {               // 检测字节间隔时间是否
                                    // 溢出
    recv_timeout--;
    if (!recv_timeout) {            // 溢出...
      recv_state=FSA_INIT;
      check_stat();
    }
  }
}
```

系统时标函数有很多系统监视器项目中的功能。但它更像时钟项目中的时标函数都是从主循环中调用。它的主要功能是维护消息定时器，允许节点从网络通信错误中恢复过来。

定时器0的中断服务程序跟踪系统时间段，当本节点的时间段到了时，它调用发送功能函数发送正在等待发送的消息。功能函数尽可能多的为队列中等待发送的消息建立缓冲区，串行发送中断服务程序把这些消息发送出去。当受到应答消息后，消息执行代码将把消息从发送队列中清除。没有接收到应答消息的消息由从系统时标函数中调用的时间溢出代码进行处理。串行口中断服务程序除了数据发送部分的代码外和系统监视器的代码一样。数据不再从发送队列中直接发出，而是从从一个新建的缓冲区中发出。

列表 0-11

```
/*********************************************************************
Function: start_xmit
Description: 把尽可能多的消息放入发送缓冲区中，然后发送第一个消息
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void start_xmit(void) {
  unsigned char maxbytes=45,      // 一个时间段中总共发送的字节数
                msgnum=0,         // 发送缓冲区中装入的总的消息数
                i;

  if (tq_head == UNUSED) {        // 如果队列是空的，就不浪费时间
    return;
  }
  while (maxbytes) {                        // 当发送缓冲区中有空间
    if (maxbytes>=tq[temp].size) {          // 确定能装下下一个字节
                                            // 拷入缓冲区并建立校验字节
```

```
      for (i=0, chksum=0, trans_size=0;
               i<tq[temp].size;
               i++, trans_size++) {
        trans_buf[trans_size]=tq[temp].buf[i];
        chksum+=tq[temp].buf[i];
      }
      trans_buf[trans_size]=chksum;         // 保存校验字节
      xmit_timeout[msgnum][0]=MSG_TIMEOUT; // 保存时间间隔信息
      xmit_timeout[msgnum][1]=tq[temp].retries;
      msgnum++;                             // 缓冲区中消息数加1
      maxbytes-=tq[temp].size+1;            // reduce amount remaining by
                                            // amount used
      temp=tq[temp].next;
      } else {
        maxbytes=0;                      // 跳出循环
      }
    }
 }

/*********************************************************************
Function: ser_xmit
Description: 处理发送中断
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void ser_xmit(void) {
  trans_ctr++;             // 发送指针加1
                           // 数据是否发送完毕
  if (trans_ctr < trans_size) {
                           // 最后一个字节发送校验位
    if (trans_ctr==trans_size-1)) {
      SBUF=trans_chksum;
    } else {
                           // 发送当前字节
      SBUF=trans_buf[trans_ctr];
                           // 更新校验字节
      trans_chksum+=trans_buf[trans_ctr];
    }
  }
}
```

现在发送队列的结构和接收队列的结构不再一样了，因为队列头不再需要应答。如果在该节点的时间段发送出了3个消息，只有第2个和第3个消息得到了应答，队列必须保存第1个消息而清除第2个和第3个消息。新的队列结构必须更加复杂来完成这种功能。

发送队列还将使用一定数量的entry结构，在每个结构间有一定的联系而不是简单的按照顺序关系排列。entris数组有两个连接列表—一个使用，一个未被使用。当需要新的存储结构时，从自由的列表中获取一个结构，并把它连接到使用列表中。当需要删除一个结构时，把该结构从使用列表中取出，并放回到自由列表中。新发送队列的源代码见列表0-12。

列表 0-12

```
/*********************************************************************
Function: tq_init
Description: 为发送队列设置列表
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void tq_init(void) {
  tq_head=tq_tail=UNUSED;              // 将头尾指针置为空
  tq_free=0;                           // 初始化空列表
  for (i=0; i<QUEUE_SIZE; i++) {
    tq[i].next=i+1;
  }
  tq[QUEUE_SIZE-1].next=UNUSED;        // 列表循环
}

/*********************************************************************
Function: tq_push
Description: 分配发送队列中的自由结构给调用者
Parameters: 无.
Returns: 返回被分配的结构单元，如果没有可分配的单元返回0xFF
Side Effects: 无.
*********************************************************************/
unsigned char tq_push(void) {
  unsigned char temp;
  if (tq_free==UNUSED) {          // 如果没有空余的存储区...
    return UNUSED;                // 告知调用者
  }
  temp=tq_free;                   // 得到第一个空结构
  tq_free=tq[tq_free].next;       // 自由列表头指向下一个空结构
  tq[temp].next=UNUSED;           // 该结构是使用列表中的
                                  // 最后一个结构
  tq[tq_tail].next=temp;          // 当前正在使用的列表尾指向
                                  // 新结构
  tq_tail=temp;
  return temp;                    // 返回新结构
 }
```

```
/*********************************************************************
Function: tq_pop
Description: 从发送队列中弹出指定的结构.
Parameters: entry - unsigned char. 指定要弹出的结构
Returns: 0 如果队列为空, 1 如果队列中还有数据
Side Effects: 无.
*********************************************************************/
bit tq_pop(unsigned char entry) {
  unsigned char temp, trail;
  if (tq_head==UNUSED || entry>(QUEUE_SIZE-1)) {    // 队列是否为空
                                                    // 或该结构无效
    return (tq_head==UNUSED) ? 0 : 1;
  }

  if (entry==tq_head) {                  // 如果弹出的是队列头
                                         // 作特殊处理
    temp=tq_head;
    tq_head=tq[tq_head].next;            // 移动头指针
    tq[temp].next=tq_free;               // 把旧的结构放入自由队列头中
    tq_free=temp;
  } else {
    temp=trail=tq_head;                   // 设置跟踪指针
    while (temp!=entry && temp!=UNUSED) { // 查表直到找到该结构
                                          // 或表被查遍
      trail=temp;
      temp=tq[temp].next;
    }
    if (temp!=UNUSED) {                   // 找到结构...
      tq[trail].next=tq[temp].next;       // 删除该结构
      tq[temp].next=tq_free;              // 把结构放入空表中
      tq_free=temp;
      if (temp==tq_tail) {
        tq_tail=trail;
      }
    }
 }
 return (tq_head==UNUSED) ? 0 : 1;
}
```

3 保持节点器件同步

处理网络工作的代码相对来说是比较简单的。TDMA网络通信确保了每个节点都能得到同等的时间来发送数据。但是，像前面所提到的，如果每个节点不是精确的在同一时刻复位，那么是不能保持同步的。确保同步最简单的方法是发给每个节点器件一个同步信号。

这需要重新设计网络。在每个时段循环的开始，我们用PC发出一个由高到低的跳变脉

冲，这个脉冲使每个节点器件都调整到时段0。每个节点都用一个中断服务程序来处理这个信号，因此把信号接到8051的INT0引脚上。ISR将重装并启动定时器0，而不是通过主程序中的代码来完成。定时器将负责时间的复位。当PC再次发出同步脉冲时，各节点又将恢复到时段0。

列表 0-13

```
/*********************************************************************
Function: start_tdma
Description: 中断服务程序应答从网络主程序发送过来的信号，重新
开始时段计数。启动定时器0。
Parameters: 无.
Returns: 无.
*********************************************************************/
void start_tdma(void) interrupt 0 {
  TH0=RELHI_50MS;              // 设置定时器0
  TL0=RELLO_50MS;
  TF0=0;
  TR0=1;
  curslot=0xFF;       // 从时段0重新开始
}
```

这样做有两个好处。第一，这使得网络中的节点较容易保持同步，因为它们都参照同一个启动信号。第二，它使PC有能力控制网络通信。你修改一下定时器0的中断服务程序，当时段计数完成一个循环后就停止定时器，这样只有收到PC发送的同步信号才能开始重新通信。假设，PC想发一个很长的信号给网络中的某个节点，但是又不想等好几个时段来发送，这时PC就可以停止网络时段计数，等它把数据发送完毕后再重新开始网络通信。

另外一个网络协议的小改动是让PC能够给那些需要立即应答的节点发送消息。换句话说就是不必把消息放入队列中等待时段进行发送，而是直接发送。这个网络就成为TDMA系统和查询系统的混合系统，这种设计使得网络节点向PC传输的数据最大化。

4 CSMA网络

当网络中的所有器件都充分利用了自己的时段发送数据时，前面所讲的TDMA网络是十分灵活而高效的，无疑是网络通信中一个很好的解决方案。

然而，并不是所有的系统都适用TDMA方案。有些节点并不传送很多消息，这样分配给该节点的时段并没有被充份应用，而有些节点的数据很多，时段对他们来说不够用。这时，TDMA方法显然不再适用。

要解决这个问题，需要每个节点在需要的时候都能够进行通信。但是不可避免有两个节点同时需要通信的情况发生，这时将产生冲突。要解决冲突，节点需要某种方法来探测网络是否被使用，和当自己在传输数据的时候是否有冲突发生。具有这种功能的网络称为CSMA（Carrier Sense-Multiple Access）网络。CSMA网络的关键问题是有一套底层的指令可使设备不产生冲突的联入网络。

把8051接入CSMA网络，网络节点的硬件必须使内置8051的串行口能够从所有的节点传送过来的数据，包括他自己。其次处理器必须使用8052，这样就多了一个定时器，新类型的网络的通信方式比较简单，首先，TDMA网络中定时器0的时标功能函数原封不动的放入定时器2中断服务程序中，定时器0用来从接受到最后一个字节开始计时。每次RI引起中断

时，定时器0都要重装计数值，同时置位网络忙标志位。当定时器溢出时，执行定时器0中断服务程序，停止定时器，清零标志位。发送消息的程序代码将检测这个标志位，如果该标志位清零，才开始发送消息。

定时器0中断服务程序将作为CSMA网络的核心程序。每当从网络中接收到一个字节时，定时器都要重装，此外，当发生多个节点的冲突时，定时器将进行随机延时。为了执行这两个功能，用一个标志位来决定是否将随机数装入定时器0中。当定时器溢出后，它将重新开始发送消息（如果网络空闲的话），并装入定时器的正常值。定时器0的中断服务程序见列表0-14。

列表 0-14

```
/*********************************************************************
Function: network_timer
Description:定时器0中断服务程序。当字节间最大限制时间溢出或
            网络隔离过程结束时引发中断
Parameters: none.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void network_timer(void) interrupt 1 {
  TR0=0;                      // 停止定时器
  if (delay_wait) {           // 是否因为网络冲突正在等待
    delay_wait=0;             // 清除辨证外标志位
    trans_restart();          // 重新开始发送
  }
  network_busy=0;             // 网络不再繁忙
  check_status();             // 是否开始发送消息
  }
/*********************************************************************
Function: trans_restart
Description:开始发送缓冲区中的消息。假设消息正确，
            重复变量和消息大小变量已经设置好了
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void trans_restart(void) {
  SBUF=trans_buf[0];          // 输出第一个字节
  last_out=trans_buf[0];      // 保存，作为冲突检测
  trans_ctr=0;                // 缓冲区指针指向第一个字节
  trans_chksum=trans_buf[0]; // 设置校验字节
 }
```

每个写入SBUF的字节将被存储在一个临时地址中，当产生接收中断时和接收到的数据相比较。如果临时地址中的数据和SBUF中的数据不符，就认为数据发送中出现了问题。这个节点将随机等待一端时间再重新发送消息。下面是处理发送和接收中断的代码。

列表 0-15

```
/*********************************************************************
Function: ser_xmit
Description: 处理串行发送中断.
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void ser_xmit(void) {
  trans_ctr++;       // 数据输出指针加1
                     // 数据是否发送完毕
  if (trans_ctr < trans_size) {
                     // 最后发送校验字节
    if (trans_ctr==trans_size-1)) {
      SBUF=trans_chksum;
      last_out=trans_chksum;
    } else {
                      // 发送当前字节
      SBUF=trans_buf[trans_ctr];
      last_out=trans_buf[trans_ctr];
                     // 更新校验字节
      trans_chksum+=trans_buf[trans_ctr];
    }
  }
}

/*********************************************************************
Function: ser_recv
Description: 当系统波特率为9600时，处理串行接收中断.
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void ser_recv(void) {
  unsigned char c, temp;
  c=SBUF;
  if (TH0 > NET_DELAY_HI)
    TR0=0;                    // 设置延迟时间
    TH0=NET_DELAY_HI;
    TL0=NET_DELAY_LO;
    TR0=1;
  }

 if (transmitting) {          // 如果这个节点正在发送
                              // 消息...
```

```
    if (c!=last_out) {          // 当前字节应该和上次写入SBUF
                                // 的字节一样
      trans_hold();             // 不一样——网络传输发生错误
    }
  } else {
    switch (recv_state) {
    ...                         // 分析输入数据
    }
  }
}

/*********************************************************************
Function: trans_hold
Description: 设置一个随机网络隔离时间从2.0ms到11.76ms
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
void trans_hold(void) {
  unsigned int holdoff;
  trans_chksum=trans_ctr=0;         // 复位发送计数器
  holdoff=(unsigned int) rand();    // 得到随机数
  holdoff/=3;                       // 把随机数控制在需要的范围
  holdoff+=TWO_MS;                  // 增加一个常数确保延时2ms
  holdoff=(0xFFFF-holdoff)+1;       // 转换成重装值
  TR0=0;                            // 重新启动定时器
  TL0=(unsigned char) (holdoff & 0x00FF);
  TH0=(unsigned char) (holdoff / 256);
  delay_wait=1;                     // 表明节点因为网络冲突
                                    // 正处于等待状态
  TR0=1;
}
```

可以看到，处理发送中断的代码没什么变化，最大的不同就在于变量 last_out 必须设为最后写入SBUF的值。记住，这个值将作为下一个完整字节进入串行口，如果不这样的话，就会出现网络错误。

CSMA网络节点的其它代码和系统监视器中的代码很像。网络中传送的消息要么是命令或对另一个节点的请求，它需要一个应答，要么是对网络中其它节点的回复，这样系统监视器中的数据结构和命令代码稍微改动一下就可以重用。

5 结论

这章介绍了几种使用8051控制器进行网络工作的方法。这些并不是唯一的几种方法。如果你需要更多关于网络设计和分析的信息，可以查阅其它书籍。

第九章 控制编译和连接

1 把C代码转变成Keil C代码

当你把对其它处理器操作的，已存在的代码移植到8051上时，或把基于8051的代码进行转变，使之符合Keil C开发工具，这无疑是明智之举。因为8051和Keil C相结合的功能是十分强大的。把现有的C代码转化成Keil C代码是很简单的工作，因为C51编译器完全支持C语言的ANSI标准。只要你的代码中不存在非ANSI的语句，直接用C51编译器进行编译就没有问题。

当开始进行代码转换时，必须注意几个问题。声明的变量和代码结构应该适合在8051上运行。根据这个原则，你应该确保代码转换向着8051的发方向进行。

如果你的代码以前是用在其它控制器上的，就应该特别注意第三章说讲的如何优化你的代码。第一点应该注意的就是8051是8位微控制器，尽量把所有变量和数据元素的存储范围控制在8位的范围内。对那些作为标志位的变量应声明为位变量，如果经常要对这个位变量寻址，就用bdata来声明它。

还应注意的一点是指针的使用。这在第三章也提到过，但值得重声。如果在声明指针的时候把它限制在某一存储区域并通知编译器，那么代码的长度和代码执行的时间都会都会缩短不少。编译器会为使用这些指针的原代码写出更好的汇编代码。

一旦已经完成了上面所提到的优化过程，就要开始检查软件的结构，确定哪些是中断服务程序，那些被主函数调用的程序。当建立了中断服务程序之后，对它进行编译和连接，连接器将对那些有多重中断调用的函数产生警告信息。这些警告信息使你知道哪些代码是有潜在的错误的。这些部分可能是由于递归调用或中断结构在同一时间被调用多次。由于8051的结构，C51编译器不会自动产生代码通过单独的调用树去处理这些递归和多重调用。如果你使用的是像80x86这样的处理器，就能为每个功能调用建立相应的调用结构，但是8051的堆栈空间没有这么大，对于那些必须递归调用的功能函数，可以把他们定义成再入函数。这时C51编译器将使用一定的堆栈空间建立一个模拟栈，这时会占用内存和延长处理时间，因此要尽量少的使用关键字'reentrant'。

并不是所有连接器产生的警告都会导致错误。有时候，连接器警告某个功能函数被多个中断调用了，但实际上却不可能。例如，有个函数被定时器0和外部中断1的ISR调用，但这两个中断被设为同一个中断优先级，因此在同一时间只能执行一个中断服务程序，在执行中断服务程序的过程中不会被同级中断所中断，因此是十分安全的。一种除去连接警告的方法是从一个调用树中删除参考，这样就不会产生你不想要的再入栈。关于这点，我们将在后面仔细讨论。

当上面所有一切都完成之后，你要考虑对外部存储区的寻址方式了。很多C程序员，当他们需要对某个物理地址进行寻址的时候，都会声明一个指针，用指针对这个物理地址进行操作。这种方法在C51中仍然适用，但最好使用像CBYTE，CWORD，XBYTE，XWORD，DBYTE，DWORD，POBYTE，PWORD 这些由absacc.h提供的宏定义，它们使外部存储区看起来像一个（char,int,long）的数组，使程序更具有可读性。另外，如果你的硬件结构有点特殊，不能简单使用MOVX对外部存储区进行寻址，你可以重新改写宏定义来适应新的寻址方式。

如果你的代码以前用的是像Archimedes或Avocet这样的编译包，你必须把关键字转换成Keil的形式。因为其它的编译器不支持像bdata,variables,reentrant函数和特殊功能寄存器组这些特征。转化后的代码应该充分利用Keil支持的这些功能。我曾经把一个项目从Archimedes转而使用C51，结果不但节省了CODE和XDATA空间，而且速度也大大加快了，以至于不得不想办法把速度降下来。从这个例子可以看出，如果使用得好的话，C51确实可以让你获益非浅。

2 把汇编代码转换成Keil汇编代码

把汇编代码转换成Keil汇编代码中要注意的问题不是很多，主要一点是使段名和Keil段名的命名规则兼容，这样和Keil C连接起来更加简单。如果程序是用C和汇编共同编写的，请参考第三章关于C和汇编联合编程的叙述。

我很少在使用Keil进行反汇编时碰到问题。实际上，唯一碰到的问题是删除从Avocet汇编程序转化过来的程序PCON寄存器的定义。原因是Avocet汇编太老了，它是在节电模式引入8051之前产生的，需要用PCON的地址直接定义。

3 使用"using"关键字

你应该记得8051系列微处理器有4个寄存器组，每组有8个寄存器。这32个字节位于DATA存储区的最底层。每个寄存器组都有一个号码（从0到3）。PSW SFR中的RS0和RS1的默认值是0，选择寄存器组0。软件可以改变RS0和RS1的值，选择四组寄存器中的任意一组。第三章讨论了在中断服务程序中使用寄存器组的问题。比较了使用using和不使用using选项时所产生的汇编代码的不同处，当使用了using选项时，寄存器不会被压入堆栈，这里我们将讨论如何利用这一点。

第三章表明，通过为中断服务程序指定寄存器组，在中断调用时可以节省32个指令周期。为了利用这点，建议在程序中为每个中断级指定一个寄存器组。例如主循环程序和初始化代码将使用默认寄存器组0。中断优先级为0的中断服务程序将使用寄存器组0，中断优先级为1的中断服务程序将使用寄存器组2。任何被中断服务程序调用的功能要么必须使用和调用者相同的寄存器组，要么使用汇编指令NOAREGS，使之不受当前寄存器组的影响。下面的代码说明了为ISR选择寄存器组的基本设计方法。

列表 0-1

```
void main(void) {
  IP=0x11;              // 串行中断和外部中断0有
                        // 高优先级
  IE=0x97;              // 使能串行中断,外部中断1
                        // 定时器0和外部中断0
  init_system();
  ...
  for (;;) {
    PCON=0x81;          // 进入空闲模式
  }
}

void serial_intr(void) interrupt 4 using 2 {
                        // 串行口中断有高优先级
                        // 使用寄存器组2
  if (_testbit_(RI)) {
    recv_fsa();
  }
  if (_testbit_(TI)) {
    xmit_fsa();
  }
}
```

```
void recv_fsa(void) using 2 {  // recv_fsa 必须使用和串行中断
                               // 同样的寄存器组，因为串行中断
                               // 将调用它
  ...
}

void xmit_fsa(void) using 2 {  // xmit_fsa 必须使用和串行中断
                               // 同样的寄存器组，因为串行中断
                               // 将调用它
  ...
}

void intr_0(void) interrupt 0 using 2 {
                                     // 高中断优先级 – 使用
                                     // 寄存器组2
  handle_io();
  ...
}

void handle_io(void) using 2 {       // 被使用RB2的中断服务程序调用
                                     // 必须使用RB2
  ...
}

void timer_0(void) interrupt 1 using 1 {
                                     // 低优先级中断 – 使用
                                     // 寄存器组1
  ...
}

void intr_1(void) interrupt 2 using 1 {
                                     // 低优先级中断 – 使用
                                     // 寄存器组1
  ...
}
```

ISR和ISR调用的程序使用同一个寄存器组。任何被主程序调用的功能函数不需要指定寄存器组，因为C51会自动使用寄存器组0。下面是这个简单例子的调用树，分支并不交叉。

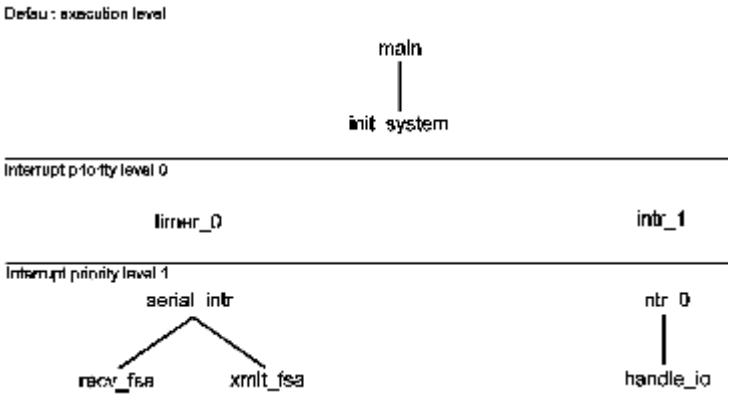

图0-1 简单调用树

很多实时时钟系统的调用树并不像上面那样简单。有些程序除了被主程序调用外，还被多个程序调用。如下面的代码。显示功能函数被主函数和两个中断级调用。

列表 0-2

```
void main(void) {
  IP=0x11;              // 串行中断和外部中断0
                        // 为高优先级
  IE=0x97;              // 使能串行中断,外部中断1
                        // 定时器0中断和外部中断0
  init_system();
  ...
  display();            // 向显示板发送一个
                        // 消息
  for (;;) {
    PCON=0x81;          // 进入空闲模式
  }
}

void serial_intr(void) interrupt 4 using 2{
                        // 串行口中断有高优先级
                        // 使用寄存器组2
  if (_testbit_(RI)) {
    recv_fsa();
  }
  if (_testbit_(TI)) {
    xmit_fsa();
  }
}

void recv_fsa(void) using 2 {  // recv_fsa 必须使用和串行中断
                               // 同样的寄存器组，因为串行中断
                               // 将调用它
  ...
  display();                   // 向显示板写入一个
                               // 状态
```

```
}

void xmit_fsa(void) using 2 {  // xmit_fsa 必须使用和串行中断
                               // 同样的寄存器组，因为串行中断
                               // 将调用它
  ...
}

void intr_0(void) interrupt 0 using 2 {
                               // 高优先级-使用
                               // 寄存器组 2
  handle_io();
  ...
}

void handle_io(void) using 2 { // 被使用RB2的中断程序调用
                               // 必须使用RB2
  ...
}

void timer_0(void) interrupt 1 using 1 {
                                        // 低中断优先级 – 使用
                                        // 寄存器组 1
  ...
  display();                            // 向显示控制器写入一个
                                        // 时间溢出消息
}

void intr_1(void) interrupt 2 using 1 {
                                        // 低优先级中断 – 使用
                                        // 寄存器组 1
  ...
}

void display(void) {
    ...
}
```

display函数被8051的各个执行级调用。这意味着display函数可被其它调用display函数的中断中断。记住每个中断函数都有它自己的寄存器组，因此不会保存当前寄存器组中的任何数据。默认时，编译器将使用寄存器组0绝对寻址对display函数进行编译。这意味着编译器将不再产生R0...R7类似的寄存器寻址方式，而是代以绝对地址。在这里，将使用定时器0的绝对地址00...07。

这时问题就产生了，当中断服务程序调用display函数的时候，那些使用寄存器组0的代码的数据会被破坏。如果display函数仅仅被一个中断服务程序调用，那还好办，只要

指定它和中断服务程序使用同样的寄存器组就可以了。但在这个例子中，它却被多个中断函数调用。

多个中断级调用display函数的时候，还会使连接器产生警告信息，这些我们留到以后再处理。目前重要的是，如何让编译器处理寄存器组的冲突。方法就是让编译器使用当前正在使用的寄存器组，而不是寄存器组0。通过对display函数使用编译控制指令NOAREGS来实现。这时编译器产生的代码将使用R0...R7来对寄存器进行寻址，而不是绝对地址。display功能函数本身不变，在他前面加上一条 NOAREGS 编译指令，使它对寄存器组的变化不敏感，在它后面的编译指令 AREGS 允许文件中的其它函数按照C51的默认值进行编译。

```
#pragma NOAREGS
void display(void) {
...
}
#pragma AREGS
```

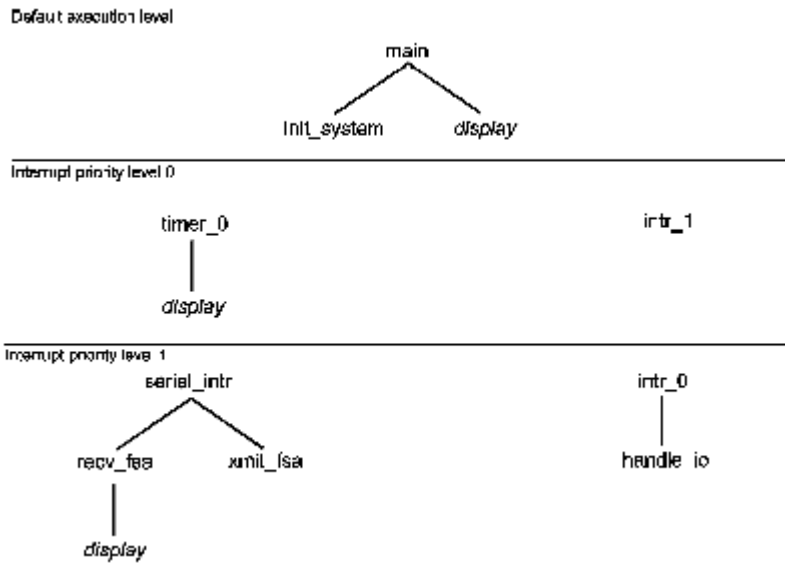

图0-2 多层中断级调用的调用树

现在还有另外一个功能。假设display使用几个局部变量来完成它的工作。C51将在压缩栈中为这些变量分配空间。根据编译器优化的结果，这些空间可能是存储器段，或一个寄存器。然而不管处于调用树的什么位置，每次调用都是使用同一存储空间。这是因为8051没有像80x86或680x0堆栈那样的功能堆栈。一般情况下，这不是什么问题，但当递归调用或使用再入函数时，将不可避免的出现局部变量冲突。

假设定时器0中断执行时调用display函数。在函数的执行过程中发生了一个串行中断，中断调用了recv_fsa 函数，而该函数又需要display函数。display函数执行完之后，局部变量的值也改变了。因为寄存器组的切换，那些使用寄存器的变量不会被破坏，而那些没有使用寄存器的变量就被覆盖了。当串行中断服务程序执行完毕之后，控制权交回定时器中断服务程序，这时正处于display程序的调用过程中。所有在默认存储段中的局部变量都已经改变了。

为了解决这个问题，Keil C51允许使用者把display函数定义成再入函数，编译器将为它产生一个模拟栈。每次调用这个函数都会在模拟栈中为它的局部变量分配存储空间。我们按下面的形式定义display函数。

```
#pragma NOAREGS
void display(void) reentrant {
```

```
...
}
#pragma AREGS
```

如果定义了再入函数，程序的存储空间和执行时间都会增加，因此要谨慎使用。除此之外，还要为再入函数划分足够多的模拟栈空间。模拟栈空间的大小通过估计同一时间内调用再入函数的次数来决定。C51可让你来决定所需模拟栈的大小。栈的设计是从顶部（如XDATA的0FFFFH）开始向你的变量发展（被分配在程序存储区的底部）。当你编译和连接完你的程序后，应该仔细观察'.M51'文件，确保有足够的再入栈空间。

4 控制连接覆盖过程

可能出现这种情况，因为C51没有真正的堆栈，不能实现从多个调用树中调用功能函数。看下面的例子。

列表 0-3

```
void main(void) {
  IP=0x00;              // 所有中断有相同的优先级
  init_system();
  ...
  display(0);
  IE=0x8A;              // 使能定时器0中断和外部中断0
  for (;;) {
    PCON=0x81;          // 进入空闲模式
  }
}

void timer_0(void) interrupt 1 using 1 {
                        // 低优先级中断 – 使用
                        // 寄存器组1
  ...
  display(1);
}

void intr_1(void) interrupt 2 using 1 {
                         // 低优先级中断 – 使用
                         // 寄存器组1
  ...
  display(2);
}

void display(unsigned char x) {
  ...
}
```

因为函数display除了被主函数调用外，还被定时器0中断和外部中断1调用，产生了冲突，连接器将给出警告。

```
*** WARNING 15: MULTIPLE CALL TO SEGMENT
```

SEGMENT: ?PR?_DISPLAY?INTEXAM
CALLER1: ?PR?TIMER_0?INTEXAM
CALLER2: ?PR?INTR_1?INTEXAM

*** WARNING 15: MULTIPLE CALL TO SEGMENT
SEGMENT: ?PR?_DISPLAY?INTEXAM
CALLER1: ?PR?INTR_1?INTEXAM
CALLER2: ?C_C51STARTUP

连接器警告你display函数可能被中断服务程序中断，而中断服务程序也调用display函数，这就导致了局部变量冲突。第一个警告说定时器0中断和外部中断1都调用了display函数。第二个警告说主函数和外部中断1之间也存在冲突。仔细检查代码结构后，可把display函数定义成再入函数。

定时器0和外部中断1具有同样的中断优先级，这两者之间不会导致冲突，可以不用考虑第一个警告。主程序中调用display函数时可被中断，这也不要紧，因为当主程序调用display函数时，中断还没被使能。这两个警告都被证明是安全的。这并不说明连接器出错了，它已经作了自己的工作，那就是当没有把多重调用的函数声明为再入函数时给出警告信息。连接器不会为你作代码分析，哪个中断会发生，在什么时候？这是工程师的工作。

当确认不必担心警告后，该怎样做呢？最简单的方法就是忽略不管，但这会影响连接器连接模块和为可重定位目标分配地址。虽然连接器还是会输出一个可执行文件，但却没有充分利用存储空间，因为连接器不能正确的进行覆盖分析了，所以不应忽略警告。

有两种方法可以除去警告，一种是告诉连接器不进行覆盖分析，这会使连接出来的代码使用很多不必要的DATA空间，但很容易实现。第二种是帮助连接器进行覆盖分析，迫使他忽略由调用树产生的参考信息。一旦你告诉它只保留一棵树的参考信息，就不会在产生警告，覆盖分析也能正常进行了。显然，第二种方法是比较好的。但如果你时间不多，且存储空间比较大的时候也可选择第一种方法。

我们用L51这个连接命令进行代码的连接。

L51 example.obj

要让L51不进行覆盖分析，只要在连接选项对话框中取消"enable variable overlaying"就可以了。

第二种方法有点麻烦，但是值得。你需要去掉三个功能调用中两个产生的调用参考信息，这需要使用命令行中的覆盖选项。display函数被'main','timer_0','intr_1'三者调用，你必须去掉其中两个产生的参考信息，一般来说留下调用次数最多的那一项。在这里，外部中断1很少发生，定时器0是系统时标，经常产生中断，因此留下定时器0调用树中的参考项。新的L51命令如下，命令行中的覆盖部分应该被输入连接设置对话框中的"Additonal"框中。

L51 example.obj overlay(main ~ _display, intr_1 ~ _display)

很多代码在第一次连接的时候都会产生多重调用警告信息。采用上面提到的方法或声明再入函数可以消除这些警告信息。你不可能一步消除所有的警告信息，多试几次，确保把它们都消除。

5 使用64K（或更多）RAM

如果你用8051开发复杂的系统，有可能不得不使用64K字节的RAM，而且还要进行I/O寻址操作，这时I/O器件地址和RAM地址将重叠。可使用端口1的引脚或通过锁存器的引脚使能或禁能RAM。下面是一个例子。

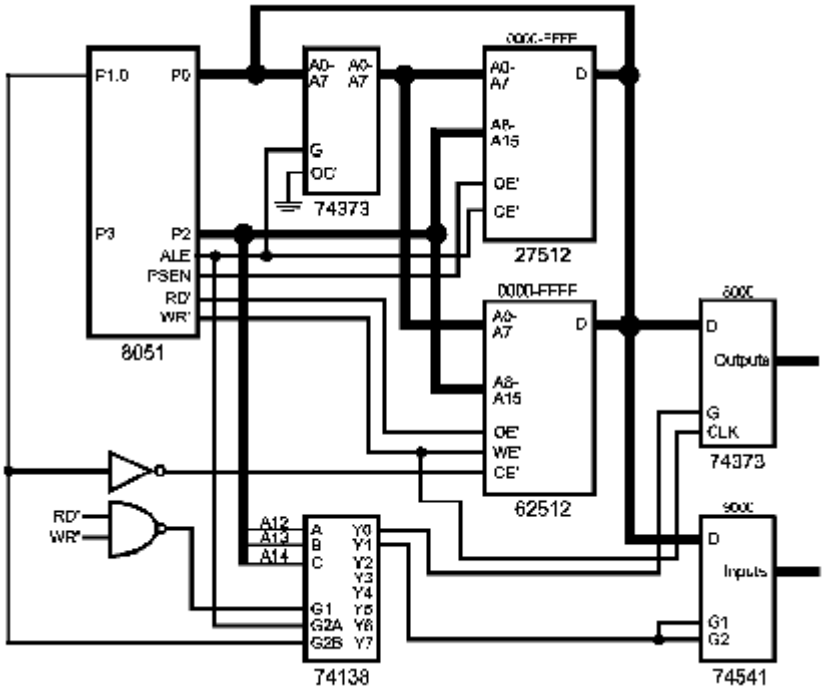

图0-3 RAM和I/O重叠

要对RAM操作时，先把P0.1置高，然后再进行寻址。当8051复位时，P0.1应该为高，使能RAM。如果软件要通过总线对I/O器件寻址，只需要把P1.0拉低禁能RAM并使能地址解码器，就可以对器件寻址了。

软件正常执行时，RAM被使能。如果要对外部I/O器件进行操作时，调用一个特殊功能禁能外部RAM，通过内部RAM输入输出数据，当操作完成后，再使能RAM，软件继续正常运行。进行I/O操作的功能函数见列表0-4。这个功能通过内部RAM传递参数，不需要使能外部RAM。

列表 0-4

```
#include <reg51.h>
#include <absacc.h>

sbit SRAM_ON = P1^0;

/*************************************************************************
功能: output
描述: 向指定XDATA地址写入数据
参数: 地址 - unsigned int. 要写入数据的地址
            数据 - unsigned char. 保存需要输出的数据
返回: 无
负面影响: 外部RAM被禁能，这时不能发生中断。因此中断系统暂时被挂起
*************************************************************************/
void output(unsigned int address, unsigned char value) {
  EA=0;                    // 禁止所有中断
  SRAM_ON=0;               // 禁能外部RAM
  XBYTE[address]=value;    // 输出数据
  SRAM_ON=1;               // 使能RAM
```

```
  EA=1;                    // 使能中断
}

/*********************************************************************
功能: input
描述: 从XDATA地址中读入数据
参数: 地址 - unsigned int.读取数据的地址
返回: 读取的数据.
负面影响: 外部RAM被禁能，这时不能发生中断。因此中断系统暂时被挂起
*********************************************************************/
unsigned char input(unsigned int address) {
  unsigned char data value;
  EA=0;                    // 禁止所有中断
  SRAM_ON=0;               // 禁能RAM
  value=XBYTE[address]; // 读入数据
  SRAM_ON=1;               // 使能RAM
  EA=1;                    // 允许中断
  return value;
}
```

禁能和使能RAM的这种概念可以被扩展，使你的RAM超过64K。在大多数情况下64K RAM对8051系统来说已经足够了。但是如果碰到大量的操作或存储大量数据的情况时，所需要的RAM可能就不止64K了。可把RAM的特殊地址线接到P1口或74HC373上，用软件来选择所需的RAM页面。这个例子中，第0页大多时候被使能，对其它页面的操作像上面的系统中对I/O器件的操作一样。RAM页面0用来存储程序变量，传递参数，作为压缩栈等。其它的RAM页面用作存放系统事件表，查询表，和一些不经常使用的数据。系统连接见图0-4，和前面不同的是P1.1和P1.2用来作为256K RAM的高两位地址线。

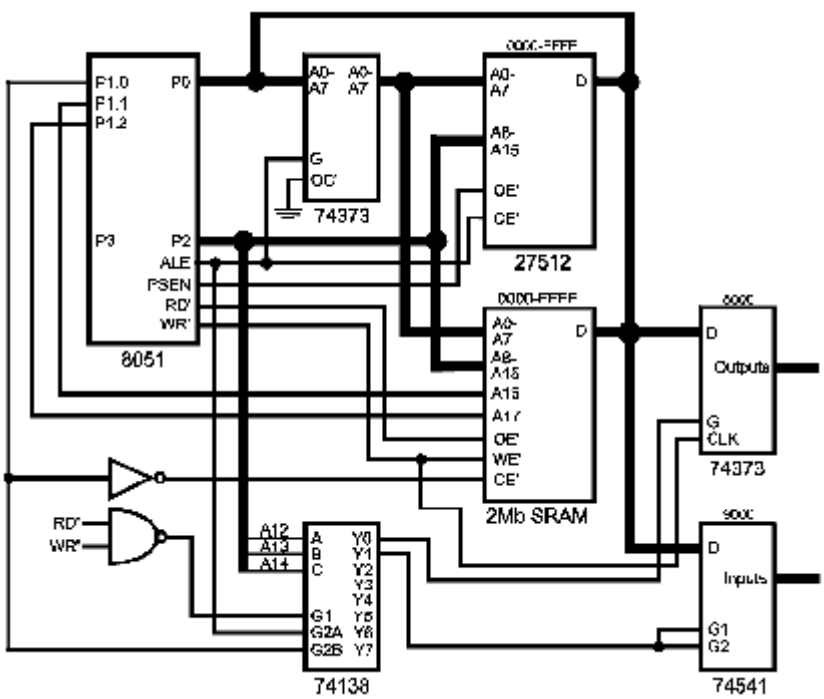

图0-4 页寻址RAM

连接在总线上的I/O器件的寻址方法和前面所讲的一样。再重申一遍，系统默认的状态是P1.0为高，以使能RAM。P1.1和P1.2为低选择RAM页面0。如果在地址选择线上面加了反向器，那么上电复位时将自动选择页面0。否则的话，如果你XDATA中有变量要在编译的时候初始化，则需要在startup.a51中加入代码清零P1.1和P1.2。

向其它RAM页面写入数据或从中读取数据是以块的方式。使用一个内部RAM缓冲区，这就避免了页面间的频繁的切换，缩短的操作的时间，但是要占用一定的内部RAM空间，并且每次传输的最大数据量有一个限制。RAM页面寻址操作的代码见列表0-5。

列表 0-5

```
#include <reg51.h>
#include <absacc.h>

sbit SRAM_ON = P1^0;

unsigned char data xfer_buf[32];

/*********************************************************************
功能: page_out
描述: 向指定XDATA地址写入数据
参数: 地址 - unsigned int. 数据写入地址
      页面 - unsigned char. 使用的RAM页面
      数量 - unsigned char. 写入的字节数
返回: 无
负面影响:外部RAM禁能，因此允许发生中断，
         中断系统被暂时挂起。
*********************************************************************/
void page_out(unsigned int address, unsigned char page,
                     unsigned char num) {
  unsigned char data i;
  unsigned int data mem_ptr; // 通过移动指针来进行数据拷贝
  mem_ptr=address;
  num&=0x1F;                 // 最大字节数为32
  page&=0x03;                // 页选面为0..3
  page<<=1;
  page|=0x01;                // 外部RAM使能
  EA=0;                      // 关闭所有中断
  P1=page;                       // 选择新页面
  for (i=0; i<num; i++) {
    XBYTE[mem_ptr]=xfer_buf[i];  // 向指定地址写入数据
    mem_ptr++;
  }
  P1=1;                          // 选择页面0
  EA=1;                            // 使能中断
}
```

```
/*********************************************************************
功能: page_in
描述: 从指定RAM页面的地址处读取数据
参数: 地址 - unsigned int. 读取数据的地址
      页面 - unsigned char. 使用的页面
      数量 - unsigned char. 读取数据的字节数
返回: 无
负面影响: 外部RAM禁能，因此允许发生中断，
          中断系统被暂时挂起。
*********************************************************************/
void page_input(unsigned int address, unsigned char page,
                    unsigned char num) {
  unsigned char data i;
  unsigned int data mem_ptr; // 通过移动指针来进行数据拷贝
  mem_ptr=address;
  num&=0x1F;                  // 限制最大字节数为32
  page&=0x03;                 // 页面选择从0..3
  page<<=1;
  page|=0x01;                 // 使能外部RAM
  EA=0;                       // 关闭所有中断
  P1=page;                    // 页面选择
  for (i=0; i<num; i++) {
    xfer_buf[i]=XBYTE[mem_ptr]; // 读取下一个地址数据
    mem_ptr++;
  }
  P1=1;                        // 选择页面0
  EA=1;                        // 使能中断
}
```

这里我采用了局部变量'mem_ptr'。for循环每次使地址加1，而汇编后产生的代码将把这个地址存储在XDATA区中，为了避免对XDATA区进行寻址，就在DATA区声明局部变量来保存地址。

上面的页面功能在很多情况下都是可行的。一些程序员可能希望有像C51库函数那样提供一套存储区操作功能，如'memcpy'。如果通用指针能对RAM页面进行寻址的话，可编写出一套和函数库功能相似的函数。Keil C的通用指针包含3个字节，两个字节存放地址，一个选择字节确定指针选择的存储空间。根据存储空间的不同，选择字节的范围从1到5。这样，字节的前几位没有被用到。现在，当寻址空间是XDATA区时，将用它们来表示指针所指的RAM页面。这个简单的改动将使新的库函数和以前的很像。我们在这里给出'mempcy'的代码，其余的留给你们去编写。'page_mempcy'的声明如下。

```
void *page_memcpy(void *dest,void *source,int num);
```

由于要在页面间进行快速的切换，所以用汇编来写，并使用大存储模式。

列表 0-6

```
?PR?PAGE_MEMCPY?PAGE_IO   SEGMENT CODE
?XD?PAGE_MEMCPY?PAGE_IO   SEGMENT XDATA OVERLAYABLE
```

```
PUBLIC      _page_memcpy,     ?_page_memcpy?BYTE

RSEG        ?XD?PAGE_MEMCPY?PAGE_IO
?_page_memcpy?BYTE:
dest:           DS        3
src:            DS        3
num:            DS        2

;------------------------------------------------------------------------
; 功能: _page_memcpy
; 描述: 从源指针所指的地址拷贝一定数量的字节到目的指针所指的地址处
;       允许xdata区指针通过使用指针区域选择字节的高位指定RAM页面
; 参数: 目的– 可选择RAM页面的通用指针，通过R1..R3传递。指明拷贝数据
;            的目的地址
;       源- 可选择RAM页面的通用指针，指明被拷贝数据的开始地址
;       数量 - unsigned integer. 指明拷贝的字节数
; 返回: 目的地址
; 负面影响: 无
;------------------------------------------------------------------------
RSEG            ?PR?PAGE_MEMCPY?PAGE_IO
_page_memcpy: PUSH 07       ; 保存寄存器数据
              PUSH 06
              PUSH 02
              PUSH 01
              PUSH 00
              PUSH ACC
              PUSH B
              MOV  DPTR, #?_page_memcpy?BYTE+6
              MOVX A, @DPTR       ; 取拷贝字节数
              MOV  06, A
              INC  DPTR
              MOVX A, @DPTR
              MOV  07, A
              ORL  A, 06

              JZ   GTFO           ; if (!num) { return }

              MOV  DPTR, #?_page_memcpy?BYTE
              MOV  A, 03          ; 装入目的指针
              MOVX @DPTR, A
              INC  DPTR
              MOV  A, 02
              MOVX @DPTR, A
```

```
                INC  DPTR
                MOV  A, 01
                MOVX @DPTR, A

L1:             LCALL GETSRC   ; 取下一个源字节
                LCALL PUTDEST  ; 写入下一个字节

                MOV   A, 07    ; num--
                CLR   C
                SUBB  A, #1
                MOV   07, A
                MOV   A, 06
                SUBB  A, #0
                MOV   06, A
                ORL   A, 07

                JZ    GTFO      ; if (!num) { return }
                JMP L1
GTFO:           POP B           ; 恢复所有寄存器
                POP ACC
                POP 00
                POP 01
                POP 02
                POP 06
                POP 07
                RET

;*********************************************************************
; 功能: GETSRC
; 描述: 从源指针所指的地址读入数据,指针加1
;       返回所读数据
; 参数: 无
; 返回: 把数据读入A
; 负面影响: 无
;*********************************************************************
GETSRC:     MOV DPTR, #?_page_memcpy?BYTE+3
            MOVX A, @DPTR           ; 得到源地址页面选择字节
            MOV B, A                ; 保存
            DEC A                   ; scale selector to 0..4
            ANL A, #00FH            ; 除去RAM页面
            MOV DPTR, #SEL_TABLE1
            RL A
            JMP @A+DPTR             ; 存储器类型选择
```

```
SEL_TABLE1: AJMP SEL_IDATA1          ; idata
            AJMP SEL_XDATA1          ; xdata
            AJMP SEL_PDATA1          ; pdata
            AJMP SEL_DATA1           ; data or bdata
            AJMP SEL_CODE1           ; code

SEL_PDATA1: MOV 00, #00              ; 对于 pdata, 地址头字节
                                     ; 必定为 00
            MOV DPTR, #?_page_memcpy?BYTE+5
            JMP L2

SEL_XDATA1: MOV DPTR, #?_page_memcpy?BYTE+4
            MOVX A, @DPTR            ; 读入地址
            MOV 00, A
            INC DPTR
L2:         MOVX A, @DPTR
            MOV DPH, 00                  ; set DPTR to XDATA address
            MOV DPL, A
            MOV A, B                     ; 得到RAM页面地址
            ANL A, #0F0H
            SWAP A
            RL A
            ORL A, #01H
            MOV P1, A                    ; 选择RAM页面
            MOVX A, @DPTR                ; 读入字节
            MOV 01, A                    ; 保存
            MOV P1, #01H                 ; 恢复 RAM page
            INC DPTR
            MOV 00, DPL
            MOV A, DPH
                                         ; 保存新的地址
            MOV DPTR, #?_page_memcpy?BYTE+4
            MOVX @DPTR, A
            INC DPTR
            MOV A, 00
            MOVX @DPTR, A
            MOV A, 01                    ; 把返回字节存入A中
            RET

SEL_CODE1:  MOV DPTR, #?_page_memcpy?BYTE+4
            MOVX A, @DPTR                ; 取得当前源地址
            MOV 00, A
            INC DPTR
            MOVX A, @DPTR
```

```
            MOV DPH, 00                    ; 用当前地址设置DPTR
            MOV DPL, A
            CLR A
            MOVC A, @A+DPTR                ; 读入字节
            MOV 01, A
            INC DPTR                       ; 指针加1
            MOV 00, DPL
            MOV A, DPH
            MOV DPTR, #?_page_memcpy?BYTE+4
            MOVX @DPTR, A                  ; 保存指针
            INC DPTR
            MOV A, 00
            MOVX @DPTR, A
            MOV A, 01                      ; 返回字节
            RET

SEL_IDATA1:
SEL_DATA1:  MOV DPTR, #?_page_memcpy?BYTE+5
            MOVX A, @DPTR                  ; 取一个字节地址
            MOV 00, A
            MOV A, @R0
            INC R0                         ; 指针加1
            XCH A, 00
            MOVX @DPTR, A                  ; 保存指针
            XCH A, 00                      ; 返回字节
            RET

;*********************************************************************
; 功能: PUTDEST
; 描述: 将A中的字节写入目的指针所指向的地址
;       然后指针加1
; 参数: 无.
; 返回: 无.
; 负面影响:无.
;*********************************************************************
PUTDEST:    MOV 02, A                  ; 保存输出数据
            MOV DPTR, #?_page_memcpy?BYTE
            MOVX A, @DPTR              ; 取得目的指针类型字节
            MOV B, A                   ; 保存类型字节
            DEC A
            ANL A, #00FH
            MOV DPTR, #SEL_TABLE2
            RL A
            JMP @A+DPTR                ; 类型选择
```

```
SEL_TABLE2:  AJMP SEL_IDATA2      ; idata
             AJMP SEL_XDATA2       ; xdata
             AJMP SEL_PDATA2       ; pdata
             AJMP SEL_DATA2        ; data or bdata
             AJMP SEL_CODE2        ; code

SEL_PDATA2:  MOV 00, #00           ; 对pdata区地址高字节
                                   ; 必定为0
             MOV DPTR, #?_page_memcpy?BYTE+2
             JMP L4

SEL_XDATA2:  MOV DPTR, #?_page_memcpy?BYTE+1
             MOVX A, @DPTR         ; 读入地址
             MOV 00, A
             INC DPTR

L4:          MOVX A, @DPTR
             MOV DPH, 00           ; 设置DPTR为外部地址
             MOV DPL, A
             MOV A, B              ; 取得RAM页面
             ANL A, #0F0H
             SWAP A
             RL A
             ORL A, #01H
             MOV P1, A             ; 选择RAM页面
             MOV A, 02
             MOVX @DPTR, A         ; 输出数据
             MOV 01, A             ; 保存
             MOV P1, #01H          ; 恢复RAM页面
             INC DPTR              ; 指针加1
             MOV 00, DPL
             MOV A, DPH
                                   ; 保存新地址
             MOV DPTR, #?_page_memcpy?BYTE+1
             MOVX @DPTR, A
             INC DPTR
             MOV A, 00
             MOVX @DPTR, A
             RET

SEL_CODE2:   RET                   ; 不能写入CODE区

SEL_IDATA2:
SEL_DATA2:   MOV DPTR, #?_page_memcpy?BYTE+2
```

```
MOVX A, @DPTR          ; 取一个字节地址
MOV 00, A
MOV @R0, 02
INC R0                   ;指针加1
XCH A, 00
MOVX @DPTR, A            ;保存指针
RET
END
```

改变以上功能的存储模式,只需要修改地址寻址的代码就可以了。以上面为例，其它的存储区操作功能函数也可很容易编写出来。

6 使用64K以上的代码空间

在十分复杂的8051控制系统中,软件的规模随着功能的加强而不断的扩大，可执行代码的长度也不断的增加。当代码的长度超过64K时，问题就变得复杂了。你将面临几个选择。

第一个选择就是把增强的功能去掉，这可能会影响产品的市场。第二个选择就是通过优化程序代码来腾出空间存放新增代码，但这是比较困难的，特别对那些已经优化过了的代码。这个方法虽然可行但不是长久之计。第三个选择就是重新设计系统，起用新的，可支持大代码空间的控制器。这意味着重新更新你的软硬件和使用新的开发工具----非常糟糕的选择。最后一个选择就是稍微改变一下硬件设计并增加系统中EPROM的数量，使之超过64K。你可能会产生疑虑，因为8051最大的寻址范围才64K呀？不用担心，Keil提供的功能包可使8051寻址的代码空间达到1MB。

使用Keil的BL51，可使用类似于前面RAM页面寻址的方式来增加代码空间。EPROM被分页，每页的大小和在页间进行跳转的方式取决于你的应用。还有就是要有一个共用空间，这个空间是处理器在任何时候都能够寻址的。这个区域存储包括中断向量，中断功能函数（可能调用其它EPROM页面的函数），C51库函数，在页面间跳转的代码和被多个页面代码使用的常量。有两种方法提供共用空间，一是使用单独的EPROM，二是在每页的底部都复制公共代码。我是比较倾向于第二种方法，因为这可以最大限度的使用页面空间。

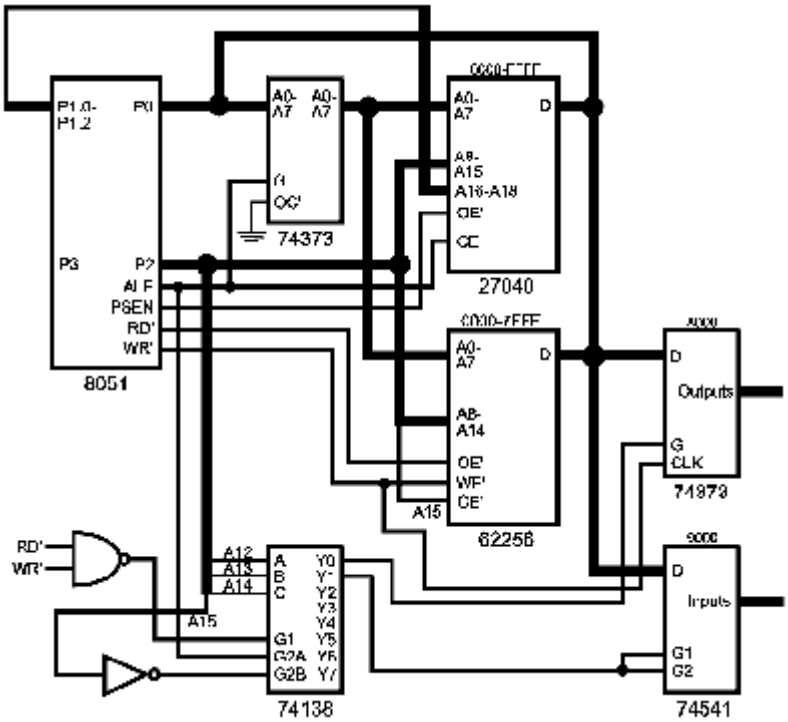

图0-5 页寻址EPROM

假设你有一个20K的公共区域，如果使用第二种方法，则每页仍有44K的可用代码区。如果使用第一种方法，则需要一个32K的EPROM，这样你每页的可用的代码区最大只有32K。

把EPROM的地址线或使能端连接到P1口或数据锁存器的引脚上，方法和RAM页面寻址一样，如图0-5所示。这里没有使用8个64K字节的EPROM，而是512K字节的EPROM。我们所作的一切就是换个大点的EPROM并接上几根地址线就行了。

使用新增的程序代码存储器是很简单的。像以往一样编写软件，你只需要把相关的功能放在一个文件中，这样当用连接器连接的时候就可以把他们放在同一个CODE页面中，这样在程序执行过程中可以尽量减少页面中的切换。页面切换的操作越少，处理其它任务的时间就越多。除此之外，还应该尽量减少常量或变量的使用，他们被单独保存在一个文件中，这样可以减少公共区的大小，使得每页有更多的代码空间。

如何对你的功能函数进行分类呢？最好的分析方法就是建立每个中断的调用树。这样你就可以知道那些功能应该放在一起，以减少不需要的页面跳转。然后建立另外一个列表显示功能函数使用那些变量和常量。被一组功能使用的常量被放在同一个文件中，这个文件和这组功能函数又在同一个代码页面中。这时你将不得不在公共区大小和页面跳转次数之间进行权衡。当你的代码空间足够时，可以进行功能组合优化。否则，应该减少常量和变量的分布。

结束功能和模块的分组之后，你需要修改L51_BANK.A51文件。告诉它代码页的数量，在页面间跳转的方式等。这些修改见列表0-7。

列表 0-7

```
$NOCOND DEBUGPUBLICS
;------------------------------------------------------------------------------
; This file is part of the BL51 Banked Linker/Locater package
; Copyright (c) KEIL ELEKTRONIK and Keil Software GmbH 1989-1994
; Version 1.2
;------------------------------------------------------------------------------
;********************** Configuration Section ***************************
```

change one - ensure that the proper number of code banks is set

```
?B_NBANKS     EQU   8        ; Define max. Number of Banks                *
;                                                                         *
?B_MODE       EQU   0        ; 0 for Bank-Switching via 8051 Port         *
;                            ; 1 for Bank-Switching via XDATA Port        *
;                                                                         *
IF ?B_MODE = 0;                                                           *
;-------------------------------------------------------------------------*
; if ?BANK?MODE is 0 define the following values                          *
; For Bank-Switching via 8051 Port define Port Address / Bits            *
?B_PORT       EQU   P1       ; default is P1                              *
```

change two - set the bank switching LSB

```
?B_FIRSTBIT   EQU    0       ; default is Bit 3                           *
;-------------------------------------------------------------------------*
ENDIF;
```

将修改后的L51_BANK.A51文件编译后产生的目标文件和你的源代码目标文件连接。所使用的命令不再是L51，而是BL51。BL51是Keil提供的增强连接器，可进行多代码页面和实时操作系统的处理。

BL51支持L51使用的命令，还有一些命令可以指定如何在代码页面中安排模块和段，这点在功能集成中十分重要。

运行BL51时，需要提供一些参数，使BL51能够对你的各个段正确定位。你需要使用BL51的第一条指令是'BANKAREA','BANKAREA'告诉BL51代码页面的物理地址，在本例中为0000H-FFFFH。'COMMON'指令告诉连接器那些功能和模块放入公共区域，并被装入到每个页面中。'COMMON'指令在BL51手册中定义。你还能使用'BANKx'指令指定装入各个代码页面中的功能，段和模块。下面是本例中的BL51连接指令。

```
BL51   COMMON{C_ROOT.OBJ}, &
       BANK0{BANK0.OBJ}, &
       BANK1{BANK1.OBJ}, &
       BANK2{BANK2.OBJ}, &
       BANK3{BANK3A.OBJ,BANK3B.OBJ}, &
       BANK4{BANK4.OBJ}, &
       BANK5{?PR?MYFUNC?MISC, BANK5.OBJ}, &
       BANK6{TABLES.OBJ,BANK6.OBJ}, &
       BANK7{BANK7.OBJ,MISC.OBJ} &
       BANKAREA(0000H,0FFFFH)
```

你可以在一个页面中放入多个模块，也可指定模块中的某一段。对公共区域也是这样。最后一点是，连接器会在每个页面中根据各段列出的顺序，为它们分配地址。如果你想一个功能段在代码段的低地址，把它作为BANKx指令的第一个参数。

7 结论

这章我们讨论了如何使用Keil提供的开发包来提高你的程序。给中断服务程序分配寄存器组，控制连接器的覆盖可以提高代码的性能。还可以对系统的RAM和ROM进行扩展。希望读者能够充分的利用这些功能。下一章我们将讨论设计方面的技术，这些技术可以说是未来的趋势。

第十章 8051的模糊控制

1 介绍

软件开发就和社会一样，都是向前发展的。在嵌入式控制系统，特殊控制系统中，最新的发展趋势是模糊逻辑控制。美国的工程师已经开始对模糊逻辑进行研究，发现它在解决某些问题方面是一个非常好的工具。但是现在有不少人鼓吹模糊逻辑能够解决任何系统问题，要你们买他的开发包。你应该清醒的认识到，模糊逻辑虽然对不少系统来说是个非常好的解决方案，但对很多系统来说帮助不大。这章你将了解不需要去购买昂贵的模糊逻辑开发工具，有一种简单的方法可在8051上使用模糊控制。在此之前，你要先了解一下模糊控制。

2 什么是模糊逻辑

我们认识事物的一般逻辑是，要么对，要么错，不可能两者都是。举个例子，5比10小是对的。这种逻辑和很多情况（如线形问题）是相符合的，而且也可用于曲线的情况。它的优点就是，很适用于计算机这种使用二进制数的机器。但是在很多情况下这种逻辑并不适用。

在真实的世界中，很多事物在某种程度上是真确的或是错误的。一部份真并且一部份假在模糊逻辑中是基本概念。模糊逻辑中是用一个数据点在某个指定范围内出现的程度来表示这个概念的。1表示一定在范围内，0表示一定不在范围内。在0和1之间有无限种程度（如.25，.5，.75等）。 表0-1

Type of Day	Degree of Membership
Cold	0.00
Chilly	0.00
Mild	0.00
Warm	0.25
Hot	1.00

假设，如果室外温度为90度，那么对于这个温度用能用表0-1列出的天气类型来描述。

在这里，每种天气类型可以看成是一个范围，数据点（90度）属于这个范围的程度被列了出来。对于一个数据点，它属于某个范围的程度是有严格定义的。这个例子用一张图表给出了它们之间的关系。cold,chilly,mild,warm,hot分别对应着一个函数。

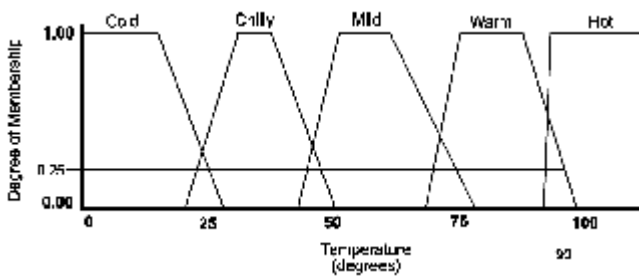

图0-1 温度函数

从图中可以看出，90度两个函数（warm,hot）有交点,因此，它们的模糊度不为0，而其它的函数没有交点，所以模糊度为0。90度和warm函数的交点的纵坐标为0.25，即它的模糊度，同样，在hot范围内的模糊度为1.00。

功能函数曲线可以为任何形状，最为常见的为梯形，其它形状都可以从梯形演变过来。下面是一些常见的模糊功能函数图。

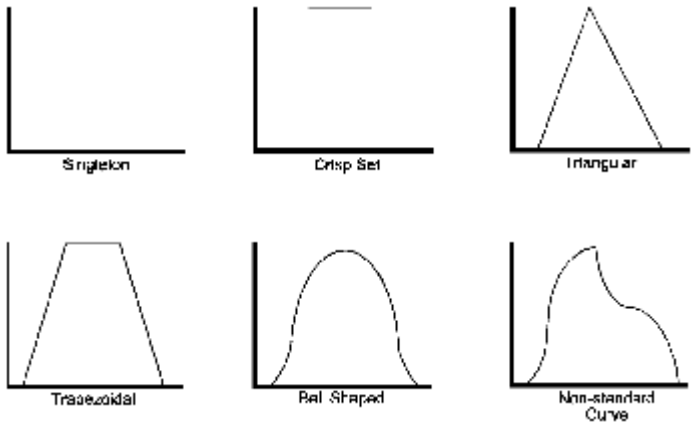

图0-2 一些模糊函数类型

前两个曲线形状（singleton和crisp）类似于布尔逻辑，要么u=0，要么u=1。其余的函数曲线u可以为任何值（u为模糊度）。改变trapezoidal曲线的曲度可以使之成为singleton，crisp，triangular型曲线。这样，就可以用一种模糊曲线表示四种模糊设置（trapezoidal，triangular，crisp和singleton）。另外两种函数比较难表示。

模糊分析最基本的要素就是模糊函数类型。一个模糊系统由模糊函数和相应的操作组成。例如，一个基于温度的风扇调速模糊控制系统遵循这样的原则：如果温度升高，则风扇速度加快。通过现在的温度得到一个热的模糊度，然后将这个模糊度和标准值进行比较以决定是否是热的状态，然后采取相应的行动。

一般来说一个模糊逻辑规则包括'if'部分,和'then'部分。'if'部分可包含多个条件,'then'部分也可包含多个结果，条件和结果都可用AND，OR，NOT这样的逻辑操作联系起来。当然，还有其它的模糊操作，但这三种是最常见的。这三种操作的算术意义见表0-2。

Operator	Implementation
$\mu_{(a \text{ AND } b)}$	$\min(\mu_a, \mu_b)$
$\mu_{(a \text{ OR } b)}$	$\max(\mu_a, \mu_b)$
$\mu_{(\text{NOT } a)}$	$1-\mu_a$

表0-2

一个逻辑系统由一系列逻辑规则组成，而每个逻辑规则又可有多个条件和结果。使用逻辑规则的数量由系统决定，这一系列逻辑规则被称为规则基。还可以为每个规则提供一个权。在大多数模糊系统中，每个规则的权都被置1，表明每个规则的重要性都是一样的。但是有些系统中，其中一些规则比其它的规则重要，那么它的权可以取大一些。例如把更加重要的规则的权值取为1，其它规则的权值小于1。这是因为模糊逻辑中处理的值一般都是从0到1。

规则基的大小取决于要解决问题的大小。一般的模糊系统的规则基都比较小（大约15个规则）；越复杂的系统规则越多，但是即使系统是大系统，规则的数量都控制在60以下。因为规则越多，系统作出决定所需要的时间就越长。你不必把所有可能的规则都放到系统中去，一个小的规则基一样可以控制系统的操作，但规则越多系统就越稳定，这使模糊逻辑系统能够容忍异常的输入信号。

3 模糊系统的结构

一个模糊逻辑系统需要三个操作阶段：输入预处理、模糊推断、反模糊处理。三者的关系见图0-3。

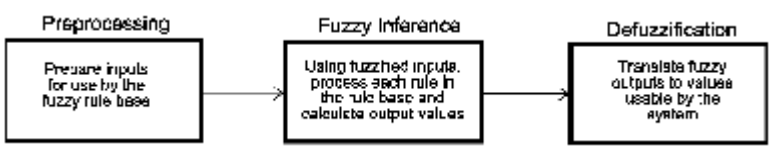

图0-3 模糊系统结构

预处理阶段进行数据采集，这个阶段包括测量输入的数据，得到模糊函数规定范围内

的值，然后用模糊规则进行评估。如果需要的话，要对输入进行计算。例如，有一个用于汽车的模糊系统把加速度作为一个输入，则需要每隔一定的时间对速度进行采样，然后计算加速度。

模糊推断估计每个模糊规则，并计算它对输出结果的影响。用前面所提供的方法来得到模糊度。由条件得到的u值反映出结果是真的程度。当规则的输出u值大于0时，就称这个规则触发了。

每个规则的结果的输出是遵循模糊设置的。评估阶段存储每个模糊输出在每个可能设置下的最大的u值。用上面的例子“如果温度是热，那么风扇转速是高”，设输入温度为90度，则对应的u值为1.00，因为90度完全在热的范围内。因此“风扇转速是高”的正确程度是1.00。如果当前“风扇转速是高”的模糊度是0，那么现在就变成了1。但是“风扇转速是高”的模糊度是1并不意味着风扇转速就将置高，这还要看其它模糊功能输出函数的结果。

反模糊处理阶段利用所得到的各种模糊输出值和数学方法计算最终的系统输出值。有几个常用方法，最简单的是最大值法。最大值法规定用给定输出的u的最大值决定与输出相关的操作。例如，给出输出转速的真实度

$u_{low} = 0.00$

$u_{medium} = 0.57$

$u_{high} = 1.00$

因此转速将被置高，因为设置high对应着最高值。最大值法很容易实现，但是，当数据点落在多个区域中的话，这种方法就没有体现模糊控制的优点。

反模糊输出的一般方法是重心法。还是利用前面的条件，可以作出下面的图

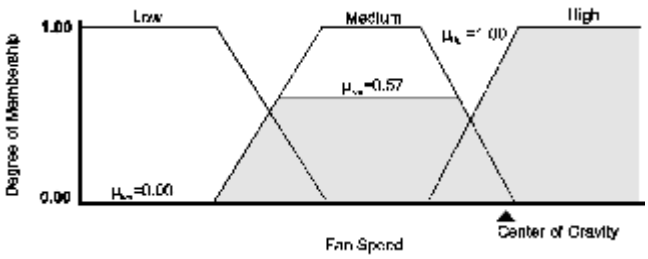

图0-4 反模糊处理

计算阴影部分的重心，这个重心值就为反模糊处理的输出值。这种方法虽然好，但是计算非常复杂。重心法一种简单的处理是把图形当成矩形处理，有下面的公式：

$$\frac{\sum_{i=0}^{n}(Vi * Ui)}{\sum_{i=0}^{n} Ui}$$

这里n是输出的设置数，Vi是定义的矩形的长度，Ui是每种设置的真实度。由这个公式得到的结果和重心法得到的结果很相似，而且容易实现。

4 模糊控制使用的场合

讲到这里，有读者可能会问，哪些场合适合使用模糊控制，哪些场合不适合使用模糊控制。一般准则是，如果你的系统已经有了一个精确的、能够使用传统逻辑有效处理的数学模型的时候就不要使用模糊控制了。而对一些不能够准确描述的系统，但是又能够凭经验控制，这一般都是一些复杂的非线形系统。这时，可以请一些有经验的专家来指定系统的操作规则。模糊控制系统能够根据这些规则得到正确的输出。

模糊控制的一个优点就是，你能够根据用语言来表达系统的解决方案。这使得解决方

案更加符合人的思考习惯。除此之外，模糊系统很容易通过修改规则和模糊函数来进行调整。模糊控制的一个缺点是，你必须证明你的方案对所有的输入都是有效的，就像传统的逻辑方案一样。

5 进行模糊控制

有了一些关于模糊的基本概念之后，我们来设计一个小的系统，从中可以学到一些模糊控制系统的设计方法。进行设计之前，先用简单的语言来描述这个系统，这意味着你要知道什么是输入量，什么是输出量和它们的类型。

假如你设计一个系统，控制一辆自动力汽车在某一点停下来。为了实现这个目的，需要知道汽车现在离这一点的距离和汽车现在的速度。而输出将控制刹车的力度。

用上述语言描述系统之后，我们确定了两个输入量：汽车离停车点的距离和汽车现在的速度。系统的输出量为刹车的程度。下面的任务就是要定义输入和输出的模糊范围。你先不要考虑具体的数值，而是做一些基本的描述。

根据对系统的感性认识用语言来描述它。这种认识可以来自专家或通过调查和研究。举个例子，可以找一个驾驶汽车有20年之久的司机，从他那里了解操作汽车的各种参数。在这里，我根据驾驶汽车一般的经验来进行设计。

首先考虑离停车点的距离。当汽车和停车点的距离达到一定的程度之后才启动系统，因为驾驶员不会在汽车离停车点还距离一公里的时候就去考虑放慢速度，准备刹车的。当汽车离停车点还有几百米的时候才会开始减速。我们可以把汽车离停车点的距离（语言符号为DISTANCE）分为几类：FAR，NEAR，CLOSE，VCLOSE(very close)。其中VCLOSE设置中包括汽车已经到达停车点。

第二个输入量是汽车的速度（语言符号是VELOCITY）。它也被分成几类：VSLOW，FAST，MEDIUM，SLOW，VSLOW。其中VSLOW设置中包括速度0。

输出量为刹车的力度用符号BRAKE表示，也被分为了几类：NONE，LIGHT，MEDIUM，VHARD。它们的数学意义将在以后定义。

有了输入输出量之后就可以定义规则了，有些模糊控制系统的设计者认为，成员函数应该在规则定义之前定义，不过这只是个人喜好问题。我先定义规则的原因是，这样可以更加全面的了解系统。

定义规则基最简单的办法是用输入量建立一个表格，然后填入输出量。这使系统更加具体。但要注意表格只适用于输入量为AND操作的情况。如下表：

		DISTANCE			
		FAR	NEAR	CLOSE	VCLOSE
VELOCITY	VFAST				
	FAST				
	MEDIUM				
	SLOW				
	VSLOW				

表0-3

一旦建立了上面的表格之后，就把它看成真值表，把结果填进去。举个例子，如果速度为VFAST并且距离为FAR，那么结果为MEDIUM。下面为完成的表格。

		DISTANCE			
		FAR	NEAR	CLOSE	VCLOSE
VELOCITY	VFAST	MEDIUM	HARD	VHARD	VHARD
	FAST	MEDIUM	HARD	VHARD	VHARD
	MEDIUM	LIGHT	MEDIUM	HARD	HARD
	SLOW	NONE	NONE	LIGHT	MEDIUM
	VSLOW	NONE	NONE	NONE	LIGHT

表0-4

有些时候，一些规则可被简化成一个规则。假设，规则“如果VELOCITY是VSLOW并且

DISTANCE是VCLOSE，那么BRAKE是LIGHT”改成“如果VELOCITY是VSLOW并且DISTANCE是VCLOSE，那么BRAKE是NONE”，表的最底层可简化为“如果VELOCITY是VSLOW，那么BRAKE是NONE”。

系统规则被建立起来之后，就开始建立每个模糊设置的成员函数。你要知道每个输入的范围。例如，要建立VELOCITY的模糊成员函数，你要知道它的范围是从0MPH到25MPH。下面是VELOCITY的成员函数。

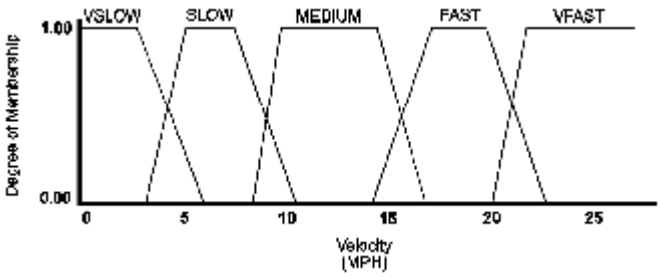

图0-5 VELOCITY的成员函数

成员函数的定义没有一定的规则，根据实际系统的具体情况而定，不一定要包含横坐标上面所有的点，也不一定非要有数据点同时对应着两个函数。下面是DISTANCE和BRAKE的成员函数。

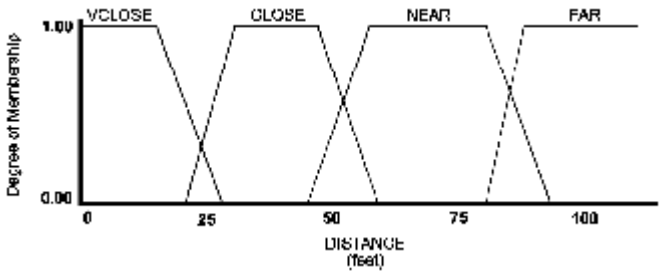

图0-6 DISTANCE的成员函数

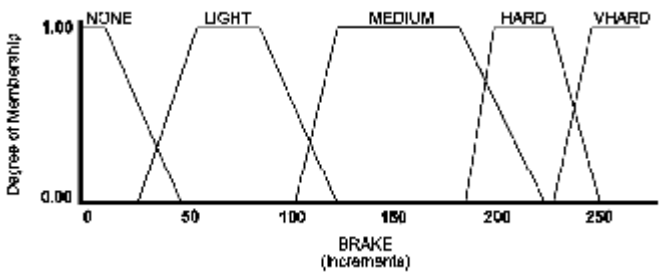

图0-7 BRAKE的成员函数

当所有这些完成之后，就开始编写程序来实现这些模糊功能。

6 模糊功能的实现

用8位控制器进行逻辑控制。首先对规则和成员函数进行定义，定义可以手工进行，也可以使用买来的工具。这里我们使用手工的方法。

在8位机上实现逻辑控制，要考虑如何在系统中表达这些逻辑规则。最好把条件和结果都用8位数据表示，这样就不能使用规则的权和使用括号把一些操作放入条件中。这里讨论的模糊逻辑不提供AND和OR操作，以及超过8位数据的输入量和输出量处理。

逻辑规则基被放入一个存在代码区的数组中。数组中的每个元素为一个字节，包含了逻辑规则的一个分支。如果存储空间够的话，也可以把数据放在一个结构中，这能使你快速的得到分支的信息。以字节存储分支的结构如下：

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 = antecedent 1 = consequence	membership function for input or output used			0 = AND 1 = OR	input or output number to use		

表0-5

如果第7位为0就认为这个分支为条件分支，2到0位的值就是输入数据。6到4位的值就是输入量的成员函数。第3位表示对这个条件得到的u值是进行或操作还是与操作。

如果第7位为1就认为这个分支为结果分支，2到0位的值就是输出数据。第3位对结果分析没有影响。

知道了分支的内部结构之后，就很容易建立规则了。还是汽车的例子，设输入0为速度，输入1为距离，输出只有一个刹车力度，设为输出0。功能函数也从0到n，本例中n的最大值为5，因为速度输入有5个成员函数。在代码中为每个可能的分支定义常量，例如：“VELOCITY IS VFAST”的常量名为“VEL_VFAST”。列表0-1为常量定义。

列表 0-1

```
// 定义速度输入常量
#define VEL_VSLOW 0x00
#define VEL_SLOW 0x10
#define VEL_MEDIUM 0x20
#define VEL_FAST 0x30
#define VEL_VFAST 0x40
// 定义距离输入常量
#define DIST_VCLOSE 0x01
#define DIST_CLOSE 0x11
#define DIST_NEAR 0x21
#define DIST_FAR 0x31
// 定义刹车输出常量
#define BRAKE_NONE 0x80
#define BRAKE_LIGHT 0x90
#define BRAKE_MEDIUM 0xA0
#define BRAKE_HARD 0xB0
#define BRAKE_VHARD 0xC0
```

有了上面的常量定义后，规则的描述就很简单了。对于汽车系统来说，规则的形式为“如果输入x1为y1且输入x2为y2，那么输出x3为y3”。你也可以通过分配分支的位置使用像AND和OR这样的连接词。例如要表达“如果velocity为vfast或velocity为slow且距离为far，那么刹车为none”，可用下面的方法：

```
VEL_FAST, VEL_SLOW | 0x08, DIST_FAR, BRAKE_NONE
```

把建立的所有规则存入数组中，在汽车的例子中，所有可能的规则都存在一个规则基数组中。

列表0-2。

```
unsigned char code rules[RULE_TOT]={ // 模糊系统规则
//    if...      and...       then...
  VEL_VSLOW,   DIST_VCLOSE, BRAKE_LIGHT,
  VEL_VSLOW,   DIST_CLOSE,   BRAKE_NONE,
  VEL_VSLOW,   DIST_NEAR,    BRAKE_NONE,
```

```
  VEL_VSLOW,   DIST_FAR,     BRAKE_NONE,
  VEL_SLOW,    DIST_VCLOSE,  BRAKE_MEDIUM,
  VEL_SLOW,    DIST_CLOSE,   BRAKE_LIGHT,
  VEL_SLOW,    DIST_NEAR,    BRAKE_NONE,
  VEL_SLOW,    DIST_FAR,     BRAKE_NONE,
  VEL_MEDIUM,  DIST_VCLOSE,  BRAKE_HARD,
  VEL_MEDIUM,  DIST_CLOSE,   BRAKE_HARD,
  VEL_MEDIUM,  DIST_NEAR,    BRAKE_MEDIUM,
  VEL_MEDIUM,  DIST_FAR,     BRAKE_LIGHT,
  VEL_FAST,    DIST_VCLOSE,  BRAKE_VHARD,
  VEL_FAST,    DIST_CLOSE,   BRAKE_VHARD,
  VEL_FAST,    DIST_NEAR,    BRAKE_HARD,
  VEL_FAST,    DIST_FAR,     BRAKE_MEDIUM,
  VEL_VFAST,   DIST_VCLOSE,  BRAKE_VHARD,
  VEL_VFAST,   DIST_CLOSE,   BRAKE_VHARD,
  VEL_VFAST,   DIST_NEAR,    BRAKE_HARD,
  VEL_VFAST,   DIST_FAR,     BRAKE_MEDIUM
};
```

规则建立完毕，下面开始定义模糊成员函数，我们认为你的输入功能函数要么是梯形的，要么可以从梯形转变过来。而为了简化反模糊处理，输出功能函数都为矩形。

当输入功能函数为梯形时，用4个字节就可以描述它。我们把梯形看成一个切去头部的三角形。软件通过存储折点和斜率来描述这个三角形。图0-8是一个例子。

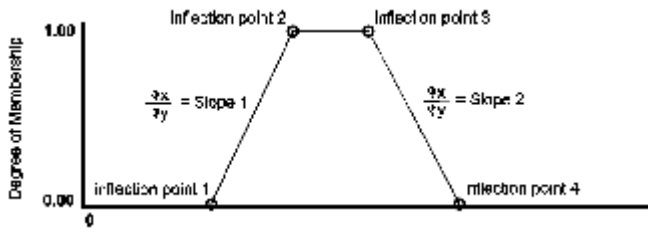

图0-8

存储点1和点3，斜率1和斜率2。有了这4个值，软件就可以得到u值。u值用一个无符号字节来表示：FFH为全真，00H为全假。用整型来计算u值，以防范围超过范围，当超过范围时，可以把数值截取在00H到FFH之内。功能函数值被存储在一个3维数组中。见列表0-3：

列表 0-3

```
unsigned char code input_memf[INPUT_TOT][MF_TOT][4]={
      // 输入功能函数以点斜式方式存储. 第一维是输入号
      // 第二维是成员函数标号，第三维是点斜式数据
  // 速度功能函数
  {
    { 0x00, 0x00, 0x1E, 0x09 },      // VSLOW
    { 0x1E, 0x0D, 0x50, 0x09 },      // SLOW
    { 0x50, 0x0D, 0x96, 0x0D },      // MEDIUM
    { 0x8C, 0x06, 0xC8, 0x09 },      // FAST
    { 0xC8, 0x0D, 0xFF, 0x00 }       // VFAST
  },
```

```
  // 距离功能函数
  {
    { 0x00, 0x00, 0x2B, 0x0A },  // VCLOSE
    { 0x33, 0x08, 0x80, 0x0A },  // CLOSE
    { 0x6E, 0x07, 0xC7, 0x08 },  // NEAR
    { 0xC7, 0x0A, 0xFF, 0x00 }   // FAR
  }
};
```

表中的值由你的功能函数决定。当功能函数的取值范围确定以后，把它们转换成新的范围（从0到FF），4个点按照这个范围进行转换。当你有了这4个点的16进制值之后，可用下面的程序把他们在转化成所需要的斜率。程序的列表如下：

列表 0-4

```
#include <stdio.h>

void main(void) {
  unsigned char val[4], output[4], ans, flag;
  do {
    printf("\n\nenter 4 hex points: ");
    scanf(" %x %x %x %x", &val[0], &val[1], &val[2], &val[3]);
    output[0]=val[0];
    output[2]=val[2];
    if (val[1]-val[0]) {
      output[1]=(0xFF+((val[1]-val[0])/2))/(val[1]-val[0]);
    } else {
      output[1]=0;
    }
    if (val[3]-val[2]) {
      output[3]=(0xFF+((val[3]-val[2])/2))/(val[3]-val[2]);
    } else {
      output[3]=0x00;
    }
    printf("\nThe point-slope values are: %02X %02X %02X
                  %02X\n\n",output[0], output[1], output[2], output[3]);
    do {
      flag=1;
      printf("run another set of numbers? ");
      while (!kbhit());
      ans=getch();
      if (ans!='y' && ans!='Y' && ans!='n' && ans!='N') {
        flag=0;
        printf("\nhuh?\n");
      }
    } while (!flag);
  } while (ans=='y' || ans=='Y');
```

```
  printf("\nlater, hosehead!\n");
}
```

这个小程序可以帮你建立系统的输入功能函数，输入功能函数建立起来了后，就该建立输出功能函数了，把输出功能函数看成矩形，并选择函数体的中点，这样做可以简化反模糊处理的数学过程。

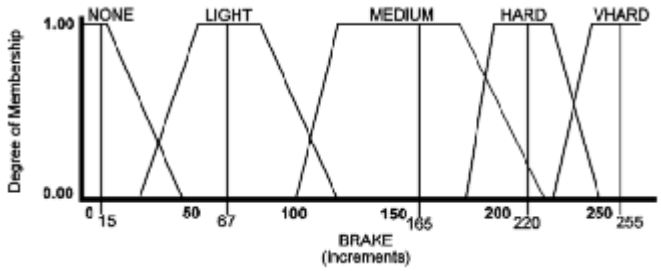

图0-9 输出功能函数

这些输出值被存储在一张表中

列表 0-5

```
unsigned char code output_memf[OUTPUT_TOT][MF_TOT]={
      // 输出成员函数
      // 第一维是输出号,第二维是成员函数标号
  { 15, 67, 165, 220, 255, 0, 0, 0 } // braking force singletons:
                                     // NONE, LIGHT, MEDIUM, HARD,
                                     // VHARD
};
```

模糊控制函数通过遍历规则基数组进行估计。分析条件时，把当前规则中的u值保存在变量'if_val'中，条件检测结束后开始估计结果,模糊控制函数通过比较'if_val'和当前输出的参考u值来得出结果，如果当前保存在'if_val'中的数大于参考的输出值，则就把'if_val'中的值作为新的输出值。一旦结果分析完毕，开始一个新的规则查询时，恢复'if_val'值。

模糊控制的源代码见列表0-6。当前的正在进行分析的分支被保存在可位寻址区，以便对里面的位进行快速寻址。

列表 0-6

```
/*********************************************************************
Function: fuzzy_engine
Description: 实施规则基中的规则
Parameters: 无
Returns: 无.
Side Effects: 无
*********************************************************************/
unsigned char bdata clause_val;      // 保存当前的分支进行
                                     // 快速访问
sbit operator = clause_val^3;        // 这位表示所使用的模糊操作
sbit clause_type = clause_val^7;     // 表示分支是否是条件分支
                                     // 或者是结果分支
void fuzzy_engine(void) {
  bit then;                          // 当正在分析结果时
                                     // 置位
  unsigned char if_val,              // 保存当前规则中条
```

```
                                    // 件分支中的值
          clause,                   // 规则基中当前的分支
          mu,                       // 保存当前分支中的值
          inp_num,                  // 当前条件使用的输入
          label;                    // 被条件使用的成员函数
  then=0;                           // 设第一个分支是条件分支
  if_val=MU_MAX;                    // max out mu for the first rule
  for (clause=0; clause<RULE_TOT; clause++) { // 遍历每条规则
    clause_val=rules[clause];       // 读入当前的分支
    if (!clause_type) {             // 当前的分支是不是条件分支
      if (then) {                   // 是否正在分析结果...
          then=0;
          if_val=MU_MAX;            // 复位mu
      }
      inp_num=clause_val & IO_NUM;  // 得到当前输入号
      label=(clause_val & LABEL_NUM) / 16; // 得到功能函数
      mu=compute_memval(inp_num, label);   // 得到条件分支的值
      if (operator) {                      // 如果是 OR
                                           // 操作...
        if (mu > if_val) {                 // 取最大值
          if_val=mu;
        }
      } else {                             // 如果是 AND操作
          if (mu < if_val) {               // 取最小值
          if_val=mu;
        }
      }
    } else {                               // 当前分支是结果
        then=1;                            // 置位标志位
        // 如果当前规则的mu比参考的值要大,保存这个值作为新的模糊输出
        if (outputs[clause_val & IO_NUM]
                [(clause_val & LABEL_NUM) / 16] < if_val) {
          outputs[clause_val & IO_NUM]
                [(clause_val & LABEL_NUM) / 16]=if_val;
        }
      }
   }
   defuzzify();                    // 用COG方法计算模糊输出
                                   // 和反模糊输出
}
```

通过调用'compute_memval'函数来估计每个给定输入的分支的u值。把这段代码放在一个函数中是为了当功能函数改变时，可以很方便的修该其代码。

列表 0-7

```
/*****************************************************************
```

```
Function: compute_memval
Description: 计算条件分支的mu值,设功能函数是以点斜式方式存储的
Parameters: inp_num - unsigned char. 使用的输入号
            label - unsigned char. 输入使用的功能函数
Returns:    unsigned char. 计算的 mu值
Side Effects: 无
*********************************************************************/
unsigned char compute_memval(unsigned char inp_num,
                             unsigned char label) {
  int data temp;
  if (input[inp_num] < input_memf[inp_num][label][0]) {
                                  // 如果输入不在曲线下
                                  // u值为0
  return 0;
  } else {
    if (input[inp_num] < input_memf[inp_num][label][2]) {
      temp=input[inp_num];          // 用点斜式计算mu
      temp-=input_memf[inp_num][label][0];
      if (!input_memf[inp_num][label][1]) {
        temp=MU_MAX;
      } else {
          temp*=input_memf[inp_num][label][1];
      }
      if (temp < 0x100) {           // 如果结果不超过1
        return temp;                // 返回计算结果
      } else {
          return MU_MAX;            // 确保mu值在范围内
      }
     } else {                       // 输入落在第二条斜线上
         temp=input[inp_num];       // 用点斜式方法
                                    // 计算 mu
         temp-=input_memf[inp_num][label][2];
         temp*=input_memf[inp_num][label][3];
         temp=MU_MAX-temp;
         if (temp < 0) {            // 确保结果不小于0
           return 0;
         } else {
             return temp;           // mu为正 – 返回结果
         }
      }
  }
  return 0;
}
```

当遍历完所有规则后，相应的输出被保存在outputs数组中，模糊控制函数调用

defuzzify功能把数组中的输出值转变成可被系统使用的COG输出值。计算的方式采用我们以前所讨论过的简化了的重心法。这个过程占用了模糊控制中的大部分时间。下面是该函数的代码。

列表 0-8

```
/*****************************************************************
Function: defuzzify
Description: 计算模糊输出的重心，并调用函数把它
             转换成可被系统使用的输出量
Parameters: 无.
Returns: 无.
Side Effects: outputs[][] 数组被清零.
*****************************************************************/
void defuzzify(void) {
  unsigned long numerator, denominator;
  unsigned char i, j;
  for (i=0; i<OUTPUT_TOT; i++) {          // 对所有的输出...
    numerator=0;                          // 恢复总数值
    denominator=0;
    for (j=0; j<MF_TOT; j++) {            // 计算总和值
      numerator+=(outputs[i][j]*output_memf[i][j]);
      denominator+=outputs[i][j];
      outputs[i][j]=0;                    // 清零输出作为参考使用
    }
    if (denominator) {                    // 确保分母是0的情况不发生
      fuzzy_out[i]=numerator/denominator; // 确定 COG
    } else {
        fuzzy_out[i]=DEFAULT_VALUE;       // 没有规则被触发
    }
  }
  normalize();                        // 把模糊输出作为正常输出
}
```

7 方案调整

前面所描述的模糊控制系统使用的存发现只使用了3字节的内部RAM，80字节的用来存放数组)。对于它较强的功能来说这些得到了下面的结果。

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	29576
12	35	31115
57	29	31714
80	43	33167
D0	D0	37112
FF	00	33283
Average processor cycles		32661

我们看到程序执行的时间是很长的。如果你的存储空间不大的话，就不得不忍受这种速度而节省空间。如果你认为这种速度太慢了，那么就不得不牺牲一些存储空间以获得速度上的提高。

最快速的改善系统的方法就是改变模糊功能函数的实现方式。以前采用的点斜式存储方式，这种方式可以节省存储空间，但

处理起来的时间较长，如果你愿意放弃一些存储空间，可以把输入范围内的256个数据点存储下来，避免进行数值计算。在汽车系统中将占用2304个字节的EPROM空间。除此之外，还要限制输入和输出数组的表长度。这样，汽车模糊系统中就有一个输出数组，两个输入数组和五个功能函数。做了这个改变后，系统的处理速度将更快。得到u值所需的时间也会缩短，以前所用的'compute_memval'函数被下面一行代码所替代

```
mu=input_memf[inp_num][label][input[inp_num]];
```

做了以上改动后，就可以得到如下效果。

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	7506
12	35	7549
57	29	7579
80	43	7549
D0	D0	7568
FF	00	7561
Average processor cycles		7552

系统的执行速度差不多提高了4倍，7500个指令周期的执行时间比33000个指令周期的执行时间更容易被系统所接受。新的系统使用了1个字节内部RAM，8个字节外部RAM和3010个字节的EPROM，其中2625个字节用来存放功能函数表。

还可以对系统进行改进。因为在这个例子中一旦一个条件的u值为0，后面的条件和结果的字节内部RAM，8个字节外部RAM和3031个字节的统的执行效果如下表。

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	5684
12	35	5750
57	29	6076
80	43	5750
D0	D0	6118
FF	00	5739
Average processor cycles		5853

列表0-9是新的汽车模糊处理的源代码。所有的代码都保存在一个文件中，为它定义一个头文件，以便和系统的其他部分进行接口。

列表 0-9

```
#define OUTPUT_TOT 1
#define MF_TOT 5
#define INPUT_TOT 2
#define MU_MAX 0xFF
#define IO_NUM 0x07
#define LABEL_NUM 0x70
#define DEFAULT_VALUE 0x80
unsigned   char outputs[MF_TOT],     // 模糊输出mu值
           fuzzy_out;                // 模糊控制值
unsigned char input[INPUT_TOT] ={   // 模糊输入
  0, 0
};
unsigned char code input_memf[INPUT_TOT][MF_TOT][256]={
// 输入功能函数
{
  { // velocity: VSLOW
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xF6,
0xED, 0xE4, 0xDB, 0xD2, 0xC9, 0xC0, 0xB7, 0xAE, 0xA5, 0x9C, 0x93, 0x8A, 0x81, 0x78,
```

```
0x6F, 0x66,
0x5D, 0x54, 0x4B, 0x42, 0x39, 0x30, 0x27, 0x1E, 0x15, 0x0C, 0x03, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
  },
  { // velocity: SLOW
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x0D,
0x1A, 0x27, 0x34, 0x41, 0x4E, 0x5B, 0x68, 0x75, 0x82, 0x8F, 0x9C, 0xA9, 0xB6, 0xC3,
0xD0, 0xDD,
0xEA, 0xF7, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xF6, 0xED, 0xE4, 0xDB, 0xD2, 0xC9, 0xC0, 0xB7, 0xAE, 0xA5, 0x9C, 0x93, 0x8A,
0x81, 0x78,
0x6F, 0x66, 0x5D, 0x54, 0x4B, 0x42, 0x39, 0x30, 0x27, 0x1E, 0x15, 0x0C, 0x03, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00
  },
  { // velocity: MEDIUM
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x0D,  0x1A,  0x27,  0x34,  0x41,  0x4E,  0x5B,  0x68,  0x75,  0x82,  0x8F,  0x9C,  0xA9,
0xB6,  0xC3,
0xD0,  0xDD,  0xEA,  0xF7,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,
0xFF,  0xFF,
0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,
0xFF,  0xFF,
0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,
0xFF,  0xFF,
0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xFF,  0xF2,  0xE5,  0xD8,  0xCB,  0xBE,  0xB1,  0xA4,
0x97,  0x8A,
0x7D,  0x70,  0x63,  0x56,  0x49,  0x3C,  0x2F,  0x22,  0x15,  0x08,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
0x00,  0x00,
0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,  0x00,
```

```
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
  { // velocity: FAST
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06,
0x0C, 0x12,
0x18, 0x1E, 0x24, 0x2A, 0x30, 0x36, 0x3C, 0x42, 0x48, 0x4E, 0x54, 0x5A, 0x60, 0x66,
0x6C, 0x72,
0x78, 0x7E, 0x84, 0x8A, 0x90, 0x96, 0x9C, 0xA2, 0xA8, 0xAE, 0xB4, 0xBA, 0xC0, 0xC6,
0xCC, 0xD2,
0xD8, 0xDE, 0xE4, 0xEA, 0xF0, 0xF6, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF6, 0xED, 0xE4, 0xDB, 0xD2,
0xC9, 0xC0,
0xB7, 0xAE, 0xA5, 0x9C, 0x93, 0x8A, 0x81, 0x78, 0x6F, 0x66, 0x5D, 0x54, 0x4B, 0x42,
0x39, 0x30,
0x27, 0x1E, 0x15, 0x0C, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
  },
  { // velocity: VFAST
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0D, 0x1A, 0x27, 0x34, 0x41,
0x4E, 0x5B,
0x68, 0x75, 0x82, 0x8F, 0x9C, 0xA9, 0xB6, 0xC3, 0xD0, 0xDD, 0xEA, 0xF7, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF
  }
},
{
{ // distance: VCLOSE
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF5, 0xEB,
0xE1, 0xD7,
0xCD, 0xC3, 0xB9, 0xAF, 0xA5, 0x9B, 0x91, 0x87, 0x7D, 0x73, 0x69, 0x5F, 0x55, 0x4B,
0x41, 0x37,
0x2D, 0x23, 0x19, 0x0F, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
{ // distance: CLOSE
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38, 0x40, 0x48, 0x50,
0x58, 0x60,
0x68, 0x70, 0x78, 0x80, 0x88, 0x90, 0x98, 0xA0, 0xA8, 0xB0, 0xB8, 0xC0, 0xC8, 0xD0,
0xD8, 0xE0,
0xE8, 0xF0, 0xF8, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xF5, 0xEB, 0xE1, 0xD7, 0xCD, 0xC3, 0xB9, 0xAF, 0xA5, 0x9B, 0x91, 0x87, 0x7D,
0x73, 0x69,
0x5F, 0x55, 0x4B, 0x41, 0x37, 0x2D, 0x23, 0x19, 0x0F, 0x05, 0x00, 0x00, 0x00, 0x00,
```

```
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
{ // distance: NEAR
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x07,
0x0E, 0x15, 0x1C, 0x23, 0x2A, 0x31, 0x38, 0x3F, 0x46, 0x4D, 0x54, 0x5B, 0x62, 0x69,
0x70, 0x77,
0x7E, 0x85, 0x8C, 0x93, 0x9A, 0xA1, 0xA8, 0xAF, 0xB6, 0xBD, 0xC4, 0xCB, 0xD2, 0xD9,
0xE0, 0xE7,
0xEE, 0xF5, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF7, 0xEF, 0xE7, 0xDF, 0xD7, 0xCF,
0xC7, 0xBF,
0xB7, 0xAF, 0xA7, 0x9F, 0x97, 0x8F, 0x87, 0x7F, 0x77, 0x6F, 0x67, 0x5F, 0x57, 0x4F,
0x47, 0x3F,
0x37, 0x2F, 0x27, 0x1F, 0x17, 0x0F, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
{ // distance: FAR
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0A, 0x14, 0x1E, 0x28, 0x32, 0x3C,
0x46, 0x50,
0x5A, 0x64, 0x6E, 0x78, 0x82, 0x8C, 0x96, 0xA0, 0xAA, 0xB4, 0xBE, 0xC8, 0xD2, 0xDC,
0xE6, 0xF0,
0xFA, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF
  }
}
};
unsigned char code output_memf[MF_TOT]={
15, 67, 165, 220, 255     // braking force singletons:
                          // NONE, LIGHT, MEDIUM, HARD,
                          // VHARD
```

```
};

//*****************************************************************
// 规则基定义如下， 每个分支可以是条件，也可以是结果。如果第7位为0
// 则为条件，反之为结果。6到4位确定规则对应的功能函数标号。第3位表
// 明条件的操作规则。为0说明是AND操作，为1说明是OR操作。2到0位确定
// 分支使用的输入或输出函数号
//*****************************************************************
// define constants for the velocity input
#define VEL_VSLOW  0x00
#define VEL_SLOW   0x10
#define VEL_MEDIUM 0x20
#define VEL_FAST   0x30
#define VEL_VFAST  0x40
// define constants for the distance input
#define DIST_VCLOSE 0x01
#define DIST_CLOSE  0x11
#define DIST_NEAR   0x21
#define DIST_FAR    0x31
// define constants for the brake output
#define BRAKE_NONE 0x80
#define BRAKE_LIGHT 0x81
#define BRAKE_MEDIUM 0x82
#define BRAKE_HARD 0x83
#define BRAKE_VHARD 0x84
#define RULE_TOT 60
unsigned char code rules[RULE_TOT]={ // 模糊系统规则
// if...       and...        then...
VEL_VSLOW,   DIST_VCLOSE,    BRAKE_LIGHT,
VEL_VSLOW,   DIST_CLOSE,     BRAKE_NONE,
VEL_VSLOW,   DIST_NEAR,      BRAKE_NONE,
VEL_VSLOW,   DIST_FAR,       BRAKE_NONE,
VEL_SLOW,    DIST_VCLOSE,    BRAKE_MEDIUM,
VEL_SLOW,    DIST_CLOSE,     BRAKE_LIGHT,
VEL_SLOW,    DIST_NEAR,      BRAKE_NONE,
VEL_SLOW,    DIST_FAR,       BRAKE_NONE,
VEL_MEDIUM,  DIST_VCLOSE,    BRAKE_HARD,
VEL_MEDIUM,  DIST_CLOSE,     BRAKE_HARD,
VEL_MEDIUM,  DIST_NEAR,      BRAKE_MEDIUM,
VEL_MEDIUM,  DIST_FAR,       BRAKE_LIGHT,
VEL_FAST,    DIST_VCLOSE,    BRAKE_VHARD,
VEL_FAST,    DIST_CLOSE,     BRAKE_VHARD,
VEL_FAST,    DIST_NEAR,      BRAKE_HARD,
VEL_FAST,    DIST_FAR,       BRAKE_MEDIUM,
```

```
VEL_VFAST,   DIST_VCLOSE,    BRAKE_VHARD,
VEL_VFAST,   DIST_CLOSE,     BRAKE_VHARD,
VEL_VFAST,   DIST_NEAR,      BRAKE_HARD,
VEL_VFAST,   DIST_FAR,       BRAKE_MEDIUM
};

/*********************************************************************
Function: defuzzify
Description: 计算模糊输出的重心，并把结果转化成系统控制量
Parameters: 无
Returns: 无.
Side Effects: outputs[] 数组被清零
*********************************************************************/
void defuzzify() {
  unsigned long numerator, denominator;
  unsigned char j;
  numerator=0;                          // 和值清零
  denominator=0;
  for (j=0; j<MF_TOT; j++) {            // 累加结果
    numerator+=(outputs[j]*output_memf[j]);
    denominator+=outputs[j];
    outputs[j]=0;                       // 结果使用完毕后被清零
  }
  if (denominator) {                    // 确认分母不为0
    fuzzy_out=numerator/denominator;    // 计算重心
  } else {
    fuzzy_out=DEFAULT_VALUE;            // 没有规则被触发
                                        // 输出为默认值
  }
  normalize();                          // 将模糊输出转变为
                                        // 控制量
}

/*********************************************************************
Function: fuzzy_engine
Description: 处理逻辑规则基
Parameters: 无.
Returns: 无.
Side Effects: 无.
*********************************************************************/
unsigned char bdata clause_val;     // 对当前分支进行快速寻址
sbit operator = clause_val^3;       // 定义寻址位
sbit clause_type = clause_val^7;    // 该位表明分支是条件
                                    // 还是结果
```

```
void fuzzy_engine() {
  bit then;                          // 进行结果处理时置位
  unsigned char if_val,              // 保存当前规则中的u值
           clause,                   // 当前规则基中的分支
           mu,                       // 当前分支中的mu 值
           inp_num,                  // 条件所使用的输入号
           label;                    // 条件使用的成员函数
                                     // 的标号
  then=0;                            // 确认第一个分支是条件分支
  if_val=MU_MAX;                     // 输出值初始化为最大值
  for (clause=0; clause<RULE_TOT; clause++) { // 遍历所有规则
  clause_val=rules[clause];                   // 把当前分支读入
                                              // bdata区
  if (!clause_type) {                         // 如果当前分支是条件...
  if (then)
    then=0;
    if_val=MU_MAX;
  }
  inp_num=clause_val & IO_NUM;                // 得到输入号
  label=(clause_val & LABEL_NUM) / 16;        // 所使用的功能函数
  mu=input_memf[inp_num][label][input[inp_num]];// 得到该条件的值
  if (!mu) {                                   // 如果条件没有被触发
    do {                                       // 跳过这个条件
      clause++;
    } while (clause<RULE_TOT && !(rules[clause]&0x80));
                                               // 跳过结果
    while (clause+1<RULE_TOT && (rules[clause+1]&0x80)) {
      clause++;
    }
    if_val=MU_MAX;                             // 为下一个规则设定
  } else {
      if (mu < if_val) {                       // 取最小值
        if_val=mu;
      }
    }
} else {                                       // 当前分支是结果分支
  then=1;                                      // 进行结果处理
                                               // 标志位置1
     // 如果当前规则的mu值比参考值大,就保存这个值
     // 作为新的模糊输出值
    if (outputs[clause_val & 0x07] < if_val) {
        outputs[clause_val & 0x07]=if_val;
    }
  }
```

```
}
defuzzify();                                    // 计算模糊输出值和系统的
                                                // 控制值
}
```

8 结论

模糊控制是一种新的控制方法，它并不能解决所有的问题，但确实可以使一些问题解决起来更加方便。当你需要设计一个逻辑控制工程的时候，可以借助一些工具来设计功能函数和规则基。有些工具还能在PC上对你的模糊控制系统进行仿真和测试。在有了一定的设计经验后，这些工具使用起来是十分方便的。记住，不要去购买那些昂贵的进行模糊控制应用设计的软件。

总结

这本书向你展示了用8051进行工程设计时的许多问题。希望你读完本书后，对8051的认识能有较大的提高。如果你现在还没有够买C编译器，你应该马上去买一个。采用C语言可是你的系统设计更简单、维护更方便！

这本书覆盖面较大，从C和汇编的代码优化到8051的网络设计再到模糊控制。希望你从本书中学到的知识对你今后的系统设计有所帮助。