MSP430 系列单片机实用 C 语言程序设计

张　晞　王德银　张　晨　编著

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

MSP430 系列单片机实用 C 语言程序设计 / 张晞，王德银，张晨编著.
—北京：人民邮电出版社，2005.9

ISBN 7-115-13664-5

Ⅰ. M… Ⅱ. ①张…②王…③张… Ⅲ. ①单片微型计算机，MSP430
②C 语言－程序设计 Ⅳ. ①TP368.1②TP312

中国版本图书馆 CIP 数据核字（2005）第 089732 号

内 容 提 要

本书从应用角度出发，主要介绍 MSP430 的硬件基础部分和 IAR 公司的 MSP430 C 编译器 EW430，并对 MSP430 中的各功能模块给出了应用实例。这些实例程序按照结构化编写，经作者的封装后，读者在开发中只需稍加修改即可直接调用。书中还介绍了 MSP430 的几种典型应用，如软件模拟串行口、在线刷新 FLASH、实现中断嵌套等，并为其编写了完整代码，读者完全可以将其直接组合在自己的项目中。书中还讲解了单片机领域编写程序的规范、程序结构的安排以及如何提高编码效率等实际应用中的问题。随书光盘中包括了本书的所有程序代码。

MSP430 系列单片机实用 C 语言程序设计

◆ 编 著 张 晞 王德银 张 晨
责任编辑 刘映欣

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：17.5 插页：1
字数：426 千字 2005 年 9 月第 1 版
印数：1 – 5 000 册 2005 年 9 月北京第 1 次印刷

ISBN 7-115-13664-5/TP・4784

定价：35.00 元（附光盘）

读者服务热线：（010）67132692 印装质量热线：（010）67129223

前　言

以前出版的有关 MSP430 的书籍主要是从 MSP430 内部结构的角度来讲述的，读者可以从中获得比较全面的相关知识，但这些书的描述往往过于底层，使得读者使用时必须花费相当多的时间认真推敲各个寄存器或者模块之间的关系。本书所描述的重点与以往的相关书籍不同，本书并不准备详细地将 MSP430 重新描述一遍，相关的知识可以从 MSP430 的数据手册或者类似书籍中获得。本书将通过实际应用来讲述如何用 C 语言编写 MSP430 的程序，所提供的例子涵盖 MSP430 大多数的功能模块，而且都是采用结构化编程方法，使得这些例子中的代码能够被读者重复利用，或者只需在例子的框架内进行修改，就可以满足需要。

MSP430 的功能非常强大，其内部的寄存器非常多，关系也比较复杂，对于初学者，往往会有无从下手的感觉。通过学习本书中的例子，初学者能够尽快渡过初学 MSP430 时的瓶颈期。对于已经有经验的读者，本书更像是一本编程的代码库，因为书中的例子都立足于应用，可以重复利用，能够大大提高工作效率。本书还详细讲述了 IAR 公司的 C 语言开发工具，可以作为 C 语言开发 MSP430 的使用手册。书中所有的例程都在 IAR 公司的 C 编译器 EW430 3.1A 版本下完成，并在北京东方美源公司提供的实验板上使用 JTAG 仿真器调试通过。由于没有详细介绍 MSP430 内部的寄存器，因此，读者阅读本书的时候，手边最好有相应的 MSP430 芯片手册，以便随时查阅。

本书由张晞、王德银、张晨三人编写而成。张晞，中国矿业大学机电学院副教授，机械制造及其自动化学科带头人，长期从事单片机的教学及研究工作。王德银，高级工程师，现任河南平顶山煤业集团五矿矿长，主要从事煤矿机电自动化和安全生产监测系统的研究。张晨，现任北京东方美源公司研发部主管，多年从事单片机的应用设计工作。本书在成书过程中得到北京东方美源科技有限公司的大力支持，在此表示感谢。

由于作者水平有限，成书时间仓促，书中必定会存在不妥之处，恳请广大读者批评指正，以便在以后的版本中及时修正，联系地址：mcu@mcu163.com（作者）或 liuyingxin@ptpress.com.cn（编辑）。欢迎访问作者的个人网站 http://www.mcu163.com，欢迎与作者进行技术交流。

编　者

2005 年 8 月

目　　录

第 1 章　**MSP430** 硬件基础知识 …… 1

1.1　概述 …… 2
1.2　存储器结构 …… 2
1.3　复位 …… 4
1.4　系统时钟 …… 6
1.5　低功耗模式 …… 8
1.6　中断 …… 10
1.7　外围模块 …… 16

第 2 章　**C** 语言基础知识 …… 17

2.1　标识符与关键字 …… 18
2.1.1　标识符 …… 18
2.1.2　关键字 …… 18
2.2　数据类型 …… 19
2.2.1　基本型 …… 19
2.2.2　构造型 …… 20
2.2.3　指针型 …… 21
2.3　运算符 …… 22
2.4　函数 …… 26
2.5　数组 …… 27
2.6　指针 …… 27
2.7　位运算 …… 29
2.8　存储类型 …… 29
2.8.1　变量 …… 29
2.8.2　函数 …… 30
2.9　预处理功能 …… 30
2.9.1　宏定义 …… 30
2.9.2　条件编译 …… 31
2.9.3　文件包含 …… 31
2.10　程序的基本结构 …… 32
2.10.1　顺序结构 …… 32
2.10.2　选择结构 …… 32
2.10.3　循环 …… 34

2.10.4 跳转 ······ 35

第 3 章 IAR C 编译器的使用 ······ 37

3.1 概述 ······ 38
3.1.1 特性 ······ 38
3.1.2 软件结构 ······ 38
3.1.3 文件类型 ······ 39
3.2 开发调试环境 ······ 40
3.2.1 创建一个工程 ······ 40
3.2.2 编译链接项目 ······ 43
3.2.3 项目设置 ······ 44
3.2.4 调试 ······ 50
3.3 语言扩展 ······ 55
3.3.1 扩展关键字 ······ 56
3.3.2 内部函数 ······ 57
3.3.3 扩展定义 ······ 60
3.4 C 语言与汇编语言混合使用 ······ 63
3.4.1 调用内部函数 ······ 63
3.4.2 直接嵌入 ······ 63
3.4.3 调用汇编模块 ······ 64
3.5 编写高质量的代码 ······ 67
3.6 函数库 ······ 68

第 4 章 开发工具 ······ 97

4.1 JTAG 仿真器、编程器 ······ 98
4.2 BSL 编程器 ······ 99

第 5 章 程序设计的规范与结构 ······ 101

5.1 程序规范 ······ 102
5.2 程序结构 ······ 106
5.3 框架程序 ······ 109

第 6 章 MSP430 异步串行通信 ······ 123

6.1 串行通信简介 ······ 124
6.2 串行通信软件实现 ······ 125

第 7 章 定时器 ······ 137

7.1 16 位定时器 ······ 138
7.1.1 定时中断 ······ 139

7.1.2　PWM 输出 ······ 142
7.1.3　捕获脉冲信号周期 ······ 149
7.1.4　软件模拟异步串行通信 ······ 153
7.2　基本定时器 ······ 161

第 8 章　FLASH 的读写、擦除与 I/O 端口 ······ 167

8.1　FLASH 的读写和擦除 ······ 168
8.2　I/O 端口 ······ 174
8.2.1　非行列式键盘 ······ 175
8.2.2　行列式键盘 ······ 185

第 9 章　DMA 数据传输与 IIC 总线 ······ 193

9.1　DMA 数据传输 ······ 194
9.2　IIC 总线 ······ 200

第 10 章　FLL+锁频环与液晶屏驱动模块 ······ 215

10.1　FLL+锁频环 ······ 216
10.2　液晶屏驱动模块 ······ 218

第 11 章　AD、DA 转换 ······ 233

11.1　ADC12 ······ 234
11.2　DAC12 ······ 244

第 12 章　比较器 A ······ 251

12.1　斜边 AD 转换 ······ 252
12.2　电阻值测量 ······ 260

第 13 章　特殊处理 ······ 267

13.1　中断嵌套 ······ 268
13.2　程序异常处理 ······ 270

附录 A　MSP430 基本电路图

第 1 章　MSP430 硬件基础知识

MSP430 是美国 Texas Instruments（TI）公司于 1996 年开始推向市场的超低功耗微处理器。除了超低功耗这一特点外，在 MSP430 系列产品中还集成了很多功能模块，可以应用于各个方面，使得用一片 MSP430 芯片就能完成以前要用多片芯片组合才能完成的功能，大大地缩小了产品的体积，并且降低了成本。因此，MSP430 系列一经推出，就受到了业内人士的欢迎。迄今为止，MSP430 已经广泛应用于各个领域，尤其是便携式仪器仪表、监测、医疗器械以及汽车电子等领域。

本章将对 MSP430 的基础特性进行简明扼要地介绍，外围模块的具体操作则完全略过。对基础特性的了解是操作任何一种微处理器所必备的，仔细阅读本章内容可为学习全书打下良好的基础。

1.1 概　　述

当前市场上存在很多种微处理器，每种微处理器都各有其特色。针对不同的应用选择合适的微处理器非常重要。MSP430 的主要特点为：

（1）超低功耗　拥有 5 种低功耗模式，以适应不同的需要。CPU 从低功耗模式被唤醒，这个过程最多只需要 6μs，因此，在某些需要迅速作出反应的应用中，CPU 能够及时退出低功耗模式，进入工作模式。

（2）灵活的时钟使用方式　除了片内集成一个晶体振荡器外，还可外接 1～2 个晶体振荡器。不同的内部功能模块可根据需要使用不同的晶体振荡器，在不需要时可以通过设置寄存器将其关闭，以降低功耗。

（3）高速的运算能力　16 位 RISC 架构，125ns 指令周期。

（4）丰富的功能模块　集成了大量的功能模块，这些功能模块包括：

- 多通道 10～14 位 AD 转换器
- 双路 12 位 DA 转换器
- 比较器
- 液晶驱动器
- 电源电压检测
- 串行口 USART（UART/SPI）
- 硬件乘法器
- 看门狗定时器和多个 16 位、8 位定时器（可进行捕获、比较、PWM 输出）
- DMA 控制器
- FLASH 存储器。它可以在运行过程中由程序控制写操作和段的擦除（In system programmable），不需要额外的高电压。

（5）廉价专业的开发工具　MSP430 的芯片上包括 JTAG 接口，因此在仿真调试程序时，通过一个廉价的 JTAG 接口转换器就可以完成以往用昂贵的仿真器才能完成的功能，如设置断点、单步执行程序、读写寄存器等。

（6）灵活快速的编程方式　可通过 JTAG 和 BSL 两种方式向 CPU 内装载程序。

（7）高保密性　只需按照特定的方式将 MSP430 内部的熔丝熔断，JTAG 口便被物理性地阻断。BSL 方式所需要的密码长达 256 位，排列组合出来的密码量为 2 的 256 次幂，如此巨大的数量被破解的可能性微乎其微。

（8）低电源电压范围　1.8～3.6V。

1.2 存储器结构

MSP430 内部存储器的类型有：程序存储器 FLASH、数据存储器 RAM、外围模块寄存

器、特殊功能寄存器。

典型微型计算机的存储器都是采用冯·诺依曼结构，也称普林斯顿结构，即存放程序指令的存储器——程序存储器和存放数据的存储器——数据存储器采取统一的地址编码结构。程序存储器与数据存储器分开的地址编码结构称为哈佛结构，如 MCS-51 系列微处理器。MSP430 采用冯·诺依曼结构。其安排如图 1-1 所示，全部寻址空间为 64K 字节。需要注意的是，虽然 MSP430 是 16 位的微处理器，但其寻址空间还是按照字节来计算的。

- 从 0 至 0xF 为特殊功能寄存器。共 16 个字节，包含中断标志寄存器 1（IFG1）、中断标志寄存器 2（IFG2）、中断允许寄存器 1（IE1）、中断允许寄存器 2（IE2）、模块允许寄存器 1（ME1）、模块允许寄存器 2（ME2）。
- 从 0x10 至 0x1FF 为外围模块寄存器。包含被定时器、AD 转换器、对外端口等模块用到的寄存器。
- 从 0x200 开始为数据存储器 RAM。不同型号中数据存储器的大小不同，但都是从 0x200 地址开始向高端地址扩展。如 MSP430F149 的数据存储器容量为 2KB，其地址范围即为 0x200～0x9FF。
- 从 0x0C00 到 0x0FFF 为 BOOT ROOM。其中存储的内容是生产芯片时掩模在芯片内的一段代码，此段代码用来完成 BSL（bootstrap）功能。使芯片的保密熔丝熔断以后，仍然可以通过 BSL 方式修改芯片内的代码。

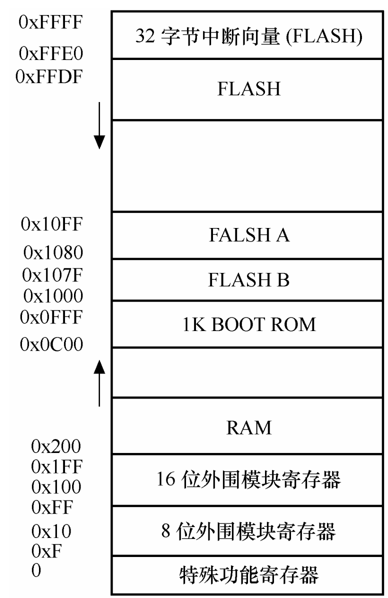


图 1-1　MSP430 存储器结构

- 从 0x1000 到 0x107F 是 128 个字节的 FLASH 存储器，称为信息存储器 B。此段存储器与高端地址存储代码的 FLASH 存储器本质上没有任何不同，同样也可以存储代码并执行，只是这一段存储器的长度较小，只有 128 个字节。主要用来存储一些掉电后仍需保存的数据。由于它是 FLASH 存储器，因此可以按照字或者字节写入，但必须整段擦除。
- 从 0x1080 到 0x10FF 为信息存储器 A。功能与信息存储段 B 相同。
- 程序存储器从 0xFFFF 开始向低端地址扩展，不同型号中程序存储器的容量不同，但都是从 0xFFFF 开始向下扩展。如 MSP430F133 的程序存储器容量为 8KB，其地址范围即为 0xE000～0xFFFF；MSP430F149 的容量为 60KB，其地址范围为 0x1100～0xFFFF。需要注意的是，在程序存储器容量为 60KB 的芯片中，程序存储器与信息存储器 A、B 发生了重合，从地址 0xFFFF 向低端地址扩展 60KB，其地址范围为 0x1000～0xFFFF，而信息存储器 A 和 B 的地址范围为 0x1000～0x10FF。程序存储器是 flash 存储器，分为若干段进行管理，可以按照字或者字节写入，擦除时无法按照字或者字节擦除，每次至少擦除一段，每段长度为 512 字节。
- 0xFFE0～0xFFFF 是程序存储器的一部分，共 32 个字节。MSP430 规定用这一段存储器来存储各种中断的中断向量。

由于程序存储器、信息存储器、数据存储器都是统一寻址的，所以，程序在这 3 种存

储器中均可执行。不同的是，程序放在数据存储器中，掉电后就会丢失，并且很容易在执行中被改写，故一般情况不会将程序放在数据存储器中执行。修改程序存储器中的内容必须经过解锁操作才能进行，否则会引起系统复位（PUC，参见 1.3 节）。有关解锁的操作会在后续章节中介绍。

MSP430 的存储结构使得使用 C 语言编程所得到的代码仍然有很高的执行效率，因此，除非对程序的大小和执行时间有很高的要求，否则都应该选用 C 语言编写程序。

1.3 复 位

复位是微处理器开始工作的起点，因此了解复位过程和复位结束时微处理器的状态对正确使用微处理器是至关重要的。

图 1-2 是 MSP430 复位的内部逻辑图。MSP430 的复位信号有两种：上电复位信号（POR）和上电清除信号（PUC）。某些型号的芯片用 Brownout Reset（BOR）模块取代了产生 POR 的模块。从图中可以看到，触发复位的信号除了通过 POR 监测电路检测系统上电而产生 POR 信号之外，还有多种信号能够触发 POR 和 PUC 信号，其中 5 种来自看门狗，1 种来自复位管脚，1 种来自写 FLASH 键值出现错误所产生的信号。

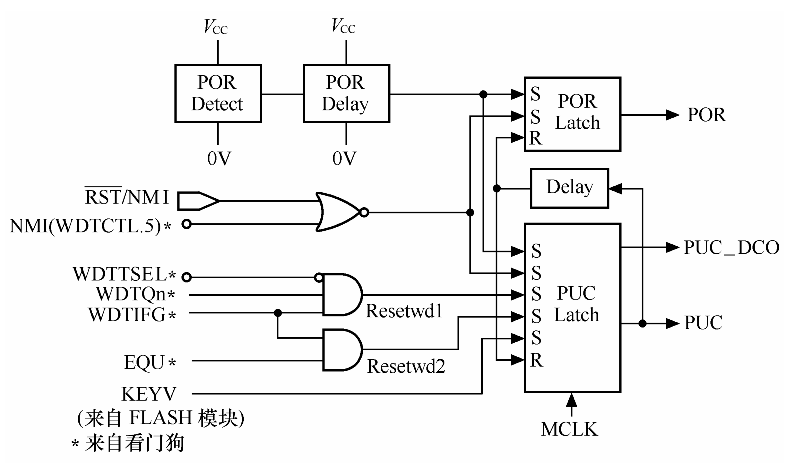


图 1-2 MSP430 内部复位逻辑电路

POR 信号只在两种情况下产生：

- 微处理器上电。
- RST/NMI 管脚被设置为复位功能，在此管脚上产生低电平时系统复位。

POR 的过程如图 1-3 所示，在微处理器上电的过程中，当电源电压超过 V_{POR} 时，延迟一段时间（POR_DELAY）后才使微处理器复位。在运行的过程中，如果电源电压 Vcc 下降到 $V_{(MIN)}$以下再恢复至超过 V_{POR}，则仍然会延迟一段时间（POR_DELAY）后才使微处理器复位。延迟时间的作用是使芯片内各模块都能够有足够的时间有效地复位。如果电源电压未下降到

$V_{(MIN)}$以下，则不会产生 POR 过程。

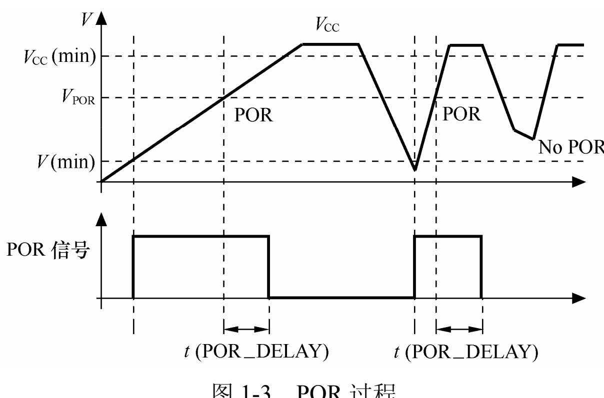


图 1-3　POR 过程

RST/NMI 管脚设置为复位功能时，管脚上产生低电平，同样会导致 POR 过程，其过程与微处理器上电产生的 POR 一致。

BOR 与 POR 类似，最终产生的信号也称为 POR 信号，其过程如图 1-4 所示。BOR 检测电源电压的上升和下降，上电和掉电都可以产生 POR 信号，导致系统复位。当电源电压上升超过 $Vcc_{(start)}$时，POR 信号开始产生，一直持续到电源电压超过 $V_{(B_IT+)}$，延迟 $t_{(BOR)}$时间后，POR 信号结束。只有电源电压下降至 $V_{(B_IT-)}$以下才能再次产生 POR。$V_{(B_IT-)}$比图 1-3 中 POR 过程中的 $V_{(MIN)}$高很多，所以，BOR 过程比 POR 过程严格，也更能保证程序运行的安全性。

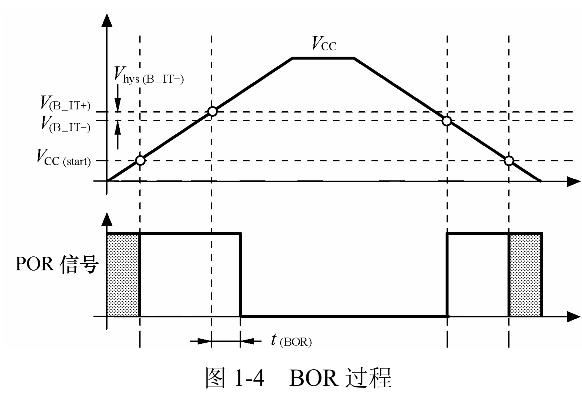


图 1-4　BOR 过程

PUC 信号产生的条件为：

- POR 信号产生。
- 看门狗有效时，看门狗定时器溢出。
- 写看门狗定时器安全健值出现错误。
- 写 FLASH 存储器安全键值出现错误。

无论是 POR 信号还是 PUC 信号触发的复位，都会使 MSP430 在地址 0xFFFE 处读取复位中断向量，程序从中断向量所指的地址处开始执行。触发 PUC 信号的条件中，除了 POR 产生触发 PUC 信号外，其他的都可以通过读取相应的中断向量来判断是何种原因引起的 PUC

信号，以便作出相应的处理。

POR 信号的出现会导致系统复位，并产生 PUC 信号。而 PUC 信号不会引起 POR 信号的产生。系统复位后（POR 之后）的状态为：

- RST/MIN 管脚功能被设置为复位功能。
- 所有 I/O 管脚被设置为输入。
- 外围模块被初始化，其寄存器值为相关手册上注明的默认值。
- 状态寄存器（SR）复位。
- 看门狗激活，进入工作模式。
- 程序计数器（PC）载入 0xFFFE（0xFFFE 为复位中断向量）处的地址，微处理器从此地址开始执行程序。

典型的复位电路有以下 3 种：

（1）由于 MSP430 具有上电复位功能，因此，上电后只要保持 RST/NMI（设置为复位功能）为高电平即可。通常的做法为，在 RST/NMI 管脚接 100kΩ 的上拉电阻，如图 1-5（a）所示。

（2）除了在 RST/NMI 管脚接 100kΩ 的上拉电阻外，还可以再接 0.1μF 的电容，电容的另一端接地，可以使复位更加可靠。如图 1-5（b）所示。

（3）由于 MSP430 具有极低的功耗，如果系统断电后立即上电，则系统中电容所存储的电荷来不及释放，此时系统电压不会下降到最低复位电压以下，因而 MSP430 不会产生上电复位，同时 RST/NMI 管脚上也没有足够低的电平使 MSP430 复位。这样，系统断电后立即上电，MSP430 并没有被复位。为了解决这个问题，可增加一个二极管，这样断电后储存在复位电容中的电荷就可以通过二极管释放，从而加速电容的放电。二极管的型号可取 1N4008。如图 1-5（c）所示。

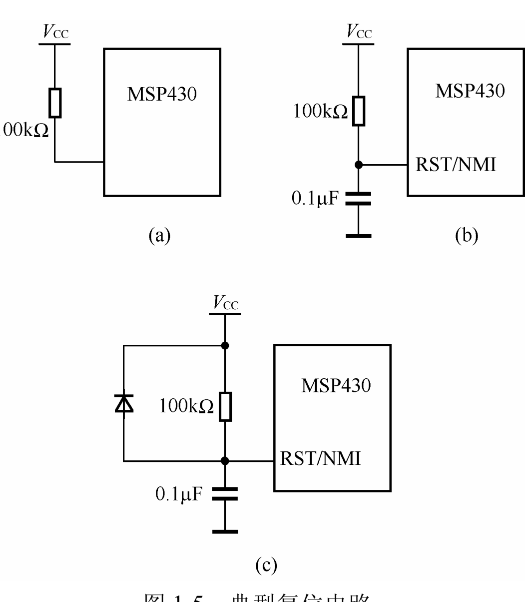


图 1-5　典型复位电路

1.4　系 统 时 钟

耗电量与微处理器的工作频率有直接的关系。MSP430 微处理器根据型号的不同最多可以使用 3 个振荡器。使用者根据需要选择合适的振荡频率，并可以在不需要时随时关闭其中一些振荡器，以节省功耗。这 3 个振荡器分别为：

（1）DCO　数控 RC 振荡器，位于芯片内部。不用时可以关闭。图 1-6 为 DCO 振荡器的逻辑控制图。

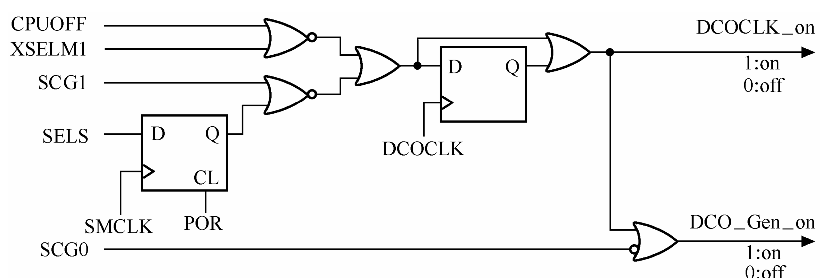


图 1-6　DCO 振荡器逻辑控制

（2）LFXT1　可以接低频振荡器，典型的如 32.768kHz 的钟表振荡器，此时振荡器不需要接负载电容。也可以接 450kHz～8MHz 的标准晶体振荡器，此时振荡器需要接负载电容。图 1-7 为 LFXT1 振荡器逻辑控制图。

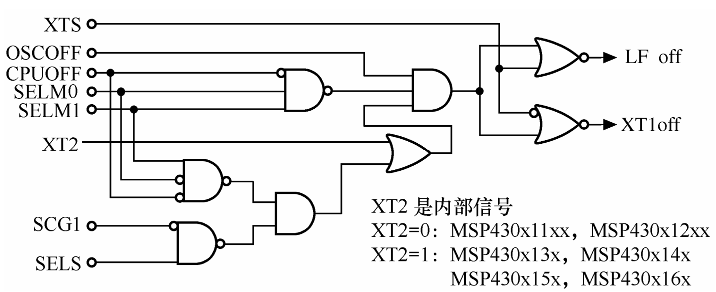


图 1-7　LFXT1 振荡器逻辑控制

（3）XT2　接 450kHz～8MHz 的标准晶体振荡器，此时振荡器要接负载电容，不用时可以关闭。图 1-8 为 XT2 振荡器逻辑控制图。

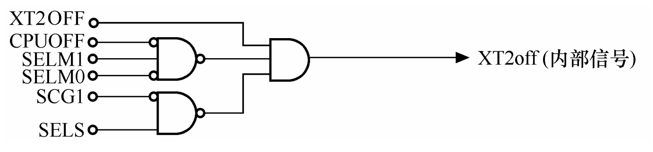


图 1-8　XT2 振荡器逻辑控制

通常低频振荡器用来降低能量消耗，例如使用电池的系统。高频振荡器用来对事件作出快速反应或者供 CPU 进行大量运算。

DCO 的振荡频率会受周围环境温度和 MSP430 工作电压的影响而产生变化，并且同一型号的芯片所产生的频率也不相同。DCO 的调节功能可以改善它的性能。DCO 的调节分为以下 3 步：

（1）选择 BCSCTL1.RSELx 确定时钟的标称频率。

（2）选择 DCOCTL.DCOx 在标称频率基础上分段粗调。

（3）选择 DCOCTL.MODx 的值进行细调。

MSP430 定义了 3 种时钟信号，分别为：

（1）MCLK　系统主时钟。除了 CPU 运算使用此时钟信号外，外围模块也可以使用。MCLK 可以选择任何一个振荡器产生的时钟信号并进行 1、2、4、8 分频作为其信号源。

（2）SMCLK　系统子时钟。外围模块可以使用，并且在使用之前可以通过各模块的寄存器实现分频。SMCLK 可以选择任何一个振荡器产生的时钟信号并进行 1、2、4、8 分频作为其信号源。

（3）ACLK　辅助时钟。外围模块可以使用，并且在使用之前可以通过各模块的寄存器实现分频。ACLK 只能由 LFXT1 进行 1、2、4、8 分频作为其信号源。

PUC 结束时，MCLK 和 SMCLK 的信号源为 DCO，DCO 的振荡频率约为 800kHz（详见相关手册）。ACLK 的信号源为 LFXT1。

MSP430 内部含有晶体振荡器失效监测电路，监测 LFXT1（工作在高频模式）和 XT2 输出的时钟信号。当时钟信号丢失大约 50μs 时，监测电路捕捉到振荡器失效，如果 MCLK 信号来自 LFXT1 或者 XT2，那么 MSP430 自动把 MCLK 的信号源切换为 DCO，这样可以保证程序继续运行。MSP430 不对工作在低频模式的 LFXT1 进行监测。

1.5　低功耗模式

低功耗模式共有 5 种，为 LPM0～LPM4（LOW POWER MODE）。CPU 运行状态称为 AM（ACTIVE MODE）模式。图 1-9 显示出各种工作模式的耗电量，可以看出，AM 耗电最多，LPM4 耗电最省，仅为 0.1μA，几乎可以忽略不计。工作电压对于功耗也有影响，电压越低功耗也越低。

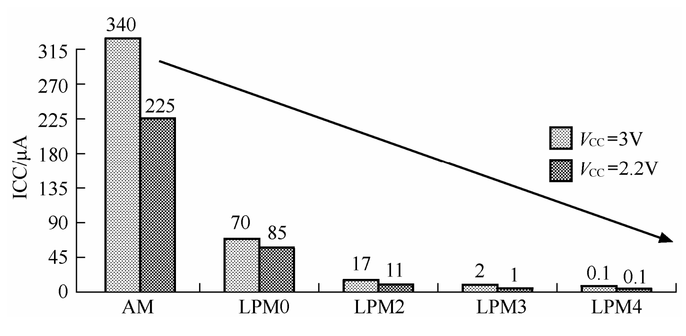


图 1-9　各模式耗电量比较

图 1-10 为 MSP430 工作模式的转换图。上电和外部复位信号产生 POR 信号，POR 信号会引起 PUC 信号。看门狗激活时，定时器溢出和安全键值错误也会引起 PUC 信号。PUC 信号结束后，MSP430 进入 AM 状态。在 AM 状态程序可以选择进入任何一种低功耗模式，然后在适当的时机，由外围模块的中断使 CPU 退出低功耗模式。

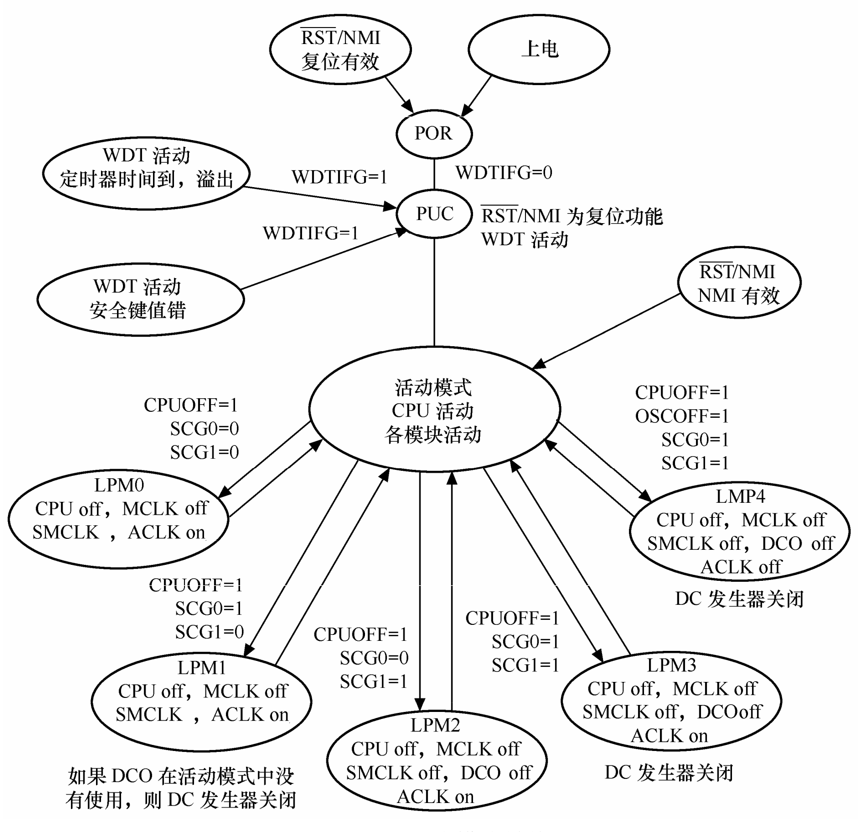


图 1-10　工作模式转换

表 1-1 显示了与选择工作模式有关的控制位。

表 1-1　　工作模式选择

SCG1	SCG0	OSCOFF	CPUOFF	模式	CPU 和时钟状态
0	0	0	0	AM	CPU 活动 MCLK 活动 SMCLK 活动 ACLK 活动
0	0	0	1	LPM0	CPU 禁止 MCLK 禁止 SMCLK 活动 ACLK 活动
0	1	0	1	LPM1	CPU 禁止 MCLK 禁止 SMCLK 活动 ACLK 活动 在 AM 模式时若 DCO 没有用作 MCLK 或者 SMCLK，则 DCO 的内部直流发生器关闭，DCO 禁止

续表

SCG1	SCG0	OSCOFF	CPUOFF	模式	CPU 和时钟状态
1	0	0	1	LPM2	CPU 禁止 MCLK 禁止 SMCLK 禁止 ACLK 活动 DCO 的内部直流发生器保留，DCO 禁止
1	1	0	1	LPM3	CPU 禁止 MCLK 禁止 SMCLK 禁止 ACLK 活动 DCO 的内部直流发生器关闭，DCO 禁止
1	1	1	1	LPM4	CPU 禁止 MCLK 禁止 SMCLK 禁止 ACLK 禁止 DCO 的内部直流发生器关闭，DCO 禁止

1.6 中　　断

中断是 MSP430 微处理器的一大特色，有效地利用中断可以简化程序和提高执行效率。MS430 的中断比较多，几乎每个外围模块都能够产生中断，为 MSP430 针对事件（外围模块产生的中断）进行的编程打下了基础。MSP430 可以在没有事件发生时进入低功耗状态，事件发生时，通过中断唤醒 CPU，事件处理完毕后，CPU 再次进入低功耗状态。由于 CPU 的运算速度和退出低功耗状态的速度很快，所以，在很多应用中，CPU 大部分时间都能够处于低功耗状态，这是 MSP430 能够如此节省电能的重要原因之一。

MSP430 的中断分为：系统复位、不可屏蔽中断、可屏蔽中断。

系统复位前面已经作过介绍，其中断向量为 0xFFFE。

不可屏蔽中断向量为 0xFFFC，产生不可屏蔽中断的原因如下：

（1）RST/NMI 管脚功能选择为 NMI 时，RST/NMI 管脚上产生一个上升沿或者下降沿（具体是上升沿还是下降沿由寄存器 WDTCTL 中的 NMIES 位决定）。NMI 中断可以用 WDTCTL 中的 NMIIE 位屏蔽。需要注意的是，当 RST/NMI 管脚功能选择为 NMI 时，不要让 RST/NMI 管脚上的信号一直保持在低电平。原因是如果发生了 PUC，则 RST/NMI 管脚的功能被初始化为复位功能，而此时它上面的信号一直保持低电平，使 CPU 一直处于复位状态，不能正常工作。

（2）振荡器失效中断允许时，振荡器失效。

（3）FLASH 存储器的非法访问中断允许时，对 FLASH 存储器进行了非法访问。

不可屏蔽中断可由各自的中断允许位禁止或打开。当一个不可屏蔽中断请求被接受时，

相应的中断允许位自动复位。退出中断程序时，如果希望中断继续有效，则必须用软件将相应中断允许位置位。

图 1-11 为不可屏蔽中断的响应流程图。响应不可屏蔽中断时，硬件会自动将 OFIE、NMIE、ACCVIE 复位。软件首先判断触发中断的中断源并复位中断标志，接着执行用户代码。退出中断之前需要置位 OFIE、NMIE、ACCVIE，以便能够再次响应中断。需要特别注意的是，置位 OFIE、NMIE、ACCVIE 之后，必须立即退出中断响应程序，否则会再次触发中断，导致中断嵌套，从而导致堆栈溢出，致使程序执行的结果无法预料。

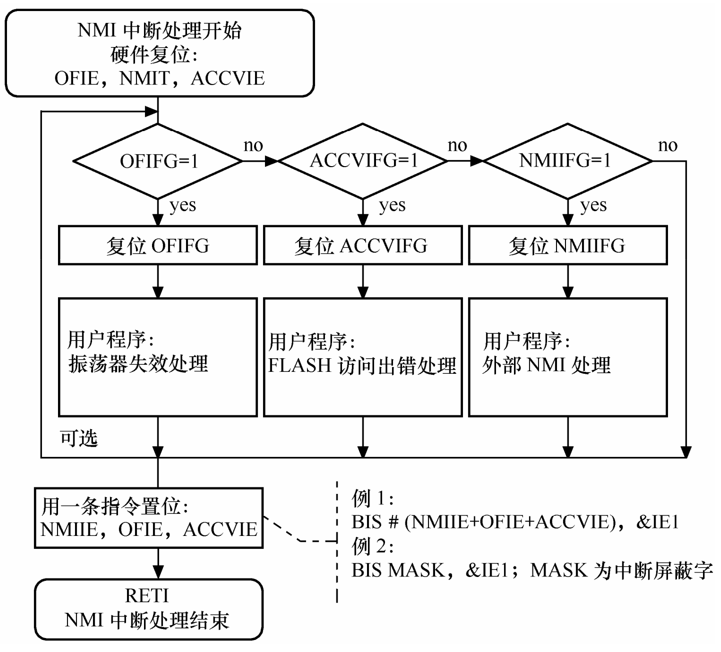


图 1-11　不可屏蔽中断响应流程

可屏蔽中断的中断源来自具有中断能力的外围模块，包括看门狗定时器（工作在定时器模式）溢出触发的中断。每一个中断都可以被自己的中断控制位屏蔽。也可以被全局中断控制位屏蔽。

多个中断请求发生时，MSP430 选择拥有最高优先级的中断响应。响应中断时，MSP430 会将不可屏蔽中断控制位 SR.GIE 复位，因此，一旦响应了中断，即使有优先级更高的可屏蔽中断出现，MSP430 也不会中断当前响应的中断，去响应另外的中断。SR.GIE 复位不影响不可屏蔽中断，所以仍可以接受不可屏蔽中断的中断请求。

中断响应的过程为：

（1）如果 CPU 处于活动状态，则完成当前指令。

（2）如果 CPU 处于低功耗状态，则退出低功耗状态。

（3）将下一条指令的 PC 值压入堆栈。

（4）将状态寄存器 SR 压入堆栈。

（5）如果有多个中断请求，则响应优先级最高的中断请求。

（6）单中断源的中断请求标志位自动复位，多中断源的标志位不变，等待软件将其复位。

（7）总中断允许位 SR.GIE 复位。SR 寄存器中的 CPUOFF、OSCOFF、SCG1、V、N、Z、C 位复位。

（8）相应的中断向量值装入 PC 寄存器，程序从此地址开始执行终端程序。

中断返回的过程为：

（1）从堆栈中恢复 SR 的值。如果响应中断前 CPU 处于低功耗模式，则仍然恢复低功耗模式。

（2）从堆栈中恢复 PC 的值。如果 CPU 不处于低功耗模式，则从此地址继续执行程序。

从中断响应和返回的过程中可以看出，如果希望在中断程序执行时仍然可以响应新的中断请求，则可以进入中断程序后将 SR.GIE 置位。这样新的中断请求出现时，MSP430 会中断当前的执行程序，响应最高优先级的中断请求，甚至包括刚被中断执行的中断程序的中断请求也可以再次被响应。但这样做一定要非常小心，对于 C 语言来说，如果不能把握中断嵌套的层次，则容易发生堆栈溢出，程序的执行必定会混乱，而 C 语言编译器是不对堆栈溢出进行检查的。

响应中断时，单中断源标志中断请求的中断请求标志位自动复位。多中断源标志则需要软件进行复位。

表 1-2、表 1-3、表 1-4 为 MSP430x1xx 系列中断向量表。表 1-5、表 1-6 为 MSP430x4xx 系列中断向量表。

表 1-2　　MSP430x11x、MSP430x11x1 系列中断向量表

中　断　源	中断标志	系统中断	中断向量	优先级
上电 外部复位 看门狗 FLASH 口令	WDTIFG KEYV	复位	0FFFEh	15 最高
NMI 晶体振荡器故障 FLASH 存储器非法访问	NMIIFG OFIFG ACCVIFG	不可屏蔽中断	0FFFCh	14
		可屏蔽中断	0FFFAh	13
		可屏蔽中断	0FFF8h	12
比较器 A	CAMPAIFG	可屏蔽中断	0FFF6h	11
看门狗定时器	WDTIFG	可屏蔽中断	0FFF4h	10
Timer_A3	CCIFG0	可屏蔽中断	0FFF2h	9
Timer_A3	CCIFG1 CCIFG2 TAIFG	可屏蔽中断	0FFF0h	8
		可屏蔽中断	0FFEEh	7
		可屏蔽中断	0FFECh	6
		可屏蔽中断	0FFEAh	5
		可屏蔽中断	0FFE8h	4

续表

中　断　源	中断标志	系统中断	中断向量	优先级
P2	P2IFG.0-7	可屏蔽中断	0FFE6h	3
P1	P1IFG.0-7	可屏蔽中断	0FFE4h	2
		可屏蔽中断	0FFE2h	1
		可屏蔽中断	0FFE0h	0 最低

表 **1-3**　　**MSP430x12x、MSP430x12x1** 系列中断向量表

中　断　源	中断标志	系统中断	中断向量	优先级
上电 外部复位 看门狗 FLASH 口令	WDTIFG KEYV	复位	0FFFEh	15 最高
NMI 晶体振荡器故障 FLASH 存储器非法访问	NMIIFG OFIFG ACCVIFG	不可屏蔽中断	0FFFCh	14
		可屏蔽中断	0FFFAh	13
		可屏蔽中断	0FFF8h	12
比较器 A	CAMPAIFG	可屏蔽中断	0FFF6h	11
看门狗定时器	WDTIFG	可屏蔽中断	0FFF4h	10
Timer_A3	CCIFG0	可屏蔽中断	0FFF2h	9
Timer_A3	CCIFG1 CCIFG2 TAIFG	可屏蔽中断	0FFF0h	8
USART0 接收	URXIFG0	可屏蔽中断	0FFEEh	7
USART0 发送	UTXIFG0	可屏蔽中断	0FFECh	6
		可屏蔽中断	0FFEAh	5
		可屏蔽中断	0FFE8h	4
P2	P2IFG.0-7	可屏蔽中断	0FFE6h	3
P1	P1IFG.0-7	可屏蔽中断	0FFE4h	2
		可屏蔽中断	0FFE2h	1
		可屏蔽中断	0FFE0h	0 最低

表 1-4　　MSP430x13、MSP430x14 系列中断向量表

中　断　源	中断标志	系统中断	中断向量	优先级
上电 外部复位 看门狗 FLASH 口令	WDTIFG KEYV	复位	0FFFEh	15 最高
NMI 晶体振荡器故障 FLASH 存储器非法访问	NMIIFG OFIFG ACCVIFG	不可屏蔽中断	0FFFCh	14
Timer_B7	BCCIFG0	可屏蔽中断	0FFFAh	13
Timer_B7	BCCIFG1 TBIFG	可屏蔽中断	0FFF8h	12
比较器 A	CAMPAIFG	可屏蔽中断	0FFF6h	11
看门狗定时器	WDTIFG	可屏蔽中断	0FFF4h	10
USART0 接收	URXIFG0	可屏蔽中断	0FFF2h	9
USART0 发送	UTXIFG0	可屏蔽中断	0FFF0h	8
ADC	ADCIFG	可屏蔽中断	0FFEEh	7
Timer_A3	CCIFG0	可屏蔽中断	0FFECh	6
Timer_A3	CCIFG1 CCIFG2 TAIFG	可屏蔽中断	0FFEAh	5
P1	P1IFG.0-7	可屏蔽中断	0FFE8h	4
USART1 接收	URXIFG1	可屏蔽中断	0FFE6h	3
USART0 发送	UTXIFG1	可屏蔽中断	0FFE4h	2
P2	P2IFG.0-7	可屏蔽中断	0FFE2h	1
		可屏蔽中断	0FFE0h	0 最低

表 1-5　　MSP430x41x 系列中断向量表

中　断　源	中断标志	系统中断	中断向量	优先级
上电 外部复位 看门狗 FLASH 口令	WDTIFG KEYV	复位	0FFFEh	15 最高
NMI 晶体振荡器故障 FLASH 存储器非法访问	NMIIFG OFIFG ACCVIFG	不可屏蔽中断	0FFFCh	14
Timer_A5	TA1CCR0 CCIFG	可屏蔽中断	0FFFAh	13

续表

中　断　源	中断标志	系统中断	中断向量	优先级
Timer_ A5	TA1CCR1～TA1CCR4 CCIFGs TA1CTL TAIFG	可屏蔽中断	0FFF8h	12
比较器 A	CAMPAIFG	可屏蔽中断	0FFF6h	11
看门狗定时器	WDTIFG	可屏蔽中断	0FFF4h	10
		可屏蔽中断	0FFF2h	9
		可屏蔽中断	0FFF0h	8
		可屏蔽中断	0FFEEh	7
Timer_A3 /Timer0_A3	TACCR0/TA0CCR0 CCIFG	可屏蔽中断	0FFECh	6
Timer_A3/Timer0_A3	TACCR1/TA0CCR1 TACCR2/TA0CCR2 CCIFGs TACLT/TA0CTL TAIFG	可屏蔽中断	0FFEAh	5
P1	P1IFG.0-7	可屏蔽中断	0FFE8h	4
		可屏蔽中断	0FFE6h	3
		可屏蔽中断	0FFE4h	2
P2	P2IFG.0-7	可屏蔽中断	0FFE2h	1
Basic Timer1	BTIFG	可屏蔽中断	0FFE0h	0 最低

表 1-6　　MSP430x43x、MSP430x44x 系列中断向量表

中　断　源	中断标志	系统中断	中断向量	优先级
上电 外部复位 看门狗 FLASH 口令	WDTIFG KEYV	复位	0FFFEh	15 最高
NMI 晶体振荡器故障 FLASH 存储器非法访问	NMIIFG OFIFG ACCVIFG	不可屏蔽中断	0FFFCh	14
Timer_B7	TBCCR0 CCIFG	可屏蔽中断	0FFFAh	13
Timer_ B7	TBCCR1～TBCCR6 CCIFGs TBIFG	可屏蔽中断	0FFF8h	12
比较器 A	CAMPAIFG	可屏蔽中断	0FFF6h	11
看门狗定时器	WDTIFG	可屏蔽中断	0FFF4h	10
USART0 接收	URXIFG0	可屏蔽中断	0FFF2h	9

续表

中　断　源	中断标志	系统中断	中断向量	优先级
USART0 发送	UTXIFG0	可屏蔽中断	0FFF0h	8
ADC12	ADC12IFG	可屏蔽中断	0FFEEh	7
Timer_A3	TACCR0 CCIFG	可屏蔽中断	0FFECh	6
Timer_A3	TACCR1 CCIFG TACCR2 CCIFG	可屏蔽中断	0FFEAh	5
P1	P1IFG.0-7	可屏蔽中断	0FFE8h	4
USART1 接收	URXIFG1	可屏蔽中断	0FFE6h	3
USART0 发送	UTXIFG1	可屏蔽中断	0FFE4h	2
P2	P2IFG.0-7	可屏蔽中断	0FFE2h	1
Basic Timer1	BTIFG	可屏蔽中断	0FFE0h	0 最低

1.7 外 围 模 块

MSP430 中有多种外围模块，不同型号的 CPU 含有的模块也有所不同。每个模块完成一定的功能。每个模块拥有各自的控制寄存器，通过它控制其工作模式。

大部分模块都有开关控制位，在不使用时可以被关闭，以节省功耗。

大部分模块在 CPU 处于低功耗模式的时候可以独立运行，并可以产生中断，从而将处于低功耗模式的 CPU 唤醒，执行某一段代码。

本书后面所举的例子包含了对各种模块的操作，对各模块也进行了必要的解释，但本书不准备作为一本数据手册提供给读者，所以关于外围模块中寄存器的详细用法，请参考 MSP430 的相关数据手册。

第 2 章　C 语言基础知识

C 语言所具备的优秀特性使它成为计算机领域使用最为广泛的高级语言之一。高级语言最终需要被翻译成微处理器能够识别的机器码才能够执行，这个翻译的过程叫做编译。不同的处理器执行不同的指令集，因此，同样一段用 C 语言编写的程序，必须用不同的 C 编译器来编译，才能够被不同的处理器接受。不同的处理器有不同的特点，为了发挥处理器各自的性能优势，相应的 C 编译器除了执行 C 语言标准（ANSI 标准）的规定外，还做了一些扩展。符合 C 语言标准的那部分程序可以被另一种处理器的 C 编译器编译，从而在另一种处理器上执行，这个过程称为移植。扩展的那部分程序则不可以被移植。如果希望程序有很好的可移植性，则编程时应该尽量少使用扩展部分，不过这样却会丧失部分发挥处理器性能优势的机会。

本书不是一本介绍 C 语言的书，因此，不准备详细介绍 C 语言的细节。本章将简要介绍 C 语言中与 MSP430 有关的部分。

2.1 标识符与关键字

2.1.1 标识符

标识符用来定义变量、函数、标号以及用户定义对象的名称。标识符由字母和数字组成，但第一个字符必须是字母或下划线。C 语言中大、小写字母被认为是不同的字符。如：at、AT、At 就是 3 种不同的标识符。标识符与保留的关键字都不能与库函数名和自定义的函数名相同。

2.1.2 关键字

关键字是一种已经被编译器定义过的标识符，具有特定的含义，因此也称作保留字，意思是不可以再被用户定义了。表 2-1 列出了 ANSI C 定义的标准关键字。

表 2-1 ANSI C 标准关键字

关 键 字	用 途	说 明
auto	变量存储类型	声明变量为局部变量。省略时，变量默认为此类型
break	程序控制语句	退出当前循环体
case	程序控制语句	switch 结构的选择语句
char	数据类型	单字节整型或字符型数据类型
const	变量存储类型	声明变量为只读，内容在程序执行过程中不可更改
continue	程序控制语句	转向下一次循环
default	程序控制语句	switch 结构中的默认选择项
do	程序控制语句	构成 do...while 循环结构
double	数据类型	双精度浮点数
else	程序控制语句	构成 if...else 结构
enum	数据类型	枚举
extern	变量存储类型	外部全局变量
float	数据类型	单精度浮点数
for	程序控制语句	构成 for 循环结构
goto	程序控制语句	goto 跳转语句
if	程序控制语句	条件跳转语句
int	数据类型	基本整型数
long	数据类型	长整型数
register	变量存储类型	变量被分配到 CPU 内部寄存器
return	程序控制语句	函数返回语句

续表

关　键　字	用　　途	说　　明
short	数据类型	短整型数
signed	数据类型	有符号数
sizeof	运算符	计算表达式或者数据类型的字节数
static	变量存储类型	静态变量
struct	数据类型	结构数据类型
switch	程序控制语句	switch 结构语句
typedef	数据类型	定义新的数据类型
union	数据类型	联合数据类型
unsigned	数据类型	无符号数
void	数据类型	无类型数据
volatile	数据类型	在程序执行中可改变数据类型
while	程序控制语句	条件判断，构成 while 或 do...while 结构

另外，针对不同的微处理器，还有一些扩展的关键字，扩展关键字将在后续章节中介绍。

2.2　数 据 类 型

数据类型有基本型、构造型、指针型。

2.2.1　基本型

基本型数据类型如表 2-2 所示。

表 2-2　　基本型数据类型

数据类型	位　　数	范　　围
char	8bit	0～255
signed char	8bit	−128～127
unsigned char	8bit	0～255
short	16bit	−32768～32767
unsigned short	16bit	0～65535
int	16bit	−32768～32767
unsigned int	16bit	0～65535
long	32bit	-2^{31}～$2^{31}-1$
unsigned long	16bit	0～$2^{32}-1$
long long	64bit	-2^{63}～$2^{63}-1$

续表

数据类型	位　　数	范　　围
unsigned long long	64bit	0 ～ 2^{64}−1
float	32bit	±1.18E−38～±3.39E+38
double*	32bit	±1.18E−38～±3.39E+38
double*	64bit	±2.23E−308～±1.79E+308

*依赖编译器。

2.2.2　构造型

构造型是由基本数据类型组成的集合体。有数组、结构、联合、枚举 4 种形式。在对单片机的编程中，使用得最多的是数组。枚举用于定义一组常量，增加程序的可读性。由于包含结构的程序运行效率低，因此结构很少使用。联合的应用范围有限，所以也很少用到。

1．数组

数组是同一种数据类型的集合体，如 unsigned char Da[5]，表示 5 个 unsigned char 类型的数据组成了 Moon 数组。

使用时通过下标来存取数组中的数据，如 Da[2]=100 的含义为为数组 Da 中的第三个元素赋值 100。数组的第一个元素的下标为 0，Da[0]为第一个元素。按照数组 Da 的定义，数组 Da 只有 5 个元素，所以，其最后一个元素为 Da[4]。

2．结构

结构是可以有多种数据类型组成的混合数据集合体。使用时首先定义结构类型，如：

```
struct Moon
{
    unsigned char Data,  //无符号 8 位字符类型
    unsigned int  Year   //无符号 16 位整数类型
};
```

以上代码定义了结构类型 Moon，其中 Data 和 Year 的数据类型不同。struct 为定义结构的关键字。

使用结构类型时首先定义结构变量。如：

```
struct Moon Mo;  //定义结构变量 Mo 为 Moon 类型
```

存取结构变量时，通过结构类型中的变量名来进行，结构变量的名称和结构中的变量名中间用“.”隔开，如：

```
Mo.Data=15;   //Mo 中的 Data 元素被赋值为 15
Mo.Year=3;    //Mo 中的 Year 元素被赋值为 3
```

3．联合

联合是可以有多种数据类型组成的混合数据集合体。联合与结构的区别在于放置在内存中的数据的存放方式不同。结构中的每个成员是顺序存放的，都拥有自己的内存单元，结构

所占用的内存长度为所有成员占用内存长度的和。联合的内存长度为占用内存最多的成员的长度，所有的成员都放在同一内存地址，每次只能存放成员中的一种，而且有效的只能是最后一次放置的成员。

使用时首先定义联合类型，如：

```
union Yeah
{
  unsigned char da0;  //无符号 8 位字符类型
  unsigned int da1;   //无符号 16 位整数类型
};
```

代码中定义了联合类型 Yeah，其中 da0 和 da1 的数据类型不同。union 为定义联合的关键字。

使用联合类型时首先定义联合变量。如：

```
union Yeah Ye;  //定义联合变量 Ye 为 Yeah 类型
```

存取联合变量时，通过联合类型中的变量名来进行，联合变量的名称和联合中的变量名中间用“.”隔开，如：

```
Ye.da0=15;    //Ye 中的 da0 元素被赋值为 15
Ye.da1=3;     //Ye 中的 da1 元素被赋值为 3
```

4．枚举

枚举就是列举一个变量所有的取值。这样做的好处是增加程序的可读性。

举例：

```
menu Week
{
    Mon,Tues,Wed,Thurs,Fri,Sta,Sun
}
```

编译器按照顺序为它们取值，Mon=0，Tues=1……Sun=6。也可以强制定义枚举的取值。

举例：

```
menu Week
{
    Mon,Tues=5,Wed,Thurs,Fri,Sta,Sun
}
```

程序运行的结果为：Mon=0，Tues=5，Wed=6，Thurs=7……

2.2.3　指针型

指针型用来存储地址，可以是变量的地址，也可以是函数的地址。指针的应用非常灵活，能够提高程序的效率。

举例：

```
unsigned char Sun;
unsigned char *Moon=Sun;
```

Moon 中存放了变量 Sun 的地址。

2.3 运 算 符

运算符是进行基本运算的符号。

1．赋值运算符

赋值运算符为“=”，作用是把等号右边的值赋给等号左边的变量。
举例：

```
Moon=100;
```

变量 Moon 的值为 100。
除了“=”，还有复合赋值运算符：+=、−=、*=、/=、%=、<<=、>>=、&=、|=、^=、～=。
举例：

```
q0 += 5;        //q0=q0+5
q0 -= 5;        //q0=q0-5
q0 *= 5;        //q0=q0*5
q0 /= 5;        //q0=q0/5
q0 %= 5;        //q0=q0%5
q0 <<= 5;       //q0=q0<<5
q0 >>= 5;       //q0=q0>>5
q0 &= 5;        //q0=q0&5
q0 |= 5;        //q0=q0|5
q0 ^= 5;        //q0=q0^5
q0 ～= 5;       //q0=q0～5
```

2．算术运算符

算术运算符有+、−、*、/、%、−−、++、−−，下面分别加以解释：

\+ 加法运算。

− 减法运算。

* 乘法运算。

/ 除法运算，结果只保留整数，小数部分被舍去。

% 取余运算，要求两个运算对象都为整数，如 9%8，结果为 1。

++ 增量运算，自加 1。例如 Moon++等价于 Moon=Moon+1。

−− 减量运算，自减 1。例如 Moon−−等价于 Moon=Moon−1。

3．关系运算符

关系运算符用于判别两个值之间的关系，判别的结果只有真和假两种结果。判别结果如

果为真，则表示为 1，判别结果为假则表示为 0。关系运算符有以下几种：

\>　大于

\>=　大于等于

<　小于

<=　小于等于

==　等于

!=　不等于

4．逻辑运算符

常用的逻辑运算符有以下几种：

&&　逻辑与，相与的两个表达式同时为真，则输出真，否则输出假。

||　逻辑或，相或的两个表达式同时为假，则输出假，否则输出真。

!　逻辑非，表达式为真，则输出假，否则输出真。

5．位运算符

按位运算符应用于字节对字节或者字对字的操作，不能应用于 float、double、void 和构造类型。位运算符有以下几种：

&　按位与

|　按位或

^　按位异或

～　按位取反

\>>　位右移

<<　位左移

6．地址运算符

地址运算符与指针有密切的关系。地址运算符有以下几种：

&　取地址运算符

*　取值运算符

举例：

```
unsigned char *Moon,Day=10,See;
Moon=&Day;    //指针 Moon 中保存了变量 Day 的地址,*Moon 的值为 10
See=*Moon;    //变量 See 取得了 Moon 中保存的地址中所保存的值,See=10
```

7．条件运算符

条件运算符的一般形式为：

```
表达式 1?表达式 2:表达式 3
```

运算的含义是：先判断表达式 1 的值，如果为真，则把表达 2 的值作为运算结果，否则

把表达式 3 的值作为运算结果。

8．逗号运算符

逗号运算符把几个表达式串在一起，各表达式从左至右运算，最后一个表达式的结果作为最终的运算结果。

举例：

```
Moon=（Day=10,Year=5,Day+Year）;
```

运算结果为 Moon=15。

9．强制转换运算符

算术运算中不同的数据类型不可避免地会进行混合运算，编译器要求不同的数据类型必须转换为同一种数据类型才能够进行运算，所以，运算时不同类型的数据必须转换为同一数据类型。有两种数据类型转换方式：隐式转换和显式转换。隐式转换是在没有进行强制转换时由编译器进行的自动类型转换。转换遵循以下规则：

（1）所有 char 型转换成 int 型。

（2）运算符两边的两个不同的数据类型的操作数按照以下规则转换：如果一个操作数是 float 型，则另外一个也转换成 float 型；如果一个操作数是 long 型，则另外一个也转换成 long 型；如果一个操作数是 unsigned 型，则另外一个也转换成 unsigned 型。

（3）对变量赋值时将“=”右边表达式的类型转换为“=”左边变量的数据类型。例如：把整型（16 位）的数据赋给字符类型（8 位）的变量， 整型数据的高 8 位会丢失。把浮点型的数据赋给整型变量，小数部分会丢失。

（4）能够进行隐式转换的只有基本数据类型，即 char、int、long、float。其余的数据类型不能进行隐式转换。

如果需要将某种数据类型转换成另一特定的数据类型，就要进行强制类型转换，即显式转换。例如，从数据存储器中的某一地址通过指针读写数据，表示地址的变量是整数，可以通过强制类型转换把地址变量转换为指针类型。强制转换运算符的一般形式为：

```
（数据类型）表达式
```

括号中的数据类型是要转换成为的数据类型。强制转换并不改变变量原来的数据类型，只是输出一个中间的具有新数据类型的值。如：

```
float fDay;
unsigned int Moon;
Moon=（unsigned int）fDay;
```

fDay 的数据类型不会改变，仍然是浮点数据类型，Moon 的值为 fDay 的整数部分，前提是 fDay 的整数部分的值不能大于整数类型的数据范围 65 535，否则结果仍然不正确。

当多个运算符出现在一个表达式中时，运算的先后次序按照运算符的优先级来进行，运算符的优先级如表 2-3 所示。

表 **2-3**　　运算符优先级

优　先　级	符　　号	含　　义	运算对象个数	结合方向
1	( ) [ ] −> .	圆括号 下标运算符 指向结构成员运算符 结构成员运算符		自左向右
2	! ～ ++ − − − (类型) * & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 取地址运算符 长度运算符	单操作数	自右向左
3	* / %	乘法运算符 除法运算符 求余运算符	双操作数	自左向右
4	+ −	加法运算符 减法运算符	双操作数	自左向右
5	<< >>	左移运算符 右移运算符	双操作数	自左向右
6	<、<=、>、>=	关系运算符	双操作数	自左向右
7	== !=	等于运算符 不等于运算符	双操作数	自左向右
8	&	按位与运算符	双操作数	自左向右
9	^	按位异或运算符	双操作数	自左向右
10	|	按位或运算符	双操作数	自左向右
11	&&	逻辑与运算符	双操作数	自左向右
12	||	逻辑或运算符	双操作数	自左向右
13	?:	条件运算符	三操作数	自右向左
14	=、+=、−=、*=、/=、%=、<<=、>>=、&=、|=、^=、～=	赋值运算符	双操作数	自右向左
15	，	逗号运算符		自左向右

2.4 函　　数

函数是具有一定功能的程序模块。一个用 C 语言编写的程序实际上就是由多个函数组成的。函数的一般形式是：

```
返回值数据类型  函数名（参数数据类型 形式参数,...,参数数据类型 形式参数）
{
    变量说明
    语句
    返回值
}
```

举例：

```
int AddIt（int x,int y）    //函数定义，x、y 为形参
{
     int z;
     z=x+y;
     return z;
}

void main()
{
     int iq0=10,iq1=30,iq3;
     iq3=AddIt(iq0,iq1);    //函数调用，iq0、iq1 为实参
}
```

函数必须先定义，然后才可以调用。在定义函数时声明的参数叫做“形式参数”，简称形参。在调用函数时，被调用函数的参数成为“实际参数”，简称实参。实参必须与形参的类型保持一致。

函数的参数传递方式分为值传递和地址传递。

当参数是基本类型变量时，主调函数传递给被调函数参数值的过程，就是为被调函数重新分配内存，将参数的值复制到内存中。函数执行过程中对参数的任何修改都不会改变主调函数中原来的值。函数调用结束时，函数分配的内存被释放，复制的参数也就不存在了。

地址传递时，主调函数同样传递参数的值给被调函数。不过由于参数是指向某个内存的地址，此内存不是函数被调用时分配的，因此也不会在函数结束时释放。所以，被调函数对此地址指向的内存的数值进行的修改在函数结束后仍然有效。

如果定义的函数没有参数，可以没有形式参数表，但圆括号不能省略。

返回值数据类型定义了函数运行结束时返回（return）的值的数据类型。如果没有返回值，那么最后可以没有 return 语句，对应的返回值数据类型要为 void。

main 函数是程序的起点，是 C 语言的规定。用户所编写的程序是从 main 函数的第一句开始执行的。

2.5　数　　组

数组是由同一数据类型组成的集合体。数组在使用前必须先声明，声明的一般形式为：

```
数据类型 数组名[常量表达式]
```

数据类型说明了数组中的各个元素的类型。数组名代表整个数组。常量表达式说明了数组中元素的个数。

举例：

```
unsigned char Moon[5];
```

以上代码的含义为名字叫作 Moon 的数组中保存了 5 个元素，元素的数据类型为 unsigned char。

使用数组的方法是通过下标确定元素，规定数组的下标从 0 开始。如上例中 Moon[0]代表数组 Moon 的第一个元素，Moon[4]代表最后一个即第 5 个元素。

数组可以是多维的，只要在声明数组时增加[常量表达式]就可以了。如 Moon[5][6]代表一个 5 行 6 列的矩阵。

数组在声明的时候可以赋初值，也可以不赋。

举例：

```
unsigned char Moon[5]={0,10,2,30,4};
```

以上代码表示 Moon 中的 5 个元素都被赋了初值。

2.6　指　　针

指针的特点是灵活、高效。指针表示的是一个地址，保存指针的变量称为指针变量。指针变量的一般形式为：

```
数据类型 *指针变量名;
```

举例：

```
unsigned int iq0=0;      //声明一个无符号整数的变量 iq0 且赋初值为 0
unsigned char Moon0=5;   //声明一个无符号字符的变量 Moon0 且赋初值为 5
unsigned int Moon1=500; //声明一个无符号整数的变量 Moon1 且赋初值为 500
unsigned char *pq0;      //声明一个无符号字符的指针变量 pq0
unsigned int *piq0;      //声明一个无符号整数的指针变量 piq0

pq0=&Moon0;         //pq0 指向变量 Moon0，即取得了 Moon0 的地址
piq0=&Moon1;        //piq0 指向变量 Moon1，即取得了 Moon1 的地址
```

```
iq0=*pq0;            //iq0 的值不再是 0，而是 5
iq0=*piq0;           //iq0 的值不再是 5，而是 500
*pq0=100;            //Moon0 的值不再是 5，而是 100
*piq0=0;             //Moon1 的值不再是 500，而是 0
```

这个例子使用了前面提到的地址运算符*和&。

指针可以进行几种基本运算，以上例的指针为例：

```
pq0++;        //指针+1，由于是 unsigned char 型指针，所以，pq0 内的值+1
piq0++;       //指针+1，由于是 unsigned int 型指针，所以，piq0 内的值+2
(*pq0)++;     //pq0 所指的变量的值+1
*(pq0+1);     //pq0 内的值+1，然后再取 pq0 所指的变量的值
```

可以用指针对数组进行操作：

```
unsigned char Moon[5];
unsigned char *pq0=Moon;    //指针指向数组 Moon
```

C 语言规定数组的名字为数组第一个元素的地址，上例中下列几种情况左边表达式与右边表达式的值是等价的：

```
pq0 Moon
*pq0  Moon[0]
*(pq0+1) Moon[1]
```

也可以直接获取数组某个元素的地址，赋给指针：

```
pq0=&Moon[4];   //pq0 的值是 Moon[4]的地址
pq0--;          //pq0 的值是 Moon[3]的地址
```

指向数组的指针可以超出数组的边界进行运算，而编译器不会提出警告。这需要非常小心，因为有可能破坏程序中其他部分所使用的变量或者读取到错误的值，造成程序运行混乱。这是使用指针不利的一面。

指针同样也可以对结构和联合进行操作。操作的方法与数组类似。

举例：

```
struct Li //定义结构 Li
{
    unsigned char Ka;
    float Dae;
};
struct Li sst,*psst;  //声明结构 Li 的实例 sst 和结构 Li 类型的指针 psst

psst=&sst;            //psst 指向 sst
(*psst).Ka=100;       //为结构 sst 中的变量 Ka 赋值 100
psst->Ka=100;         //为结构 sst 中的变量 Ka 赋值 100
```

(*psst).Ka=100 与 psst->Ka=100 的含义完全相同。“->”的含义为通过指针获取结构中的变量。

数组和结构、联合的成员也可以是指针。

2.7 位 运 算

位运算通过位运算符进行。

举例：

```
unsigned char q0=0x35,q1=0x47;
q0=q0<<3;      //q0 的值左移 3 位
q1=q1>>5;      //q1 的值右移 5 位
```

执行结果为：q0=0x1A8　q1=0x2

有些处理器寄存器中的每一位也有其位地址，所以可以对寄存器中的位单独操作，如 MCS51 系列微处理器。MSP430 没有位地址，所以，只能采用位屏蔽的方法对寄存器中某一位进行操作，而避免改变同一寄存器中其他位的内容。如：

```
P1DIR=P1DIR | 0x1;
P1OUT=P1OUT & 0xFE;
```

P1DIR 为 MSP430 的 P1 口的方向寄存器，P1OUT 为 P1 口的输出寄存器。以上代码中第一行语句将 P1 口的最低位（一共 8 位）输入输出方向改为输出，其他位方向不变。第二行语句让最低位输出 0，其他位不变。

2.8 存 储 类 型

变量和函数都有其有效区域，称为作用域。程序如果使用范围之外的变量和函数，则编译器会报错。

2.8.1　变量

变量包括局部变量、全局变量、外部变量、静态变量。

- 局部变量　在函数内部定义的变量程称为局部变量，它只在函数内有效，退出函数时所分配的内存被释放。
- 全局变量　在程序开始执行的时候就被分配了内存，一直保持到程序结束，可以被任何模块调用。在函数之外定义的变量为全局变量。
- 外部变量　在其他文件中定义但在本文件中使用的变量，称为外部变量，用 extern 标识。
- 静态变量　寿命相当于全局变量，但只允许在定义的函数内使用的变量称为静态变量，静态变量在退出定义的函数时，其值仍然保留，直至下一次进入定义它的函数中，执行程序对它进行修改。

变量的声明形式为：

```
作用域类型 数据类型 变量名;
```

举例：

```
extern int  Moon;
```

变量的作用域如表 2-4 所示。

表 2-4 变量作用域

类　　型	作用域和寿命	说　　明
auto	局部变量	变量默认类型为 auto
register	局部变量，只是提示编译器将此变量分配在 CPU 的寄存器中	寄存器中变量的使用速度比较快，但寄存器个数有限，最终还是由编译器决定是否为变量分配寄存器
extern	全局变量	外部变量
static	全局变量	静态变量

2.8.2　函数

函数的存储类型有 static 和 extern 两种。声明形式为：

```
作用域类型 返回数据类型 函数名(参数表);
```

声明为 static 的函数称为内部函数或者静态函数。静态函数只能在定义此函数的文件中被调用，而不能被其他文件中的函数调用。

extern 称为外部函数。除非被声明为静态函数，函数都可以在其他文件中被调用。编译器函数默认为外部函数，因此，extern 通常可以省略。

2.9　预处理功能

编译器在编译程序之前先进行预处理。预处理包括宏定义、条件编译和文件包含。

2.9.1　宏定义

宏定义的命令为#define。如：

```
#define CAR 100
```

以上代码的含义是定义 CAR 为 100，执行预处理时编译器将有效范围内所有的 CAR 替换为 100。这样做的好处是增加程序的可读性和可维护性。

也可以使用带参数的宏，例如：

```
#define SUM(x,y)     (x+y)
Moon=SUM(1,2)*2;
```

以上宏定义在预处理时会将 SUM（1，2）替换为（1+2），成为：

```
Moon=(1+2)*2;
```

注意括号的重要性，如果宏写为：

```
#define SUM(x,y)  x+y
```

则替换的结果为：

```
Moon=1+2*2;
```

即使运算的顺序发生了变化，结果也会不同。

2.9.2　条件编译

由于某些原因，编译时会想要得到不同的结果。比如某一系列的产品所使用的程序，绝大部分都是相同的，只是由于型号不同需要在某处根据不同的型号做不同的处理，此时就需要条件编译，由编译器根据所选的型号（条件编译的条件）自动选择所需要的程序段。

条件编译的命令有#if、#ifdef、#ifndef、#else、#elif、#endif。

格式 1：

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

判断是否定义了标识符，如果定义了标识符，则选择程序段 1 编译，忽略程序段 2；否则选择程序段 2，忽略程序段 1。

格式 2：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

判断是否定义了标识符，如果未定义，则选择程序段 1 编译，忽略程序段 2；否则选择程序段 2，忽略程序段 1。

格式 3：

```
#if 常量表达式 1
    程序段 1
#elif 常量表达式 2
    程序段 2
...
#else
    程序段 n
#endif
```

以上形式的代码顺序判断哪一个常量表达式为真，并编译相应的程序段，忽略其他的程序段。

2.9.3　文件包含

由于各种原因，不会将一个程序的所有部分都放置在一个文件中。另外，C 语言和编译器也定义了很多常量、数据结构以及函数。编译器在编译的时候必须要知道一个文件中的程

序使用到的这些内容在什么文件中作出了定义，一般编写每一个 c 文件（程序文件以.c 为后缀）都会再编写一个头文件（.h 为后缀），c 文件中用到的常量、数据结构、函数都在头文件中声明。需要调用此 c 文件的内容的文件只需要用#include<***.h>或者#include“***.h”语句调用就可以了。<>与“”的作用一样，不过习惯上在包含编译器定义的库时用<>，包含用户自己定义的文件时用“”。

在头文件中，往往还会看到一种宏指令，如在 msp430x14x.h 中有：

```
#ifndef __msp430x14x
#define __msp430x14x
...
#endif
```

其中 msp430x14x 是这个头文件的文件名，含义是如果未定义__msp430x14x，则定义__msp430x14x，并编译随后的内容。以后编译器遇到其他的文件也包含此头文件时，由于__msp430x14x 已经被定义过了，不符合#ifndef __msp430x14x 条件，所以不会再次编译头文件的内容。它的作用是防止同一个头文件被重复包含，从而避免造成头文件中的内容重复定义。编译器会对重复定义提出警告或者报错。

2.10 程序的基本结构

程序有 3 种基本结构：顺序、选择、循环。

2.10.1 顺序结构

在顺序结构中语句依次顺序执行。顺序结构是程序的基本结构。

2.10.2 选择结构

选择结构由判断和分支两部分组成。选择结构有两种。

1．if

有 3 种形式：

（1）形式为：

```
if (表达式)
{
    程序段
}
```

如果表达式为真则执行程序段，否则跳过程序段执行本选择结构之后的程序。

（2）形式为：

```
if (表达式)
{
```

```
    程序段 1
}
else
{
    程序段 2
}
```

如果表达式为真则执行程序段 1，否则执行程序段 2。

（3）形式为：

```
if (表达式 1)
{
    程序段 1
}
else if (表达式 2)
{
    程序段 2
}
...
else
{
    程序段 n
}
```

如果表达式 1 为真则执行程序段 1，然后执行本选择结构之后的程序；如果表达式 2 为真，则执行程序段 2，然后执行本结构后的程序。如此依次判断表达式是否为真，如果为真则执行相应的程序段，然后执行本结构后的程序。

2．switch

其形式为：

```
switch (表达式)
{
    case 常量 1:
                程序段 1
    break;
    case 常量 2:
                程序段 2
    break;
    ...
    default:
    程序段 default
}
```

计算表达式的值，判断与哪一个常量相等，则执行哪一段程序，然后执行本结构后的程序。如果没有一个值与表达式的值相等，则执行程序段 default。break 指令的含义为跳出选择体，如果省去，则在执行完某程序段后就不会跳出，而是继续进行下面的判断。在多个不同的常量都使表达式执行同一段程序时可以写为：

```
case 常量 1:
case 常量 2:
            程序段
break;
```

2.10.3 循环

1．for

形式为：

```
for(初值设定表达式;循环条件表达式;更新表达式)
{
    程序段
}
```

执行过程如图 2-1 所示。

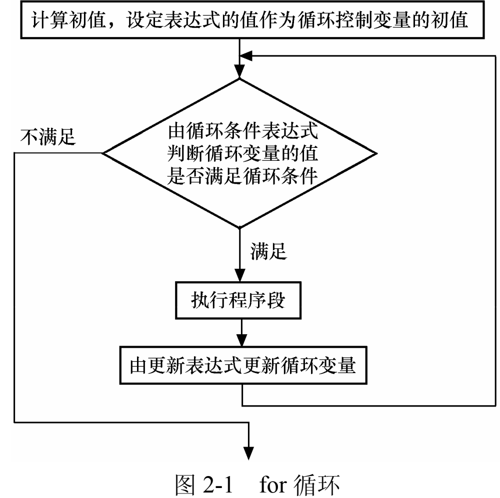


图 2-1　for 循环

2．while

形式为：

```
while（条件表达式）
{
    程序段
}
```

执行过程如图 2-2 所示。

3．Do…while

形式为：

```
do
{
    程序段
}
while（条件表达式）;
```

执行过程如图 2-3 所示。

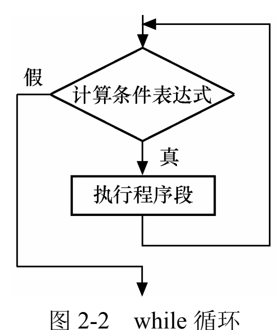


图 2-2　while 循环

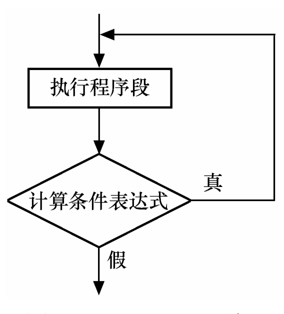


图 2-3　do…while 循环

2.10.4　跳转

1．goto

形式为：

```
goto 语句标号;
```

语句标号是一个带冒号“:”的标识符。执行 goto 语句将使程序无条件跳转到语句标号标示的那一句程序。goto 语句可以从循环结构中跳出，但不能跳入循环结构。结构化编程不推荐使用此语句，但在某些情况下，使用它可以使程序表现得更为简洁。

举例：

```
tt:
  x=1;
  ...
goto:tt;
```

2．continue

形式为：

```
continue;
```

一般用在循环结构中，功能是忽略循环结构中本语句之后尚未执行的程序，直接进入下一个循环周期。是否退出循环体仍需由循环条件来确定。

举例：

```
for(q0=0;q0<100;q0++)
{
       if(q1[q0]==50)
           continue;
       else
           q1[q0]=0;
}
```

结果：依次判断数组 q1 中下标从 0～99 的元素的值，如果值等于 50，则忽略（continue），否则这些元素被赋值为 0。

3．break

形式为：

```
break;
```

在前面遇到过此语句。break 语句一般用在循环结构中，其作用是无条件跳出循环结构。在 switch 结构中的 break 即是此含义。

4．return

形式为：

```
return（表达式）;
```

或：

```
return;
```

return 语句的作用为中止函数的执行，程序返回到调用函数的位置的下一句执行。对于需要返回值的函数应返回一个值，所以 return 后要跟表达式。不需要返回值的函数使用 return 即可。

第 3 章　IAR C 编译器的使用

编写程序的过程主要包括编写代码、编译、调试，这 3 个过程是交替进行的。编译器的生产厂商通常将代码编辑器、编译器、调试器集成在同一个软件包中，称为集成调试环境，这样的软件包简称为编译器。熟练掌握编译器的使用方法是非常重要的，往往能够大幅度提高编写、调试代码的效率。

本章从使用的角度出发，详细讲述了 IAR C 编译器的使用方法和 C 语言的扩展部分，以及如何利用编译器的特性提高代码的执行效率。最后，列出了 C 语言的库函数并给出解释，便于读者查阅使用。

3.1 概　　述

3.1.1 特性

IAR 公司成立于 1983 年，为 8 位微处理器开发汇编编译器。IAR 公司的编译器已成为业界领先的嵌入式开发平台，广泛应用于移动电话、GPS 系统、远程控制、游戏机等含有微处理器的系统的开发中。IAR 公司的编译器可以支持超过 30 种不同的 8 位、16 位、32 位处理器。

针对 MSP430 的开发平台全称为 IAR Embedded Workbench EW430，以下简称为 EW430，其功能非常强大，而且仍然在以很快的速度更新版本。它的基本特性为：

（1）支持 ANSI C 并包含对 Embedded C++的支持。

（2）内建 MSP430 特性扩展优化。

（3）代码长度和速度有多级优化。

（4）支持 32 位和 64 位浮点数。

（5）支持硬件乘法器。

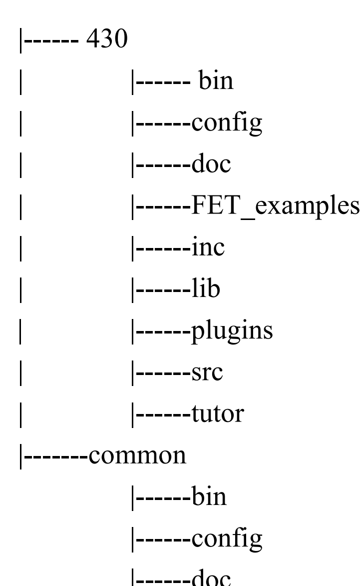
（6）内部函数支持低功耗模式。

（7）支持 C 和汇编语言混合编程。

3.1.2 软件结构

EW430 中包含 C 编译器 ICC430、汇编编译器 A430、调试器 C-SPY。EW430 软件包安装完毕的目录如下：

IAR Systems、Embedded Workbench Evaluation 4.0



```
|
    |------ 430
    |            |------ bin
    |            |------config
    |            |------doc
    |            |------FET_examples
    |            |------inc
    |            |------lib
    |            |------plugins
    |            |------src
    |            |------tutor
    |-------common
                 |------bin
                 |------config
                 |------doc
```

|------plugins

|------ src

430 目录下为与 MSP430 有关的内容。

（1）430\bin 下为 C/C++、汇编编译器和调试器程序。

（2）430\config 下为开发环境和工程的设置记录文件。

（3）430\doc 下为关于 EW430 的补充信息和使用手册，这些手册是用户最重要的资料来源。以下列出所补充的信息和使用手册的名称，以及他们所包含的内容：

- a430.htm　A430 版本发布信息，包含版本新特性和已经发现的问题
- a430_msg.htm　A430 的警告信息和错误信息
- cs430.htm　C-SPY 版本发布信息
- ew430.htm　EW430 版本发布信息
- icc430.htm　ICC430 版本发布信息
- icc430_msg.htm　ICC 警告信息和错误信息
- migration.htm　关于将旧版本 EW430 下的程序转换到当前版本的信息
- clib.pdf　C 库函数手册
- EW430_AssemblerReference.pdf　A430 编译器手册
- EW430_CompilerReference.pdf　ICC430 编译器手册
- EW430_UserGuide.pdf　EW430 开发环境（IDE）使用手册
- EW430Help.chm　EW430 编译器在线帮助
- EW430Help_c.chm　EW430 开发环境（IDE）在线帮助
- EW430Help_d.chm　C/EC++函数库在线帮助
- EW430Help_s.chm　EW430 开发环境工程选项在线帮助
- readme.htm　软件包发布信息

（4）430\FET_examples 下为 MSP430 的一些程序实例。

（5）430\inc 下为 C\C++和汇编编译器使用的头文件。

（6）430\lib 为编译器预定义的库文件。

（7）430\plugins 下为嵌入模块的可执行文件。

（8）430\src 下为允许用户修改的库函数的源代码和应用实例。

（9）430\tutor 下为 EW430 学习指导用到的实例。

common 目录下为 IAR SYSTEM 共用部分。前面介绍过 IAR SYSTEM 支持多种器件的开发。

（1）common\ bin 下为开发环境、连接器等程序。

（2）common\ config 下为开发环境的设置记录文件。

（3）common\ doc 下为关于 common 部分的补充信息和使用手册。

（4）common\ plugins 下为嵌入模块的可执行文件。

（5）common\ src 下为 common 部分学习指导用到的实例。

3.1.3　文件类型

一个工程文件会包含多种不同类型的文件，编译器也会生成一些中间类型的文件。表 3-1 列出了几种重要文件的类型和用途。

表 3-1　　文件类型和用途

文件类型	用　　途
.asm	汇编源文件
.s43	汇编源文件
.c	c 源文件
.d43	编译器生成的包含调试信息的目标文件
.ewp	工程项目文件
.eww	工程集合（workspace）文件
.h	c 或 c++头文件
.inc	汇编头文件
.lst	c 和汇编编译器生成的列文件
.txt	在指定输出目录下，编译器生成的可以下载到处理器运行的目标机器码文件，称为 msp430-txt，其格式为普通的文本文件

3.2　开发调试环境

3.2.1　创建一个工程

要开发一个完整的软件，其源文件通常会有多个，而且文件类型也有多种。为了便于使用和管理，将这些文件的集合称为 project（工程或者项目）。其中有几个文件是专门用来记录编译器设置、文件列表等开发环境的，称为工程文件。开发软件的时候，首先要创建一个 project，设置工程参数，然后才可以编译和链接。EW430 除了创建工程，还需要创建一个 workspace（工程集合）。workspace、project、源文件的关系如图 3-1 所示。

工程集合中可以只有一个工程。每个工程中也可以只有一个源文件，视需要而定。源文件的种类可以是 C 文件、汇编文件，也可以是库文件。

创建工程的过程为：

（1）从 File 菜单中选择 New 选项，出现的对话框如图 3-2 所示。

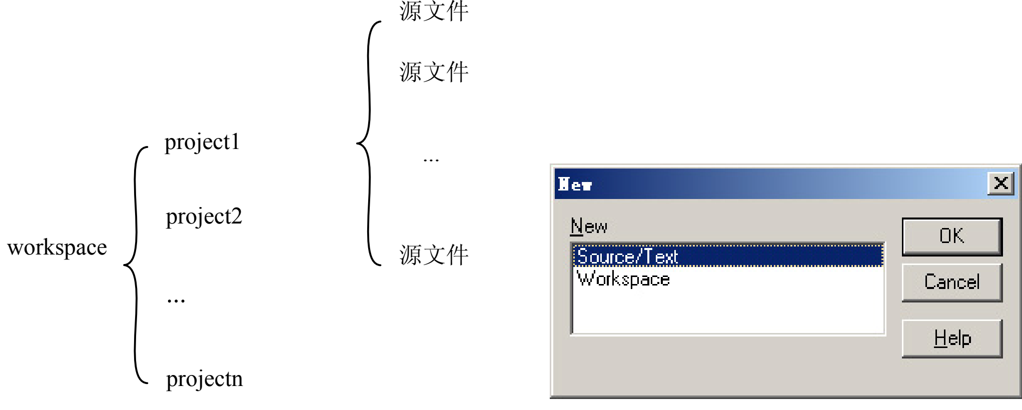


图 3-1　工程集合、工程、源文件的关系　　　　图 3-2　新建对话框

选择 Workspace 项，新建一个工程集合。出现的对话框如图 3-3 所示。

图 3-3　新建的 workspace

（2）从 Project 菜单中选择 Creat New Project 选项。出现的对话框如图 3-4 所示。

在 4 个选项中选择一个。其中 Empty project 表示建一个空的工程，asm 表示建一个纯使用汇编语言的工程，C++表示建一个使用 C++语言的工程，C 表示建一个使用 C 语言的工程。选择后 3 种都会为用户生成一个主程序的框架，并对编译和链接的选项选择相关的设置。不过，它不会包办所有的工作，一般用户在后面仍然需要按照自己的要求进一步调整选项。

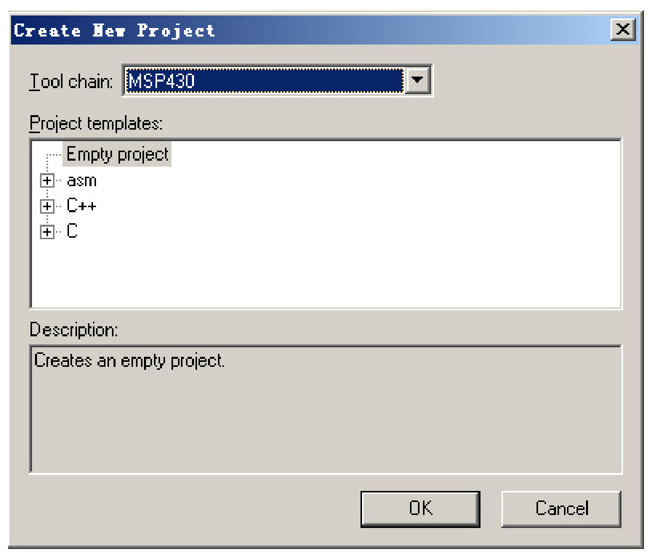


图 3-4　创建新工程

选择使用 C 语言的工程，按 OK 按钮。出现对话框，要求填写工程名称以及保存路径，工程名称填写为 study。填写完毕后，按 OK 按钮出现创建完毕的工程，如图 3-5 所示。

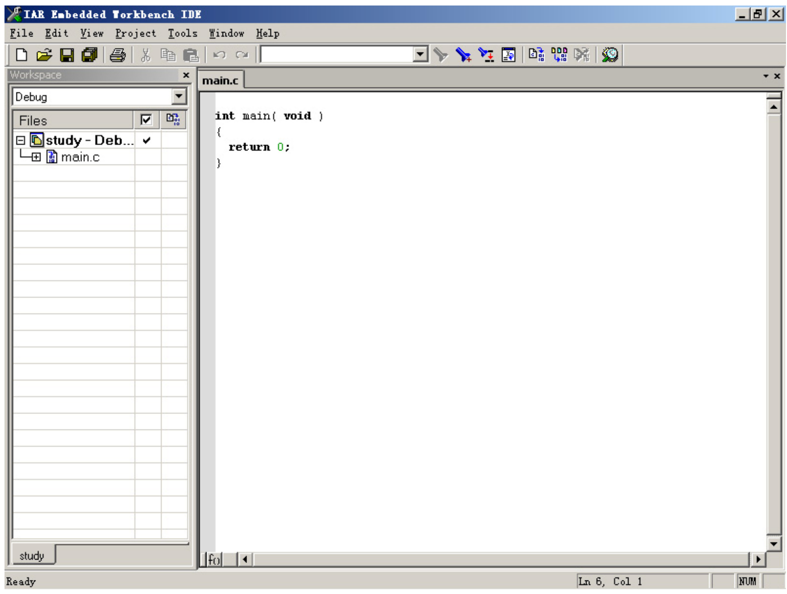


图 3-5　创建的工程

可以看到，创建了一个工程名字叫做 study，study 下包含一个 main.c 文件。main.c 是由编译器自动生成的。main.c 中有一个空 main 函数。EW430 还同时为同一个工程创建了两种编译选项：Debug 和 Release。Debug 编译时在目标文件中包含调试信息，进行低级别（low）优化，用于调试程序。Release 编译时不在目标文件中包含调试信息，且对目标代码文件进行了优化，生成的目标文件为 msp430-txt 格式，用于最终软件发布。用户也可以创建自己的编译方式。

向工程中添加其他文件。将鼠标移动到工程窗口上，单击右键，出现如图 3-6 所示的弹出菜单。选择 Add Files 选项，出现添加文件选择对话框，添加需要的文件。

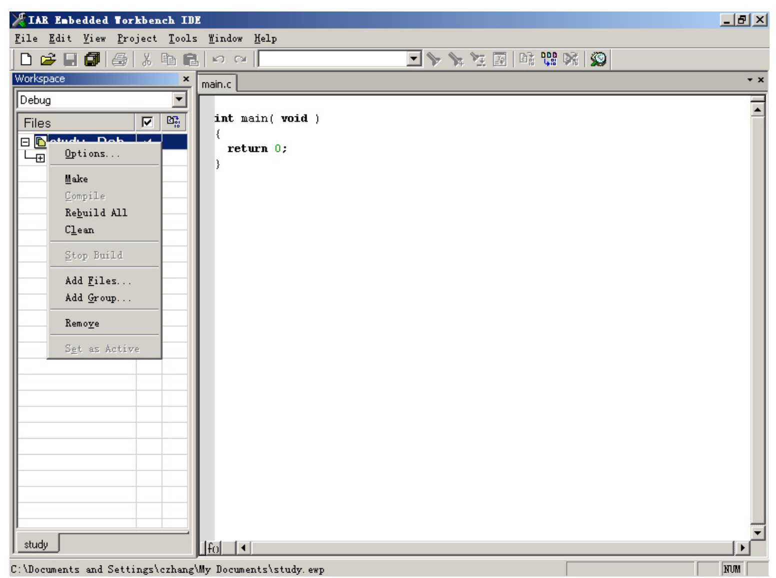


图 3-6　向工程中添加文件

添加文件完毕的窗口如图 3-7 所示，工程中有两个文件，main.c 和 24c02.c。

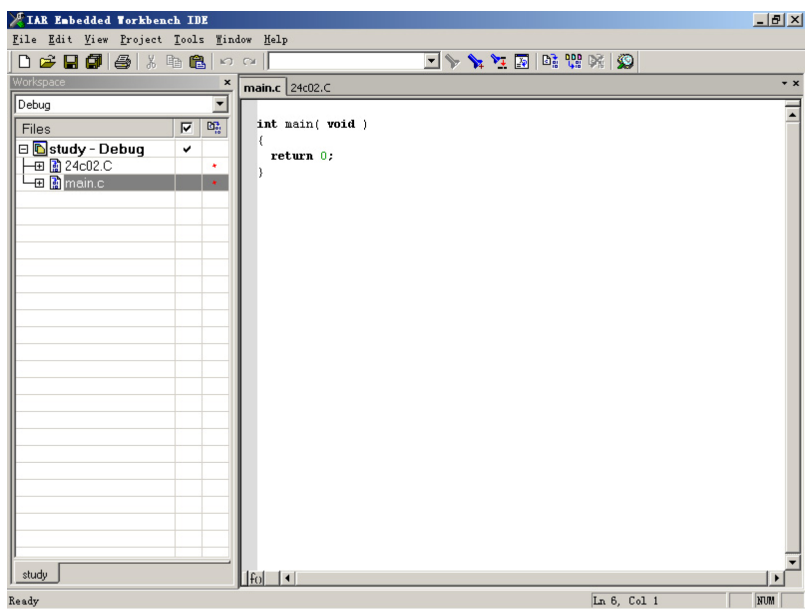


图 3-7　添加文件完毕

3.2.2　编译链接项目

C 语言源文件被转变为目标机器码的过程分为编译和链接两步，都由编译器完成，有时也会将两步合起来称为编译。编译开始之前先要对编译器的参数作一些设置。在图 3-6 中的弹出菜单中选择 Options 选项，注意单击右键之前鼠标要放在 study-Dubeg 栏上，而不是放在工程中的文件（如 main.c、24c02.c）上。如果是放在工程中的文件上，设置的仅是这个源文件的参数，而不是整个工程的参数。编译器允许每个源文件使用各自的编译、链接参数。在图 3-6 中的弹出菜单中选择 Options 选项后，出现 Options 对话框，如图 3-8 所示。

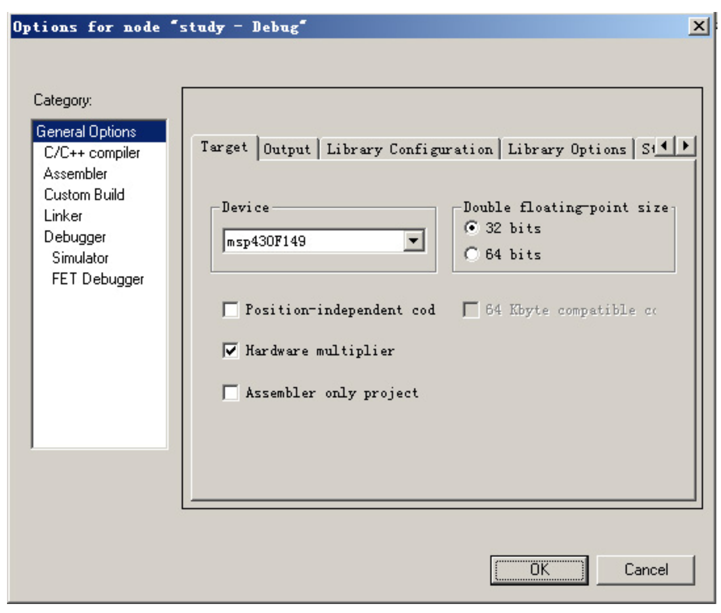


图 3-8　参数设置

图 3-8 所示的对话框中选项很多，具体内容将在项目设置中介绍。

设置完毕后，单击 OK 按钮返回。在菜单 Project 项中选择 Make 选项，或者单击图标按钮进行编译和链接。如果成功，则结果如图 3-9 所示，在 Message 窗口中显示如下内容：

...

42 bytes of CODE memory

80 bytes of DATA memory

2 bytes of CONST memory

...

Total number of errors:0

Total number of warrings:0

该结果表示这段程序最终在 CPU 中占据了 42 字节的程序存储器和 80 字节的数据存储器。CONST memory 是只读数据存储段，实际上也占据的是程序存储器（FLASH）。

通过编译链接的程序错误总数必须为 0。如果不为 0，必须要将错误排除，否则编译器不会生成目标文件。

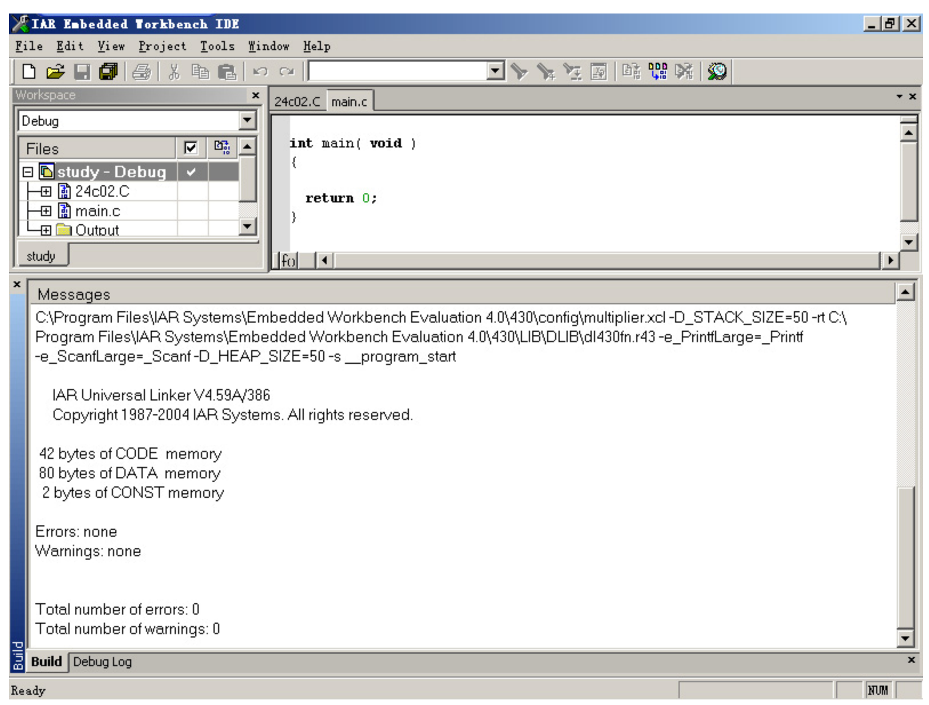


图 3-9　编译链接成功

如果警告不为 0，编译器也会生成目标文件，这要视警告的性质而定，用户有时可以忽略这些警告。但对于一个严谨的程序，警告也应该全部排除。

3.2.3　项目设置

下面具体介绍图 3-8 中项目设置（Category）中的内容，为了节省篇幅，有些晦涩且极

少用到的选项将会略过，使用时选择默认值就可以了。

1．General Options

General Options 是普通设置，包括 Target、Output、Library Configuration、Library Options、Stack/Heap 选项卡。下面对各选项卡进行具体介绍。

（1）Target 选项卡　包括 Device、Double floating-point size、Position-independent code、Hardware multiplier、Assembler only project 项。

- Device　选择 CPU 类型。
- Double floating-point size　浮点数的长度，可以选择 64 位或者 32 位。应当尽可能地选择 32 位。
- Position-independent code　选中此项后所生成的目标代码与存储地址无关，编译器在编译时，所有与地址有关的操作都编译为执行相对地址的指令。因此放置在 CPU 可执行地址范围内的任意地址都可以执行。缺点是代码长度会增加。
- Hardware multiplier　选中此项后所编译生成的代码会使用硬件乘法器，但只在 Device 中选择的 CPU 类型中有乘法器时才有效，否则自动忽略。
- Assembler only project　如果工程中只有汇编源文件，则选中此项时，编译器会自动进行一些参数的设置。

（2）Output 选项卡　包括 output file、Output Directories 项。

- output file　如果准备生成一个可在目标 CPU 中执行的完整代码，则选择 Executable。如果准备生成一个供其他工程调用的库，则选择 Library。
- Output Directories　生成目标文件保存的路径。

（3）Library Configuration 选项卡　包括 Library、Description、Compact math libraries 项。

- Library　选择编译时调用的运行库的类型。可选库的类型如表 3-2 所示。DLIB 库支持用户进行裁减、修改、重建等操作。但在大多数情况下，选择 Normal DLIB 或者 CLIB 即可以满足要求。
- Description　描述所选择的库的性能。
- Compact math libraries　选择此项时，编译器启用一个较小的浮点运算库，这可以减小代码长度，但同时会缩减某些运算特性。

表 3-2　可选库类型

库	说　明
None	不选择任何库
Normal DLIB	一般的 DLIB 库，支持 C++，完全兼容于 ISO/ANSI C
Full DLIB	完整的 DLIB 库，支持 C++，完全兼容于 ISO/ANSI C
Custom DLIB	自定义 DLIB 库，有用户选择所需要的库文件。完全兼容于 ISO/ANSI C
CLIB（default）	CLIB 库，不支持 C++。不完全兼容于 ISO/ANSI C
Custom CLIB	自定义 CLIB 库，不完全兼容于 ISO/ANSI C

（4）Library Options 选项卡　仅包括 Printf Formatter/ Scanf Formatter 一项。

- Printf Formatter/ Scanf Formatter　printf 和 scanf 是 C 语言中与格式转换有关的两个函数。选择合适的模式可以提高代码的效率，减小消耗。

（5）Stack/Heap 选项卡　包括 stack size、Heap size 项。

- stack size　堆栈分配的字节数。
- Heap size　堆分配的字节数。

2．C/C++ compiler

C/C++ compiler 是对 C/C++程序的编译设置，包括 Language、Code、Output、List、Preprocessor、Diagnostics 选项卡。下面对各选项卡进行具体介绍。

（1）Language 选项卡　包括 Language、Require prototypes、Language conformance、Plain ‘char’ is、Enable multibyte support、Enable IAR migration preprocessor extensions 项。

- Language　选择使用的语言，如表 3-3 所示。
- Require prototypes　强制编译器检查所调用的函数是否有正确的函数原型。
- Language conformance　包括 Allow IAR Extensions、Relaxed ISO/ANSI、Strict ISO/ANSI 三个选项。Allow IAR Extensions 允许使用与 MSP430 硬件特性有关的扩展，这些扩展不遵守 ISO /ANSI C 规范，所以，程序可移植性较差。Relaxed ISO/ANSI 不允许使用与 MSP430 硬件有关的扩展，但是允许采用比较宽松的 ISO/ANSI。Strict ISO/ANSI 要求程序严格遵守 ISO/ANSI 规范。
- Plain ‘char’ is　选择 char 的含义。选择 signed 表示编译器认为 char 是 signed char。选择 unsigned 表示编译器认为 char 是 unsigned char。
- Enable multibyte support　有效使编译器支持多字节的字符集。编译器会根据所在计算机使用的字符集编辑或者打开、保存源文件。
- Enable IAR migration preprocessor extensions　这一项有效时编译器对早期版本的编译器兼容。

表 3-3　选择语言

语　言	说　明
C	支持 ANSI/ISO C，不支持 C++
Embedded C++	支持 C++，需要选择 DLIB
Extended Embedded C++	支持 C++及扩展性能。可以使用一些扩展特性，如模板库
Automatic	编译器根据文件后缀确定使用的语言。.c 使用 C 语言。.cpp 使用 C++

（2）Code 选项卡　包括 Optimizations、Enabled transformations、R4 utilization、R5 utilization 项。

- Optimizations　优化设置。选择 size 表示尽量减小生成代码的长度。选择 speed 表示尽量提高代码的执行速度。在此两种选择的基础上可以进一步选择优化的级别，其中 NONE 级别表示不进行优化。在调试程序时应该选择 NONE 级别，否则某些变量的值在调试时就不能被跟踪。
- Enabled transformations　优化的过程中，编译器会对程序作一些调整，优化类型如表 3-4 所示。

- R4 utilization　选择寄存器 R4 的用法。Normal use 表示由编译器使用 R4。_regvar 表示 R4 用于保存寄存器变量。Not used 表示完全由用户决定如何使用。
- R5 utilization　选择寄存器 R5 的用法。选项与 R4 utilization 一样。

表 3-4　优化类型

类　型	说　明
Common subexpression eliminationg	消除需要重复计算的表达式，由编译器只做一次运算，可以提高运行速度，减小代码长度
Loop unrolling	对于循环次数不依赖于程序运行结果的循环，编译器通过多次复制一个循环次数较少的循环来代替。这样可以加快运行速度，但会增加代码长度
Function inlining	对于简单的函数，编译器把整个函数体直接嵌入调用它的函数中，省去了调用函数用到的如入栈、出栈等消耗。这样可以加快运行速度，但会增加代码长度
Code motion	将一些重复使用的代码进行移动，以减少运算次数。这样可以提高运行速度，减少代码长度
Type based alias analysis	编译器通过分析数据的传递过程，消除不必要的中间传递过程。可以提高运行速度，减少代码长度

（3）Output 选项卡　包括 Make library module、Object library module name、Generate debug info 项。

- Make library module　选择此项将生成一个库模块，而不是程序模块。
- Object library module name　通常生成的库模块的名字为源文件的名字去掉后缀，此选项由用户自己确定生成的库模块的名字。
- Generate debug info　选择此项编译器会在生成的目标文件中加入调试信息。如果没有调试信息将无法通过 C-SPY 或者其他的调试器调试程序。选择此项会少量增加代码长度。

（4）List 选项卡　包括 Output list file、Output assembler file 项。

- Output list file　选中此项表示编译的同时生成后缀为.lst 的列表文件。选择 Assembler mnemonics 表示在列表文件中加入与源文件对应的汇编语句。选择 Diagnostics 表示在列表文件中加入诊断信息。
- Output assembler file　选中此项表示编译器编译的同时生成后缀为.s43 的汇编文件。Include source 表示在生成文件中包含源文件。Include compiler runtime information 表示在生成文件中包含有关程序运行库的一些信息。

（5）Preprocessor 选项卡　包括 Include paths、Defined symbols、Preprocessor output to file 项。

- Include paths　为#include 宏指令用到的头文件指定搜索路径。
- Defined symbols　在这里可以定义一个宏变量，与在源文件中使用#define 定义的效果一样。某些情况下，相同的源文件需要根据不同的需求作出不同的处理，产生不同的代码。可以建立多个工程，在这里定义不同的宏变量，并在源文件中通过宏对这些宏变量进行判断，从而使编译器自动生成不同的代码，以适应不同的需求。
- Preprocessor output to file　将预处理执行的结果输出到文件，文件名为源文件的名

字，后缀为.i。选择 Preserve comments 表示生成文件的时候保留注释。

（6）Diagnostics 选项卡　编译器在编译过程中发现源文件中的问题，会根据其严重程度给出错误信息或警告信息。此处可以对发现的问题重新归类，一般不需要重新设置。

3．Assembler

Assembler 是对汇编程序的编译设置，包括 Lanauage、Output、List、Preprocessor、Diagnostics 选项。请参考对 C/C++程序的编译设置。

4．Custom Build

可以在 Custom Build 项中增加 IAR EW430 之外的第三方工具。只包含 Custom Tool Configuration 一个选项卡。

5．Linker

Linker 将编译完成的各个文件模块链接、定位，使之成为一个可以执行的完整的程序，包括 Output、Extra Output、#define、Diagnostics、List、Config、Processing 选项卡。下面对各选项卡进行具体介绍。

（1）Output 选项卡　包括 Output file、Format 项。

- Output file　链接完成后，产生文件默认的名字为工程名加相应的后缀。未选中 Override default 表示使用默认名，显示在下面的文本框中，呈灰色。如选中则表示用户可以自己填充生成的文件名。
- Format　包括 Debug information for C-SPY 和 other 两个选项。
  - Debug information for C-SPY　选择此项表示生成的文件中包含调试信息，产生文件的后缀为.d43。选中 With runtime control modules 表示会增加一些有关控制程序挂起、退出、异常的调试信息。选中 With I/O emulation modules 会增加对 I/O 进行操作的调试信息，并将重定位 stdin 和 stdout 的工作方式，用户可以通过计算机上的终端对话框（Terminal I/O window）模拟 CPU 的 I/O 操作。如果计算机与 CPU 的通讯速度比较慢，可以选中 Buffered terminal output，缓冲从 CPU 传出的数据。选中 Allow C-SPY-specific extra output file 将使 Extra Output 页中的选项有效。
  - Other　选择此项表示产生的文件中不包含任何调试信息。通过 Output 下拉框选择输出格式。TI 公司将 MSP430 的标准输出格式命名为 msp430-txt。

（2）Extra Output 选项卡　在某些情况下，需要输出两个含调试信息的文件，可以在此处指定第二个文件。

（3）#define 选项卡　可以定义在链接时使用的宏变量，其作用与编译时定义的宏变量类似。

（4）Diagnostics 选项卡　编译器链接时有关错误和警告的设置。一般不需要重新设置。

（5）List 选项卡　产生链接的列表文件。文件名同工程文件名，后缀为.map 。文件中主要列出链接后程序各模块所占用地址空间和各个段（代码段、数据段、堆栈段）所占用地址

空间的情况。

（6）Config 选项卡　包括 Linker command file、Override default program 项。

- Linker command file　在 General Options 中选择了 CPU 的型号后，编译器会自动选择相应的文件，一般不需要修改。用户可以先选中 Override default，然后在文件选择对话框中选择需要的文件。
- Override default program　如果不希望使用编译器指定的程序入口标识符 _program_start，可以选中此项，然后自己定义。一般不需要修改。

（7）Processing 选项卡　包括 Fill unused code memory、Generate checksum 项。

- Fill unused code memory　选中此项则将 CPU 没有用到的程序存储器用 Fill pattern 文本框中的数值中填充。
- Generate checksum　选中此项将生成校验字节。有几种不同的算法可供选择，计算出的校验字节由编译器放置在 CHECKSUM 段中，通常紧跟在生成的程序代码的后面。此项功能可以用来检查程序在程序存储器中是否有错误。

6. Debugger

Debugger 是有关调试程序选项的设置，包括 Setup、Plugins 选型卡。下面对各选项卡进行具体介绍。

（1）Setup 选项卡　包括 Driver、Run to、Device description file 项。

- Driver　选择调试方式。FET Debuger 选择 JTAG 调试方式。Simulator 选择软件模拟方式。
- Run to　可以设置为 CPU 复位后，程序首先运行到某个地方停下，等待用户下一步的指令。
- Device description file　在 General Options 中选择了 CPU 的型号后，编译器会自动选择相应的文件，一般不需要修改。用户可以先选中 Override default，然后在文件选择对话框中选择需要的文件。

（2）Plugins 选项卡　调试时可以插入辅助调试模块。

7. FET Debuger

FET Debuger 是有关 FET 仿真器的设置，包括 verify download 和 Download control 两项内容。下面对各项进行具体介绍。

（1）verify download 复选框　选中此项表示下载程序后进行校验。

（2）Download control 组合选择框　包括 Suppress download、Erase main memory、Erase main and information memory、Retain unchanged main memory 项。

- Suppress download　如果程序已经下载到 CPU 中了，则选中此项将跳过下载的过程。选中 Ask when downloading，将出现一个对话框，由用户选择是否重新下载程序。
- Erase main memory　下载前先擦除程序存储器。
- Erase main and information memory　下载前先擦除程序存储器和信息存储器。
- Retain unchanged main memory　只擦除程序存储器中程序用到的段，其他的保留不变。

（3）Use virtual breakpoints 复选框　硬件断点有数量限制，硬件断点用完以后，如果选中此项，C-SPY 就会使用虚拟断点。使用虚拟断点迫使程序以单步的方式执行，所以程序执行的速度会很慢。

（4）System breakpoints on　仅在使用 CLIB 库时有效。如果不使用 Terminal I/O Window，并且也不需要在程序退出时触发一次中断，则可以不选择 exit、putchar、getchar，这样可以节约断点资源。但如果使用的是 DLIB，则此选项无效。

3.2.4　调试

本节所讲的调试针对的是通过 JTAG 端口连接硬件进行仿真的情况。连接好 JTAG 仿真器后（参见第 4 章），在菜单 Project 项中选择 Debug 选项或者单击图标按钮，正常情况下，将会进入调试状态，如图 3-10 所示。进入调试界面之前，编译器会检查项目中的文件在修改后是否被编译过，如果没有，先进行编译和链接。这样可使生成的目标代码和源文件总是保持一致。

1．运行

图 3-10 调试界面上多了一些图标按钮，这些图标都有对应的菜单选项。因为使用频率很高，所以提供了图标按钮。下面分别对这些图标按钮加以介绍：

Reset　复位

Step Over　单步运行。遇到函数调用时，将遇到的函数当作单独的一步执行。

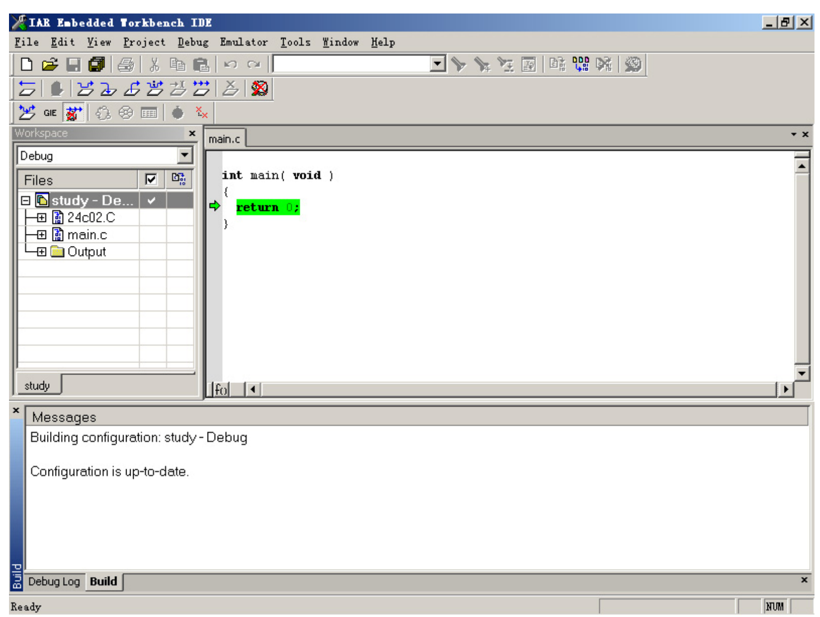


图 3-10　调试界面

Step Into　单步运行。遇到函数调用时，进入所遇到的函数中，并执行一句语句。

Step Out　单步运行。将本函数执行完毕，退出本函数后停止程序运行，等待用户新的命令。

Next Statement　单步运行。C 语言的一句语句中还可以包含多个表达式，Step Over、Step Into、Step Out 将每一个表达式作为一步。Next Statement 不考虑包含的表达式，直接执行 C 语言完整的一句语句。参见随书光盘中例 3-1：MoonRiver=Moon(100)+River(200)；单击 Step Over 图标将先执行 Moon（100），然后再执行 River（200）。单击 Step Into 图标将进入到 Moon（100）和 River（200）中一步一步地执行 Moon 函数中的程序。如果进入了 Moon 或者 River 函数中，则执行 Step Out 会立即执行完当前函数并退出此函数，然后停下。Next Statement 将一次执行完 MoonRiver=Moon(100)+River(200)这一整句，然后停下。

Run to Cursor　运行到光标所在位置停下。

Go　运行。

break　暂停。

Toggle Breakpoint　建立或者取消光标所在位置的断点。

Stop Debugging　退出调试界面。

GIE Enable/disable GIE bit　将 CPU 内 SR 寄存器中的全局中断控制位 GIE 置位或者清零。

Leave target running　退出调试后，使目标 CPU 继续运行。

2．查看

程序暂停时，将光标放在变量上，可以查看此变量当前的值。对于局部变量，仅限于程序停止时所在位置的函数的局部变量，因为局部变量是程序进入函数的时候才分配的。

程序暂停时，将光标放在变量上，在右键菜单中选择 Quick Watch 选项，可以查看和修改变量的值。

在菜单 View 选项中：

- Disassembly　显示汇编窗口。
- Memory　显示存储器窗口。MSP430 是统一寻址的，通过此窗口可以看到所有地址空间内存储的内容，并可以进行修改。
- Register　显示寄存器窗口。可以查看和修改所有特殊功能寄存器以及 CPU 寄存器。
- Watch　显示和修改用户指定的变量和表达式的值。
- Locals　显示和修改光标所在函数内局部变量的值。
- Auto　显示和修改光标所在位置的表达式中出现的变量的值。
- Live Watch　显示和修改静态变量的值。
- Call Stack　显示当前哪些函数被调用。
- Trace　跟踪记录程序执行时用户指定的表达式的值。
- Stack　查看 CPU 内的堆栈使用情况，包括当前堆栈指针位置、堆栈深度等。

3．断点

EW430 可以设置程序断点和数据断点。执行到断点所在的位置时，程序会停下，此时用户可以检查程序的执行情况，此为程序断点。数据断点可使某个变量的值发生变化时产

生中断。

设置断点的方法有两种：

（1）使用按钮或者菜单 Edit 中的 Toogle breakpoint 选项，在光标所在位置添加或者删除程序断点。

（2）使用菜单 Edit 中的 Breakpoints 选项，在 Breakpoints 对话框中添加程序或数据断点，并可以设置复杂的断点响应条件。

打开断点对话框，对话框是分页的。图 3-11 所示为设置程序断点对话框，图 3-12 所示为设置数据断点对话框，图 3-13 所示为设置条件断点对话框，条件断点的范围比较广，包含了程序断点和数据断点。

对话框的 Break At 栏用于填充断点的位置。已经有的断点在对话框下方列出。单击 Edit 按钮，出现对 Enter Location 话框。其中有 3 种断点类型可以选择：

（1）Expression　如图 3-14 所示。在 Expression 中填充一个计算结果为某一地址的表达式。如填某一函数的名称，则会在进入此函数后的第一句设置一程序断点。 如填某一变量的名称，则会为此变量设置一数据断点。

（2）Absolute address　如图 3-15 所示。在 Adress 中填写程序或者变量的地址来设置断点。对于 C 语言的调试来说，这种方式不方便，所以一般不使用。

（3）Source location　如图 3-16 所示。通过确定某一源文件的行和列来设置断点。对于 C 语言的调试来说，这种方式不方便，所以一般不使用。

3 种断点设置对话框的设置过程如下：

（1）程序断点对话框　图 3-11 程序断点对话框中的 Conditions 下的 Expression 文本框中可以填充一表达式作为触发断点的另一条件，满足触发断点条件的方式有以下 3 种：

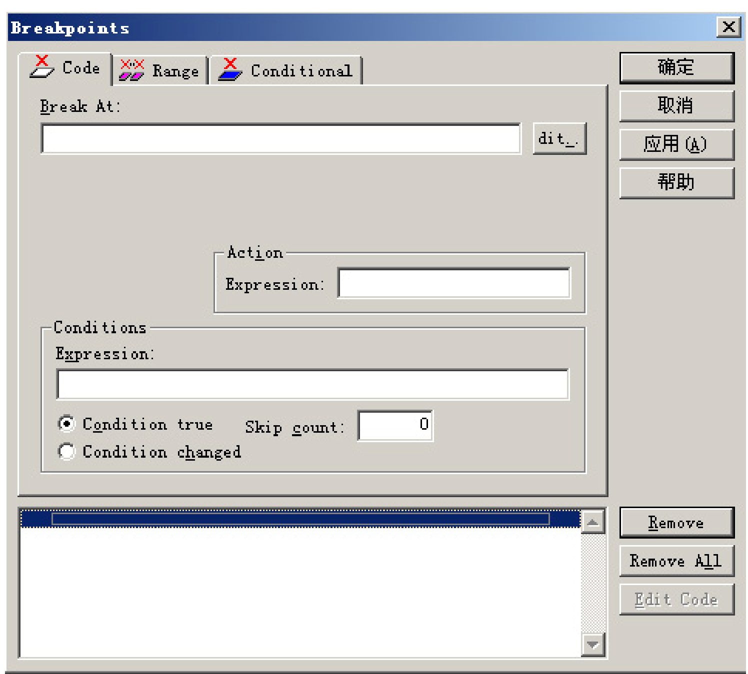


图 3-11　程序断点对话框

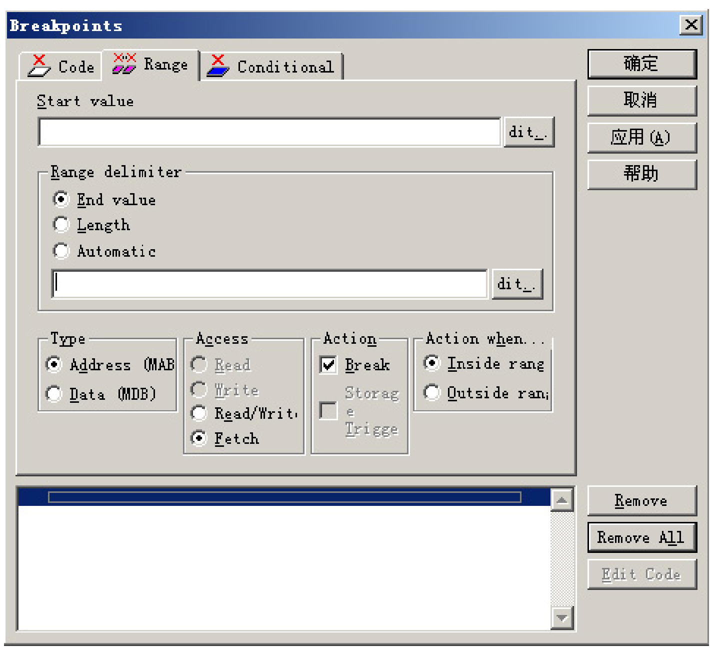


图 3-12　数据断点对话框

- Condition true　表达式计算结果为真。
- Condition change　表达式计算结果改变。
- skip count　填充一数值 n，程序前 n 次经过断点位置不满足条件，在第 n+1 次满足。

当程序运行到断点位置且表达式满足 3 种条件之一时，将会触发断点。

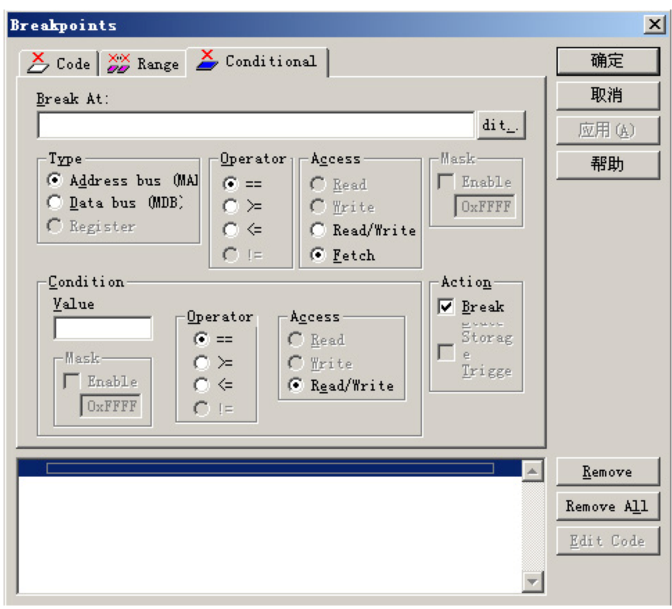


图 3-13　条件断点对话框

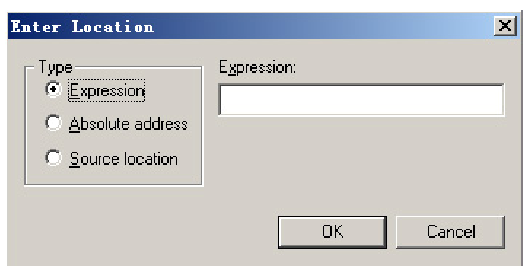


图 3-14　Enter Location 对话框 1

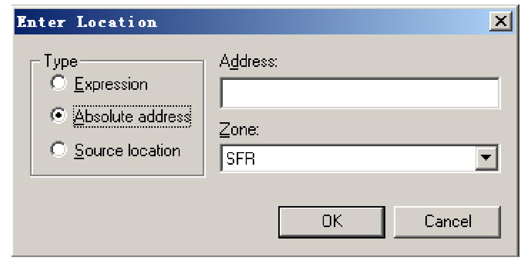


图 3-15　Enter Location 对话框 2

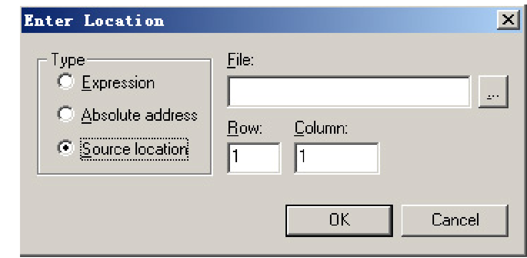


图 3-16　Enter Location 对话框 3

在 Action 处填充一表达式，断点被触发时将会自动运行此表达式。

（2）数据断点对话框　图 3-12 所示的数据断点对话框设置的数据断点可以是针对变量、数组、结构或者一段地址范围内的数据，其中任何数据的改变如果满足了触发条件都会触发断点。触发条件如下：

- Start value　起始地址。在图 3-12 中单击 Edit 按钮（图中显示为 dit 按钮），出现 Enter Location 对话框，填充断点位置。如果要为变量、数组、结构设置断点，则在 Expression 中填充名称。如果要设置一段地址范围，则填充起始地址。
- Range delimiter　确定触发断点的地址范围，选项如下：
  - End value　结束地址，用十六进制格式填充。或者按在图 3-12 中单击 Edit 按钮，通过 Enter Location 对话框填充。
  - Length　地址范围的长度，用十六进制格式填充。或者在图 3-12 中单击 Edit 按钮，通过 Enter Location 对话框填充。
  - Antomatic　如果 Start value 中填充的是变量、数组或者结构，EW430 会自动计算地址范围。
- Type　断点触发的类型，选项如下：
  - Address bus　需要在 Break At 框中填充的表达式最终计算结果为一个地址，可以是变量也可以是函数，或者是某一句程序。当此地址被访问时，断点被触发。Address bus 的含义就是此地址出现在地址总线上时，触发断点。
  - Data bus　当某一个值出现在数据总线上时，触发断点。需要在 Break At 框中填充的表达式计算结果为一个值。
- Access　对触发中断的行为作出限制，选项如下：
  - Read/Write　对某一地址读写时，触发中断。
  - Fetch　取指令时，触发中断。
- Action　选择触发中断后作出的反应。break 为暂停程序执行。

- Action when　什么情况下触发中断，选项如下：
  - Inside range　访问开始前触发中断。
  - Outside range　访问结束后触发中断。

（3）条件断点对话框比较复杂，如图 3-13 所示。下面对触发条件进行解释：

- Break At　用法与程序断点对话框中的一样。在其中填充变量名称时，就可以设置一数据断点。需要注意的是，一般情况下要填充全局变量，否则，其结果是不确定的。而且，在设置数据断点时，要让程序至少单步执行一步，确保全局变量已经被分配完毕。
- Type　断点触发的类型，选项如下：
  - Address bus　需要在 Break At 框中填充的表达式最终计算结果为一个地址，可以是变量也可以是函数，或者是某一句程序。当此地址被访问时，断点被触发。Address bus 的含义就是此地址出现在地址总线上时，触发断点。
  - Data bus　当某一个值出现在数据总线上时，触发断点。需要在 Break At 框中填充的表达式计算结果为一个值。
- Operator　对触发中断的条件作出限制。可以选择程序满足==、>=或者<=文本框 Break At 中的填充内容时触发中断。如 Operator 取==，Break At 取一个值，Type 取 Data bus。当数据总线上出现等于 Break At 中的值时，触发中断。如果 Operator 取>=，则当数据总线上出现大于等于 Break At 中的值时，触发中断。
- Access　对触发中断的行为作出限制，选项如下：
  - Read/Write　对某一地址读写时，触发中断。
  - Fetch　取指令时，触发中断。

这两种形式触发中断的位置的区别十分微妙，一般调试时几乎不用区分。

- Condition　以上选项可以构成一个完整的断点。不过如果是数据断点，还可以在 Condition 中进一步添加条件，构成一个复合数据断点，在条件满足后触发断点。选项如下：
  - Value　数据总线或寄存器中的值。
  - Operator　选择当数据总线或寄存器中的值==、>=或<=Value 中的值时满足条件。
- Action　选择触发中断后作出的反应。break 为暂停程序的执行。

3.3　语 言 扩 展

为了发挥 MSP430 的优点，EW430 在 ISO/ANSI 标准 C 语言规范上进行了扩展，用户可以利用扩展部分编写出效率更高、功能更强的程序，并且可以大大减小开发的工作量。但这样做的代价是降低了可移植性，因为扩展的这部分代码直接与 MSP430 的硬件结构相关，而其他种类的处理器的硬件结构必定与 MSP430 有所不同。

通常的作法是将程序分层编写或包装。通用的算法严格遵守标准 C 语言规范，包装一组文件或者函数，保证可移植性。与 MSP430 硬件相关的部分，如操纵寄存器等，包装成一组文件或者函数，移植的时候只要修改这一部分，使之适应新的硬件就可以了。

语言扩展包含关键字扩展和函数扩展，扩展的函数称作内部函数，内部函数全部由汇编

语言实现，编译器编译时对内部函数进行嵌入（inline）处理。

3.3.1 扩展关键字

关键字的概念前面已经介绍过。下面是除了 C 语言标准关键字之外的扩展部分，这里只介绍常用的扩展关键字。

1．asm

也可以写成 __asm。功能是在 C 程序中直接嵌入汇编语言。

语法：

```
asm ("string");
```

其中 string 必须是有效的汇编语句。

2．__interrupt

放在函数前面，标志中断函数。下面这段程序是异步串行口 UART0 的接收中断函数。UART0RX_VECTOR 为异步串行口 UART0 的接收中断向量。

举例：

```
#pragma vector=UART0RX_VECTOR
__interrupt void UART0_R(void)   //UART0 接收中断
{
    TXBUF0=RXBUF0;
}
```

3．__monitor

放在函数前面，功能是当这一函数执行的时候自动关闭中断。应该尽量缩短这样的函数，否则，中断事件无法得到及时的响应。

4．__no_init

放在全局变量前面，功能是使程序启动时不为变量赋初值。

5．__raw

编译中断函数时，编译器会自动生成一段代码，首先保存当时所用到 CPU 内寄存器的内容，退出中断程序时再进行恢复。将__raw 放在中断函数前可以禁止保存 CPU 内寄存器的过程，当然退出时也不会恢复。是否为中断函数使用此关键字要根据需要而定。

6．__regvar

放在变量前面，作用是声明变量为寄存器变量。可以用于整数、指针、32 位浮点数以及只含有一个元素的结构和联合。寄存器变量的地址只能为 R4 或者 R5，也不能用指针指向这个寄存器变量，而且必须用__no_init 禁止初始化。如：

```
__regvar __no_init unsigned char q0 @ __R4;
```

其他不常用的关键字还有：__data16、__intrinsic、__noreturn、__root、__task、__word16。

3.3.2　内部函数

本节将介绍内部函数的原型和功能。

1．__bcd_add_short

```
unsigned short __bcd_add_short(unsigned short, unsigned short);
```

功能：两个 16 位 BCD 格式的数字相加，返回和。

2．__bcd_add_long

```
unsigned long __bcd_add_long(unsigned long, unsigned long);
```

功能：两个 32 位 BCD 格式的数字相加，返回和。

3．__bcd_add_long_long

```
unsigned long long __bcd_add_long_long(unsigned long long,unsigned long long);
```

功能：两个 64 位 BCD 格式的数字相加，返回和。

4．__bic_SR_register

```
void __bic_SR_register(unsigned short);
```

功能：将 CPU 中 SR 寄存器中的某些位清 0。其参数为屏蔽码，需要清 0 的位为 1。

5．__bic_SR_register_on_exit

```
void __bic_SR_register_on_exit(unsigned short);
```

功能：用于一个中断函数或者不可中断函数（标志为 __monitor）返回时，将 CPU 内 SR 寄存器中的某些位清 0。其参数为屏蔽码，需要清 0 的位为 1。

6．__bis_SR_register

```
void __bis_SR_register(unsigned short);
```

功能：将 CPU 中 SR 寄存器中的某些位置 1。其参数为屏蔽码，需要置 1 的位为 1。

7．__bis_SR_register_on_exit

```
void __bis_SR_register_on_exit(unsigned short);
```

功能：用于一个中断函数或者不可中断函数（标志为 __monitor）返回时，将 CPU 内 SR 寄存器中的某些位置 1。其参数为屏蔽码，需要置 1 的位为 1。

8．__disable_interrupt

```
void __disable_interrupt(void);
```

功能：关闭全局中断。先执行 DINT 指令，关闭全局中断，然后再执行 NOP 指令。空指令是为了确保关闭了全局中断之后再执行下面的程序。

9．__enable_interrupt

```
void __enable_interrupt(void);
```

功能：使用 NINT 指令打开全局中断。

10．__even_in_range

```
unsigned short __even_in_range(unsigned short value, unsignedshort upper_limit);
```

功能：只能与 switch 语句结合使用，判断 value 是否为偶数且小于等于 upper_limit。

举例：

```
unsigned int MoonRiver,iq0;
iq0=2;
switch(__even_in_range(iq0,4))
{
    case 0:
         MoonRiver=0;
    break;
    case 2:
         MoonRiver=2;
}
```

结果：假设 iq0 的值为 2，执行完毕时 MoonRiver=2。否则，与普通的 switch 语句一样，跳过 case 部分，直接执行下面的程序。使用__even_in_range 的好处是可以生成效率比较高的代码，在判断多中断源的中断的来源时可以使用此函数。

11．__get_interrupt_state

```
istate_t __get_interrupt_state(void);
```

功能：返回当前的中断状态。返回值 istate_t 为一结构，通过使用此函数可以获得当前的中断状态并保存，将来可以使用__set_interrupt_state 恢复中断状态。

12．__get_R4_register

```
unsigned short __get_R4_register(void);
```

功能：返回寄存器 R4 的值，只在 R4 被锁定时有效。

13．__get_R5_register

```
unsigned short __get_R5_register(void);
```

功能：返回寄存器 R4 的值，只在 R5 被锁定时有效。

14. __get_SP_register

```
unsigned short __get_SP_register(void);
```

功能：返回堆栈指针寄存器 SP 的值。

15. __get_SR_register

```
unsigned short __get_SR_register(void);
```

功能：返回 CPU 中状态寄存器 SR 的值。

16. __get_SR_register_on_exit

```
unsigned short __get_SR_register_on_exit(void);
```

功能：用于一个中断函数或者不可中断函数（标志为 __monitor）返回时，返回状态寄存器 SR 的值。只在中断函数或者不可中断函数中有效。

17. __low_power_mode_n

```
void __low_power_mode_n(void);
```

功能：进入低功耗模式 0～4。

18. __low_power_mode_off_on_exit

```
void __low_power_mode_off_on_exit(void);
```

功能：从一个中断函数或者不可中断函数（标志为 __monitor）返回时退出低功耗模式。只在中断函数或者不可中断函数中有效。

19. __no_operation

```
void __no_operation(void);
```

功能：执行 NOP 指令。

20. __op_code

```
__op_code(unsigned short);
```

功能：在指令流中插入一个常数。

21. __segment_begin

```
void * __segment_begin(segment);
```

功能：segment 是段的名字，必须是字符串。返回指向 segment 段的地址。此处的段是程序中定义的数据段、代码段、堆栈段等，一般用户可以使用编译器的默认设置。

22. __segment_end

```
void * __segment_end(segment);
```

功能：segment 是段的名字，必须是字符串。返回 segment 段结束后的第一个字节的地址。

23．__set_interrupt_state

```
void __set_interrupt_state(istate_t);
```

功能：恢复 istate_t 中保存的中断状态。

24．__set_R4_register

```
void __set_R4_register(unsigned short);
```

功能：将 unsigned short 值赋给寄存器 R4，只在 R4 被锁定时有效。

25．__set_R5_register

```
void __set_R5_register(unsigned short);
```

功能：将 unsigned short 值赋给寄存器 R5，只在 R5 被锁定时有效。

26．__set_SP_register

```
void __set_SP_register(unsigned short);
```

功能：给堆栈指针寄存器 SP 赋值。

27．__swap_bytes

```
unsigned short __swap_bytes(unsigned short);
```

功能：一个 16 位的无符号整数，高 8 位与低 8 位进行交换。如 0x1234 交换后为 0x3412。

3.3.3 扩展定义

为了使用方便，EW430 还作了一些定义，有些定义不属于 C 语言的一部分，而是一种二次包装，如对内部函数的包装。它们的功能完全可以用其他函数或者表达式实现，用户也可以自己重新定义，但使用它们会使编写程序更加简单易懂。

下面 1～4 项都是在不同类型 CPU 的头文件中定义的，如 mspx14x.h，其中对 CPU 内的各寄存器和模块的各种工作模式都作了详尽的定义，编程时应当尽可能地利用。

1．PxIN、PxOUT、PxDIR、PxSEL

x 为端口号。IN 为端口输入寄存器，OUT 为端口输出寄存器，DIR 为端口方向控制寄存器，SEL 为端口第二功能选择寄存器。

举例：

```
Moon=P1IN;       //读端口 P1 的值，赋给变量 Moon
P3Out=5;         //P3 端口输出 5
P2DIR=0xF0;      //P2 端口的高 4 位为输出，低 4 位为输入
P6SEL=0xF;       //P6 端口的高 4 位用作 I/O 端口，低 4 位用于第二功能
```

2．BITx

x 的取值范围为 0～F。代表寄存器的某一位。其定义为：

```
#define BIT0            (0x0001)
#define BIT1            (0x0002)
...
#define BITE            (0x4000)
#define BITF            (0x8000)
```

BIT0 为最低位，BITF 为最高位。MSP430 是不支持位操作的，如果想对位操作，最好的方法就是通过位屏蔽来实现。

举例：

```
P1OUT  |=  BIT0;    //将 P1 口的最低位输出置 1
P1OUT  &=  ~BIT7;   //将 P1 口的最高位输出清 0，P1 口只有 8 位
```

3．LPMx

x：0～4。进入 0～4 低功耗模式。其定义为：

```
#define LPM0    _BIS_SR(LPM0_bits) //进入低功耗模式 0
...
#define LPM4    _BIS_SR(LPM4_bits) //进入低功耗模式 4
```

从以上代码可以看出扩展定义是对内部函数的二次包装。

举例：

```
LPM0;   //进入低功耗模式 0
LPM4;   //进入低功耗模式 4
```

4．LPMx_EXIT

x：0～4。退出 0～4 低功耗模式。其定义为：

```
#define LPM0_EXIT _BIC_SR_IRQ(LPM0_bits)  //退出低功耗模式 0
...
#define LPM4_EXIT _BIC_SR_IRQ(LPM4_bits)  //退出低功耗模式 4
```

举例：

```
LPM0_EXIT;  //退出低功耗模式 0
LPM4_EXIT;  //退出低功耗模式 4
```

5．_EINT( )

打开全局中断控制，使 GIE=1。

6．_DINT( )

关闭全局中断控制，使 GIE=0。执行__disable_interrupt 指令。

7．_NOP( )

空操作。执行__no_operation 指令。

8．_OPC(x)

在指令流中插入一个常数。x 为 unsigned char 类型。执行__op_code 指令。

9．_SWAP_BYTES(x)

x 是一个 16 位的无符号整数，高 8 位与低 8 位进行交换。执行__swap_bytes 指令。

10．__no_init [数据类型] 变量名 @ 地址

在某一固定地址处定义一个不进行初始化的变量，地址可以在 RAM 或 FLASH 内。如果使用此方式定义在 RAM 内的变量需要赋值，那么必须首先定义，然后才能赋值。

举例：

```
/*分配变量 MoonRiver 在 RAM 地址 0x210*/
__no_init unsigned int MoonRiver @ 0x210;  //没有初始化
MoonRiver=100;                             //初始化 Moonriver 为 100
/*分配变量 MoonRiver 在 FLASH 地址 0xFFC0*/
__no_init float MoonRiver @ 0xFFC0;
/*分配数组 MoonRiver[3]在 RAM 地址 0x200*/
__no_init char MoonRiver[3] @ 0x200;
/*分配结构 sMoonRiver 在 RAM 地址 0x200*/
typedef struct
{
  unsigned char q0;
  unsigned int iq0;
}sMoonRiver;        //定义一个结构型的数据类型，取名为 sMoonRiver
__no_init sMoonRiver MoonRiver @ 0x200; //声明变量 MoonRiver，其数据类型为
sMoonRiver
MoonRiver.q0=100;
MoonRiver.iq0=1000;    //为 MoonRiver 赋初值
```

11．const [数据类型] 变量名 @ 地址

在某一固定地址处定义一个只读变量，并且只能在定义的时候赋初值。这种定义变量的方式在 FLASH 的固定地址处分配变量时非常有用。

举例：

```
/*分配变量 MoonRiver 在 RAM 地址 0x210*/
const unsigned int MoonRiver @ 0x210 =100;  //初始化 MoonRiver 为 100
/*分配变量 MoonRiver 在 FLASH 地址 0xFFC0*/
const float MoonRiver @ 0xFFC0=32.5;        //初始化 MoonRiver 为 32.5
/*分配数组 MoonRiver[3]在 FLASH 地址 0xFF00*/
```

```
    const char MoonRiver[3] @ 0xFF00={0,1,2};
    /*分配结构 sMoonRiver 在 RAM 地址 0xFFD0*/
    typedef struct
    {
      unsigned char q0;
      unsigned int iq0;
    }sMoonRiver;          //定义一个结构型的数据类型，取名为 sMoonRiver
    const sMoonRiver MoonRiver @ 0xFFD0={2,500}; //声明变量 MoonRiver，其数据类型
为 sMoonRiver
```

如果不需要确定地址，只想在 FLASH 中分配一个变量，则形式为：

```
    const [数据类型] 变量名;
```

举例：

```
    const unsigned int MoonRiver =100;
```

3.4　C 语言与汇编语言混合使用

C 语言与汇编语言混合使用有 3 种方式：

- 内部函数
- 直接嵌入
- 调用汇编模块

3.4.1　调用内部函数

因为内部函数本身是由汇编语言实现的，所以调用内部函数本身就是在 C 语言中嵌入汇编语言。不过内部函数数量很少，只能实现特定的功能。

3.4.2　直接嵌入

使用__asm 或 asm 扩展关键字。

举例：

```
    void foo()
    {
      while (!flag)
      {
        asm("MOV.B &P1IN,&flag");
      }
    }
```

这种方法使用起来很简单，但编译器只是简单地将汇编语句嵌入程序中，不考虑与前后语句是否匹配，因此，有可能会造成不稳定。它有几条限制：

（1）编译器编译时使用不同的优化级别会忽略嵌入的汇编语句，或者不进行优化。

（2）一些汇编指令不能嵌入。

（3）不能访问局部变量。

（4）不能声明语句标号。

不推荐使用这种方法。如果没有内部函数可以调用，那么最好通过调用汇编模块的方法嵌入汇编语言。这部分内容将在 3.4.3 节中具体介绍。

3.4.3　调用汇编模块

调用汇编模块时主要有 3 点需要注意：

（1）编写汇编模块时，必须严格遵从调用规则。

（2）必须在汇编模块中把函数声明为 PUBLIC。

（3）调用时，或者将汇编函数声明为 extern，或者为汇编模块编写头文件，以便编译器可以找到函数的位置。

这里对调用规则作出解释。编译器编译时，按照固定的规律将函数的参数放在寄存器中或者压入堆栈中，因此，编写汇编程序时，必须按照此规律来获取传入的参数。函数返回时也是如此。

在使用时通常将 MSP430 内的寄存器分为 2 组：

（1）临时性寄存器。可以用作临时性寄存器的有：

- CPU 寄存器 R12～R15
- 用于返回地址的寄存器
- CPU 寄存器 R11:R10:R8:R9 组合起来用于传递一个 64 位参数的时候

临时性寄存器有点像局部变量，每一级函数都可以任意使用。因此，如果在调用下一级函数后还想使用这些寄存器中的值，就必须在调用前将这些值保存到 RAM 中，以便将来使用时能够将这些值恢复。

（2）受保护的寄存器。CPU 寄存器 R4～R11 在不作为返回地址的寄存器时是受保护的寄存器。但是当 R11:R10:R8:R9 组合起来用于传递一个 64 位参数时，这 4 个寄存器是不需要保护的。每一级函数都可以使用受保护的寄存器，但在使用前必须保存它们的值，并在退出函数之前将这些值恢复。

参数传递可以通过寄存器或者堆栈进行，通过寄存器传递参数的效率比较高，所以，编译器被设计成尽量通过寄存器传递参数。由于可以使用的寄存器有限，下面 3 类参数只能通过堆栈传递：结构和联合类型、double 类型、参数不确定的函数。

寄存器传递参数的安排如下：

（1）第一个参数使用 R12（8、16 位参数）或 R13:R12（32 位参数）。

（2）第二个参数使用 R14（8、16 位参数）或 R15:R14（32 位参数）。

（3）如果有更多的参数，则使用堆栈。

（4）64 位的参数使用 R15:R14:R13:R12 或 R11:R10:R9:R8。

堆栈传递参数的安排比较简单。先看看堆栈的结构，如图 3-17 所示。

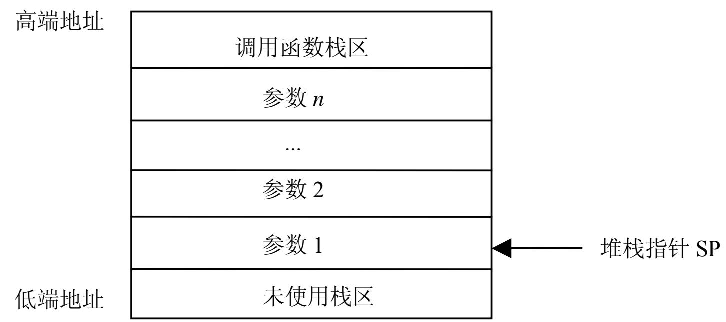


图 3-17　函数调用堆栈传递参数

参数入栈是按照函数声明的顺序从右向左进行的，先压入最右边的参数，然后依次向左入栈。最左边的两个参数如果无法用寄存器传递，那么也压入堆栈。

函数的返回值通过寄存器传递。安排如下：

（1）返回值为 8 位或 16 位时使用寄存器 R12。如 unsigned char、指针类型。

（2）返回值为 32 位时使用寄存器 R13:R12。如 float 类型。

（3）返回值为 64 位时使用寄存器 R15:R14:R13:R12 或 R11:R10:R9:R8。

（4）返回值为结构、数组或者联合时使用寄存器 R12，返回的是指向它们的指针。

下面举一个完整的例子，asm.s43 的功能是计算 arg1+arg2+arg3+arg4 并返回和。

```
/*-------汇编头文件 asm.h-------*/
unsigned char Func1(unsigned char arg1,unsigned int arg2,unsigned char arg3,
unsigned char arg4);
/*-------汇编头文件 asm.h 结束-------*/

/*-------汇编文件 asm.s43-------*/
PUBLIC Func1                //函数声明为 PUBLIC

Func1:
    ADD.B   R14, R12        //R12 和 R14 中存放 arg1 和 arg2
    ADD.B   0x2(SP), R12
    ADD.B   0x4(SP), R12    //arg3 和 arg4 在堆栈中
    RET                     //返回值在 R12 中

END
/*-------汇编文件 asm.s43 结束-------*/

/*-------c 文件 main.c-------*/
#include "asm.h"

int main( void )
{
```

```
    unsigned int iq0=1,iq1=2,iq2=3,iq3=4,iq4;

    iq4=Func1(iq0,iq1,iq2,iq3);
    return 0;
  }
  /*-------c 文件 main.c 结束-------*/
```

也可以看一看编译器编译完成后将 main.c 转换成汇编语言的形式，这样更有助于理解调用汇编的过程。在工程的 options 选项中选择 C/C++ compiler->List，使 Output assembler file 有效，如果希望在生成的汇编文件中包含原有的 C 代码作为注释，可以再使 Include Source 有效。重新编译一下，在工程所在的目录\Debug\List 下，将会出现文件 main.s43。

```
  /*-------文件 main.s43 -------*/
        NAME main

        RSEG CSTACK:DATA:SORT:NOROOT(1)

        EXTERN ?longjmp_r4
        EXTERN ?longjmp_r5
        EXTERN ?setjmp_r4
        EXTERN ?setjmp_r5

        PUBWEAK ?setjmp_save_r4
        PUBWEAK ?setjmp_save_r5
        PUBLIC main

        EXTERN Func1

        RSEG CODE:CODE:REORDER:NOROOT(1)
  main:
     MOV.W   #0x1, R12
     MOV.W   #0x2, R14
     MOV.W   #0x3, R13
     MOV.W   #0x4, R15
     PUSH.B  R15
     PUSH.B  R13
     CALL    #Func1
     MOV.B   R12, R15
     AND.W   #0xff, R15
     MOV.W   #0x0, R12
     ADD.W   #0x4, SP
```

```
    RET

       RSEG CODE:CODE:REORDER:NOROOT(1)
?setjmp_save_r4:
    REQUIRE ?setjmp_r4
    REQUIRE ?longjmp_r4

    RSEG CODE:CODE:REORDER:NOROOT(1)
?setjmp_save_r5:
    REQUIRE ?setjmp_r5
    REQUIRE ?longjmp_r5

    END
```

仔细研究这个文件，对于理解调用汇编模块的过程十分有帮助。从以上代码中可以清楚地看到 PUSH.B R15 和 PUSH.B R13 两句将后两个参数压入了堆栈。返回值保存在寄存器 R12 中。

在编写程序的时候，也可以使用这种方法。先编写一个很简单的 C 程序，保证使用的参数和返回值都与将来要编写的汇编语言程序一致，通过编译器将其转变为汇编语言程序。这样，将来要编写的汇编语言程序与 C 语言程序之间的接口就包含在转变好的汇编语言程序中，剩下的工作只要在这段汇编语言程序中找出保存变量的寄存器，添加自己编写的内容就可以了。

3.5　编写高质量的代码

编写高质量的代码不仅可以提高程序执行效率，缩减代码长度，而且对于保证程序的可靠性也是相当重要的。事实证明，低效冗长的代码更容易出现错误。一段高质量代码需要很多方面的综合配合，如明晰的项目需求、简洁合理的程序结构、出色的文档、高效的语句表达等，这是一个复杂的工程。这里只讨论如何编写出高效的表达语句。后面编程实例中会介绍如何安排好程序结构。对此类问题感兴趣的读者可以阅读软件工程方面的相关书籍。

微处理器一般用于特定环境和特定用途，出于成本、功耗和体积方面的考虑，一般都要求尽量节省使用资源。并且，由于微处理器的硬件一般都不支持有符号数、浮点数的运算，且运算位数有限，因此，分配变量时必须仔细。另外要说明的是，速度和存储器的消耗经常是两个不可兼顾的目标，多数情况下，编程者必须根据实际情况作出权衡和取舍。

需要注意的事项如下：

（1）通常在满足运算需求的前提下，尽量选择为变量定义字节数少的数据类型。

（2）尽量不用过长的数据类型，如 long long 和 double。

（3）MSP430 不支持位寻址，所以运算中尽量减少位操作。对于只有“是”和“否”

两种取值的变量，如果 RAM 容量允许，则可分配为 unsigned char 类型，这样可以提高运算速度。如果分配成某一字节中的某一位，则可以减少存储器的消耗，但会降低运算速度。

（4）避免使用浮点数，尽量使用定点数进行小数运算。如果必须使用浮点数，则尽量使用 32 位的 float，而不是 64 位的 double。

（5）尽量将变量分配为无符号数据类型。

（6）对于指针变量，如果声明后其值不再改变，则声明成 const 类型，这样编译器编译时能够更好地优化所生成的代码。

（7）尽可能使用局部变量而不是全局变量或者静态变量（static）。这样有利于编译器编译时更好地优化所生成的代码。

（8）避免对局部变量使用&取地址符。因为这样会使编译器无法把此变量放在 CPU 的寄存器中，而是放在 RAM 中，从而失去了优化的机会。

（9）仅在模块内使用的变量声明为 staic 类型，这样有利于编译器优化。

（10）如果堆栈空间有限，则尽量减少函数调用的层次和递归调用。

（11）如果传送参数过多，则可以将参数组成一个数组或者结构，然后用指针传递。

（12）某些变量在中断程序和普通级别程序中都会被用到，所以必须加以保护。将变量声明为 volatile 类型，编译器优化时就不会移动它，对它的访问也就不会被延迟。应该保证对 volatile 的变量的访问不被打断，为此，可以在访问它的部分加上__monitor 声明。

3.6 函 数 库

EW430 内包含两种库：DLIB 和 CLIB。

DLIB 是一个完整的 ISO/ANSI C 和嵌入式 C++库，它支持遵循 IEEE 754 格式的浮点数，并且可以根据不同的需要进行裁减，以提高效率。

CLIB 不完全兼容 ISO/ANSI C，也不支持 IEEE 754 格式的浮点数和嵌入式 C++。CLIB 是一个简便易用的库，在很多情况下比 DLIB 使用起来更容易，这是因为目前在 MSP430 这样存储器有限而总线又没有开放的芯片内，很多 C++的特性实际上难以发挥。

在 EW430 的编辑器内打开任何一个文件，然后输入函数的名字，按 F1 键就可以获得关于函数的帮助。需要注意的是，这些帮助来自 DLIB。如果想获得 CLIB 函数的帮助，可从 EW430 安装目录\430\doc\clib.pdf 获得。

下面列出 CLIB 的库函数并给出了使用说明，这些库函数来自 clib.pdf，供读者使用时查阅。说明中给出了函数的功能，解释了参数和返回值的含义，在描述中详细说明了函数的基本用法和用途。有些比较难以理解的函数还给出了使用的实例。

abort

void abort(void)

功能：非正常结束程序。

参数：无。

返回值：无。

描述：非正常结束程序，不返回到调用函数。

头文件：stdlib.h

abs

int abs(int j)

功能：求绝对值。

参数：j。

返回值：j 的绝对值。

描述：计算 j 的绝对值。

头文件：stdlib.h

acos

double acos(double arg)

功能：求反余弦值。

参数：arg　双精度浮点数，取值范围为[-1，+1]。

返回值：arg 的反余弦值，取值范围为[0，pi]。

描述：计算 arg 的反余弦值，单位为弧度。

头文件：math.h

asin

double asin(double arg)

功能：求反正弦值。

参数：arg　双精度浮点数，取值范围为[-1，+1]。

返回值：arg 的反正弦值，取值范围为[-pi/2，+pi/2]。

描述：计算 arg 的反正弦值，单位为弧度。

头文件：math.h

assert

void assert (int expression)

功能：检查表达式。

参数：expression　要检查的表达式。

返回值：无。

描述：这是一个检查表达式的宏。如果检查结果为假，则打印一条消息到 stderr 并调用 abort，结束程序运行。打印消息的格式为：File name; line num # Assertion failure "expression"。若想忽略 assert 调用，则在#include <assert.h>语句前一行放置#define DEBUG 语句。

头文件：assert.h

atan

double atan(double arg)

功能：求反正切值。

参数：arg　双精度浮点数。

返回值：arg 的反正切值，取值范围为[−pi/2，+pi/2]。

描述：计算 arg 的反正切值，单位为弧度。

头文件：math.h

atan2

double atan2(double arg1, double arg2)

功能：求反正切值。

参数：arg1　双精度浮点数。

　　　arg2　双精度浮点数。

返回值：arg1/arg2 的反正切值，取值范围为[−pi，+pi]。

描述：计算 arg1/arg2 的反正切值，根据 arg1 和 arg2 的符号确定返回值的象限。

头文件：math.h

atof

double atof(const char *nptr)

功能：将字符串转换成双精度浮点数。

参数：nptr　指向字符串的指针，字符串中包含 ASCII 形式的数字。

返回值：由字符串转换成的双精度浮点数。

描述：将字符串 nptr 转换成双精度浮点数，忽略空白字符并在遇到不可转换为数字的字符时停止转换。

头文件：stdlib.h

举例：

atof(" −3K") 的结果为 −3.00。

atof(".0006")的结果为 0.0006。

atof("1e−4")的结果为 0.0001。

atoi

int atoi(const char *nptr)

功能：将字符串转换成整数。

参数：nptr　指向字符串的指针，字符串中包含 ASCII 形式的数字。

返回值：由字符串转换成的整数。

描述：将字符串 nptr 转换成整数，忽略空白字符并在遇到不可转换为数字的字符时停止转换。

头文件：stdlib.h

举例：

atoi("−3K")的结果为−3。

atoi("6")的结果为 6。

atoi("149")的结果为 149。

atol

long atol(const char *nptr)

功能：将字符串转换成长整型数。

参数：nptr　指向字符串的指针，字符串中包含 ASCII 形式的数字。

返回值：由字符串转换成的长整型数。

描述：将字符串 nptr 转换成长整型数，忽略空白字符并在遇到不可转换为数字的字符时停止转换。

头文件：stdlib.h

举例：

atol("−3K")的结果为−3。

atol("6")的结果为 6。

atol("149")的结果为 149。

bsearch

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare)(const void *_key, const void *_base));

功能：搜索。

参数：key　指向要搜索对象的指针。

base　指向被搜索数组的指针。

nmemb　被搜索数组含有元素的个数。

size　被搜索数组每个元素占用的字节数。

compare　搜索时调用的比较函数，比较_key 与_ base 的值。对 bsearch 而言，*compare 的返回值为：

<0　　_key<_base

=0　　_key=_base

>0　　_key>_base

返回值：成功　返回指向 base 中并且与 key 指向对象相匹配的指针。

失败　返回空指针 NULL。

描述：在 base 指向的数组中搜索与 key 所指向的对象相匹配的元素。nmemb 是数组中元素的个数，如 int MoonRiver[5]中含有 5 个元素。size 是每个元素占用的字节，如 MoonRiver 中的元素是整数，为 2 字节。size_t 是 unsigned int 类型。

头文件：stdlib.h

calloc

void *calloc(size_t nelem, size_t elsize)

功能：内存分配。

参数：nelem　要分配的内存中元素的个数。

elsize　每个元素占用的字节数。

返回值：成功　指向所分配内存的首地址的指针。

失败　如果没有足够的内存，则返回 0。

描述：分配一块内存，包含 nelem 个元素，每个元素占用 elsize 个字节。为了确保 elsize 的值是正确的，应该使用 sizeof 操作符来获取。

头文件：stdlib.h

ceil

double ceil(double arg)

功能：计算大于等于 arg 的整数。

参数：arg　双精度浮点数。

返回值：大于等于 arg 的整数。

描述：计算大于等于 arg 的整数。

头文件：math.h

cos

double cos(double arg)

功能：计算余弦值。

参数：arg。

返回值：arg 的余弦值。

描述：计算 arg 的余弦值。

头文件：math.h

cosh

double cosh(double arg)

功能：计算双曲余弦值。

参数：arg。

返回值：arg 的双曲余弦值。

描述：计算 arg 的双曲余弦值。

头文件：math.h

div

div_t div(int numer, int denom)

功能：整数除法。

参数：number　分子。

　　　denom　分母。

返回值：number/denom 的结果，放在 div_t 的结构中。div_t 的定义为：

```
typedef struct
{
    int quot, rem;  // quot：商  rem：余数
} div_t;
```

描述：number 除以 denom，结果放在 div_t 的结构中。

头文件：math.h

exit

void exit(int status)

功能：正常终止程序运行。

参数：status。

返回值：无。

描述：正常终止程序运行，不会返回调用者。

头文件：stdlib.h

exp

double exp(double arg)

功能：计算 e 的 arg 次方。

参数：arg。

返回值：e 的 arg 次方。

描述：计算 e 的 arg 次方。

头文件：math.h

exp10

double exp10(double arg)

功能：计算 10 的 arg 次方。

参数：arg。

返回值：10 的 arg 次方。

描述：计算 10 的 arg 次方。

头文件：math.h

fabs

double fabs(double arg)

功能：计算绝对值。

参数：arg。

返回值：arg 的绝对值。

描述：计算 arg 的绝对值。

头文件：math.h

floor

double floor(double arg)

功能：计算小于等于 arg 的整数。

参数：arg。

返回值：小于等于 arg 的整数。

描述：计算小于等于 arg 的整数。

头文件：math.h

fmod

double fmod(double arg1, double arg2)

功能：计算浮点数除法的余数。

参数：arg1　分子。

arg2　分母。

返回值：arg1/arg2 的余数。

描述：计算 arg1/arg2 的余数。

头文件：math.h

free

void free(void *ptr)

功能：释放已分配的内存。

参数：ptr　指向一块内存的指针。

返回值：无。

描述：释放已分配的内存，这块内存是之前用 malloc、calloc 或 realloc 分配的。

头文件：stdlib.h

frexp

double frexp(double arg1, int *arg2)

功能：将浮点数的指数和尾数分离。

参数：arg1　被分离的浮点数。

　　　arg2　指向一个整数的指针，整数内放置分离出的 arg1 的指数部分。

返回值：arg1 的尾数部分，取值范围为[0.5，1.0]。

描述：将浮点数的指数和尾数分离，指数放在*arg2 中，尾数作为返回值返回。

头文件：math.h

getchar

int getchar(void)

功能：读字符。

参数：无。

返回值：读取的字符。

描述：从标准输入流中读一个 ASCII 格式的字符。用户需要根据硬件环境定制此函数，函数的源文件在 getchar.c 中。

头文件：stdio.h

gets

char *gets(char *s)

功能：读字符串。

参数：s　读取的字符串的指针。

返回值：指向从标准输入流中读取的字符串的指针，与 s 相同。

描述：从标准输入流中读取字符串，并放入 s 中。如果遇到行尾或者文件结尾，则字符串结束。行结束符被 0 取代。

头文件：stdio.h

isalnum

int isalnum(int c)

功能：检验一个字符是否是字母或者数字。

参数：c　代表字符的整数。

返回值：如果 c 是字母或者数字则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是字母或者数字。

头文件：ctype.h

isalpha

int isalpha(int c)

功能：检验一个字符是否是字母。

参数：c　代表字符的整数。

返回值：如果 c 是字母则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是字母。

头文件：ctype.h

iscntrl

int iscntrl(int c)

功能：检验一个字符是否是控制符。

参数：c　代表字符的整数。

返回值：如果 c 是控制码则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是控制码，如行结束符、回车符等。

头文件：ctype.h

isdigit

int isdigit(int c)

功能：检验一个字符是否是数字。

参数：c　代表字符的整数。

返回值：如果 c 是整数则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是数字。

头文件：ctype.h

isgraph

int isgraph(int c)

功能：检验一个字符是否是非空格可打印字符。

参数：c　代表字符的整数。

返回值：如果 c 是非空格可打印字符则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是非空格可打印字符。

头文件：ctype.h

islower

int islower(int c)

功能：检验一个字符是否是小写字母。

参数：c　代表字符的整数。

返回值：如果 c 是小写字母则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是小写字母。

头文件：ctype.h

isprint

int isprint(int c)

功能：检验一个字符是否是可打印字符（含空格）。

参数：c　代表字符的整数。

返回值：如果 c 是可打印字符（含空格）则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是可打印字符（含空格）。

头文件：ctype.h

ispunct

int ispunct(int c)

功能：检验一个字符是否是非空格、字母、数字的可打印字符。

参数：c　代表字符的整数。

返回值：如果 c 是非空格、字母、数字的可打印字符则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是非空格、字母、数字的可打印字符。

头文件：ctype.h

isspace

int isspace (int c)

功能：检验一个字符是否是空格字符。

参数：c　代表字符的整数。

返回值：如果 c 是空格字符则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是空格字符。空格字符可定义为：

空格	‘ ’
走纸	\f
换行	\n
回车	\r
水平制表符	\t
垂直制表符	\v

头文件：ctype.h

isupper

int isupper(int c)

功能：检验一个字符是否是大写字母。

参数：c　代表字符的整数。

返回值：如果 c 是大写字母则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是大写字母。

头文件：ctype.h

isxdigit

int isxdigit(int c)

功能：检验一个字符是否是十六进制数字。

参数：c　代表字符的整数。

返回值：如果 c 是十六进制数字则返回非 0 整数，否则返回 0。

描述：检验一个字符是否是十六进制数。不区分大小写。十六进制数包括 0～9、a～f、A～F 这些字符。

头文件：ctype.h

labs

long int labs(long int j)

功能：求绝对值。

参数：j。

返回值：j 的绝对值。

描述：计算 j 的绝对值。

头文件：stdlib.h

ldexp

double ldexp(double arg1,int arg2)

功能：乘以 2 的幂。

参数：arg1　乘数。

　　　arg2　次方。

返回值：arg1　乘以 2 的 arg2 次方的值。

描述：计算 arg1 乘以 2 的 arg2 次方的值。

头文件：math.h

ldiv

ldiv_t ldiv(long int numer, long int denom)

功能：长整数除法。

参数：number　分子。

　　　denom　分母。

返回值：number/denom 的结果，放在 ldiv_t 的结构中。ldiv_t 的定义为：

```
typedef struct
{
    long quot, rem;  // quot:商 rem: 余数
}ldiv_t;
```

描述：number 除以 denom，结果放在 ldiv_t 的结构中。

头文件：math.h

log

double log(double arg)

功能：计算自然对数的值。

参数：arg。

返回值：arg 的自然对数值。

描述：计算 arg 的自然对数的值。

头文件：math.h

log10

double log10(double arg)

功能：计算以 10 为底的常用对数的值。

参数：arg。

返回值：arg 的以 10 为底的常用对数值。

描述：计算 arg 的以 10 为底的常用对数值。

头文件：math.h

longjmp

void longjmp(jmp_buf env, int val)

功能：长跳转。

参数：env　为 jmp_buf 结构。调用 setjmp 函数时，将环境值保存在 jmp_buf 结构中，作为调用 longjmp 函数的参数。

val　调用 setjmp 函数时的返回值。

返回值：无。

描述：恢复调用 setjmp 函数时保存在 jmp_buf 结构中的环境值，使程序从调用 setjmp 函数的那个位置开始执行。

头文件：setjmp.h

malloc

void *malloc(size_t size)

功能：分配内存。

参数：size　分配对象占用的字节数，size_t 为双字节无符号整数。

返回值：成功　被分配对象的首字节地址。

失败　如果没有足够的内存空间分配，则返回 0。

描述：为大小为 size 的对象分配内存。分配的内存放置在堆（heap）中。

头文件：stdlib.h

memchr

void *memchr(const void *s, int c, size_t n)

功能：在内存对象中搜索字符。

参数：s　指向内存对象的指针。

c　代表字符的整数。

n　对象占用的字节数。

返回值：成功　指向首次出现 c 的地址指针。

失败　NULL。

描述：在 s 指向的内存对象中搜索首次出现的 c。s 中所有的字符都被当作无符号字符处理。

头文件：string.h

memcmp

int memcmp(const void *s1, const void *s2, size_t n)

功能：比较内存。

参数：s1　指向第一个对象的指针。

s2　指向第二个对象的指针。

n　比较的字符数。

返回值：>0　s1>s2

=0　s1=s2

<0　s1<s2

描述：比较两个对象前 n 个字符的内容是否一致。

头文件：string.h

memcpy

void *memcpy(void *s1, const void *s2, size_t n)

功能：复制字符。

参数：s1　指向目标对象的指针。

s2　指向源对象的指针。

n　复制的字符数。

返回值：s1。

描述：从 s2 复制 n 个字符到 s1，如果对象交叠，则结果是不确定的，必要时应用 memmove 代替。

头文件：string.h

memmove

void *memmove(void *s1, const void *s2, size_t n)

功能：移动字符。

参数：s1　指向目标对象的指针。

s2　指向源对象的指针。

n　复制的字符数。

返回值：s1。

描述：从 s2 复制 n 个字符到 s1。与 memcpy 的过程不同，这里先将 s2 要复制的内容复制到一个缓冲区内，然后再从缓冲区复制到 s1，这样就不会像 memcpy 那样产生交叠。s1、s2 可以指向同一个对象。

头文件：string.h

memset

void *memset(void *s, int c, size_t n)

功能：填充内存。

参数：s　指向要填充的内存对象的指针。

c　要填充的字符。

n　要填充的字符数。

返回值：s。

描述：将 s 指向的对象的前 n 个字符填充为 c，填充前 c 被转换为无符号字符。

头文件：string.h

modf

double modf(double value, double *iptr)

功能：分离小数和整数部分。

参数：value　要分离的数。

iptr　分离后的整数部分放在 iptr 所指的双精度浮点数中。

返回值：分离出的小数部分。

描述：将 value 的小数和整数部分分离，两部分的符号都与 value 一样。

头文件：math.h

pow

double pow(double arg1, double arg2)

功能：计算乘方值。

参数：arg1， arg2。

返回值：arg1 的 arg2 次方值。

描述：计算 arg1 的 arg2 次方值。

头文件：math.h

printf

int printf(const char *format, …)

功能：格式化输出。

参数：format　指向格式化字符串的指针。

...　在 format 控制下待打印的数据。

返回值：成功　输出的字符数。

失败　负值。

描述：将数据按照 format 指定的格式输出到标准输出流。完整的格式化输出会消耗大量的内存，所以，应该根据需要选择不同的格式化程序（见工程设置中 library options 的选择）。format 的转换形式为：%[flafs][field_width][.precision][length_modifier] conversion，[ ]内的项可选解释如下：

flags　内容如表 3-5 所示。

field_width　指输出的最大字符数。如有必要，可用空格填充。负数表示左对齐。使用* 的含义是相应的下一个参数（必须为整数）指出输出字符数。

Precision　对于整数（d、i、o、u、x、X）是输出的数字个数。对于浮点数（e、E、f）是输出的小数的位数。对于 G 和 g 形式的转换是小数点后有效数字的个数。

表 3-5　**flags** 所代表的内容

Flag	效　果
−	左对齐
+	有符号数均以正负号开头
空格	数值总是以符号或者空格开头
#	选择格式转换： octal　首位数字总是 0 G,g　输出小数点和小数点后的 0 E,e,f　输出小数点，即使小数点后没有数字也如此 X　非 0 值前面加 0X
X	非 0 值前面加 0X
0	用 0 填充 field_width （对于 d、 i、o、u、x、 X、 e、 E、 f、 g、G 有效）

length_modifier　内容如表 3-6 所示。

表 3-6　**length_modifier** 所代表的内容

length_modifier	用　途
h	在 d、 i、 u、 x、 X 之前表示短整型或者无符号短整型值
l	在 d、 i、 u、 x、 X 之前表示长整型或者无符号长整型值
L	在 E、 f、 g、 G 之前表示长双精度值

Conversion　内容如表 3-7 所示。

表 3-7　　conversion 所代表的内容

conversion	结　　果
d	有符号十进制数
i	有符号十进制数
o	无符号八进制数
u	无符号十进制数
x	无符号十六进制数，使用小写（0～9、a～f）
X	无符号十六进制数，使用大写（0～9、A～F）
e	双精度值，形式为[-]d.ddde+dd
E	双精度值，形式为[-]d.dddE+dd
f	双精度值，形式为[-]ddd.ddd
g	双精度值，形式取 f 或者 e 中更合适的双精度值
G	双精度值，形式取 F 或者 E 中更合适的双精度值
C	单字符常数
s	字符串常数
p	指针（地址）
n	无输出。但将被写的字符数保存在下一个参数所指的整数中
n%	%字符

注意：转换规则把所有字符和短整型参数转换为整型，把浮点数转换为双精度数。

pringf 调用 putchar 函数，putchar 函数必须根据硬件重新定制。printf 的源代码在文件 printf.c 中，长度较小且消耗堆栈空间较少的简化版本在 intwri.c 中。

头文件：stdio.h

举例。运行下列程序：

```
int i=6, j=-6;
char *p = "ABC";
long l=100000;
float f1=0.0000001;
f2=750000;
double d=2.2;
```

调用不同的 printf 函数的结果如表 3-8 所示，表中的 0 表示空格。

表 3-8　　执行结果

语　　句	输　　出	字　符　数
printf("%c",p[1])	B	1
printf("%d",i)	6	1

续表

语　　句	输　　出	字　符　数
printf("%3d",i)	^{00}6	3
printf("%.3d",i)	006	3
printf("%-10.3d",i)	0060000000	10
printf("%10.3d",i)	0000000006	10
printf("Value=%+3d",i)	Value=0+6	9
printf("%10.*d",i,j)	000-000006	10
printf("String=[%s]",p)	String=[ABC]	12
printf("Value=%lX",l)	Value=186A0	11
printf("%f",f1)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",f1)	1.000000e-07	12
printf("%16e",d)	00002.200000e+00	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",f1)	1e-07	5
printf("%g",f2)	750000	6
printf("%g",d)	2.2	3

putchar

int putchar(int value)

功能：输出一个字符到标准输出。

参数：value　代表字符的整数。

返回值：成功　value。

　　　　失败　EOF。

描述：输出一个字符到标准输出。用户需要根据硬件环境定制此函数，函数的源文件在 putchar.c 中。printf 函数调用此函数。

头文件：stdio.h

puts

int puts(const char *s)

功能：输出字符串和换行符到标准输出流。

参数：s　指向字符串的指针。

返回值：成功　非负值。

　　　　失败　-1。

描述：输出字符串和换行符到标准输出流。

头文件：stdio.h

qsort

void qsort (const void *base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base));

功能：排序。

参数：base　指向要排序的数组的指针。

nmemb　数组的元素个数。

size　数组中每个元素占用的字节数。

compare　比较时调用的函数，比较_key 与_base 的值。对*compare 而言返回值为：

<0　_key<_base

=0　_key=_base

>0　_key>_base

返回值：无。

描述：对 base 所指的数组中的元素进行排序。

头文件：stdlib.h

rand

int rand(void)

功能：产生伪随机数。

参数：无。

返回值：伪随机数序列的下一个整数。

描述：输出伪随机数序列中下一个数值，取值范围为[0，最大随机数]，为伪随机序列设置初值的方法参见 srand 函数。

头文件：stdlib.h

realloc

void *realloc(void *ptr, size_t size)

功能：改变已分配内存的大小。

参数：ptr　指向要改变大小的内存的指针。

size　内存占用的字节数。

返回值：成功　指向内存首地址的指针。

失败　如果没有足够的内存空间用来分配，则返回 Null。

描述：改变已分配内存的大小，这些内存之前必须是用 malloc、calloc、realloc 分配的。

头文件：stdlib.h

scanf

int scanf(const char *format, …)

功能：格式化输入。

参数：format　指向格式化字符串的指针。

...　可选项。指向准备接收数据的变量的指针。

返回值：成功　输入的字符数。

失败　负值。

描述：从标准输入流读数据，并按照 format 指定的格式进行转换。完整的格式化输入会消耗大量的内存，所以，应该根据需要选择不同的格式化程序（见工程设置中 library options 的选择）。format 的转换形式为：%[assign_suppress][field_width][length_modifier]conversion，[ ]内的项为可选项。

assign_suppress　如果为*，则进行扫描，但不做任何输出。

field_width　输入的最大字符数。默认情况下，一直扫描直到发生不匹配的情况。

length_modifier　内容如表 3-9 所示。

conversion　内容如表 3-10 所示。

表 3-9　**length_modifier** 所代表的内容

<table>
<tr><th>length_modifier</th><th>在...之前</th><th>意　义</th></tr>
<tr><td rowspan="3">l</td><td>d、i 或 n</td><td>长整型而非整型</td></tr>
<tr><td>o、u 或 x</td><td>无符号长整型而非无符号整型</td></tr>
<tr><td>e、E、g、G 或 f</td><td>双精度浮点数而非单精度浮点数</td></tr>
<tr><td rowspan="2">h</td><td>d、i 或 n</td><td>短整型而非整型</td></tr>
<tr><td>o、u 或 x</td><td>无符号短整型而非无符号整型</td></tr>
<tr><td>L</td><td>e、E、g、G 或 f</td><td>长双精度浮点数而非单精度浮点数</td></tr>
</table>

表 3-10　**conversion** 所代表的内容

conversion	含　义
d	可选的有符号十进制整数值
i	可选的有符号整数值，格式按照 C 语言标准：十进制、八进制（0n）、十六进制（0xn 或 0Xn）
o	可选的有符号八进制整数值
u	无符号十进制整数值
x	可选的有符号十六进制整数值
X	可选的有符号十六进制整数值（等价于 x）
f	浮点常数
e E g G	浮点常数（等价于 f）
s	字符串
c	一个或者 field_width 个字符
n	不真正输入，但将读入的字符数保存在下一个参数所指的整数中
p	地址指针
[	任意数目的字符都可以与]之前的任意字符匹配。如：[abc]可认为是 a、b 或者 c
[ ]	任意数目的字符与在]之前的字符匹配。如：[ ]abc]可认为是]、a、b 或者 c
[^	任意数目的字符都可以与]之前的任意字符不匹配。如：[^abc]认为不是 a、b 或者 c
[^]	任意数目的字符与在]之前的字符不匹配。如：[^abc] 认为不是]、a、b 或者 c
%	%字符

转换时除了 c、n 和[、[ ]、[^、[^]，前导空格符均被跳过。scanf 直接调用 getchar

函数，gerchar 函数必须按照所使用的硬件重新定制。

头文件：stdio.h

举例。执行下列程序。

```
int n, i;
char name[50];
float x;
n=scanf("%d%f%s", &i, &x, name)
对于输入内容:
25 54.32E-1 Hello World
```

结果为：

```
n=3, i=25, x=5.432, name="Hello World"
```

执行下列程序：

```
scanf("%2d%f%*d %[0123456789]", &i, &x, name)
```

对于输入内容：

```
56789 0123 56a72
```

结果为：

```
i=56, x=789.0, name="56" (0123 未赋值)
```

setjmp

int setjmp(jmp_buf env)

功能：设置跳转返回点。

参数：env　保存返回点的环境，以便返回时恢复。jmp_buf 是一个数据结构。

返回值：0。

描述：设置跳转返回点，保存返回点的环境。setjmp 必须与相应的 longjmp 调用在同一函数中或者在高于 longjmp 调用的嵌套中。

头文件：setjmp.h

sin

double sin(double arg)

功能：计算正弦值。

参数：arg。

返回值：arg 的正弦值。

描述：计算正弦值，单位为弧度。

头文件：math.h

sinh

double sinh(double arg)

功能：计算双曲正弦值。

参数：arg。

返回值：arg 的双曲正弦值。

描述：计算双曲正弦值，单位为弧度。

头文件：math.h

sprintf

int sprintf(char *s, const char *format, …)

功能：格式化输出到字符串。

参数：s　指向接收格式化数据字符串的指针。

format　指向格式化字符串的指针。

...　可选项，在 format 控制下将被打印。

返回值：成功　输出的字符个数。

失败　负值。

描述：格式化输出到字符串。使用方式与 printf 相同，只是输出目标为字符串。sprintf 不调用 putchar，因此即使 putchar 没有根据硬件重新定制，也可以使用。

头文件：stdio.h

sqrt

double sqrt(double arg)

功能：计算平方根值。

参数：arg。

返回值：arg 的平方根值。

描述：计算 arg 的平方根值。

头文件：math.h

srand

void srand(unsigned int seed)

功能：设置伪随机数序列。

参数：seed　选择特定的伪随机数序列。

返回值：无。

描述：选择一个伪随机数序列。函数 rand 从伪随机数序列中取随机数，如果调用 rand 之前没有调用 srand 函数，则所选择的随机数序列相当于调用 srand（1）选择的随机数序列。

头文件：stdlib.h

sscanf

int sscanf(const char *s, const char *format, …)

功能：格式化输入字符串。

参数：s　指向要格式化输入的字符串的指针。

format　指向格式化字符串的指针。

...　可选项。指向准备接收数据的变量的指针。

返回值：成功　输入的字符数。

失败　负值。

描述：从字符串中格式化输入。除了是从字符串中读取外，其他工作完全与 scanf 一样。sscanf 不调用 getchar，因此，即使 getchar 没有根据硬件重新定制也可以使用。

头文件：stdio.h

strcat

char *strcat(char *s1, const char *s2)

功能：连接字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针

返回值：s1。

描述：复制第二个字符串的内容到第一个字符串的后面。第二个字符串的第一个字符将第一个字符串原来的终结符覆盖。

头文件：string.h

strchr

char *strchr(const char *s, int c)

功能：在字符串中搜索字符。

参数：s　指向字符串的指针。

　　　c　要搜索的字符。

返回值：成功　指向字符串中第一次出现 c 的指针。

　　　　失败　没有找到 c，则返回 null。

描述：在 s 所指的字符串中搜索第一个出现的 c 字符。字符串的终结符也被认为是字符串的一部分。

头文件：string.h

strcmp

int strcmp(const char *s1, const char *s2)

功能：比较字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

返回值：比较的结果：

返回值	比较结果
>0	s1>s2
=0	s1=s2
<0	s1<s2

描述：比较两个字符串。

头文件：string.h

strcoll

int strcoll(const char *s1, const char *s2)

功能：比较字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

返回值：比较的结果：

返回值	比较结果
>0	s1>s2

=0　　　　　　s1=s2

<0　　　　　　s1<s2

描述：比较两个字符串。功能与 strcmp 一样，此函数为了满足兼容性而提供。

头文件：string.h

strcpy

char *strcpy(char *s1, const char *s2)

功能：复制字符串。

参数：s1　指向目的字符串的指针。

　　　s2　指向要复制字符串的指针。

返回值：s1。

描述：复制 s2 到 s1。

头文件：string.h

strcspn

size_t strcspn(const char *s1, const char *s2)

功能：搜索字符串中完全不含另一个字符串中字符的最长部分。

参数：s1　指向被搜索字符串的指针。

　　　s2　指向搜索字符串的指针。

返回值：s1 中完全不含 s2 中字符的最长部分的字节数。

描述：搜索 s1 字符串中完全不含 s2 字符串中字符的最长部分。

头文件：string.h

strerror

char * strerror (int errnum)

功能：返回错误消息字符串。

参数：errnum　要返回的错误的号码。

返回值：如表 3-11 所示。

描述：根据错误号码，返回错误的字符串信息。

表 3-11　　strerror 函数返回值

错误号码	返回字符串
EZERO	no error
EDOM	domain error
ERANGE	range error
errnum<0||errnum>Max_err_num	unknow error
其余号码	error No. errnum

头文件：string.h

strlen

size_t strlen(const char *s)

功能：计算字符串的长度。

参数：s　指向字符串的指针。

返回值：s 为所占用的字节数。

描述：计算 s 所占用的字节数。

头文件：string.h

strncat

char *strncat(char *s1, const char *s2, size_t n)

功能：将一个字符串的一部分添加到另一个字符串上。

参数：s1　指向目标的字符串的指针。

　　　s2　指向源字符串的指针。

　　　n　　添加的字符个数。

返回值：s1。

描述：从 s2 中复制不超过 n 个字符到 s1 的末尾。

头文件：string.h

strncmp

int strncmp(const char *s1, const char *s2, size_t n)

功能：部分比较两个字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

　　　n　　比较的字符个数。

返回值：

返回值	含义
>0	s1>s2
=0	s1=s2
<0	s1<s2

描述：比较两个字符串前 n 个字符。

头文件：string.h

strncpy

char *strncpy(char *s1, const char *s2, size_t n)

功能：将指定长度的字符从一个字符串复制到另一个字符串中。

参数：s1　指向目标字符串的指针。

　　　s2　指向源字符串的指针。

　　　n　　复制字符的个数。

返回值：s1。

描述：将 s2 中前 n 个字符复制到 s1 中。

头文件：string.h

strpbrk

char *strpbrk(const char *s1, const char *s2)

功能：搜索一个字符串中的字符是否在另一个字符串中。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

返回值：成功　指向在 s1 中第一次出现 s2 中字符的地址的指针。

　　　　失败　如果没有搜索到，则返回 null。

描述：在 s1 中搜索是否有 s2 中的字符。

头文件：string.h

strrchr

char *strrchr(const char *s, int c)

功能：从字符串右端开始搜索字符。

参数：s　指向被搜索的字符串的指针。

　　　c　要搜索的字符。

返回值：成功　指向最先搜索到的 c 字符的地址的指针。

　　　　失败　没有搜索到，则返回 null。

描述：从字符串 s 的右端开始搜索字符 c。字符串的末尾终结符也被认为是字符串的一部分。

头文件：string.h

strspn

size_t strspn(const char *s1, const char *s2)

功能：在一个字符串中搜索另一个字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

返回值：s1 中包括 s2 的初始段的长度。

描述：找出 s1 中完全包括 s2 的初始段。

头文件：string.h

strstr

char *strstr(const char *s1, const char *s2)

功能：在一个字符串中搜索另一个字符串。

参数：s1　指向第一个字符串的指针。

　　　s2　指向第二个字符串的指针。

返回值：成功　指向 s1 中第一次出现 s2（不包含字符串结束符）的地址的指针。

　　　　失败　如果没有搜索到或者 s2 为一长度为 0 的字符串，则返回 null。

描述：在 s1 中搜索 s2 第一次出现的位置。

头文件：string.h

strtod

double strtod(const char *nptr, char **endptr)

功能：将字符串转换为双精度浮点数。

参数：nptr　指向字符串的指针。

　　　endptr　指向字符串指针的指针。

返回值：成功　返回由 nptr 转换来的双精度浮点数，endptr 指向常数后的第一个字符。

　　　　失败　返回 0，endptr 指向第一个非空格字符。

描述：将一个 ASCII 格式的字符串转换成双精度浮点数，忽略所有前导空格。endptr 用

来作诊断。

头文件：stdlib.h

strtok

char *strtok(char *s1, const char *s2)

功能：拆分字符串。

参数：s1　指向要被拆分的字符串。

s2　指向标记字符串。

返回值：成功　指向标记的指针。

失败　0。

描述：在 s1 中搜索 s2，s2 作为拆分 s1 的标记。第一次调用 strtok 时，s1 为要拆分的字符串，strtok 保存 s1。再次调用 strtok 时，s1 为 null，strtok 在它保存的字符串中搜索标记字符串 s2。s2 可以与前一次调用时使用的不同。如果搜索到一个标记，则返回指向 s1 中找到的标记的第一个字符的地址指针，否则，返回 null。如果标记不在字符串末尾，则 strtok 用 null 字符取代标记。

头文件：string.h

strtol

long int strtol(const char *nptr, char **endptr, int base)

功能：转换字符串为长整数。

参数：nptr　指向字符串的指针。

endptr　指向字符串指针的指针。

base　转换的基数。

返回值：成功　返回由 nptr 转换来的长整数，endptr 指向常数后的第一个字符。

失败　返回 0，endptr 指向第一个非空格字符。

描述：将一个 ASCII 格式的字符串转换成长整数，忽略所有前导空格。endptr 用来作诊断。base 必须在 2～36 之间。如果 base 是 0，则转换结果为普通的长整数，否则，转换结果将根据 base 的值，产生由数字和字母组成的数值。字母[a，z]和[A，Z]代表 10～35。如果 base 是 16，则 0x 可以作为十六进制整数的前导。

头文件：stdlib.h

strtoul

unsigned long int strtoul(const char *nptr, char **endptr,base int)

功能：将字符串转换为无符号长整形值。

参数：nptr　指向字符串的指针。

endptr　指向字符串指针的指针。

base　转换的基数。

返回值：成功　返回由 nptr 转换来的无符号长整数，endptr 指向常数后的第一个字符。

失败　返回 0，endptr 指向第一个非空格字符。

描述：将一个 ASCII 格式的字符串转换成无符号长整数，忽略所有前导空格。endptr 用来作诊断。base 必须在 2～36 之间。如果 base 是 0，则转换结果为普通的无符号长整数，否则，转换结果将根据 base 的值，产生由数字和字母组成的数值。

字母[a，z]和[A，Z]代表 10～35。如果 base 是 16，则 0x 可以作为十六进制整数的前导。

头文件：stdlib.h

strxfrm

size_t strxfrm(char *s1, const char *s2, size_t n)

功能：转换字符串，并返回字符串长度。

参数：s1　指向转换后的字符串的指针。

s2　指向待转换的字符串的指针。

n　 s1 中的最大字符数。

返回值：转换后字符串的长度，不包括结束的空字符（null）。

描述：转换过程为，如果使用 strcmp 函数比较结果字符串，那么它返回 strcoll 函数的返回值。

头文件：string.h

tan

double tan(double arg)

功能：计算正切值。

参数：arg。

返回值：arg 的正切值。

描述：计算 arg 的正切值。

头文件：math.h

tanh

double tanh(double arg)

功能：计算双曲正切值。

参数：arg。

返回值：arg 的双曲正切值。

描述：计算 arg 的双曲正切值。

头文件：math.h

tolower

int tolower(int c)

功能：转换为小写字符。

参数：c　代表字符的整数。

返回值：代表转换完成的小写字符的整数值。

描述：转换字符 c 为小写字符。

头文件：ctype.h

toupper

int toupper(int c)

功能：转换为大写字符。

参数：c　代表字符的整数。

返回值：代表转换完成的大写字符的整数值。

描述：转换字符 c 为大写字符。

头文件：ctype.h

va_arg

type va_arg(va_list ap, mode)

功能：函数调用时扩展参数。

参数：ap　va_list 类型的值。

　　mode　类型名称，如指向一个对象的指针。

返回值：见描述。

描述：函数调用时扩展参数。这是一个宏，用来扩展表达式，以使其与下一个被传递的参数具有相同的类型和值。被 va_start 初始化后，parmN 指明参数。va_arg 依次通过 ap 扩展参数。

头文件：stdarg.h

va_end

void va_end(va_list ap)

功能：停止读函数调用参数。

参数：ap　va_list 类型的值。

返回值：见描述。

描述：这是一个宏，可使程序从函数正常返回，函数的可变参数表由 va_start 指明，va_start 初始化了 va_list 类型的 ap。

头文件：stdarg.h

va_list

char *va_list[1]

功能：参数列表。

参数：无。

返回值：见描述。

描述：参数列表类型。为保存 va_end 和 va_arg 需要的信息的数组。

头文件：stdarg.h

va_start

void va_start(va_list ap, parmN)

功能：开始读函数调用参数。

参数：ap　va_list 类型的值。

　　parmN　函数定义的变量参数列表最右边参数的标识。

返回值：见描述。

描述：开始读函数调用参数。是一个初始化 ap 的宏，ap 被 va_end 和 va_arg 使用。

头文件：stdarg.h

vprintf

int vprintf(const char * format, va_list argptr)

功能：格式化输出。

参数：format　指向 format 字符串的指针。

　　argptr　参数列表。

返回值：成功　输出的字符数。

　　　　失败　负值。

描述：格式化输出到标准输出流。与 printf 的功能相同，只是参数是由指向一个 va_list 结构的指针传递的，而不是参数本身。

头文件：stdio.h

vsprintf

int vsprintf(char * s, const char * format, va_list argptr)

功能：格式化输出到缓冲区。

参数：s　指向接受转换结果的缓冲区的指针。

　　format　指向 format 字符串的指针。

　　argptr　参数列表。

返回值：成功　输出的字符数。

　　　　失败　负值。

描述：格式化输出到缓冲区。与 sprintf 的功能相同，只是参数是由指向一个 va_list 结构的指针传递的，而不是参数本身。

头文件：stdio.h

_formatted_read

int _formatted_read (const char **line, const char **format,va_list ap)

功能：读格式化数据。

参数：line　指向要扫描的数据的指针的指针。

　　format　指向标准 scanf 格式说明字符串的指针的指针。

　　ap　指向 va_list 类型的可变参数列表的指针。

返回值：成功读取数据的数量。

描述：读格式化数据。此函数是 scanf 的基本格式程序，是可重入函数（reentrant）。_formatted_read 函数运行需要 ANSI 特定的宏，其定义在 stdarg.h 中。需要特别注意是：

- 必须有 va_list 类型的 ap 变量
- 调用_formatted_read 前必须先调用 va_start 函数
- 结束当前内容时必须调用 va_end 函数
- 函数 va_start 的参数必须是针对可变参数列表左边形式参数

头文件：icclbutl.h

_formatted_write

int _formatted_write (const char *format, void outputf (char,void *), void *sp, va_list ap)

功能：写格式化数据。

参数：format　指向标准 printf/sprintf 格式说明字符串的指针的指针。

　　outputf　指向由_formatted_write 创建的实际完成单个字符写的函数的指针。函数的第一个参数是要写的字符的值。第二个参数总是与_formatted_write 第三个参数的值相同。

　　sp　底层函数需要的指向一些数据结构的指针。即使不需要，这个参数也要用

（void*）0 声明。在输出函数中也要声明。

ap　指向 va_list 类型的可变参数列表的指针。

返回值：成功写的数量。

描述：写格式化数据。此函数是 prinf 和 sprintf 的基本格式程序。通过它的通用接口可以很容易地向非标准显示器内写数据。_formatted_write 是可重入函数（reentrant）。运行时需要 ANSI 特定的宏，其定义在 stdarg.h 中。需要特别注意的是：

- 必须有 va_list 类型的 ap 变量
- 调用_formatted_write 前必须先调用 va_start 函数
- 结束当前内容时必须调用 va_end 函数
- 函数 va_start 的参数必须是针对可变参数列表左边形式参数

头文件：icclbutl.h

_medium_read

int _medium_read (const char **line, const char **format,va_list ap)

功能：读格式化数据。

参数：line　指向要扫描的数据的指针的指针。

format　指向标准 scanf 格式说明字符串的指针的指针。

ap　指向 va_list 类型的可变参数列表的指针。

返回值：成功读的数量。

描述：读格式化数据，但不包含浮点数。这是_formatted_read 的简化版本。所占用的程序空间是_formatted_read 的一半。更多的信息参见_formatted_read。

头文件：icclbutl.h

_medium_write

int _medium_write (const char *format, void outputf(char,void *), void *sp, va_list ap)

功能：写格式化数据。

参数：format　指向标准 printf/sprintf 格式说明字符串的指针的指针。

outputf　指向由_formatted_write 创建的实际完成单个字符写的函数的指针。函数的第一个参数是要写的字符的值。第二个参数总是与_formatted_write 第三个参数的值相同。

sp　底层函数需要的指向一些数据结构的指针。即使不需要，这个参数也要用（void*）0 声明。在输出函数中也要声明。

ap　指向 va_list 类型的可变参数列表的指针。

返回值：成功写的数量。

描述：写格式化数据，但不包含浮点数。这是_formatted_read 的简化版本。所占用的程序空间是_formatted_writ 的一半。更多的信息参见_formatted_writ 函数。

头文件：icclbutl.h

_small_write

int _small_write (const char *format, void outputf(char, void*), void *sp, va_list ap)

功能：写格式化数据。

参数：format　指向标准 printf/sprintf 格式说明字符串的指针的指针。

outputf　指向由_formatted_write 创建的实际完成单个字符写的函数的指针。函数的第一个参数是要写的字符的值。第二个参数总是与_formatted_write 第三个参数的值相同。

sp　底层函数需要的指向一些数据结构的指针。即使不需要，这个参数也要用（void*）0 声明。在输出函数中也要声明。

ap　指向 va_list 类型的可变参数列表的指针。

返回值：成功写的数量。

描述：写格式化数据，是_formatted_writ 的更简化的版本，只支持%%、%d、%o、%c、%s 和%x 几种类型。它不支持 filed_width 和 precision 参数，并且对于不支持的类型不进行诊断。更多的信息参见_formatted_writ 函数。

头文件：icclbutl.h

第4章 开发工具

要设计一个完整的单片机系统，开发工具是必不可少的。很多型号的单片机开发工具都很昂贵，导致使用这种单片机的门槛比较高，很大程度上限制了这种单片机的普及。学习过其他单片机的读者或许会有这样的体会，对于一些单片机来说，有机会学习但很难找到实践的机会。

MSP430 不是这样，开发 MSP430 所需要的工具费用很低，学习者不必为没有机会实际运行自己编写的程序而感到遗憾。本章介绍 MSP430 的全套开发工具，使读者可以全面了解开发 MSP430 单片机系统所需要的工具。

4.1 JTAG 仿真器、编程器

MSP430 的开发工具非常简便。MSP430 内部集成了遵循边界扫描故障诊断协议（IEEE1149）的电路，通常称为 JTAG。芯片内部 JTAG 对外的端口称为 JIAG 端口，该端口是一个双向串行端口。通过它可以控制 MSP430 的运行、读写内部寄存器的值、刷新 FLASH 的内容。因此，将 JTAG 电路集成到 CPU 内就相当于将仿真器集成到了 CPU 内部，只需要一个接口电路，将 JTAG 信号转送到调试终端（通常为 PC）就可以了。沿用以前的习惯，通常称这种接口电路为 JTAG 仿真器。

传统的仿真器一般采用为仿真特殊设计的 CPU 或者用 FPGA 模拟目标 CPU 的方法，这样做的最大缺点是仿真器成本高，导致开发和学习的门槛高。另外，无论使用哪一种方法，都与最终产品实际使用的 CPU 有差别，因此，会出现在仿真器上工作正常，而换用普通 CPU 工作却不正常的现象。

JTAG 仿真器就没有上述困扰。因为 JTAG 仿真器实际上仅是一个接口电路，所以，成本非常低，TI 公司给出了 MSP430 JTAG 仿真器的电路图，用户可以自制。CPU 本身带有仿真端口，所以不存在开发与实际应用时使用不同 CPU 的问题。JTAG 还是向 MSP430 内部下载程序的通道，对于不需要加密的程序就可以不需要编程器了，这样也降低了用户的成本。

JTAG 仿真器当然也有缺点，其可设置的断点数目有限，而且，与很高端的传统仿真器相比较，其程序跟踪能力不足。另外，JTAG 端口会占用管脚或者管脚需使用复用方式。

使用 JTAG 仿真器的时候，用户需要在设计目标板时预留 JTAG 接口，TI 公司给出了 JTAG 口引线的定义，如表 4-1 所示。

表 4-1　　JTAG 端口定义

管脚	1	3	5	7	9	11	13
定义	TDO	TDI	TMS	TCK	GND	RESET	NC
管脚	2	4	6	8	10	12	14
定义	VCC	NC	NC	TEST	NC	NC	NC

其中 1、2...14 为插头引脚号，对应于图 4-1JTAG 端口插头示意图。TDO、TDI、TMS、TCK、RESET、GND 为必需的。对于管脚少的芯片，如 F110、F123，没有多余的管脚供 JTAG 端口单独使用，则通过 TEST 管脚完成复用功能。VCC 连接仿真器与目标板的电源，除非目标板与仿真器各自独立供电，否则应当保留。早期的 JTAG 端口还有一个引脚 XOUT，现在已经废弃，可以不考虑。

用户按照图 4-2 的方式将 PC、仿真器、目标板连好，上电，就可以仿真调试了。

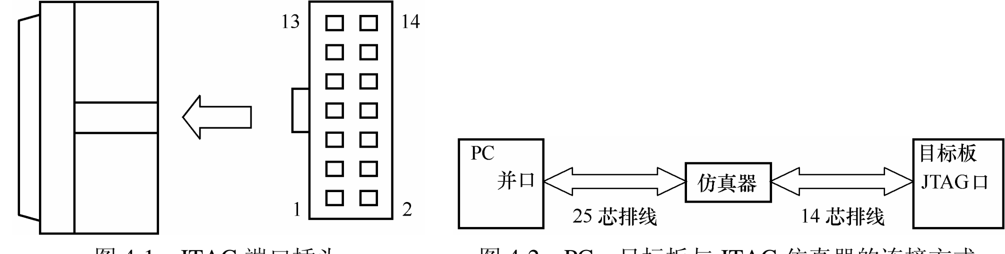


图 4-1　JTAG 端口插头　　　　图 4-2　PC、目标板与 JTAG 仿真器的连接方式

JTAG 编程器的连接方式与仿真器类似。JTAG 编程器通常有熔断熔丝的功能。

通过 JTAG 端口可以非常方便地访问 CPU 内部资源，包括程序存储器，所以，在作为发布的产品时，需要阻断 JTAG 端口，以防止产品被仿制。MSP430 芯片中设计的 JTAG 端口包含一根熔丝，通过特殊的时序和电压即可以将其熔断。熔断后的熔丝无法恢复，属于物理性不可逆转的毁坏，以此来防止 CPU 内的程序被仿制者盗用。熔断熔丝后的 CPU 自然也无法再通过仿真器进行调试。

4.2　BSL 编程器

很多情况下，发出产品的程序需要升级，而 CPU 内的熔丝已经熔断过了。此时就要用到 MSP430 上的 bootstrap loader 功能（简称 BSL）。

MSP430 出厂时在地址 0x0C00～0x0FFF 中掩模了一段代码，称为 BOOT ROOM（见图 1-1）。如果 CPU 复位时，在 TEST 或者 TCK 管脚附加一定的时序 ，就可以激活 BSL。图 4-3 是 CPU 正常复位的过程。图 4-4 是有 TEST 管脚的 CPU 激活 BSL 的过程，没有 TEST 管脚的 CPU 使用 TCK 代替 TEST。

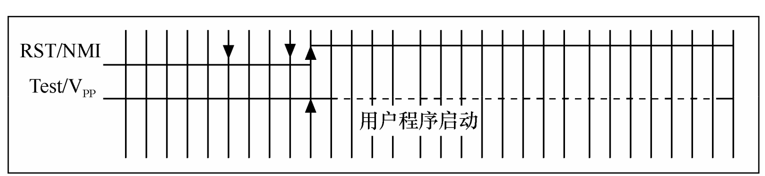


图 4-3　正常复位时序

BSL 激活后，CPU 不是从复位向量所指的地址开始执行，而是执行 BOOT ROOM 中的代码。外部设备遵循 BSL 的通信协议，向 CPU 发出指令，就可以刷新 CPU 内的程序了。

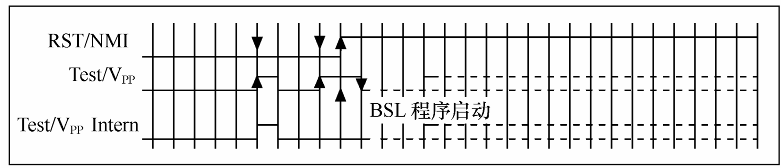


图 4-4　BSL 激活时序

BSL 的底层通信协议是 RS232，只需要 RX、TX 和 GND 就可以进行了。需要注意的是，RX、TX 与 MSP430 内的 USART 模块的异步串行通信口所使用的管脚不同，管脚如表 4-2 所示。

表 4-2　　　　BSL 通信管脚

CPU 型号	TX	RX
MSP430F1 系列	P1.1	P2.2
MSP430F4 系列	P1.0	P1.1

注：TI 公司出品的新型号也许会改变这一规律，建议读者使用前查看相应芯片手册。

BSL 通信协议中仅有固定的几条指令，分为非保护指令和保护指令。

- 非保护指令　校验密码、全部擦除（包含信息存储器）、编程、输出 BSL 版本号、改变传输波特率。
- 保护指令　块编程（输入数据块并写入 FLASH 或 RAM）、块输出（输出 CPU 内的 FLASH 或 RAM 内的数据）、段擦除、擦除校验、修改程序指针（PC）并开始执行

非保护指令不需要密码校验。保护指令必须在执行非保护指令中的校验密码并通过后，才可以执行。

由 MSP430 的内部机构所确定的 BSL 密码是程序位于 0xFFE0～0xFFFF 中的中断向量（见图 1-1），共 32 个字节，256 位。编译程序时，编译器会将生成的中断向量填入相应的地址，没有使用的中断对应的字节保持不变。由于 FLASH 擦除后，其内容全部为 0xFF，所以，没有填写中断向量的部分一般为 0xFF。为了加强保密性能，可以在程序中编写几个空中断，编译器仍然会生成中断向量，但不会激活中断程序。此密码的容量是巨大的，只要用户使用得当，强行试验是根本不可能成功破解的。

非保护指令中与代码有关的实际上只有校验密码和全部擦除，也就是说，用户只有校验密码和全部擦除两种选择。全部擦除 FLASH 后，其中的所有字节都为 0xFF，密码也就全部为 0xFF。此时可以通过 BSL 校验密码指令，接着就可以进入保护指令，对 CPU 重新编程。但由于 FLASH 已经全部被擦除，所以，不存在盗用代码的问题。

由此可见，使用 BSL 是十分安全的。不过由于使用的是串行通信方式，所以，BSL 编程的速度比 JTAG 要慢很多，一般用于产品升级。

作为 MSP430 的一种标准工具，TI 公司公布了 BSL 的制作方法以及 BSL 标准，读者可在 TI 公司的网站上查到。

JTAG 仿真器、编程器以及 BSL 编程器在市场上都可以买到，除了 TI 公司的原装产品，还有多家公司生产。有些产品将 BSL 和 JTAG 编程的功能做成了一体，使用非常方便，如北京东方美源公司生产的 DF800 型编程器。

第5章　程序设计的规范与结构

编程时应该强调的一个重要方面是程序的易读性。在保证软件的速度等性能指标满足用户需求的前提下，让程序简明易懂，使得将来代码的使用者在很短的时间内能够看懂程序的结构，理解设计的思路是十分重要的。这样的程序能够大大提高代码的可读性、可重用性、健壮性、可移植性和可维护性。目前有关 PC 程序设计规范的书籍和文章比较多，关于单片机的却很少。究其原因是单片机的程序相对 PC 的程序来说，程序所实现的功能比较简单，规模也小得多，因此，还没有受到重视。随着技术的发展，单片机的性能不断提高，所完成的功能越来越复杂，这种情况也在逐步改变，对开发人员按照一定的规范来编写程序的要求也越来越高。单片机程序比 PC 程序简单、规模小，因此，不能照搬开发 PC 机程序所使用的编程规范，而是要进行一些简化和修改，使它更适用于单片机程序的开发。5.1 节中介绍编写单片机程序所要遵循的基本规范，读者可以根据自己的需要进行扩充和修改。此规范不仅可以用于编写 MSP430 的程序，也同样适用于其他系列单片机程序的编写。

编写程序时，应当首先确定一个比较完整的程序结构，在此结构的基础上逐步细化，最终完成程序所要求的全部功能。确定程序的结构包括如何划分程序模块以及各模块之间的关系。5.2 节中循序渐进地介绍如何根据需求确定程序的结构。5.3 节中给出编写 MSP430 程序的框架，供读者编写程序时查阅。

5.1 程序规范

编程应该遵循一定的规范并贯穿程序整个生命周期的始终。初学者只要根据自己的需要制定一个规范，然后对自己的程序稍微多花费一点时间按照规范编写，就会惊喜地发现所获得的收益远远超过了所花费的时间。当习惯成为自然，所享受的就是遵循规范所带来的优势了。按照规范编写程序的好处有：

- 增加程序的可维护性。
- 当需求发生变化时，比较易于修改。
- 可重用性好，便于将来再次使用代码。
- 可移植性好，便于将代码在其他类型的微处理上使用。
- 可读性好，不会导致过了很长时间以后，没人能够理解程序的意思，或者非常难以理解。

编程规范不一定是全面和复杂的，最重要的是适用。下面是作者根据多年经验所总结并推荐的规范。

1．头文件

以下是一个头文件的框架和注释，文件名为 moonriver.h。

```
moonriver.h
#ifnde __MOONRIVER        //避免本头文件中定义的内容被重复定义
#define __MOONRIVER


//本头文件需要包含的其他头文件
#include “……”


//函数定义
……
//数据常量定义
……
//数据类型定义
……
//变量定义
……
#endif
```

头文件中的内容应该按照固定的顺序编写。首先使用#include 包含语句编写本头文件中

需要包含的其他头文件，其次声明函数原型，接着定义用到的数据常量。定义数据常量一般使用#define 语句将某个字符串定义为某一数值，在程序中使用这个字符串代表此数值，以便增加程序的可读性。然后定义新的数据类型，如结构或者联合。最后定义变量。需要注意的是，通常变量不应该在头文件中定义，而是在 c 文件中定义。这是因为如果在头文件中定义变量，那么对于包含了此头文件的任何 c 文件，定义变量的语句都会被执行，变量就会被重复定义，从而导致编译错误。

2．变量命名

变量的命名应该基本能够反映变量的数据类型和含义。一个变量由如下几部分组成：存储类型+数据类型+变量名。命名规则如下：

- 存储类型
  全局变量：大小写混用，不同部分之间用大写字母隔开。
  局部变量：全部用小写，不同部分之间用“_”隔开。
- 数据类型　全部用小写字母。

char	（无）
int	i
float	f
double	d
struct	s
指针	p
数组	a
有符号	g

- 变量名　使用缩写或者拼音都可以，大小写按照存储类型的规定进行。

举例：

```
unsigned char MoonRiver;            //全局无符号字符变量
unsigned char moon_river;           //局部无符号字符变量
char gMoonRiver;                    //全局有符号字符变量
char g_moon_river;                  //全局有符号字符变量
unsigned int iMoonRiver;            //全局无符号整数变量
unsigned int* piMoonRiver;          //全局指向无符号整数类型的指针变量
unsigned int p_moon_river;          //局部指向无符号整数类型的指针变量
```

有几种最常用到的变量，这里给出特别定义：

- q0、q1、q2...　定义为 unsigned char 类型，为在函数内部声明的局部变量。
- iq0、iq1、iq2...　定义为 unsigned int 类型，为在函数内部声明的局部变量。
- fq0、fq1、fq2...　定义为 float 类型，为在函数内部声明的局部变量。
- n0、n1、n2...　定义为 unsigned char 或者 unsigned int 类型，在函数内部作为循环次数的计数器使用。

3．缩略语

由于有几种词义在编程时经常会遇到，所以有必要为它们固定定义一个缩写，这样在遇到这些缩写的时候就能够很快知道它们所表示的内容，这些缩写称为缩略语。缩略语在定义变量和函数名的时候都可以使用。

（1）N　下标变量。如要向一个数组中按顺序填充数据，则需要一个变量保存数组的下标。如果数组定义为 unsigned int aiMoon[5]，那么可以定义保存下标的变量为 unsigned char NMoon，这样就很容易知道这个变量是数组 aiMoon 的下标，而不会将其误用为其他用途。

（2）b　布尔变量，只有“是”和“非”两种取值的变量。

（3）Tim　表示与时间有关的变量。

（4）Cnt　计数或计时变量。程序中经常会对某些事件或者时间进行计数。如变量 CntTimMs 是用来计时的变量，计时单位是毫秒。

（5）Pre　上一个变量。通过指针存取数组时，此缩略语用来表示当前指针所指地址的前一个地址。

（6）Nt　下一个变量。通过指针存取数组时，此缩略语用来表示当前指针所指地址的下一个地址。

（7）Sta　状态变量。程序中有时需要定义多个工作的状态，程序根据当前所在的状态来确定执行何种算法。含有此缩略语的变量表明此变量保存的是当前程序的工作状态。

（8）Max　最大值。

（9）Min　最小值。

（10）Init　初始化变量。

（11）Sys　系统变量。表明此变量是与整个单片机系统有关的变量。

4．定义语句

程序中需要对常量和宏进行定义。常数和宏的名称永远都用大写，相应的词之间用“_”隔开。

举例：

```
#define BIT_P1_0  0XFE          //定义屏蔽位的掩码
#define CLR_B  (P1&BIT_P1_0)    //定义复位 P1.0 的宏
```

5．注释

注释在程序中的作用非常重要，初学者往往认为注释不是程序的一部分，从而忽视注释。其实编写程序仅仅是一个程序生命周期的开始，而大部分时间都处在程序的改进和维护期。在此期间，注释是后续维护程序的人员了解程序最有效的手段，因此注释是程序非常重要的一个组成部分。注释分为 3 种：模块注释、函数注释和语句注释。

（1）模块注释　给出整个程序模块的基本信息，放在文件最开始的部位，使用/*…*/注

释符号。通常包含的内容有：

- 文件名　一个 c 文件应该只包含一个功能模块，在模块的注释中要写出文件名称。
- 模块描述　说明模块的功能、适用对象、运行环境等。
- 版本　记录版本的修改日期以及所作的修改。

（2）函数注释　给出函数的基本信息，放在函数的前面。使用/*…*/注释符号。通常包含的内容有：

- 函数描述　描述函数的功能和算法，同时标明函数中所用到的全局变量。
- 返回值　函数的返回值的含义，同时说明函数对有关全局变量所做的修改。

（3）语句注释　对程序语句和变量给出解释，放在每一语句的右边或者上边，说明此语句的作用。使用//注释符号。对变量的注释主要用来说明此变量的用途，一般只有全局变量需要注释。

6．缩进格式

每个缩进用 3 个空格，可以通过按 Tab 键或者插入空格来实现。编辑器中一般都可以调整每按一次 Tab 键所缩进的空格数。所有的{ }对的缩进量应该一致，而且单独占用一行，这样比较容易分辨清楚哪些是括号对中的内容。当每次出现{时，下一行向右增加一个缩进。当每次出现}时，减少一个缩进。这样同一对{}中的{和}总是出现在同一列上，从而使程序书写整齐。

举例：

```
unsigned char MoonRiver（unsigned char arg0,unsigned char arg1）
{
       if(arg0<100)
       {
          arg1 += 1;
          arg0=5;
       }
   else
   {
          arg1 += 2;
          arg0=0;
       }
       arg1 += arg0;
       rerurn arg1;
}
```

7．语句和表达式

语句和表达式要尽量使用简洁明了的语句，不要写过于复杂的语句。可以将比较复杂的语句拆分成多句来编写，这样并不会影响程序将来的运行性能。相反，过于复杂的语句只会

降低程序的可读性。

语句的行与行之间不应该排得太紧，实现同一功能的几条语句可以视为一组，组与组之间应该有空行。必要时也可以对一组语句给出注释。

以上这些规范并不是强迫性的，因为有时候执行起来的确会碰到一些不太容易解决的具体问题。例如在取变量名的时候，某些变量的名称含义很难恰当表述，从而使得变量名有可能很长。过长的名称也会造成不便，降低编程的效率，所以，必须采取灵活的策略，在效率和清晰之间取得平衡。

5.2 程 序 结 构

拿写字来打比方，程序的语句相当于字的笔画，程序的结构相当于字的结构。字的笔画优美固然很好，但如果结构不好，那么无论如何不能算是一幅好字。同样道理，程序的结构非常重要，从某种程度上来说，其重要程度甚至超过了程序的语句。程序的语句编写得再简洁明了也只是局部，而程序结构则决定了程序的整体构架。

编写的程序可以分为有操作系统和无操作系统两类。这两者并不矛盾，各自适用的范围有所不同。本书主要讲述无操作系统程序的几种典型结构。无操作系统的程序最基本的结构如图 5-1 所示。

程序在初始化完毕后，进入一个没有结束的循环，直至掉电。在这个循环中，程序周而复始地循环处理所有任务。可以把问题看得简单一点，所有的程序都是由数据+算法构成的。所有的程序所完成的工作都是将输入的数据按照算法进行加工，然后再输出。因此，可以将主程序部分进一步划为 3 部分，这样图 5-1 所示的程序结构就改进为图 5-2 所示的结构。

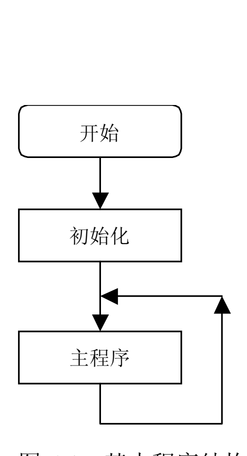


图 5-1　基本程序结构

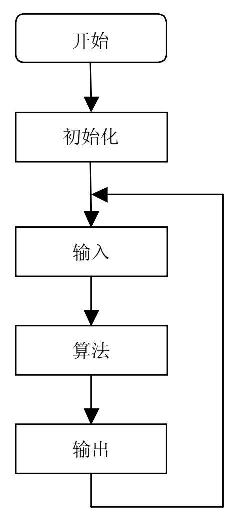


图 5-2　细化的程序结构图

理想情况下，每次循环中首先读取各种输入信号，包括各种传感器、键盘等信号，并集中所有执行算法所需的前提条件。然后算法部分根据已知的条件进行运算，得到输出结果。最后输出部分将运算出的结果变为信号输出，用以驱动各种设备，如显示屏、继电器、电机等。在这种结构下，只有输入和输出部分与硬件有关系，输入部分驱动传感器，输出部分驱动执行机构。算法与硬件隔离，所以，可以用符合 ANSI/ISO 标准的语言来写算法部分，便于将来进行移植。输入部分和输出部分的操作对象是各种设备，因此可以将这两部分再次细化为驱动各个设备的模块，如图 5-3 所示。每一个设备都有一个相对应的设备驱动模块文件。每个设备的驱动模块中一般应当包含一个用来初始化设备的函数，如果设备始终处于打开状态，一般在主程序的初始化部分调用设备的初始化函数。

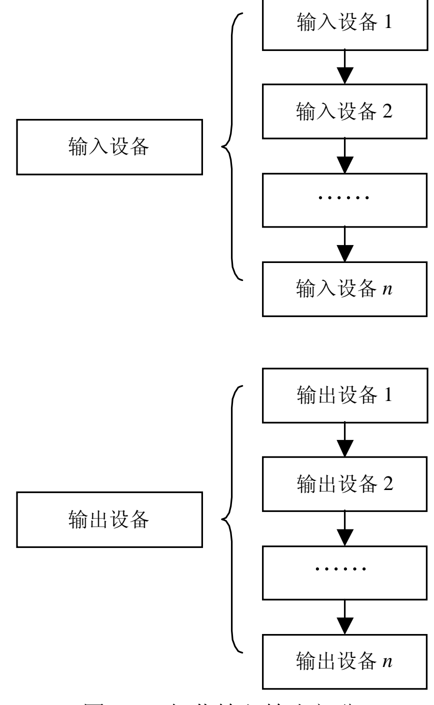


图 5-3　细化输入输出部分

总结一下目前的程序结构：主程序分为输入、算法、输出 3 部分。输入模块调用每一个输入设备的驱动模块，每一个设备驱动模块至少含有一个 c 文件，此文件完全负责对设备的各种操作。输出模块也是如此。算法部分不与任何硬件有关，只是将输入的数据进行处理，给出运算结果。

这样在编写程序的时候可以按照程序结构所划分的各个部分分别进行。因为各个设备的驱动程序只与此设备有关，所以，能够单独进行维护和测试。算法部分与硬件无关，因此可以不必等到设备的硬件完成才进行测试，而只要给出接口部分相应的测试数据就可以进行测试。当所有的模块都通过单独测试后，再将这些模块组装起来，最后再进行完整的测试。如果由于某种原因改变原有设计，需要添加、去掉或者更换某种设备，那么只需要添加、去掉或者更换相应的设备模块，让新模块与程序原来的接口相匹配就可以了。这就是自顶向下分析程序、自底向上编写程序的方式。

不过，上述结构只是理想化的模型，实际应用的情况会比较复杂，如某些信号或者事件对时间的实时性要求很高，不能等待程序循环一周再进行检测或者处理。在这种情况下，必须使信号或者事件触发中断，在中断程序中检测信号或者进行处理。这样就必须安排好中断程序与非中断程序以及各个中断程序之间的关系。

可以将要处理的事件根据紧急程度分为以下几类：

- 最紧急的是需要立即处理的事件，应该在中断程序中处理完毕。这种事件的中断优先级需要安排得比较高，以便在多个中断同时申请中断时能够优先得到响应。
- 对紧急程度比较次要的事件的处理也放在中断程序中，需要将其中断优先级安排得较低。
- 对不紧急事件的处理放在非中断程序中，主程序每循环一周才能执行一次。

但无论紧急程度如何，中断程序应该尽可能编写得短，只在中断中完成必要的功能，以便能够及时地处理其他紧急事件。可以将处理事件的程序分为两部分执行，只在中断程序中

执行紧急的部分，剩余部分在非中断程序中执行。如在串口通信时，仅在中断程序中接受或发送数据，而对数据的处理则在非中断程序中进行。

由于微处理器的速度很快，所以一般情况下，除了收发数据和对高速变化信号的监测需要在中断程序中进行，其他运算都可以在非中断程序中进行。在中断程序中完成了必要的处理后，可以通过置位标志寄存器的方法通知非中断程序进行其余部分的处理。待全部工作完成后再恢复标志寄存器，以便进行下一轮的处理。

在某些情况下可以使非中断程序的循环按照一定的周期运行，但并不要求周期十分精确。例如，可以使用一个定时器产生固定的时间间隔，如 10ms，时间到达时触发中断。每次产生中断时，在中断程序中置位标志寄存器。非中断程序在一次循环完成后，不会紧接着进行下一次循环，而是监测标志寄存器，直至标志寄存器被置位，才开始进行下一轮的循环。这样非中断程序每隔固定的时间循环一次，这要求循环一次的时间必须小于设定的固定时间。这个时间可以不十分精确，由本次循环数据处理的复杂程度而定，但基本上满足大多数情况的要求。再次修改程序结构如图 5-4 所示。

图 5-4 所示的结构能够满足相当一部分程序的需求，但不能满足所有的需求。没有一个结构能够满足所有的需求，读者理解的应该是一种思想方法，切忌死记某种模式。

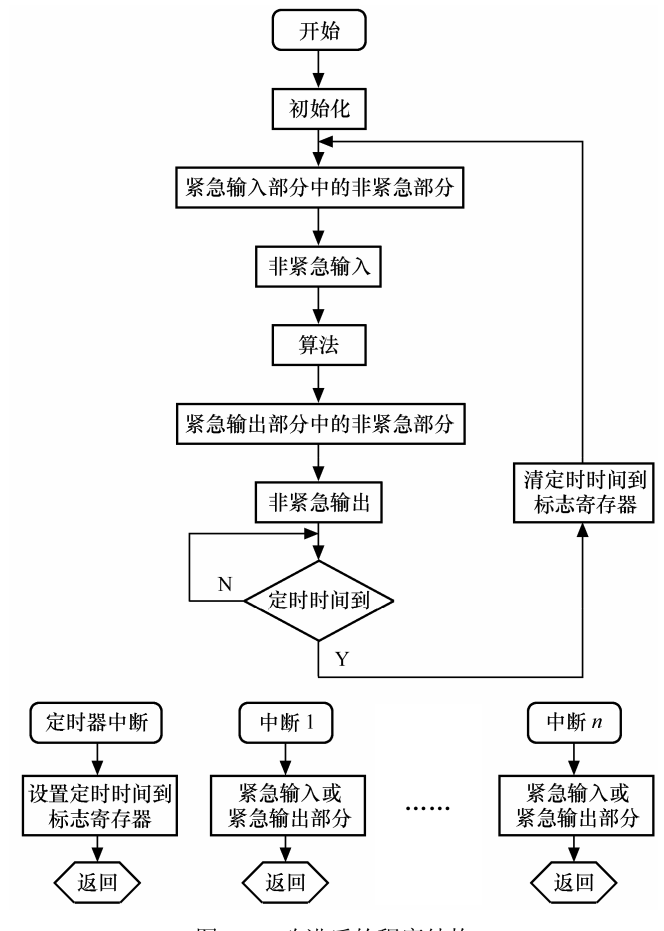


图 5-4　改进后的程序结构

需要指出的是，实现结构化编程会付出一些代价，包括使用更多的存储器和降低处理数据的速度。例如调用函数时，会将一些寄存器中的值压入堆栈，从函数返回时，再从堆栈中恢复这些寄存器的值，这样就增加了程序的长度，并且消耗了一部分时间用于数据的进栈和出栈。然而，并没有一个固定的标准来平衡消耗与获得。从某种角度来说，构造程序也是一种平衡的艺术，需要从功能、成本、开发时间等多个角度来考虑。

根据 MSP430 的特点，MSP430 还支持如图 5-5 所示的结构。

主程序不再是一个无休止的循环，而是进入了休眠状态。由某些事件或者信号触发中断，在中断程序中处理这些事件或者信号，处理完毕后再次进入休眠状态。这种程序结构更像是事件驱动的做法。

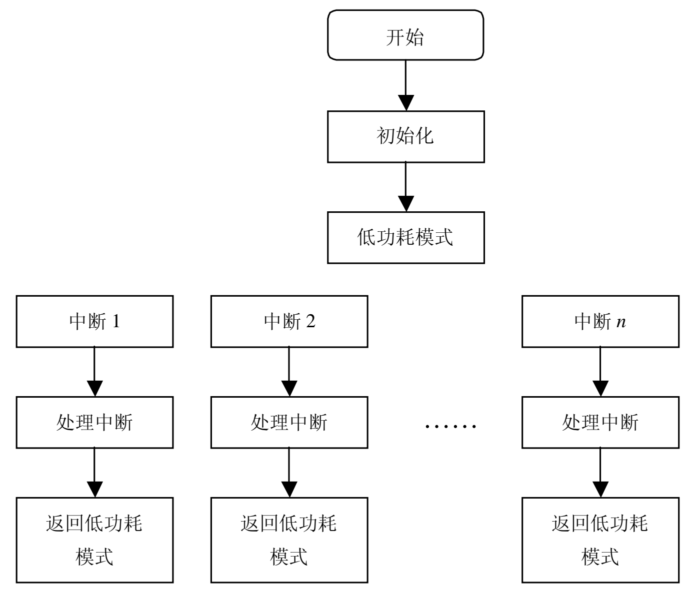


图 5-5　MSP430 的一种程序结构

5.3　框 架 程 序

下面给出 MSP430 的框架程序。读者可以在框架程序的基础上修改，然后添加自己所需的部分。

```
/***********************************************************************\
文件名：main.c
描述：MSP430 框架程序。适用于 MSP430F149，其他型号需要适当改变。
     不使用的中断函数保留或者删除都可以，但保留时确保不要打开不需要的中断。
     保留中断函数，编译器将会为 BSL 密码填充所有的字节。
版本：1.0 2005-1-13
\***********************************************************************/
```

```
//头文件
#include <MSP430x14x.h>

//函数声明
void InitSys( );

int main( void )
{
    WDTCTL=WDTPW+WDTHOLD;                   //关闭看门狗
    InitSys( );                             //初始化

start:
        //以下填充用户代码

        LPM3;    //进入低功耗模式 n，n 取值为 0~4，若不希望进入低功耗模式，屏蔽本句
        goto start;

}

/*********************************************************************
系统初始化
*********************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;                     //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                     //清除振荡器失效标志
        for(iq0=0xFF;iq0>0;iq0--);          //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!= 0);             //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                    //选择 MCLK、SMCLK 为 XT2
    //以下填充用户代码，对各种模块、中断、外围设备等进行初始化

    _EINT();        //打开全局中断控制，若不需要打开，可以屏蔽本句
```

```
}

/***********************************************************************
端口 2 中断函数
***********************************************************************/
#pragma vector=PORT2_VECTOR
__interrupt void Port2()
{
    //以下为参考处理程序，不使用的端口应当删除其对于中断源的判断。
    if((P2IFG&BIT0)==BIT0)
    {
        //处理 P2IN.0 中断
        P2IFG &= ~BIT0;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT1)==BIT1)
    {
        //处理 P2IN.1 中断
        P2IFG &= ~BIT1;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT2)==BIT2)
    {
        //处理 P2IN.2 中断
        P2IFG &= ~BIT2;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT3)==BIT3)
    {
        //处理 P2IN.3 中断
        P2IFG &= ~BIT3;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT4)==BIT4)
    {
```

```
        //处理 P2IN.4 中断
        P2IFG &= ~BIT4;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT5)==BIT5)
    {
        //处理 P2IN.5 中断
        P2IFG &= ~BIT5;    //清除中断标志
        //以下填充用户代码

    }
    else if((P2IFG&BIT6)==BIT6)
    {
        //处理 P2IN.6 中断
        P2IFG &= ~BIT6;    //清除中断标志
        //以下填充用户代码

    }
    else
    {
        //处理 P2IN.7 中断
        P2IFG &= ~BIT7;    //清除中断标志
        //以下填充用户代码

    }
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/***********************************************************************
USART1 发送中断函数
***********************************************************************/
#pragma vector=USART1TX_VECTOR
__interrupt void Usart1Tx()
{
    //以下填充用户代码

    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}
```

```
/***********************************************************************
USART1 接收中断函数
***********************************************************************/
#pragma vector=USART1RX_VECTOR
__interrupt void Ustra1Rx()
{
    //以下填充用户代码

    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/***********************************************************************
端口 1 中断函数
多中断中断源：P1IFG.0~P1IFG7
进入中断后应首先判断中断源，退出中断前应清除中断标志，否则将再次触发中断
***********************************************************************/
#pragma vector=PORT1_VECTOR
__interrupt void Port1()
{
    //以下为参考处理程序，不使用的端口应当删除其对于中断源的判断。
    if((P1IFG&BIT0)==BIT0)
    {
        //处理 P1IN.0 中断
        P1IFG &= ~BIT0;    //清除中断标志
        //以下填充用户代码

    }
    else if((P1IFG&BIT1)==BIT1)
    {
        //处理 P1IN.1 中断
        P1IFG &= ~BIT1;    //清除中断标志
        //以下填充用户代码

    }
    else if((P1IFG&BIT2)==BIT2)
    {
        //处理 P1IN.2 中断
        P1IFG &= ~BIT2;    //清除中断标志
```

```
        //以下填充用户代码

    }
    else if((P1IFG&BIT3)==BIT3)
    {
        //处理 P1IN.3 中断
        P1IFG &= ~BIT3;     //清除中断标志
        //以下填充用户代码

    }
    else if((P1IFG&BIT4)==BIT4)
    {
        //处理 P1IN.4 中断
        P1IFG &= ~BIT4;     //清除中断标志
        //以下填充用户代码

    }
    else if((P1IFG&BIT5)==BIT5)
    {
        //处理 P1IN.5 中断
        P1IFG &= ~BIT5;     //清除中断标志
        //以下填充用户代码

    }
    else if((P1IFG&BIT6)==BIT6)
    {
        //处理 P1IN.6 中断
        P1IFG &= ~BIT6;     //清除中断标志
        //以下填充用户代码

    }
    else
    {
        //处理 P1IN.7 中断
        P1IFG &= ~BIT7;     //清除中断标志
        //以下填充用户代码

    }
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
```

```
}

/************************************************************************
定时器 A 中断函数
多中断中断源：CC1~2 TA
************************************************************************/
#pragma vector=TIMERA1_VECTOR
__interrupt void TimerA1()
{
    //以下为参考处理程序，不使用的中断源应当删除
    switch (__even_in_range(TAIV, 10))
    {
        case 2:
            //捕获/比较 1 中断
            //以下填充用户代码

        break;
        case 4:
            //捕获/比较 2 中断
            //以下填充用户代码

        break;
        case 10:
            //TAIFG 定时器溢出中断
            //以下填充用户代码

        break;
    }
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/************************************************************************
定时器 A 中断函数
中断源：CC0
************************************************************************/
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0()
{
    //以下填充用户代码
```

```
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/***********************************************************************
AD 转换器中断函数
多中断源：模拟 0~7、VeREF+、VREF-/VeREF-、(AVcc-AVss)/2
***********************************************************************/
#pragma vector=ADC_VECTOR
__interrupt void Adc()
{
    //以下为参考处理程序，不使用的中断源应当删除
    if((ADC12IFG&BIT0)==BIT0)
    {
        //通道 0
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT1)==BIT1)
    {
        //通道 1
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT2)==BIT2)
    {
        //通道 2
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT3)==BIT3)
    {
        //通道 3
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT4)==BIT4)
    {
        //通道 4
```

```
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT5)==BIT5)
    {
        //通道 5
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT6)==BIT6)
    {
        //通道 6
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT7)==BIT7)
    {
        //通道 7
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT8)==BIT8)
    {
        //VeREF+
        //以下填充用户代码

    }
    else if((ADC12IFG&BIT9)==BIT9)
    {
        //VREF-/VeREF-
        //以下填充用户代码

    }
    else if((ADC12IFG&BITA)==BITA)
    {
        //温度
        //以下填充用户代码

    }
```

```
        else if((ADC12IFG&BITB)==BITB)
        {
            //（AVcc-AVss）/2
            //以下填充用户代码

        }
        LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
    }

    /**********************************************************************
    USART0 发送中断函数
    **********************************************************************/
    #pragma vector=USART0TX_VECTOR
    __interrupt void Usart0Tx()
    {
        //以下填充用户代码

        LPM3_EXIT; //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
    }

    /**********************************************************************
    USART0 接收中断函数
    **********************************************************************/
    #pragma vector=USART0RX_VECTOR
    __interrupt void Usart0Rx()
    {
        //以下填充用户代码

        LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
    }

    /**********************************************************************
    看门狗定时器中断函数
    **********************************************************************/
    #pragma vector=WDT_VECTOR
    __interrupt void WatchDog()
    {
        //以下填充用户代码

```

```
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/**********************************************************************
比较器 A 中断函数
**********************************************************************/
#pragma vector=COMPARATORA_VECTOR
__interrupt void ComparatorA()
{
    //以下填充用户代码

    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/**********************************************************************
定时器 B 中断函数
多中断源：CC1~6 TB
**********************************************************************/
#pragma vector=TIMERB1_VECTOR
__interrupt void TimerB1()
{
    //以下为参考处理程序，不使用的中断源应当删除
    switch (__even_in_range(TBIV, 14))
    {
        case 2:
            //捕获/比较 1 中断
            //以下填充用户代码

        break;
        case 4:
            //捕获/比较 2 中断
            //以下填充用户代码

        break;
        case 6:
            //捕获/比较 3 中断
            //以下填充用户代码

        break;
```

```
            case 8:
                //捕获/比较 4 中断
                //以下填充用户代码

            break;
            case 10:
                //捕获/比较 5 中断
                //以下填充用户代码

            break;
            case 12:
                //捕获/比较 6 中断
                //以下填充用户代码

            break;
            case 14:
                //TBIFG 定时器溢出中断
                //以下填充用户代码

            break;
        }
        LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
    }

/***********************************************************************
定时器 B 中断函数
中断源：CC0
***********************************************************************/
#pragma vector=TIMERB0_VECTOR
__interrupt void TimerB0()
{
    //以下填充用户代码

    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}

/***********************************************************************
不可屏蔽中断函数
***********************************************************************/
```

```
#pragma vector=NMI_VECTOR
__interrupt void Nmi()
{
    //以下为参考处理程序，不使用的中断源应当删除
    if((IFG1&OFIFG)==OFIFG)
    {
        //振荡器失效
        IFG1 &= ~OFIFG;
        //以下填充用户代码

    }
    else if((IFG1&NMIIFG)==NMIIFG)
    {
        //RST/NMI 不可屏蔽中断
        IFG1 &= ~NMIIFG;
        //以下填充用户代码

    }
    else if((FCTL3&ACCVIFG)==ACCVIFG)
    {
        //存储器非法访问
        FCTL3 &= ~ACCVIFG;
        //以下填充用户代码

    }
    LPM3_EXIT;  //退出中断后退出低功耗模式。若退出中断后要保留低功耗模式，将本句屏蔽
}
```

框架程序比较长，但却是一个完整的程序结构，对于初学的读者来说，更容易获得整体的概念。MSP430 的中断种类很多，通过这个框架程序也可以方便地查询中断的处理方式。

程序中的 main 函数是程序的起点。main 函数中调用了另外一个函数 InitSys，InitSys 作用是对系统进行初始化。其余的是中断函数，对中断进行处理。

程序的第一句将看门狗关闭，MSP430 的看门狗复位后的默认状态是打开的，如果不使用就必须关闭，否则会产生 PUC 信号，从而导致系统复位。另外，在调试时也应当关闭看门狗。接着进入 InitSys 函数，进行初始化。如果使用 XT2 振荡器，则必须先打开 XT2 振荡器。因为振荡器失效的中断标志 OFIFG 复位后是有效的，所以，首先必须复位 OFIFG，清除振荡器失效标志。因为 XT2 振荡器不会立刻起振，所以需要延时一段时间后再次测试 OFIFG。如果 OFIFG 为 1，则 XT2 没有起振，再次复位 OFIFG，等待 XT2 起振。周而复始，直至 XT2 起振再向下执行程序。

程序接着选择 MCLK 和 SMCLK 的时钟源为 XT2，当然也可以不这样选。接着根据需要对所使用的外围模块进行初始化。最后，如果使用了中断，则打开全局中断控制位。CPU 也可以根据需要选择进入低功耗模式，等待中断被触发再退出低功耗模式。

不同的中断函数处理方式有所不同。某些中断要求退出中断函数前要复位中断标志，否则会再次触发中断。多中断源中断函数需要判断中断源，如果同时有多个中断源触发中断，则按照所编写程序判断中断触发源的顺序来确定响应中断的优先级。用户可以按照自己的顺序要求来编写程序。中断函数中最后一句 LPMn_EXIT 使程序退出中断程序之后脱离低功耗模式，n 的取值范围为 0~4，代表 5 种低功耗模式。如果不需要脱离低功耗模式，则可以删除此语句。

第 6 章　MSP430 异步串行通信

因为串行通信比并行通信所耗用的硬件资源要少，结构也简单，所以应用更加广泛。实现串行通信的通信协议有很多种，如 IIC 总线协议、SPI 协议等，它们分别适用于不同的场合。

本章讲述在 MSP430 上实现的异步串行通信。异步串行通信协议使用得非常广泛，基本上为所有单片机的必备功能，常用于系统之间的相互通信。

6.1 串行通信简介

MSP430 中大部分芯片都包含有硬件的串行通信模块 USART0，有些型号有两个串行通信模块：USART0 和 USART1。MSP430 串行通信模块的功能很强，使用也非常灵活。模块在发送和接收每一字节数据时都可以触发中断，从而使 CPU 退出低功耗模式，发送和接收中断由两个独立的中断控制位控制。波特率的调整非常灵活，可通过寄存器 UxR00 和 UxR10 粗略调整波特率，然后通过寄存器 UMCTL0 进一步细调波特率。通过寄存器 UCTL0 设置串行通信的工作模式，通过 UTCTL0 设置产生波特率所使用的时钟。值得注意的是，设置串行通信模块时，应该使寄存器 UCTL0 中的 SWRST 位置位，以便使串行通信模块处于复位状态，设置完毕后，再复位寄存器 UCTL0 中的 SWRST 位，以便使串行通信模块可以正常工作。如果在串行通信模块正常工作状态设置它的寄存器，那么串行通信模块有可能无法正常工作。TXD 为发送数据的管脚，RXD 为接收数据的管脚，必须将这两个管脚设置为工作在第二功能，使这两个管脚工作在异步通信发送和接收数据的状态。模块在不使用时可以关闭，以节省能耗。设置完以上各项后，打开串行通信模块，就可以进行串行通信了。如果使用中断处理函数接收、发送数据，还应该置位相应的中断控制位，用于打开中断。

MSP430 支持两种多机通信模式：地址位多机模式和空闲多机模式。多机通信模式适用于一个主机对应多台从机通信的方式，从机与从机之间无法相互直接通信。每台从机有一个单字节的地址，各从机的地址互不重合。主机在地址帧中发送从机地址，从机收到地址帧后将收到的数据与本机的地址相比较，如果相同，则接收主机随后发出的数据帧。主机需要改变通信对象时，再次发出地址帧，从机再次判断是否与本机地址一致。

进行异步串行通信的双方必须使用相同的通信格式。非多机通信的点对点异步串行通信的格式通常为：起始位+数据位+奇偶校验位+停止位。起始位为 1 位。数据位可以选择 7 位或者 8 位，一般选择 8 位。奇偶校验位为 1 位，可以省略。停止位可以为 1 位或者 2 位。

从机地址位多机模式的通信格式为：起始位+数据位+地址/数据位+停止位。从机通过地址/数据位判别收到的是数据帧还是地址帧。

空闲多机模式的通信格式为：主机发送地址之前，使接收和发送线路上保持 10 位或者更多的空闲位。从机检测到这些空闲位后，收到的第一个字节为地址字节。

进行异步串行通信时，可靠性是一个重要的性能指标，但是由于数据传输通道的不理想以及来自各方面的干扰，出现错误是不可避免的。因此，无论是主机还是从机必须对收到的数据进行校验，否则不能保证数据传输的可靠性。校验的方式通常是将若干数据字节作为一组，与若干控制字节合在一起发送，称为数据包。控制字节一般包含包起始字节、校验字节、包长度字节。包的第一个字节是包起始字节，通信协议的制定者可以自行规定一个数值，接收端在等待接收数据的时候，只有首先收到此数值才认为是接收一个数据包的开始，否则认为是收到错误数据而加以抛弃。包长度字节表明数据包的字节数，紧接在包起始字节之后，这样接收端可以计算出将要收到的包的字节数。校验字节通常放在数据包的最后，发送端和接收端对数据包中除校验字节外的其他所有字节用同样的算法进行计算。根据包长度字节收

到足够的字节数后，接收端对收到的数据进行校验计算。如果收到的数据包无误，则接收端计算得到的结果与收到的校验字节必定是相等的，否则，认为收到的数据包有错误，可以抛弃或者通知发送端重新发送。也可以规定包长度是固定的字节数，这样每次收到的包的字节数是固定的，可以简化接收端处理程序的复杂性。校验计算的算法种类很多，在要求不高的情况下，可以取包中除校验字节外的所有字节的算术和，就是将这些字节进行累加，不考虑是否进位，最后取低 8 位作为校验字节。

6.2　串行通信软件实现

程序 6-1 是异步通信模块程序，可以用于两台单片机之间的通信，也可以用于单片机与 PC 机之间的通信。为了方便起见，在以下介绍中假设是 MSP430 与 PC 机之间进行通信。程序 6-1 的功能为接收 PC 机发送的数据包，对收到的数据包进行校验。如果校验正确，则按照通信协议的规定处理数据，将处理结果再组成一个数据包，发还给 PC 机。如果校验错误，则发送一个单字节的校验错误码给 PC 机，PC 机收到后会重新发送数据包。MSP430 发送处理结果数据包后，将会等待 PC 机对数据包进行校验。如果校验正确，那么 PC 机发送校验正确码给 MSP430。如果校验错误，则发送校验错误码，MSP430 收到校验错误码后，将会重新发送数据包。

程序 6-1 的工作环境为 XT2 接 8M 晶振，LFXT1 接 32.768K 晶振。MCLK 选用 XT2，ACLK 选用 LFXT1。串行通信模块使用 USART0，波特率时钟选用 ACLK，波特率为 9600bit/s。串行通信工作模式为 1 位起始位+8 位数据位+1 位奇校验位+1 位停止位。将串行通信接收中断打开，在中断函数中将收到的数据放入缓冲区。发送不使用中断，每发送一个字节后，通过查询标志位的方式判断是否发送完毕。

main.c 是主程序。首先调用 InitSys 函数进行系统初始化。初始化完毕后，程序进入主循环。每次循环时，程序首先调用 DoUart 函数处理串行通信接收缓冲区中的数据。处理完毕后，如果有需要发送给 PC 机的数据，就调用 SendUart 函数发送数据。发送完毕后，CPU 进入低功耗模式 LPM3。如果串行通信模块收到数据，则 CPU 退出低功耗模式，进入串行接收中断程序。退出中断程序后，CPU 不再进入低功耗模式，而是执行跳转语句 goto start，进行下一轮循环，再次调用 DoUart 和 SendUart 函数。如果没有收到数据，那么 CPU 会一直处于低功耗模式。

df_uart.c 为串行通信模块驱动程序，包括函数 UartInit、SendUart、Usart0Rx。

函数 UartInit 中首先设置串行通信模块使用的管脚 RXD、TXD 工作在第二功能。然后设置串行通信工作模式并复位串行通信模块，选择产生波特率的时钟为 ACLK，设置波特率为 9600bit/s，取消串行通信模块的复位，使通信模块可以正常工作。最后打开串行通信模块和串行接收中断。

函数 SendUart 的功能为发送数据。变量 pBuffer 为一地址指针，指向发送缓冲区内的某一元素，变量 n_byte 为要发送数据的字节数。函数 SendUart 发送从 pBuffer 所指地址开始的 n_byte 个字节，每发送一个字节，通过语句 while((IFG1&UTXIFG0)==0)查询此字节是否发送

完毕，UTXIFG0 是串行通信模块发送完毕的标志位，位于寄存器 IFG1 中。如果发送完毕，则接着发送下一字节数据，使变量 pBuffer 加 1，指向下一个要发送数据的地址。

串行接收中断函数 Usart0Rx 首先判断接收到的字节是否出现奇校验错误。如果出现错误则调用数据包处理模块 bao.c 中的 SetBaoErr 函数，通知数据包处理模块收到的包有错误。如果没有错误，则调用 bao.c 中的 AddUsData 函数，将收到的字节添加入接收缓冲区。最后使用 LPM3_EXIT 使 CPU 在退出中断函数后不再返回低功耗模式。

bao.c 对收到的数据包进行处理，包括数据包校验函数 JiaoYan、接收缓冲区添加数据函数 AddUsData、数据包处理函数 DoUart、设置数据包错误函数 SetBaoEr、执行数据包传输来的指令函数 DoCommand。

数据包校验函数 JiaoYan 对收到的数据包进行校验，校验的方法是计算数据包中除校验字节外的所有字节的算术和。如果算术和与校验字节相等，则通过校验，函数返回 1，否则返回 0。考虑到函数的通用性，应使计算出的算术和通过指针型参数 pjiao_zhi 返回给函数的调用者。参数 n_byte 为数据包的总字节数，在本例中数据包的字节数是固定的，包字节数虽然可以作为一个常量写在程序中，不过仍然将包字节数作为一个参数，目的仍然是为了提高函数的通用性。这样，将来此函数用于其他通信程序时，如果需要改变包字节数，那么调用此函数时就不必再修改函数内部的语句，因为包字节数本身是函数的参数之一。这样做虽然会增加代码长度和减缓程序的执行速度，但考虑到单片机的功能越来越强，其优点还是明显的。

函数 AddUsData 将收到的字节添加进接收缓冲区。接收缓冲区是一个数组，所以，添加数据时，有必要判断已经收到的字节数是否超出了缓冲区所能容纳的字节数，如果缓冲区已满，则不再向缓冲区中添加数据。目前在缓冲区满的时候，此函数没有返回错误信息。读者可以自行修改，如让函数在缓冲区满的时候返回一个代表错误的值，表明缓冲区已满。

函数 DoUart 判断收到的数据包是否正确无误。首先判断接收缓冲区中是否有收到的数据。如果有数据，则判断收到的第一个字节是否是包起始字节。如果不是，则向发送缓冲区中添加一个字节的错误信息 HAND_ERR。变量的内容为要发送的字节数，使 SendByte 为 1，表明有 1 字节的数据等待发送。如果收到的第一个字节是合法的包起始字节，则继续等待其余的字节。注意，合法的包起始字节可以有多个，代表不同的含义。本程序合法的包起始字节有 3 个：HAND_BAO、HAND_OK、HAND_ERR。HAND_BAO 表明将收到一个完整的数据包，含 5 个字节。在接收数据包时，HAND_OK 和 HAND_ERR 用来通知发送方接受方刚收到的数据包是否正确，只返回一个字节，没有校验字节。接收方收到这样的包起始字节后，不等待后续的字节。每次发送一个数据包后，必须等待接收方返回 HAND_OK 或者 HAND_ERR，如果返回的是 HAND_ERR，则重新发送数据包。如果收到的是 HAND_BAO，则在其余字节接受完毕后，调用 JiaoYan 函数进行数据包校验。如果校验未通过，发送 HAND_ERR 给发送方，通知发送方重新发送。如果校验通过，则从数据包中提取出指令，调用 DoCommand 函数处理指令，并将处理完毕的结果组成数据包，返回给发送方。

函数 DoCommand 处理指令，使用 switch 结构分别对不同的指令进行处理。严格来说，switch 中应该有一个 default 的选项，用来处理错误的命令，读者可以自行添加。

文件 xieyi.h 定义了数据包的结构和大小，以及数据包中包含指令的含义。可以看到一个数据包的结构为数据包头（数据包起始字节）、指令、数据高字节、数据低字节、校验字节，

共 5 个字节。指令用来告知接收方数据的含义或者如何处理数据。

图 6-1 是程序 6-1 的结构图。

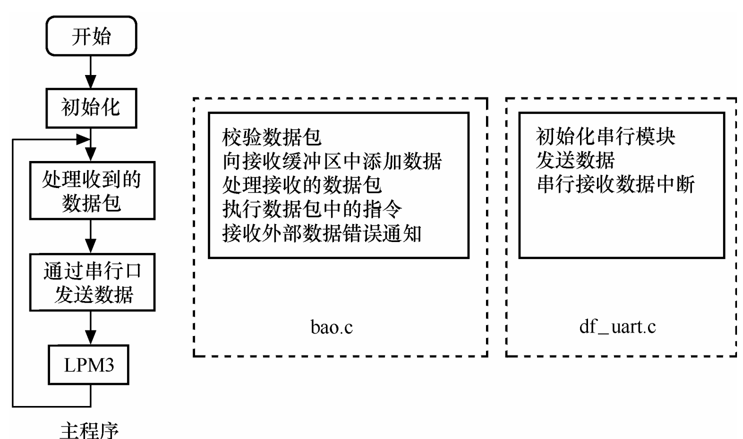


图 6-1　程序 6-1 结构图

程序 6-1：

```
main.c
#include <MSP430x14x.h>
#include "xieyi.h"
#include "bao.h"
#include "df_uart.h"

void InitSys();  //初始化

int main( void )
{
    unsigned char q0;
    unsigned char *pq0;
    WDTCTL=WDTPW+WDTHOLD;              //关闭看门狗
    InitSys();                         //初始化
start:
    pq0=DoUart(&q0);
    //发送数据
    if(q0 != 0)
    {
        SendUart(pq0,q0);
    }
    LPM3;
    goto start;
```

```
}

/*********************************************************************
系统初始化
*********************************************************************/
void InitSys()
{
    unsigned int iq0;

    BCSCTL1 &= ~XT2OFF;                         //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                         //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);           //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG) != 0);                //判断 XT2 是否起振

    BCSCTL2=SELM_2;                             //选择 MCLK 为 XT2

    UartInit();                                 //初始化 USART0
    _EINT();
}

df_uart.h
#ifndef  __DF_UART
#define  __DF_UART
void UartInit();
void SendUart(unsigned char *pBuffer,unsigned char n_byte);
#endif

df_uart.c
#include <MSP430x14x.h>
#include "df_uart.h"
#include "bao.h"
#include "xieyi.h"

#define USART_DIR P3DIR
#define USART_SEL P3SEL
#define USART_IN  P3IN
```

```
#define USART_OUT P3OUT
#define UTXD0 BIT4
#define URXD0 BIT5

void UartInit()
{
    USART_SEL |= UTXD0+URXD0;  //设置管脚为第二功能
    UCTL0=CHAR+PENA+SWRST;     //8 位数据，1 位停止位，奇校验
    UTCTL0=SSEL0;              //选择 UCLK=ACLK
    UBR00=0x3;                 //设置波特率为 9600bit/s
    UBR10=0;
    UMCTL0=0x4A;
    UCTL0 &= ~SWRST;
    ME1 |= UTXE0+URXE0;        //打开模块 USART0
    IE1 |= URXIE0 ;            //打开 USART0 接收中断
}

/*********************************************************************
发送函数。采用查询方式。
PBuffer：指向发送数据缓冲区的指针
n_byte：发送的字节数
*********************************************************************/
void SendUart(unsigned char *pBuffer,unsigned char n_byte)
{
    unsigned char q0;
    for(q0=0;q0<n_byte;q0++)
    {
        while((IFG1&UTXIFG0)==0);  //判断是否发送完毕
        TXBUF0=*pBuffer;
        pBuffer++;
    }
}

/*********************************************************************
USART0 接收中断函数
*********************************************************************/
#pragma vector=USART0RX_VECTOR
__interrupt void Usart0Rx()
{
```

```
        if((U0RCTL&RXERR)==0)
        {
            AddUsData(RXBUF0);
        }
        else
        {
            SetBaoErr();
            U0RCTL &= ~(FE+PE+OE+BRK);
        }
        LPM3_EXIT;
    }

    xieyi.h
    #ifndef __XIEYI
    #define __XIEYI

    #define N_XY_BAO 5              //通信包的字节数
    #define HAND_BAO 0xA0           //后跟数据的数据包头
    #define HAND_OK 0x0A            //收到数据校验正确
    #define HAND_ERR 0xA4           //收到数据校验错误

    //PC 传送给实验板的指令
    #define NONE_COMMAND 0          //空的指令，什么都不做
    #define ADD_COMMAND 0x10        //数据 A+数据 B
    #define SUB_COMMAND 0x20        //数据 A-数据 B
    /*****************************************************************
    //PC 传送给实验板的通信包

    每个字节代表的含义:
    1 数据包头(HAND_BAO)
    2 指令
    3 数据 A
    4 数据 B
    5 校验

    //实验板传给 PC 的通信包

    每个字节代表的含义:
    1 数据包头(HAND_BAO)
```

```
    2 指令
    3 数据高字节
    4 数据低字节
    5 校验

    发送数据包后，需等待接收对方的校验信息 HAND_OK 或者 HAND_ERR。
    收到数据包校验正确后发出 HAND_OK，否则发出 HAND_ERR。
    ***********************************************************************/
    #endif

    bao.h
    #ifndef __BAO
    #define __BAO
    unsigned char JiaoYan(unsigned char *pbuffer,unsigned char n_byte,unsigned
char *pjiao_zhi);
    void AddUsData(unsigned char sq0);
    unsigned char* DoUart(unsigned char *p0);
    void SetBaoErr();
    unsigned int DoCommand(unsigned char comd);
    #endif

    bao.c
    #include "bao.h"
    #include "xieyi.h"
    #include "df_uart.h"

    unsigned char aRxBuff[N_XY_BAO];          //接收数据缓冲区
    unsigned char NRxBuff=0;
    unsigned char aTxBuff[N_XY_BAO];          //发送数据缓冲区
    unsigned char NTxBuff=0;
    unsigned char bWaitRe=0;  //1：发送数据包后等待 PC 返回对数据包的校验结果；0：不等待
    unsigned char Command=NONE_COMMAND;       //收到的指令
    unsigned char SendByte=0;                 //准备发送的字节数
    unsigned char bUartRxErr=0;    //1：接收数据出错，如帧错、奇偶校验错等。0：没错
    /**********************************************************************
    采用算术和的方法进行
    数据包校验
    pbuffer：指向要校验的数据缓冲区的指针
    n_byte：校验的字节数
```

```
    pjiao_zhi：计算出的校验值
    返回值：校验通过为 1，校验失败为 0
    *********************************************************************/
    unsigned char JiaoYan(unsigned char *pbuffer,unsigned char n_byte,unsigned
char *pjiao_zhi)
    {
        unsigned char q0,q1=0;

        for(q0=0;q0<n_byte-1;q0++)
        {
            q1 += *pbuffer;
            pbuffer++;
        }

        *pjiao_zhi=q1;
        if(q1== *pbuffer)
            return 1;
        else
            return 0;
    }

    /*********************************************************************
    向接收缓冲区中增加一个数据
    *********************************************************************/
    void AddUsData(unsigned char sq0)
    {
        if(NRxBuff<N_XY_BAO)
        {
            aRxBuff[NRxBuff]=sq0;
            NRxBuff++;
        }
    }

    /*********************************************************************
    处理 Uart 数据包
    p0：用来返回发送的数据的字节数
    返回值：指向发送缓冲区的指针
    *********************************************************************/
```

```
unsigned char* DoUart(unsigned char *p0)
{
    unsigned char q0;
    unsigned int iq0;
    //判断接收数据
    if(NRxBuff!=0)
    {
        if(aRxBuff[0]==HAND_OK&&bWaitRe==1&&bUartRxErr==0)
        {
            //发送完数据包，等待 PC 校验结果
            bWaitRe=0;
            NRxBuff=0;
            SendByte=0;
        }
        else if(aRxBuff[0]==HAND_ERR&&bWaitRe==1&&bUartRxErr==0)
        {
            //发送完数据包，等待 PC 校验结果
            SendByte=N_XY_BAO;
            NRxBuff=0;
        }
        else if(aRxBuff[0]==HAND_BAO&&bWaitRe==0&&bUartRxErr==0)
        {
            if(NRxBuff==N_XY_BAO)
            {
                //数据包接收完毕
                if(JiaoYan(aRxBuff,N_XY_BAO,&q0)==1)
                {
                    //数据校验通过
                    Command=aRxBuff[1];
                }
                else
                {
                    //数据校验错误
                    aTxBuff[0]=HAND_ERR;
                    SendByte=1;
                }
                NRxBuff=0;
            }
        }
```

```
            else
            {
                //aRxBuff[0]不是合法的值或者数据接收错误
                aTxBuff[0]=HAND_ERR;
                SendByte=1;
                NRxBuff=0;
                bUartRxErr=0;
            }
        }
        //处理指令
        iq0=DoCommand(Command);
        if(Command!=NONE_COMMAND)
        {
            aTxBuff[0]=HAND_BAO;
            aTxBuff[2]=iq0>>8;
            aTxBuff[3]=iq0&0xFF;
            JiaoYan(aTxBuff,N_XY_BAO,&aTxBuff[4]);
            SendByte=N_XY_BAO;
            Command=NONE_COMMAND;
    }
        if(SendByte==N_XY_BAO)
            bWaitRe=1;
        *p0=SendByte;
        return aTxBuff;
    }

    /**********************************************************************
    其他模块检测到数据包有错时通知本模块数据通信有错
    **********************************************************************/
    void SetBaoErr()
    {
        bUartRxErr=1;
    }

    /**********************************************************************
    执行指令
    **********************************************************************/
    unsigned int DoCommand(unsigned char comd)
    {
```

```
        switch(comd)
        {
            case ADD_COMMAND:
                //执行 ADD_COMMAND 指令：数据 A+数据 B
                return (aRxBuff[2]+aRxBuff[3]);
            case SUB_COMMAND:
                //执行 SUB_COMMAND 指令：数据 A-数据 B
                return (aRxBuff[2]-aRxBuff[3]);
        }
    }
```

整个程序的结构清晰，很容易修改和扩展，完全可以用于实际。读者可以在函数 DoCommand 中放入自己的处理程序。校验方式也很容易换成其他的方式，只需要修改 JiaoYan 函数的内容就可以了。如果需要一次传送更多的数据，可以修改 xieyi.h 中定义数据包大小的宏#define N_XY_BAO 为所需要的数值，程序中有关数据包大小的部分都会随着 N_XY_BAO 所定义的值而变动。指令也可以自行增加和定义，仍然是修改 xieyi.h 中定义指令的宏。也可以将程序改为多机通信模式，下面介绍改为多机通信模式的步骤。

（1）修改 USART0 接收中断函数如下：

```
    #pragma vector=USART0RX_VECTOR
    __interrupt void Usart0Rx()
    {
        #define ADRESS 0x01             //本机地址
        if((U0RCTL& URXWIE)==URXWIE)
        {
            //接收数据为地址
            if(RXBUF0==ADRESS)          //判断是否是本机地址
            {
                U0RCTL &= ~URXWIE;      //是本机地址，则修改为接收数据，也可产生中断
            }
            else
            {
                U0RCTL |= URXWIE;       //不是本机地址，则以后接收到数据不产生中断
            }
        }
        else
        {
            //接收数据
            if((U0RCTL&RXERR)==0)
            {
                AddUsData(RXBUF0);
```

```
            }
            else
            {
                SetBaoErr();
                U0RCTL &= ~(FE+PE+OE+BRK);
            }
        }
    LPM3_EXIT;
}
```

（2）在中断函数中增加对接收数据是地址还是数据的判断。当收到的数据是地址的时候，将收到的字节与本机地址进行对比，如果相等，则表明主机将与本机进行通信，复位寄存器 U0RCTL 的 URXWIE 位。此后收到数据后将触发接收中断，在中断程序中将收到的数据存入接收缓冲区。如果收到的地址与本机地址不符，则说明主机以后发送的数据不是针对本机的，置位寄存器 U0RCTL 的 URXWIE 位，此后接收到数据字节将不会触发中断。

（3）修改串行口的初始化函数 UartInit( )中对工作模式的设置。

如果使用地址位多机协议，则改为：

```
UCTL0=CHAR+MM+SWRST;                //8 位数据，1 位停止位，地址位多机协议
```

如果使用线路空闲多机协议，则改为：

```
UCTL0=CHAR+SWRST;                   //8 位数据，1 位停止位，空闲位多机协议
```

最后增加一句程序使 CPU 进入多机通信状态：

```
U0RCTL |= URXWIE;                   //只有地址字符使 URXIFG 置位
```

第7章 定 时 器

定时器是微处理器中非常重要的一个模块，其功能的强弱直接影响微处理器的工作能力。MSP30 的定时器比较复杂，有基本定时器和定时器 A、定时器 B、看门狗定时器，不同型号的微处理器具有不同的定时器。定时器 A 和定时器 B 结构上基本一致，是 16 位定时器，为 MSP430 主要使用的定时器。本章 7.1 节介绍 16 位定时器的使用方法，并给出了 4 个比较典型的使用实例。

基本定时器除了有定时功能，还被作为液晶输出模块的时钟信号。基本定时器的定时功能没有 16 位定时器的功能强大。本章 7.2 节介绍基本定时器的使用方法。

7.1 16 位定时器

MSP430 的 16 位定时器能够同时支持多个比较与捕获以及 PWM 输出、独立定时的功能。定时器 A 与定时器 B 结构上很类似，大致分为 3 个单元：计数单元、捕获/比较单元、输出单元。

以定时器 B 为例，定时器的计数单元以计数器 TBR 为核心，计数时钟源可以选择 MCLK、ACLK、SMCLK、外部时钟。时钟信号可以被 1、2、4、8 分频，以降低计数频率。计数模式可以选择 4 种，如表 7-1 所示。定时器 B 中的各个模块共用一个计数单元。定时器 A 与定时器 B 相同。

表 7-1 定时器计数模式

模 式	描 述
0：停止	计数器停止计数
1：增计数	从 0 开始增计数，记到大于等于比较寄存器 TBCL0 中的值时，定时器复位，并从 0 开始重新计数
2：连续	定时器从当前值开始计数，当计到最大计数值后又重新从 0 开始计数。最大计数值与计数位数有关，8 位为 0xFF、10 位为 0x3FF、12 位为 0xFFF、16 位为 0xFFFF
3：增/减	计数器的值先增后减，当计数值增加到 TBCL0 的值时，计数器由增计数转变为减计数

定时器中的捕获/比较单元有多个，如定时器 TimerB7 中有 7 个捕获/比较单元，定时器 TimerA3 中有 3 个捕获/比较单元，每一个捕获/比较单元都有相同的结构。工作时首先将捕获/比较寄存器 CCRx 中的值载入寄存器 TBCLx。由寄存器 TBCLx 中的值与计数器中的值进行比较，相等时驱动输出单元。比较方式通常用于定时，根据计数器所选的时钟频率计算出定时时间所需的计数值，载入寄存器 TBCLx。定时器启动后，计数器中的值通过比较器与寄存器 TBCLx 中的值比较，相等则定时时间到。捕获方式通常用于记录某一事件发生的时间。当捕获源的信号满足捕获条件时，硬件自动将寄存器 TBR 中的值写入寄存器 TBCCRx。例如要测量一个方波的周期，可以定义上升沿捕获。每次捕获时，读出寄存器 TBCCRx 中的值，两次读出的计数值的差所代表的为两次上升沿之间的时间，即方波的周期。根据计数器所选择的时钟的频率与计数值可以计算出方波的真正周期。无论是捕获还是比较模式，都可以触发中断。

每个捕获/比较单元都对应一个输出单元，用于输出信号。有 8 种工作模式可以选择，分别是：输出模式、置位模式、PWM 翻转/复位模式、PWM 置位/复位模式、翻转模式、复位模式、PWM 翻转/置位模式、PWM 复位/置位模式。

定时器 A 与定时器 B 结构上的不同在于，定时器 A 中直接进行 TAR 与 CCRx 的对比，但定时器 B 中与 TBR 进行比较的寄存器不是 CCRx，而是 TBCCLx，工作前需要先将寄存器

CCRx 的值装载进寄存器 TBCCLx。

7.1.1　定时中断

定时中断是定时器最基本的应用方式，其功能为，预定一个时间值，到时间后产生中断。定时中断可以用来计时，也可以按照一定的频率执行某一段程序，如定时信号检测。

程序 7-1 的功能为，随着程序循环的次数修改定时时间，程序每循环一次，改变一次发光二极管的状态：点亮或者熄灭。程序的每一个周期循环结束后，CPU 就进入低功耗模式。定时时间到后，触发中断，使 CPU 退出低功耗模式，开始新的循环。由于定时时间不断地改变，发光二极管亮灭的周期也随着不断地改变。

程序 7-1 的工作环境为：MCLK 使用 8MHz，ACLK 使用 32.768kHz。定时器使用定时器 B 中的模块 0，计数时钟源选择 ACLK，计数器工作在增计数模式。捕获/比较单元选择比较模式。本程序没有用到输出单元。

main.c 是主程序。由于此程序的算法和执行部分过于简单，而且不具备实际意义，所以，没有将它们单独作为模块，直接写在了主程序中。首先调用 InitSys 函数进行系统初始化，然后进入程序主循环部分。程序每循环一次，变量 Cnt 减 1，当 Cnt 为 0 时，为 Cnt 恢复初值 TIME_FA。定时器的定时值保存在变量 TimeZhi 中，每次 Cnt 为 0 时，TimeZhi 加 1，如果 TimeZhi 为 0xFFFF，则 TimeZhi 赋值为 32（任意设定的）。TimeZhi 作为参数，调用 SetTime 函数修改定时器的定时时间。接着翻转发光二极管的亮灭状态。最后，CPU 进入低功耗模式，等待定时器定时时间到时被唤醒。由于每次 Cnt 减小到 0 值就改变定时时间，所以，发光二极管的亮灭周期不断地改变。

df_timerb.c 是定时器 B 的模块驱动程序。其中函数 TimerBinit 对定时器 B 进行初始化。函数 SetTime 设置定时时间。函数 GotimeDfB 打开或者关闭定时器。

函数 TimerBinit 中选择 ACLK 作为计数器的时钟源，复位定时器和输入分频器，使计数器处于停止计数状态，并打开定时器中断，使捕获/比较单元工作在比较模式。

函数 SetTime 设置定时时间。只是把参数 ti 赋值给 TBCCR0。定时器的计数频率与所选择的计数时钟频率有关，如果时钟频率不同，那么 TBCCR0 中同样的值产生的实际定时时间也不同。

函数 GotimeDfB 打开或关闭定时器。打开定时器的同时，将定时器和输入分频器作了复位。打开定时器就是使计数器开始计数，脱离停止模式。关闭定时器就是使计数器进入停止模式。

定时器中断函数 Timer_B0 在计数器的值等于 TBCCR0 时被触发，即定时时间到的时候被触发。惟一做的事情就是使 CPU 在退出中断后脱离低功耗模式。

程序 7-1 的结构非常简单，程序结构图省略。

程序 7-1：

```
df_timerb.h
#ifndef __DF_TIMER_B
#define __DF_TIMER_B
void TimerBInit();
void SetTime(unsigned int ti);
```

```
void GotimeDfB(unsigned char doit);
#endif

df_timerb.c
#include <MSP430x14x.h>
#include "df_timerb.h"

void TimerBInit()
{
    TBCTL=TBSSEL_1+TBCLR+MC_0; //选择时钟源为 ACLK
    TBCCTL0=CCIE;              //允许定时器中断
}

/*********************************************************************
设置定时时间
ti：要定时的时间，与时钟源的频率有关
*********************************************************************/
void SetTime(unsigned int ti)
{
    TBCCR0=ti;                 //定时时间
}

/*********************************************************************
打开或关闭定时器
doit：0：停止   100：运行
*********************************************************************/
void GotimeDfB(unsigned char doit)
{
    if(doit==100)
    {
        TBCTL |= MC_1+TBCLR;   //打开定时器
    }
    else if(doit==0)
    {
        TBCTL &= ~MC0;         //关闭定时器
    }
}

#pragma vector=TIMERB0_VECTOR
```

```
__interrupt void Timer_B0(void)//定时器中断函数
{
    LPM3_EXIT;
}

main.c
#include <MSP430x14x.h>
#include "df_timerb.h"

void InitSys();             //初始化

#define TIME_FA 500         //定时中断产生 TIME_FA 次后，改变定时时间
unsigned int Cnt=TIME_FA;   //中断次数记录
unsigned int TimeZhi=32;    //定时时间

#define LED_DIR P4DIR       //定义 LED 输出管脚
#define LED_OUT P4OUT
#define LED_IO BIT1
unsigned char Led=0;        //记录 LED 是亮还是灭。0：灭；其他：亮

int main( void )
{
    unsigned char q0;
    unsigned char *pq0;
    WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
    InitSys();              //初始化
start:
    if(Cnt != 0)
        Cnt--;
    else
    {
        Cnt=TIME_FA;
        if(TimeZhi<0xFFFF)                  //改变定时器的定时时间
            TimeZhi++;
        else
            TimeZhi=32;

        SetTime(TimeZhi);                   //设置定时时间
        Led=!Led;                           //翻转 LED 状态
```

```
        if(Led==0)
            LED_OUT &= ~LED_IO;             //熄灭 LED
        else
            LED_OUT |= LED_IO;              //点亮 LED
    }
    LPM3;                                   //进入低功耗模式
    goto start;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    unsigned int iq0;

    BCSCTL1 &= ~XT2OFF;                     //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                     //清除振荡器失效标志
        for(iq0=0xFF; iq0>0; iq0--);        //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!= 0);             //判断 XT2 是否起振

    BCSCTL2=SELM_2;                         //选择 MCLK 为 XT2
    LED_DIR |= LED_IO;
    TimerBInit();                           //定时器 B 初始化
    SetTime(TimeZhi);                       //设置定时时间
    GotimeDfB(100);                         //打开定时器
    _EINT();                        //打开全局中断控制，若不需要打开，可以屏蔽本句
}
```

7.1.2 PWM 输出

PWM 是 pulse-width modulated 的缩写，称为脉宽调制，意思是对输出方波的一个周期中高电平和低电平所占的时间比例进行调整。这是一项应用十分广泛的技术，比较典型的应用是利用 PWM 进行 D/A 转换。在 PWM 输出管脚的后面接一低通滤波电路，滤掉 PWM 的高频成分，由于 PWM 信号中的高低电平所占比例不同，因此对滤波电路中滤波电容的充放电程度也就不同，从而可以实现输出不同的模拟电压信号。进行 D/A 转换时改变的是输出方波信号的占空比，而不是方波的频率。

由于 MSP430 管脚的驱动能力有限，所以最好不要直接在输出管脚上接滤波电路，应该先在输出管脚上接一个缓冲电路，滤波电路接在缓冲电路后面。典型的低通滤波器如图 7-1 所示，其中电阻和电容的取值需要根据具体情况计算。

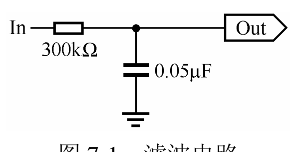


图 7-1　滤波电路

程序 7-2 的功能为接收串行口传输过来的指令。根据指令控制 PWM 输出的运行和停止以及输出的频率和占空比。

程序 7-2 的工作环境为：MCLK 使用 8MHz，ACLK 使用 32.768kHz。使用定时器 B 输出 PWM 信号。定时器 B 的计数器时钟源选用 ACLK，捕获/比较单元使用单元 0 和单元 1，输出单元工作模式选择模式 7：PWM 复位/置位模式，PWM 的输出管脚为 P4.7（TB1）。PWM 信号的输出过程如图 7-2 所示，输出电平在计数器 TBR 等于 TBCL1 时复位，等于 TBCL0 时置位。改变 TBCL0 的值可以改变输出信号的周期，改变 TBCL1 的值则可以改变输出信号的占空比。TBCL0 的值应该大于 TBCL1 的值。输出选择模式 2、3、6、7 均可以输出 PWM 信号，只是输出的波形有所区别。

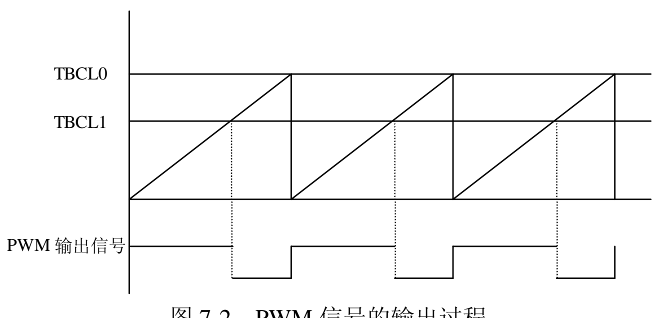


图 7-2　PWM 信号的输出过程

pwm_timer_b.c 是定时器 B 的驱动模块。函数 TimerBInit 对定时器 B 进行初始化。函数 SetScale 改变占空比。函数 SetFre 设置输出方波的频率。函数 GoPwm 打开或者关闭脉宽调制信号的输出。

函数 TimerBInit 对定时器 B 进行初始化。选择计数器的时钟源为 ACLK，输出单元工作在 PWM 输出复位/置位模式。捕获/比较模块 1 对应的输出管脚是 P4.7(TB1)，选择其工作在第二功能，信号方向为输出。PWM 信号的输出不需要程序干预，定时器和输出管脚设置完毕，进入工作状态后，完全由硬件控制 PWM 信号的输出。

函数 SetScale 改变占空比，改变寄存器 TBCL1 的值可以改变占空比。改变寄存器 TBCL1 的值的过程是先改变 TBCCR1 的值，然后再载入 TBCL1。不过函数 SetScale 没有直接改变 TBCCR1 的值，而是将参数 sc 的值赋给了一个全局变量 iScale。只有在打开 PWM 输出（调用 GoPwm 函数）的时候，才将变量 iScale 的值赋给寄存器 TBCCR1。如果不这样做，那么在需要修改定时器的多个参数时，就会出现先被修改参数已经起作用了，但还有一些参数未被修改的情况。这样，在很短的时间内，程序的输出会出现非预期的值，在某些情况下会导致严重的后果。参数 sc 的取值范围为 0~0xFFFF，且必须小于变量 iFre 的值。

函数 SetFre 的作用是改变 PWM 信号的输出频率。其工作过程与函数 SetScale 类似，只是将参数值赋给全局变量 iFre。直至调用 GoPwm 函数的时候，才会将变量 iFre 的值赋给寄存器 TBCCR0，改变 PWM 信号的输出频率。参数 fre 的取值范围为 0~0xFFFF。

函数 GoPwm 打开或者关闭脉宽调制信号的输出。参数 doit 为 0 时停止，为 100 时开始输出 PWM 信号。函数的返回值返回 PWM 信号输出的状态：正在输出还是停止输出。程序进入本函数，首先关闭定时器的计数单元，停止 PWM 的输出。如果参数 doit 的值为 100，则打开 PWM 输出，将 iScale 和 iFre 的值分别赋给寄存器 TBCCR1 和 TBCCR0，然后再打开计数器。计数器工作在模式 2，连续计数模式。

main.c 是主程序，来自程序 6-1。由于使用了定时器 B 进行 PWM 信号的输出，所以在初始化函数 InitSys 中增加了函数 TimerBInit 的调用，对定时器 B 进行初始化。串行口用来与 PC 通信，对其初始化函数 UartInit 的调用仍然保留。

在本程序中，pwm_timer_b.c 与程序 6-1 异步串行通信程序被结合在一起，由 PC 机通过串行口来控制 PWM 输出的频率、占空比以及 PWM 信号的输出与停止。从示范 PWM 输出的角度来说，没有必要这么做，pwm_timer_b.c 再加上一个简单的 main.c 文件就足够了。不过这么做是一个展示结构化编程好处的机会，可以看到，只要认真地按照结构化编程的思路去做，重复利用代码将会非常容易。

df_uart.C 与程序 6-1 中的代码相比没有任何改变。在此不再重复。

图 7-3 是程序 7-2 的结构框图。

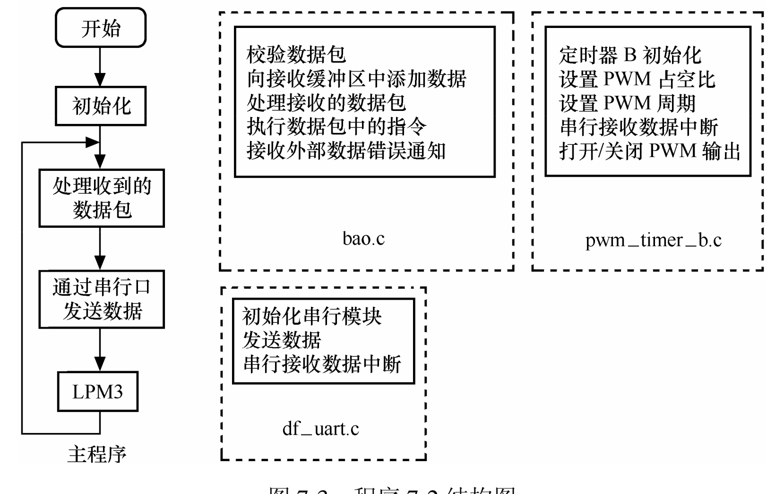


图 7-3　程序 7-2 结构图

程序 7-2：

```
pwm_timer_b.h
#ifndef __PWM_TIMER_B
#define __PWM_TIMER_B
void TimerBInit();
void SetScale(unsigned int sc);
void SetFre(unsigned int fre);
unsigned char GoPwm(unsigned char doit);
#endif
```

```
pwm_timer_b.c
#include <MSP430x14x.h>
#include "pwm_timer_b.h"

#define PWM_DIR P4DIR
#define PWM_SEL P4SEL
#define PWM_OUT P4OUT
#define PWM_IN  P4IN
#define PWM_IO BIT1

unsigned int iFre;                  //脉冲周期
unsigned int iScale;                //占空比

/**********************************************************************
初始化定时器B
**********************************************************************/
void TimerBInit()
{
    TBCTL=TBSSEL_1+TBCLR ;          //时钟源为ACLK
    TBCCTL1=OUTMOD_7;               //工作在PWM复位/置位模式

    PWM_SEL |= PWM_IO;              //选择输出端口的第二功能
    PWM_DIR |= PWM_IO;
}

/**********************************************************************
设置占空比
sc: 要修改的占空比值
**********************************************************************/
void SetScale(unsigned int sc)
{
    iScale=sc;
}

/**********************************************************************
设置频率
fre: 要修改的输出频率值
**********************************************************************/
void SetFre(unsigned int fre)
```

```
{
    iFre=fre;
}

/*********************************************************************
控制 PWM 运行或者停止
doit: 0: 停止; 100: 运行; 其他: 什么都不做，只返回运行状态
返回值: 运行状态。 同 doit 的值
******************************** *************************************/
unsigned char GoPwm(unsigned char doit)
{
    TBCTL &= ~(MC0+MC1);          //关闭定时器
    if(doit==100)
    {
        TBCCR1=iScale;
        TBCCR0=iFre;
        TBCTL |= MC_2;            //打开定时器，计数模式 2
    }
    if((TBCTL&MC_0)==0)           //判断 PWM 是否运行
        return 0;
    else
        return 100;
}

main.c
#include <MSP430x14x.h>
#include "pwm_timer_b.h"
#include "bao.h"
#include "df_uart.h"
//函数声明
void InitSys();

int main( void )
{
    unsigned char q0;
    unsigned char *pq0;
    WDTCTL=WDTPW+WDTHOLD;         //关闭看门狗
    InitSys();                    //初始化
start:
```

```
        pq0=DoUart(&q0);
        //发送数据
        if(q0!=0)
        {
            SendUart(pq0,q0);
        }
        LPM3;                          //进入低功耗模式3
        goto start;
    }

    /**********************************************************************
    系统初始化
    **********************************************************************/
    void InitSys()
    {
        unsigned int iq0;

        //使用XT2振荡器
        BCSCTL1 &= ~XT2OFF;                        //打开XT2振荡器
        do
        {
            IFG1 &= ~OFIFG;                        //清除振荡器失效标志
            for (iq0=0xFF; iq0>0; iq0--);          //延时，等待XT2起振
        }
        while ((IFG1 & OFIFG) != 0);               //判断XT2是否起振

        BCSCTL2=SELM_2+SELS;                       //选择MCLK、SMCLK为XT2
        TimerBInit();                              //初始化定时器B
        UartInit();                                //初始化串行口
        _EINT();    //打开全局中断控制，若不需要打开，可以屏蔽本句
    }
```

xieyi.h中定义的通信协议只对指令部分和传送数据的内容进行了重新定义，其他不变。重新定义部分如下:

```
    //PC传送给实验板的指令
    #define NONE_COMMAND 0          //空的指令，什么都不做
    #define STOP_COMMAND 0x10       //停止PWM输出
    #define GO_COMMAND 0x20         //开始PWM输出
    #define FRE_COMMAND 0x30        //发送PWM输出频率
    #define SC_COMMAND 0x40         //发送PWM输出占空比
```

```
/**********************************************************************
//PC 传送给实验板的通信包
每个字节代表的含义:
1 数据包头(HAND_BAO)
2 指令
3 数据高字节
4 数据低字节
5 校验

指令为 FRE_COMMAND 时数据为 PWM 输出的频率
指令为 SC_COMMAND 时数据为 PWM 输出的占空比

//实验板传给 PC 的通信包
每个字节代表的含义:
1 数据包头(HAND_BAO)
2 指令
3 数据高字节
4 数据低字节
5 校验

发送数据包后，需等待接收对方的校验信息 HAND_OK 或者 HAND_ERR。
收到数据包校验正确后发出 HAND_OK，否则发出 HAND_ERR。
**********************************************************************/
```

bao.c 是处理数据包的模块。因为执行的指令的内容发生变化，所以需要修改执行指令的函数 DoCommand。新的 DoCommand 函数如下:

```
/**********************************************************************
执行指令
**********************************************************************/
unsigned int DoCommand(unsigned char comd)
{
    unsigned int iq0;
    switch(comd)
    {
        case STOP_COMMAND:
            //执行 STOP_COMMAND 指令
            return (GoPwm(0));
        case GO_COMMAND:
            //执行 GO_COMMAND 指令
```

```
                return (GoPwm(100));
            case FRE_COMMAND:
                //执行 FRE_COMMAND 指令
                iq0=(aRxBuff[2]<<8)+aRxBuff[3];
                SetFre(iq0);
                return iq0;
            case SC_COMMAND:
                //执行 SC_COMMAND 指令
                iq0=(aRxBuff[2]<<8)+aRxBuff[3];
                SetScale(iq0);
                return iq0;
        }
    }
```

7.1.3　捕获脉冲信号周期

捕获方式是定时器的另一项基本功能。捕获方式主要用来监测数字信号的变化，分为 3 种情况：上升沿捕获、下降沿捕获、上升沿和下降沿都捕获。每次捕获时 MSP430 都会通过硬件自动将计数器中的值保存在所使用的捕获模块的寄存器 CCRx 中。如果被捕获信号发生的变化符合捕获的条件，就会触发捕获中断，从而使单片机能够对被捕获的信号进行迅速的处理。如果是周期信号，则两次上升沿或者两次下降沿之间的时间即为信号的周期，上升沿和下降沿之间的时间则为信号的脉宽（高电平或者低电平的时间长度）。在后面的软件模拟异步串行通信程序中，接收数据时就是通过捕获起始位的电平变化来开始读取串行数据的。

程序 7-3 的功能是测量从管脚 P2.2（TA0）输入的方波信号的周期。工作方法是通过捕获信号的下降沿来触发中断，然后在中断函数中读取定时器所捕获的计数器的值。两次捕获的计数器值的差值即为方波信号的周期。

程序 7-3 的工作环境为：MCLK 为 8MHz，ACLK 为 32.768kHz。使用定时器 A，计数器时钟源选择 ACLK，工作在连续计数模式。捕获模块的工作模式为下降沿捕获。要测量的方波信号从管脚 P2.2 输入。

main.c 是主程序。函数 main 首先调用函数 InitSys，进行系统初始化，然后，就进入低功耗模式。主程序中没有将测到的信号输出的部分，输出需要外部设备，如串行通信口、显示器等。每一个外部设备都需要驱动程序，这会大量增加代码的长度，而这部分与本节的主题无关，因此，测量的结果就简单地保存在程序的变量 Cyc 中。可以通过仿真器设置断点来观察测量的结果。

函数 InitSys 进行系统初始化。先初始化定时器 A，然后打开捕获功能。

df_bh_timera.c 为定时器 A 的驱动程序。函数 InitBhTimerA 对定时器 A 进行初始化。函数 GoBhTimerA 控制捕获的运行或者停止。函数 GetCyc 返回检测到的信号周期。TimerA0 为定时器 A 的中断函数。

函数 InitBhTimerA 的初始化与程序 7-2 中对定时器 B 初始化的过程类似。首先选择计数

器的时钟源和工作模式。然后选择比较/捕获单元工作在捕获模式，捕获模式为下降沿捕获。最后选择信号输入管脚 P2.2 使用第二功能，接收要捕获的信号。

函数 GoBhTimerA 控制捕获的打开和关闭。打开捕获时，打开计数器，打开定时器中断。关闭捕获时，停止计数器的运行，关闭定时器中断。

定时器 A 中断函数在发生捕获时被触发，首先计算本次寄存器 CCR0 中的值与变量 LastCCR0 的差值，LastCCR0 是上一次捕获时记录的寄存器 CCR0 的值。差值保存在全局变量 Cyc 中，Cyc 即为信号的周期。然后，将本次读取的 CCR0 值赋给全局变量 LastCCR0，在下一次捕获时使用。因为计数器工作在连续计数模式，所以，本次 CCR0 中的值可能会小于 LastCCR0。但 Cyc 是无符号整数类型，在计算差值时一旦结果为负值，就会取补码作为结果，对测量信号周期的最终结果没有影响。要注意的是，计数器最大计数值为 0xFFFF，在信号的一个周期内计数器所计的数值应该小于 0xFFFF。如果信号周期比较长，就应该降低计数器时钟的频率，可以采用分频的方法，也可以使用频率更低的晶体振荡器。

函数 GetCyc 提供一个数据接口，需要做的工作仅仅是返回检测到的信号周期。其他的程序模块可以通过调用此函数获得检测到的信号周期。

利用程序 7-3 可以做一个简单的实验：测量 PC 机串行口发出数据的周期。将 PC 串行口输出的信号接到定时器 A 的信号输入端 P2.2。要注意的是，PC 输出的信号为 RS-232 电平，必须经过电平转换才可以接到 MSP430 的管脚上，否则会损坏 MSP430。运行程序 7-3 后，使 PC 发送数据 0xAA（二进制：10101010），然后暂停程序的运行（不要把断点放在 TimerA0 中断函数中，否则会影响捕获的准确性），在 watch 窗口查看变量 Cyc。以下为测量结果：

PC 发送的波特率	测量结果
9600bit/s	Cyc=7
4800bit/s	Cyc=14
2400bit/s	Cyc=27

对测量结果进行验证。波特率的含义为串行通信每秒钟所传送的位数。由于传送的数据是 10101010，两位为一个周期，所以每秒发送的周期数为波特率×2。捕获所用的时钟频率为 32.768kHz。有如下算式：

$$Cyc=32768\times 2/\text{波特率}$$

计算结果为：

PC 发送的波特率	计算结果
9600bit/s	Cyc=6.8
4800bit/s	Cyc=13.65
2400bit/s	Cyc=27.3

由于是通过计数器的计数值来记录时间的，所以 MSP430 的测量结果不可能出现小数。对计算结果进行 4 舍 5 入取整后，可以看出测量结果与计算结果相符。

程序 7-3 的结构非常简单，程序结构图省略。

程序 7-3：

```
df_bh_timera.h
#ifndef __DF_BH_TIMERA
```

```
#define __DF_BH_TIMERA
void InitBhTimerA();
void GoBhTimerA(unsigned char doit);
unsigned int GetCyc();
#endif

df_bh_timera.c
#include <MSP430x14x.h>
#include "df_bh_timera.h"

#define BHSEL P2SEL                    //定义捕获端口
#define BH_IO BIT2

unsigned int Cyc;                      //检测到的信号周期
unsigned int LastCCR0;                 //上一次测到的 CCR0 值
/**********************************************************************
初始化
**********************************************************************/
void InitBhTimerA()
{
    TACTL=TASSEL_1+TACLR;              //定时器 A，时钟源：ACLK，连续计数模式
        CCTL0=CCIS_1+CM_2+CAP;         //选择下降沿捕获，CCIB（P2.2）为信号源
        BHSEL |= BH_IO;                //选择 P2.2 使用第二功能
}

/**********************************************************************
控制捕获运行或者停止
doit：0：停止；100：运行；其他：什么都不做
**********************************************************************/
void GoBhTimerA(unsigned char doit)
{
    if(doit==0)
    {
        TACTL &= ~MC1;             //关闭计数器
        CCTL0 &= ~CCIE;            //关闭中断
    }
    else
    {
        TACTL |= MC_2+TACLR;       //打开计数器
```

```
        CCTL0 |= CCIE;          //捕获中断允许
    }
}

/**********************************************************************
定时器 A 中断函数
中断源：CC0
**********************************************************************/
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0()
{
    Cyc=CCR0-LastCCR0;          //计算周期
    LastCCR0=CCR0;              //保存 CCR0 的值
}

/**********************************************************************
获得检测到的信号周期
返回值：检测到的信号周期（时钟计数值）
**********************************************************************/
unsigned int GetCyc()
{
        return Cyc;
}

main.c
#include <MSP430x14x.h>
#include "df_bh_timera.h"
//函数声明
void InitSys();

int main( void )
{
    WDTCTL=WDTPW+WDTHOLD;       //关闭看门狗
    InitSys();                  //初始化
        LPM3;                   //进入低功耗模式 3
}

/**********************************************************************
系统初始化
```

```
***********************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;                        //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                        //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--)           //延时，等待 XT2 起振
            ;
    }
    while ((IFG1 & OFIFG)!= 0);                //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                       //选择 MCLK、SMCLK 为 XT2

        InitBhTimerA();                        //初始化定时器 A
        GoBhTimerA(100);                       //打开捕获功能
        _EINT();
}
```

7.1.4　软件模拟异步串行通信

因为软件模拟串行通信需要使用定时器，所以将它放在定时器的这一章讲述。异步串行通信协议的基本内容已经在 6.1 节讲过。下面介绍通信的工作原理。

1．发送原理

定时器计数单元工作在连续模式，捕获/比较单元工作在比较模式。计数器连续从 0 计数到 0xFFFF，然后又从 0 开始计数。首先按照串行通信使用的波特率计算出发送的数据中两位之间的时间间隔 t，使寄存器 CCR0 值等于计数器 TAR 的当前值加上时间间隔 t，然后打开计数器。当定时器计数器 TAR 的值等于寄存器 CCR0 的值时，触发中断。数据位的输出利用定时器输出单元的两种工作模式（置位和复位）进行。如果要发送的位为高电平，则输出单元设置为置位模式，否则设置为复位模式。定时器中断时，硬件自动按照所设置的输出模式输出电平，这样要发送的位就被自动发送出去了。在中断程序中根据下一位要发送数据的电平设置输出模式。如此反复进行，直至 10 位（1 起始位+8 数据位+1 停止位）数据全部发送完毕。

2．接收原理

定时器的计数单元工作在连续模式，捕获/比较单元最初工作在捕获模式。接收数据时，

首先捕获到起始位的下降沿，并触发中断。在中断程序中将捕获/比较单元的工作模式改为比较模式。按照串行通信使用的波特率计算出发送的数据中两位之间的时间间隔 t。因为捕获是发生在起始位的前沿，如图 7-4 所示，所以转换为比较模式后，首次为 CCR0 赋的值应为 1.5t。这样定时器定时时间到时，进行第一次采样的采样点在收到的第一个数据位中点。考虑到每次采样是在中断程序中进行的，进入中断程序会有一些延迟，因此，将定时时间缩短，稍微小于 1.5t，这样能够尽量使采样点接近每一位数据的中点。因为每位数据的发送时间为 t，所以除了第一次定时时间取 1.5t，其他每次采样后都将寄存器 CCR0 的值赋为 t，这样可使每次都在数据位的中点采样。如此循环进行，直至接收完所有数据位和停止位。

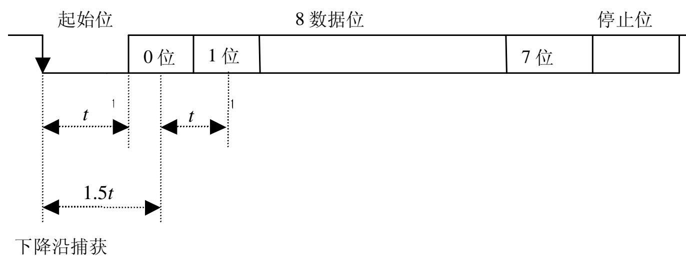


图 7-4　软件模拟串行通信接收原理示意图

此程序如果采用频率较高的时钟源，可以使用更高的波特率进行通信。

程序 7-4 的功能为利用定时器 A 的捕获和比较功能模拟串行通信，波特率为 2400bit/s，半双工。通信的内容与程序 6-1 相同，仍然为对接收到的数据包进行分解，执行数据包中的指令，然后返回处理的结果。如果数据包有错误，则返回错误信息，通知发送方重新发送。

程序 7-4 的工作环境为：MCLK 为 8MHz，ACLK 为 32.768kHz。管脚 P1.1 为接收端（RXD），管脚 P2.2 为发送端（TXD）。定时器 A 计数器时钟源选择 ACLK。所使用的串行通信格式为：1 个起始位，8 个数据位，1 个停止位，没有校验位。起始位为低电平，停止位为高电平。传输时低位在前，高位在后。

softuart.c 为软件模拟串行通信的文件模块。函数 InitSfUart 初始化定时器 A 以及通信所用到的管脚。由于是半双工通信，所以接收和发送数据不能同时进行。函数 InitRxd 使串行通信进入接收状态。函数 SendUart 发送数据。函数 TimerA0 为定时器 A 的中断函数。函数 ReadBuf 为其他程序模块提供一个接口，调用者可以通过调用此函数获得所接收到的数据。函数 GetRTx 也提供一个接口，调用者可以通过调用此函数查询发送数据是否完成，或者是否收到完整的数据。

函数 InitSfUart 初始化定时器 A，选择定时器 A 计数器工作在连续计数模式，时钟源为 ACLK。捕获/比较单元工作在比较模式，处于数据发送状态。由于停止位为高电平，在发送完数据后，数据发送管脚会保持在高电平，所以，初始化输出模式为置位模式。数据发送和接收管脚都选择第二功能。

函数 InitRxd 使通信进入接收状态。全局变量 CntWei 被赋值为要接收的位数，中断程序中根据此变量来判断全部数据位是否接收完毕。定时器 A 被初始化为捕获模式，并置位捕获中断允许位，准备捕获起始位的下降沿。

函数 SendUart 的功能是发送数据。调用者要发送的数据为 8 位，加上起始位和停止位共

为 10 位。全局变量 TxBuf 保存要发送的 10 位数据（包含起始位和停止位），为 16 位的无符号整数类型。将要发送的 8 位数据左移一位，最低位（起始位）自动为 0，然后通过位屏蔽的方法将第 9 位（停止位）置 1。这样变量 TxBuf 的第 0 位为起始位 0，第 1~8 位为数据位，第 9 位为停止位 1，发送时只要依次从第 0 位开始逐位发送就可以了。全局变量 CntWei 保存要发送的位数，这里为 10 位。将当前 TAR 的值与一个数据位的时间长度 t 之和赋值给寄存器 CCR0，确定发送第一位的数据的时间。由于停止位是高电平，所以使输出模式为置位模式。定时器第一次触发中断时，首先发送管脚自动输出高电平，然后在中断程序中才开始按照变量 TxBuf 中的数据位设置输出模式。这样通信线路上先出现一个高电平，然后再出现起始位的低电平。这样做的目的是确保通信线路上出现一个明确的起始位，增加通信的可靠性。进入中断程序后，必须首先判断是工作在发送状态还是接收状态，这可以通过寄存器 CCTL0 中的 CCIS0 位进行判断。当选择 CCI0B（捕获事件的信号由管脚 P1.1 输入）为捕获事件的输入信号源时，CCIS0 为 1，表明通信工作在接收状态，否则，工作在发送状态。由于进入了发送状态，CCIS0 被复位，CCI0B 不作为捕获事件的输入信号源。

函数 TimerA 是模拟串行通信的核心，主要分为接收部分和发送部分两部分处理。判断执行哪部分程序的依据是寄存器 CCTL0 中的 CCIS0。

在接收部分，首先判断是否工作在捕获模式。如果工作在捕获模式，则表明刚捕获的是起始位的下降沿，将工作模式修改为比较模式，定时时间设置为 1.5t。如果工作在比较模式，表明接收的是数据。收到的数据的电平通过寄存器 CCTL0 的 SCCI 位读出，TAR 的值与寄存器 CCR0 的值相等时，从 CCI0B（P1.1）输入的信号电平被锁存在 SCCI 位中。收到的数据为先低位后高位，变量 RxBuf 的作用为保存收到的数据。先将其右移一位，然后根据收到的数据值，采用位屏蔽的方法将 RxBuf 的最高位置位或者复位。如此循环，每收到一位，变量 CntWei 减 1，当 CntWei 为 0 时，所有的位接收完毕，关闭定时器中断，退出接收状态。将接收完成标志变量 bRTx 置位，退出低功耗模式，以便继续执行程序，处理收到的数据。

在发送部分，要发送的数据已经在函数 SendUart 中被保存到变量 TxBuf 中，发送时首先发送低位，根据 TxBuf 的最低位的值确定定时器的输出模式是置位还是复位，下一次定时时间到时，这一位就被发送出去。每次发送完成，TxBuf 右移一位，为下次发送作准备。每次循环判断 CntWei 是否为 0，如果为 0，则表明已经发送完毕，关闭定时器中断，将发送完成标志变量 bRTx 置位。

函数 ReadBuf 返回串行通信收到的数据，并清除 bRTx 位。

函数 GetRTx 返回 bRTx 的值，调用者通过它获知发送或者接收数据是否完成。

main.c 是主程序。由系统初始化函数 InitSys 和 main.c 组成。

函数 InitSys 调用函数 InitSfUart 初始化软件串行口，接着调用函数 InitRxd，使串行口进入接收状态。最后，打开全局中断。

函数 main 与程序 6-1 中的 main 函数基本一样。由于 softuart.c 中发送和接收的缓冲区都只有一个字节，所以，必须在接收或者发送一个字节后及时处理缓冲区。main 函数中每接收完一个字节数据就会退出低功耗模式，进行一次主程序的循环。循环首先调用函数 GetRTx，查询是否收到一个完整的字节。如果收到，则调用 ReadBuf 函数读出收到的数据，然后调用函数 AddUsData 将收到的数据添加入数据包，以等待处理。每次循环都会调用函数 DoUart

来处理收到的数据包，函数 DoUart 来自文件 bao.c，与程序 6-1 中的完全一样，这里不再重复给出。函数 DoUart 处理完数据包后，将要发送的字节数存放在变量 q0 中。如果 q0 不为 0，则表明有数据需要发送。每次循环都会调用函数 SendUart 来发送数据，并通过调用函数 GeiRTx 查询是否发送完一个字节，调用函数 ReadBuf 是为了复位 bRTx 位，为发送下一字节作准备。发送完所有字节后，串行口进入接收状态，保持对串行口的监听。

图 7-5 为程序 7-4 的结构图。

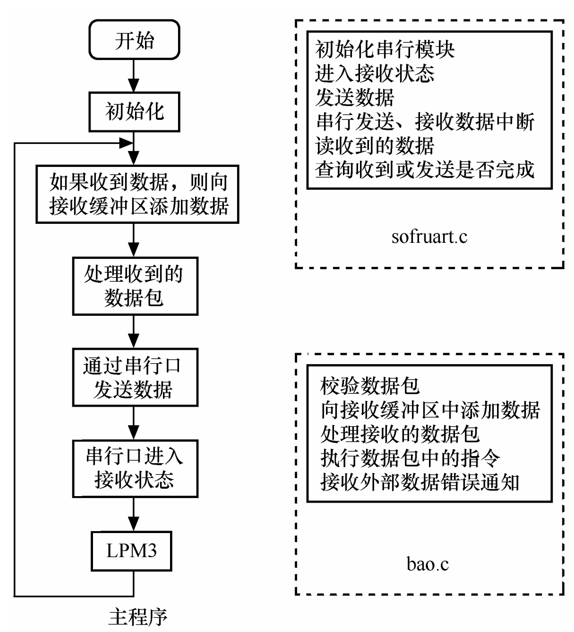


图 7-5　程序 7-4 程序结构图

程序 7-4：

```
softuart.h
#ifndef __SOFTUART
#define __SOFTUART
void InitSfUart();
void InitRxd();
void SendUart(unsigned char tx0);
unsigned char ReadBuf();
unsigned char GetRTx();
#endif

softuart.c
#include <MSP430x14x.h>
#include "softuart.h"

#define TXDSEL P1SEL
```

```
#define TXDDIR P1DIR
#define RXDSEL P2SEL
#define SF_TXD BIT1
#define SF_RXD BIT2

#define NEXT_TIME 14    //一位数据的时间，427μs，约2341bit/s
#define NEXT_TIME_H 6   //大约半位数据的时间，在此时间读RXD的数据

#define DATA_WEI_RX 8
#define DATA_WEI_TX 10

unsigned char CntWei;   //接收或者发送的位数计数器
unsigned char RxBuf;    //接收缓冲器
unsigned int TxBuf;     //发送缓冲器
unsigned char bRTx=0;   //1：发送完或收到一个字节；0：没有收到或没发送完

/**********************************************************************
初始化
**********************************************************************/
void InitSfUart()
{
    TACTL |= TASSEL_1+MC_2+TACLR;  //定时器A，时钟源：ACLK，连续计数模式
    CCTL0 |= OUTMOD_1;             //TXD空闲时输出1
    TXDSEL |= SF_TXD;              //选择P1.1使用第二功能
    TXDDIR |= SF_TXD;              //选择P1.1为输出模式
    RXDSEL |= SF_RXD;              //选择P2.2使用第二功能
}

/**********************************************************************
进入接收状态
**********************************************************************/
void InitRxd()
{
        CntWei=DATA_WEI_RX;
        CCTL0=CCIS_1+OUTMOD_1+CM_2+CAP+CCIE;
            //选择下降沿捕获，CCIB（P2.2）为信号源，捕获中断允许
}

/**********************************************************************
```

```
发送数据
tx0: 要发送的数据
************************************************************************/
void SendUart(unsigned char tx0)
{
        TxBuf=((unsigned int )tx0<<1)|0x200;
        CntWei=DATA_WEI_TX;
        CCR0=TAR+NEXT_TIME;
        CCTL0=OUTMOD_1+CCIE;   //输出 1
}

/***********************************************************************
定时器 A 中断函数
中断源: CC0
************************************************************************/
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0()
{
        CCR0 += NEXT_TIME;             //下一位到来的时间
        if((CCTL0&CCIS_1)==CCIS_1)
        {
            //处于接收状态
            if((CCTL0&CAP)==CAP)
            {
                CCTL0 &= ~CAP;         //处于捕获模式, 转比较方式
                CCR0 += NEXT_TIME_H;   //1.5 位后读取数据
            }
            else
            {
                //接收数据
                if(CntWei!=0)
                {
                    RxBuf=RxBuf>>1;
                    if((CCTL0&SCCI)==SCCI)
                        RxBuf |= 0x80;
                    CntWei--;
                }
                else
                {
```

```
                    //接收完毕
                    CCTL0 &= ~CCIE;    //关闭接收中断，退出接收状态
                    bRTx=1;            //接收到一个字节标志置位
                    LPM3_EXIT;         //退出低功耗状态
                }
            }
        }
        else
        {
            //处于发送状态
            if(CntWei!=0)
            {
                //未发送完
                CCTL0 &= ~OUTMOD2;     //输出模式1，置位
                if((TxBuf&0x1)==0)
                {
                    //发送0
                    CCTL0 |= OUTMOD2; //输出模式5，复位
                }
                TxBuf >>= 1;
                CntWei--;
            }
            else
            {
                //全部发送完
                CCTL0 &= ~CCIE;            //关闭中断
                bRTx=1;
            }
        }
}

/***********************************************************************
读收到的数据，清除有数据收到标志
返回值：收到的数据
***********************************************************************/
unsigned char ReadBuf()
{
        bRTx=0;
        return RxBuf;
```

```
}

/**********************************************************************
查询是否收到或者发送完数据
返回值：是否收到或者发送完数据的标志。1：完成；0：没有完成
**********************************************************************/
unsigned char GetRTx()
{
    return bRTx;
}

main.c
#include <MSP430x14x.h>
#include "xieyi.h"
#include "bao.h"
#include "softuart.h"

void InitSys();                         //初始化

int main( void )
{
        unsigned char q0;
        unsigned char *pq0;
    WDTCTL=WDTPW+WDTHOLD;               //关闭看门狗
    InitSys();                          //初始化
start:
        if(GetRTx()==1)
        {
            //收到数据
            q0=ReadBuf();               //从 softuart.c 模块中读数据
            AddUsData(q0);              //将数据存入数据包
        }
        pq0=DoUart(&q0);                //处理数据
        while(q0!=0)
        {
            SendUart(*pq0);             //发送数据
            while(GetRTx()==0);         //等待发送完毕
            ReadBuf();                  //清除发送完毕标志
            q0--;
```

```
            pq0++;
        }
        InitRxd();                    //串行口进入接收状态
        LPM3;
        _NOP();
        goto start;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    unsigned int iq0;

    BCSCTL1 &= ~XT2OFF;                      //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                      //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);        //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG) != 0);             //判断 XT2 是否起振

        BCSCTL2=SELM_2;                      //选择 MCLK 为 XT2
        InitSfUart();                        //初始化软件串行口
        InitRxd();                           //串行口进入接收状态
        _EINT();
}
```

7.2　基本定时器

基本定时器（Basic Timer1）存在于 MSP430F4xx 系列中，其功能比较简单。它的功能有两个：一是为液晶驱动模块提供时钟信号 f_{LCD}；二是产生周期性的中断，为系统提供周期信号。基本定时器的计数器分为两个部分，分别由寄存器 BTCNT1 和 BTCNT2 控制。BTCNT1 的时钟源只能是 ACLK, BTCNT2 的时钟源可以是 MCLK、ACLK 或者经过 256 分频的 ACLK。经过 256 分频的 ACLK 实际上来自 BTCNT1 的计数溢出，相当于 BTCNT1 与 BTCNT2 合为一个 16 位的定时器。 f_{LCD} 作为液晶模块的时钟信号，是从 BTCNT1 中引出的，可以有 4 种频率进行选择，分别是对 ACLK 的 32、64、128、256 分频。周期性的中断由 BTCNT2 产生，

可以有 8 种中断周期供选择，分别是对 BTCNT2 的时钟源的 2、4、8、16、32、64、128、256 分频。BTCNT1 和 BTCNT2 的计数值都可以用软件读出。

程序 7-5 的功能很简单：输出 f_{LCD} 信号，为液晶屏提供时钟；BTCNT2 计时时间到后触发中断，在中断函数中使 P1.1 端口输出电平翻转。

程序 7-5 的工作环境为 MCLK 选择 DCO。定时器时钟源选择 ACLK，32.768kHz。

bastimer.c 为基本定时器的驱动程序。函数 InitBasTime 初始化基本定时器。函数 GoBasTime 打开或者关闭基本定时器。函数 IntBtcn2 打开或者关闭 BTCNT2 的中断。BasTime 为基本定时器的中断函数。

函数 InitBaseTime 的功能为初始化基本定时器。选择液晶信号输出频率为 1kHz，BTCNT2 的时钟源 f_{clk2} 选择 ACLK，BTCNT2 输出的定时时间为 $f_{clk2}/4$。

函数 GoBasTime 的功能为打开或者关闭基本定时器。因为 BTCNT2 不能单独运行，只有在 BTCNT1 运行时才能打开，所以需要通过函数的参数 which 确定操作的是定时器的哪一部分。

函数 IntBtcn2 的功能为打开或者关闭 BTCNT2 的中断。BTCNT2 的中断由寄存器 IE2 中的 BITIE 位控制。通过屏蔽位的方法打开或者关闭 BTCNT2 的中断。

函数 BasTime 处理基本定时器的中断。本程序的中断处理十分简单，每次中断将 P1.1 口的电平翻转。函数中使用静态变量 out 记录 P1.1 口的输出状态，out 只能在 BasTime 函数中使用，在 BasTime 函数之外无法使用，但退出 BasTime 函数时 out 的值仍然会保留，当再次进入 BasTime 函数时，以前保存的 out 值仍然有效。

main.c 为主程序。main 函数非常简单，只调用了系统初始化函数 InitSys，然后进入低功耗模式。

函数 InitSys 为系统初始化函数。调用基本定时器初始化函数 InitBasTimer，然后调用 GoBasTimer 函数打开基本定时器中的 BTCNT1 和 BTCNT2。选择 P1.1 口工作在输出模式。打开基本定时器的中断。最后打开全局中断。

程序 7-5 的程序比较简单，程序结构图省略。

程序 7-5：

```
bastimer.h
#ifndef __BASIC_TIMER
#define __BASIC_TIMER
void InitBasTimer();
void GoBasTimer(unsigned char doit,unsigned char which);
void IntBtcn2(unsigned char doit);
#endif


bastimer.c
#include  <msp430x42x.h>
#include "bastimer.h"
/*********************************************************************
初始化基本定时器
```

```
************************************************************************/
void InitBasTimer()
{
    //确定液晶时钟信号、BTCN2的输入时钟源以及中断周期
    BTCTL=BT_fLCD_1K+BT_fCLK2_ACLK+ BT_fCLK2_DIV4;
}

/************************************************************************
打开或关闭定时器，BTCNT2只能与BTCNT1一起运行，单独打开BTCNT2没有意义。
doti：0：打开；100：关闭
which：操作定时器的哪一部分。0：BTCNT1和BTCNT2；1：BTCNT2；2：BTCNT1
************************************************************************/
void GoBasTimer(unsigned char doit,unsigned char which)
{
    if(doit==0)
    {
        if(which==0)
            BTCTL &= ~BTHOLD;   //打开BTCN1和BTCN2
        else if(which==2)
        {
            BTCTL |= BTHOLD;   //仅打开BTCN1
            BTCTL &= ~BTDIV;
        }
    }
    else if(doit==100)
    {
        if(which==0)
            BTCTL |= BTHOLD+BTDIV; //关闭BTCN1和BTCN2
        else if(which==1)
            BTCTL |= BTHOLD;       //关闭BTCN2
    }
}

/************************************************************************
打开或关闭BTCNT2输出中断
doit：0：打开   100：关闭
************************************************************************/
void IntBtcn2(unsigned char doit)
{
```

```
    if(doit==0)
        IE2 |= BTIE;   //打开
    else if(doit==100)
        IE2 &= ~BITE;  //关闭
}

/*********************************************************************
基本定时器中断函数
*********************************************************************/
#pragma vector=BASICTIMER_VECTOR
__interrupt void BasTimer()
{
    static unsigned char out=0;
    if(out==0)
    {
        out=1;
        P1OUT &= ~BIT1;
    }
    else
    {
        out=0;
        P1OUT |= BIT1;
    }
}

main.c
#include  <msp430x42x.h>
#include "bastimer.h"

void InitSys();

int main()
{
    WDTCTL=WDTPW+WDTHOLD;  //关闭看门狗
    InitSys();
    LPM3;
}

/*********************************************************************
```

```
系统初始化
*************************************************************************/
void InitSys()
{
    InitBasTimer();
    GoBasTimer(0,0);
    P1DIR |= BIT1;        //输出方波端口
    IntBtcn2(0);          //打开基本定时器中断
    _EINT();              //打开全局中断控制
}
```

第 8 章　FLASH 的读写、擦除与 I/O 端口

在某些应用中需要将程序执行过程中所生成的数据保存下来，可以选择使用 FLASH 存储器保存这些数据。MSP430 的内部程序存储器就采用 FLASH 存储器。为了避免保存数据时发生错误，从而导致保存在 FLASH 中的程序被改写，FLASH 的擦除和写都规定了严格的操作规则。8.1 节中将介绍如何读写和擦除 FLASH 存储器。

每种 CPU 都要进行 I/O 端口的操作，MSP430 的端口操作本身比较简单，8.2 节将介绍两个常用的应用：行列式键盘和非行列式键盘。

8.1 FLASH 的读写和擦除

MSP430 中用来保存程序的存储器为 FALSH 存储器，不同型号 CPU 存储器的容量大小不同，目前最大的 60K 字节，最小的 1K 字节。FALSH 分为若干段，程序存储器每段 512 字节，信息存储器每段 128 字节。程序存储器和信息存储器除了段的大小不同外，使用时没有区别，都可以存储程序和数据，具体存储的内容由用户确定。在 60KB 的芯片中，信息存储器与程序存储器合起来为 60KB，而其他的芯片存储器容量大小为：程序存储器+信息存储器。

FLASH 存储器可以按照字或者字节写入，但是不能按照字或字节来擦除，只能整段擦除，这是由 FLASH 存储器的特性所决定的。因此，在程序中最好将要擦除、改写的数据与程序放在不同的段中，以免擦除时将程序也一并擦掉。擦除的方式有段擦除和主存擦除（擦除全部程序存储器或者擦除全部 FLASH）两种。编程的方式有字/字节编程和块编程两种。因为 FLASH 在编程和擦除时处于特殊的状态，不能接受访问，所以，擦除程序不能擦除程序自己被保存的段，同样编程的程序也不能向自己被保存的段内写数据，处于被编程或者被擦除过程中间的段也不能被读出。因为主存擦除指令擦除的是所有的保存程序的存储器，所以，必然会擦除到擦除程序被保存的段，从而导致冲突。因此，主存擦除的程序只能被放在 RAM 中才能够顺利执行。同样的道理，块编程程序也只能被保存在 RAM 中执行。对 FLASH 进行擦除和编程需要适当的时钟信号，信号频率范围约为 257kHz~476kHz。时钟信号可以取自 ACLK、MCLK、SMCLK，经过分频得到。擦除和编程 FLASH 时，要求电源电压不得低于 2.7V。

因为在对 FLASH 进行擦写的过程中，一旦出现误操作，就可能会擦除或者改写保存在 FLASH 中的程序，出现不可预料的后果，所以，必须提高擦写 FLASH 的可靠性。所采取的措施是规定每次操作 FLASH 的寄存器的时候，高 8 位为安全键值。写入寄存器的时候必须保证高 8 位是正确的安全键值，否则会触发非屏蔽中断请求（NMI）。写寄存器时的安全键值为 0xA5，读出的安全键值为 0x96。

程序 8-1 的功能为，在 FLASH 中定义了两个数组并赋初值：DataB 和 DataW，还定义了两个地址：FLASHA_ADR 在信息存储器 A 中，FLASHD_ADR 在程序存储器中。程序首先擦除一段数据段和信息存储器 A，然后，将 DataB 和 DataW 中的数据分别复制到以 FLASHA_ADR 和 FLASHD_ADR 为首地址的区域内。为了验证是否写得正确，程序最后将写完的数据再读出到 RAM 数组 RDataB 和 RDataW 中，这两个数组中所有的元素都赋初值为 0。读者可以通过仿真器在适当的位置设置断点来观察读写的结果。

程序 8-1 的工作环境为：MCLK 选择 8MHz。

flash.c 为 FLASH 擦写驱动程序。FlashErase 为段擦除函数。函数 FlashBusy 测试 FLASH 是否空闲。FlashWW 为字编程函数。FlashWB 为字节编程函数。其中没有定义全部擦除 FLASH 的函数，原因是全部擦除 FLASH 的函数必须在 RAM 中执行，除非全部更新程序，否则通常没有必要全部擦除 FLASH。

函数 FlashErase 擦除 FLASH 中的一段。参数 adr 为要擦除的段范围内的任意地址。写

FLASH 控制寄存器时，高 8 位都必须是安全键值，否则会触发非屏蔽中断。这里选择 1 段擦除模式。FLASH 控制器的时钟源选择 MCKL，8MHz，经过 25 分频，最终的工作频率为 320kHz，符合要求。擦除需要一定的时间，循环调用 FlashBusy 函数，等待 FLASH 控制器空闲，只能在 FLASH 控制器空闲时才可以开始擦除操作。只要向 adr 地址写入任意数值，FLASH 控制器即开始擦除操作。再次循环调用 FlashBusy 函数，等待 FLASH 控制器完成擦除操作。完成擦除后，将 FCTL3 寄存器中的 LOCK 位置位，锁定 FLASH 控制器。

函数 FlashBusy 通过读取寄存器 FCTL3 中的 BUSY 位来确定 FLASH 控制器是否处于空闲状态。

函数 FlashWW 每次向 FLASH 中的某一地址写入一个字。参数 Adr 为要写入的地址，由 MSP430 的芯片硬件所决定，地址 Adr 应当是偶地址。参数 DataW 为要写入的字。与段擦除类似，首先选择单字/字节编程模式，然后选择 FLASH 控制器的时钟频率，与擦除 FLASH 时使用的时钟频率一样，为 320kHz。接着进行解锁操作，等待 FLASH 控制器空闲。将要写的值 DataW 写入地址 Adr，等待 FLASH 完成操作，最后锁定 FLASH 控制器。

函数 FlashWB 的功能是每次向 FLASH 中的某一地址写入一个字节，与函数 FlashWW 的写入过程一样，不同之处在于写入的数据类型为字节。

main.c 为主程序。功能比较简单，读者很容易理解，这里不进行解释。

程序 8-1 的结构比较简单，程序结构图省略。在本程序中有几个问题需要重点说明：

（1）定义一个保存在 FLASH 中的变量或者数组的方法如下：

```
const unsigned char DataB[N_DATA]={0,1,2,3,4,5,6,7,8,9};
const unsigned int DataW[N_DATA]={0,100,200,300,400,500,600,700,800,900};
```

const 标识符使数组 DataB 和 DataW 编译时分配在 FLASH 中。通过指针读取 FLASH 中某个地址的值是最方便的方法，要注意的是指向 FLASH 区域的指针必须定义为 const 类型。

（2）在 main.c 中最后使用指针读取 FLASH 中的数据到 RAM 中，已知的地址值并不是指针类型，这种情况经常会遇到。所以，必须将地址值转换为指针类型，解决方法是进行强制类型转换。

```
piq0=(unsigned int *)FLASHD_ADR;       //数值强制转换为指针类型
pq0=(unsigned char *)FLASHA_ADR;       //数值强制转换为指针类型
```

（3）一般如果没有特殊说明，编译器会在链接时自动安排函数所在的地址。为了保证操作 FLASH 的函数与要擦写的内容不在同一段内，有必要在程序中指明擦写 FLASH 的函数所在的段。flash.c 中所有函数的声明后边都有 @ "MYSET"，表示这些函数链接时都要定位在 MYSET 段中。MYSET 段的地址范围是由用户自已确定的。用户定义段的方式如下：

有关编译器链接方式的信息都被保存在一个文件中，打开工程 options 对话框，选择左边文本框中的 Linker 选项，在右边选择 Config 选项卡，在 Linker command file 栏下选择 override default。如图 8-1 所示，编辑框中列出的即为默认的链接文件，默认的链接文件被保存在 IAR 安装目录下 430\config 内。在没有选中 override default 选项时，编译器根据所选用的 CPU 型号自动选择所使用的文件，这些文件的内容用户最好不要改动，以免影响以后其他项目的链接。在本例中，原来的文件为 lnk430F149.xcl，可以将此文件复制一个，与源程序放在同一目录下，改名为 FLASHlnk430F149.xcl。在 override default 选项下方的文本框内选择链接文

件为 FLASHlnk430F149.xcl，这个文件中定义了初始段、代码段、数据段、堆栈段以及信息存储器的地址范围。在文件中可以找到代码段的定义如下：

```
// -------------------------------------------------------------------
// ROM memory (FLASH)
// -------------------------------------------------------------------

// Code

-Z(CODE)CSTART=1100-FFDF
-Z(CODE)CODE=1100-FFDF
```

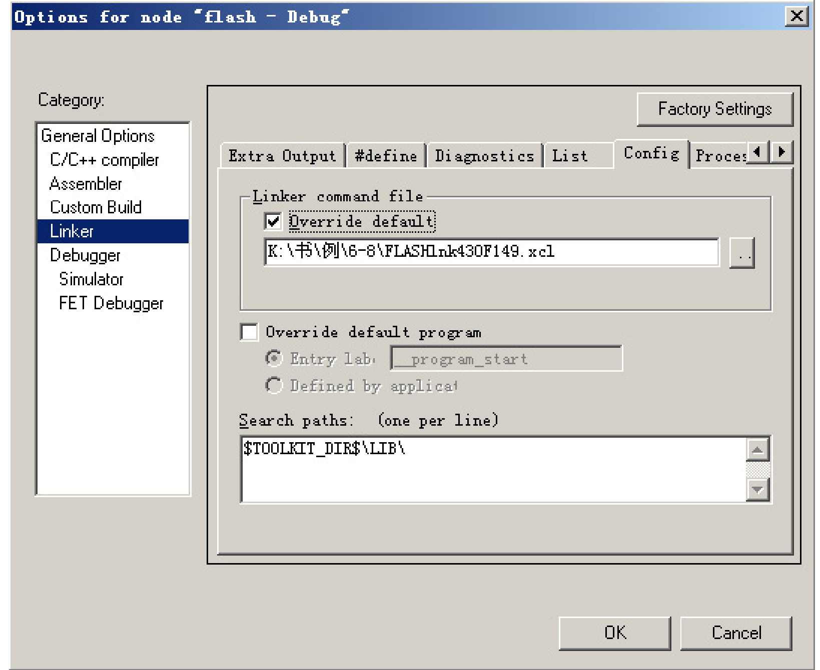


图 8-1　改变连接文件

从代码段的定义可以看到，文件中已经定义了两个段：CSTART 段和 CODE 段，它们的地址范围都是 0x1100~0Xffdf。在最后加上-Z(CONST)MYSET=FC00~FFDF 语句。定义 MYSET 段的地址范围在 0xFC00~0xFFDF，地址范围读者可以根据需要自行修改。由于编译器是从 FLASH 的低端地址开始分配程序的，所以一般定义在高端地址，但要注意不要与中断向量的 32 个字节（地址范围在 0xFFE0~0xFFFF）冲突。其他段地址的定义读者也可以修改，但修改时一定要小心。

程序 8-1：

```
flash.h
#ifndef __FLASH
#define __FLASH
void FlashErase(unsigned int adr);
```

```
unsigned char FlashBusy();
void FlashWW(unsigned int Adr,unsigned int DataW);
void FlashWB(unsigned int Adr,unsigned char DataB);
#endif

flash.c
#include  <msp430x14x.h>
#include "flash.h"

/**********************************************************************
段擦除
adr：要擦除的段内的任一地址
**********************************************************************/
void FlashErase(unsigned int adr) @ "MYSET"
{
    unsigned char *p0;

    FCTL1=FWKEY+ERASE;
    FCTL2=FWKEY+FSSEL_1+FN3+FN4;       //分频为 320kHz
    FCTL3=FWKEY;                       //解锁
    while(FlashBusy()==1)              //等待 FLASH 存储器完成操作
    ;
    p0=(unsigned char *)adr;
    *p0=0;                             //向段内地址任意写，启动擦除操作
    while(FlashBusy()==1)              //等待 FLASH 存储器完成操作
    ;
    FCTL3=FWKEY+LOCK;
}

/**********************************************************************
测试 FLASH 是否忙
返回值：1：忙；0：不忙
**********************************************************************/
unsigned char FlashBusy() @ "MYSET"
{
    if((FCTL3&BUSY)==BUSY)
        return 1;
    else
        return 0;
```

```
}

/**********************************************************************
字编程
Adr：要编程的地址。注意：不是指针类型，应当是偶地址
DataW：要编程的字
**********************************************************************/
void FlashWW(unsigned int Adr,unsigned int DataW) @ "MYSET"
{
    FCTL1=FWKEY+WRT;
    FCTL2=FWKEY+FSSEL_1+FN3+FN4;        //分频为 320kHz
    FCTL3=FWKEY;
    while(FlashBusy()==1)               //等待 FLASH 存储器完成操作
    ;
   *((unsigned int *)Adr)=DataW;
    while(FlashBusy()==1)               //等待 FLASH 存储器完成操作
    ;
    FCTL1=FWKEY;
    FCTL3=FWKEY+LOCK;
}

/**********************************************************************
字节编程
Adr：指向要编程的地址。注意：不是指针类型
DataB：要编程的字节
**********************************************************************/
void FlashWB(unsigned int Adr,unsigned char DataB) @ "MYSET"
{
    FCTL1=FWKEY+WRT;
    FCTL2=FWKEY+FSSEL_1+FN3+FN4;        //分频为 320kHz
    FCTL3=FWKEY;
    while(FlashBusy()==1)               //等待 FLASH 存储器完成操作
    ;
    *((unsigned char *)Adr)=DataB;
    while(FlashBusy()==1)               //等待 FLASH 存储器完成操作
    ;
    FCTL1=FWKEY;
    FCTL3=FWKEY+LOCK;
}
```

```
main.c
#include <MSP430x14x.h>
#include "flash.h"

void InitSys();

#define FLASHA_ADR 0x1080              //FLASH 的 A 段起始地址
#define FLASHD_ADR 0x2500              //FLASH 中某一段中的地址

//程序存储器中的源数组
#define N_DATA 10
const unsigned char DataB[N_DATA]={0,1,2,3,4,5,6,7,8,9};
const unsigned int DataW[N_DATA]={0,100,200,300,400,500,600,700,800,900};

int main( void )
{
    unsigned char q0;
    unsigned int iq1,iq2;
    const unsigned int *piq0=DataW;
    const unsigned char *pq0=DataB;
    unsigned char RDataB[N_DATA]={0,0,0,0,0,0,0,0,0,0};
    unsigned int RDataW[N_DATA]={0,0,0,0,0,0,0,0,0,0};

    WDTCTL=WDTPW+WDTHOLD;              //关闭看门狗
    InitSys();                         //初始化

    FlashErase(FLASHD_ADR);            //段擦除
    FlashErase(FLASHA_ADR);            //段擦除
    iq1=FLASHD_ADR;
    iq2=FLASHA_ADR;
    for(q0=0;q0<N_DATA;q0++)
    {
        FlashWW(iq1,*piq0);                //写字数据
        FlashWB(iq2,*pq0);                 //写字节数据
        iq1 += 2;                          //写入字地址+2
        iq2++;                             //写入字节地址+1
        piq0++;                            //源字指针+1
        pq0++;                             //源字节指针+1
```

```
    }

    piq0=(unsigned int *)FLASHD_ADR;        //数值强制转换为指针
    pq0=(unsigned char *)FLASHA_ADR;
    for(q0=0;q0<N_DATA;q0++)
    {
        RDataW[q0]=*piq0;
        RDataB[q0]=*pq0;
        piq0++;
        pq0++;
    }
    _NOP();                                 //空操作，便于测试
    LPM3;                                   //进入低功耗模式 3
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;                         //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                         //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);           //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!=0);          //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                //选择 MCLK、SMCLK 为 XT2
    _EINT();
}
```

8.2 I/O 端口

MSP430 所有的端口都有控制输入输出方向和进行输入、输出的能力 。P1、P2 端口能

够响应外部中断，大部分端口拥有第二功能。所有的管脚功能都可以单独配置。每一个端口都有 PxDIR（信号方向）、PxIN（输入）、PxOUT（输出）3 个寄存器。拥有第二功能的端口会有 PxSEL 寄存器，用来选择端口的第二功能。

P1、P2 可以配置为输入信号上升沿或者下降沿触发中断，但固定的电平不会引起中断，中断所使用的寄存器为：PxIE（中断使能）、PxIES（中断触发沿）、PxIFG（中断标志）。

利用 P1、P2 的中断功能可以实现键盘功能。非行列式键盘的缺点是占用 I/O 端口比较多，如果按键较多或 I/O 口紧张，则应该使用行列式键盘。相比较而言，行列式键盘的软件复杂，执行起来消耗 CPU 的资源也较多。

由于键盘触点按下和放开时的抖动不可避免，所以无论哪一种方式，软件中都必须有防止按键抖动的功能。

8.2.1　非行列式键盘

非行列式键盘的连接方式如图 8-2 所示。每个端口上有上拉电阻，没有按键按下时 P1.5、P1.6、P1.7 端口上为高电平，按下某键时，相应端口为低电平。程序测到某个端口的电平为低时，就可以确定与此端口连接的按键被按下。因为非行列式键盘硬件电路和执行软件简单，消耗 CPU 资源少，所以在按键少的情况下使用比较有利。

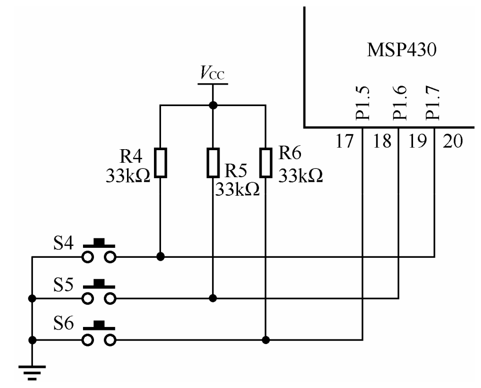


图 8-2　非行列式键盘

程序 8-2 的功能为监测按键是否按下，将按下的按键的键值和按键次数通过串行口发送出去。

程序 8-2 的工作环境为 MCLK 为 8MHz，ACLK 为 32.768kHz。使用定时器 A 定时，作为按键防抖动的延时。定时器 A 使用捕获/比较单元 0，计数器的时钟源为 ACLK。串行口使用 USART0，时钟源选择 ACLK，发送数据的波特率为 9600bit/s，数据格式为 1 位起始位+8 位数据位+1 位停止位。

程序 8-2 工作过程为：当有键按下时，产生的下降沿触发中断。在按键端口中断程序中判断中断源，如果是按键被按下，则关闭所有引脚中断，并启动定时器 A，定时时间为 20ms。20ms 后定时器 A 触发中断，在定时器 A 中断函数中调用 key.c 中的函数 IsKey 判断刚才被按下的按键是否仍然被按下。如果仍然被按下，就认为此次按键被按下是有效的，否则，认为无效。如果有效，以后每次定时器产生中断都查询一次按键，如果此按键未被释放，则累计计时时间达到 1 秒钟时，按键次数加 1。如果按键被释放，则关闭定时器 A，打开按键端口中断，监测是否有键被按下。在按键被按下期间，定时器 A 每次触发的中断在返回时都会使主程序退出低功耗模式。主程序进入循环，调用 key.c 中的函数 GeiKeyZhi 获得键值和按键次数。如果获得的键值不为 KEY_NONE，则说明有键被按下，接着调用 df_uart.c 中的 SendUart 函数发送按键的键值和按键次数。发送完毕后，进入低功耗模式，等待新的按键动作。

文件 key.c 和 df_timera.c 共同组成非行列式键盘驱动模块。

函数 InitKey 初始化按键端口，主要是设置按键端口的工作和中断模式。将按键的次数清 0，进入监测是否有键被按下的状态。

函数 GoKey 打开或者关闭键盘中断。

函数 GeiKeyZhi 为查询按键的键值和按键次数，如果没有键被按下，那么返回 KEY_NONE 值。每次查询都清除以往的键值和按键次数。如果主程序查询得及时，那么每次得到的按键次数都为 1。修改 KEY_TIME 值可以改变记录按键次数的频率。由于定时器 A 的中断时间约为 20ms，定义 KEY_TIME 值为 50，则大约连续按键 1s 时，保存按键次数的变量 KeyCnt 加 1。

函数 IsKey 判断收到的按键信号是否有效。读按键端口的值，如果所有的端口都是高电平，则说明此时没有键被按下，记录按键状态的变量 KeyDown 被赋值为 KEY_NONE。如果没有键被按下，则关闭定时器，打开键盘中断，进入等待按键被按下的状态。如果某端口为低电平，则说明有按键被按下。判断键盘是否被有效按下的另一个条件是经过延时之后，此按键是否仍被按下。每次调用函数 IsKey，保存延时时间的变量 KeyTime 都会减 1，当 KeyTime 减小为 0 时，表明延时时间到。如果变量 KeyDown 记录的键值与当前按下的按键的键值相符，则表明按键是有效的，变量 KeyCnt 加 1。程序中键盘使用的 3 个管脚被定义为 KEY_P15、KEY_P16、KEY_P17。这样做的好处是，如果需要使用其他的管脚，那么只需要重新定义一下就可以使用。例如，不使用 BIT5 管脚，而改为使用 BIT0 管脚，那么只需要改“#define KEY_P15　BIT5”为“#define KEY_P15 BIT0”就可以了，不需要在程序中逐个的修改 BIT5 为 BIT0。

函数 Port1 是端口中断函数。因为 P1 的 8 个端口共用一个中断，所以，这是一个多中断源函数。由寄存器 KEYIFG 判断由哪一个端口触发的中断。本程序中只有 P1.5、P1.6、P1.7 连接了按键，如果是这 3 个端口触发的中断，则记录按下的键值，清除寄存器 KEYIFG 中的中断标志。如果不清除，则退出中断后会立刻再次触发中断。对于非 P1.5、P1.6、P1.7 端口触发的中断，视为干扰信号，不进行处理，只是清除中断标志。如果确定 P1.5、P1.6、P1.7 端口有键按下，就关闭端口中断，打开定时器 A，准备在延时之后再检测此按键。

文件 df_timera.c 为定时器 A 的驱动模块，与定时器 B 的初始化过程基本一致，可以参考 7.1.1 节的内容，在此不再赘述。在 df_timera.h 中修改 TIME_20MS 的值可以改变键盘防抖动的延时时间。

文件 df_uart.c 为串行口驱动模块。由于只需要发送数据，不需要接收，所以比较简单。此文件是由程序 6-1 中的文件 df_uart.c 简化而来的，关于它的解释可以参考第 6 章。发送部分与键盘本身没有关系，只是为了能对按键作出明显的反应，读者可以根据需要换成其他的按键处理方式。

main.c 为主程序。在系统初始化函数 InitSys 中调用了 3 个模块的初始化函数：InitKey、TimerAInit、UartInit，分别初始化了按键端口、定时器 A、串行口。每次有按键被按下，CPU 就会退出低功耗模式，执行函数 main 中的主循环，读按键的键值和按键次数。如果读出的键值不是 KEY_NONE，则表明按键按下是有效的，于是调用函数 SendUart 发送键值和按键次数。

图 8-3 为非行列式键盘程序的结构图。

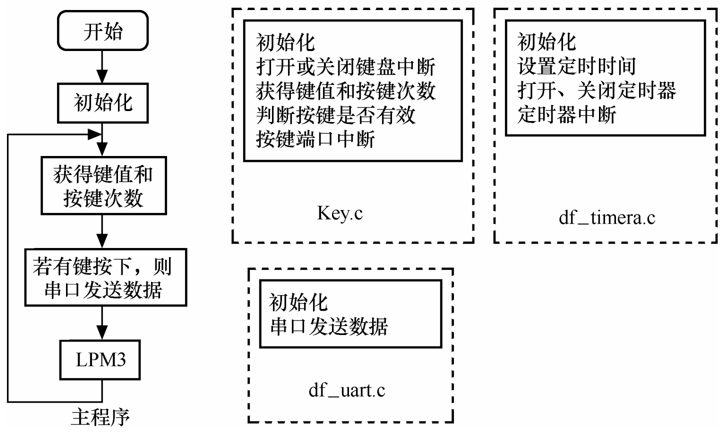


图 8-3　非行列式键盘程序结构

程序 8-2：

```
key.h
#ifndef __KEY
#define __KEY
void InitKey();
void GoKey(unsigned char sw);
unsigned char GetKeyZhi(unsigned char *key_cnt);
void IsKey();

//键值定义
#define KEY_NONE        0           //无键按下
#define KEY_P15         0x10        //P1.5 键值
#define KEY_P16         0x20        //P1.6 键值
#define KEY_P17         0x30        //P1.7 键值
#endif

key.c
#include <MSP430x14x.h>
#include "key.h"
#include "df_timera.h"

//选用端口定义
#define KEYDIR P1DIR
#define KEYIES P1IES
#define KEYIE P1IE
#define KEYIN P1IN
```

```
#define KEYIFG P1IFG

//定义键盘管脚
#define KEY0 BIT5
#define KEY1 BIT6
#define KEY2 BIT7
#define KEY_MOD (KEY0+KEY1+KEY2)

unsigned char KeyZhi=KEY_NONE;          //经过确认的键值
unsigned char KeyCnt;                   //某次连续按键的次数
unsigned char KeyDown=KEY_NONE;         //被按下的键

#define KEY_TIME 50                     //连续按键经过此时间，按键次数加 1
unsigned char KeyTime;                  //记录连续按键的时间
/**********************************************************************
初始化
**********************************************************************/
void InitKey()
{
    KEYDIR &= ~(KEY0+KEY1+KEY2);        //设置端口为输入
    KEYIES |= KEY0+KEY1+KEY2;           //设置下降沿中断
    KEYIE |= KEY0+KEY1+KEY2;            //打开端口中断
    KeyCnt=0;                           //按键次数清零
}

/**********************************************************************
打开或者关闭键盘中断
sw: 0: 关闭; 100: 打开
**********************************************************************/
void GoKey(unsigned char sw)
{
    if(sw==0)
        KEYIE &= ~(KEY0+KEY1+KEY2);         //关闭端口中断
    else
        KEYIE |= KEY0+KEY1+KEY2;            //打开端口中断
}

/**********************************************************************
```

```
获得键值和连续按下的键的有效次数，并清除记录的键值和有效次数
key_cnt：返回某一键被连续按下的有效次数
返回值：按下的有效的键的键值
***********************************************************************/
unsigned char GetKeyZhi(unsigned char *key_cnt)
{
    unsigned char q0;
    q0=KeyZhi;
    KeyZhi=KEY_NONE;                  //清除键值
    *key_cnt=KeyCnt;
    KeyCnt=0;
    return q0;
}

/***********************************************************************
判断按键的有效性
***********************************************************************/
void IsKey()
{
    if((KEYIN&KEY_MOD)==KEY_MOD)
    {
        //没有键按下
        KeyDown=KEY_NONE;
        GotimeDfA(0);                 //关闭定时器
        GoKey(100);                   //打开键盘中断
    }
    else
    {
        if(KeyTime==0)
        {
            //连续按键的时间到
            KeyTime=KEY_TIME;
            if((KEYIN|KEY0==KEY0)&&KeyDown==KEY_P15)
            {
                //P1.5 按键按下
                KeyCnt++;
                KeyZhi=KEY_P15;
            }
```

```
            else if((KEYIN|KEY1==KEY1)&&KeyDown==KEY_P16)
            {
                //P1.6 按键按下
                KeyCnt++;
                KeyZhi=KEY_P16;
            }
            else if((KEYIN|KEY2==KEY2)&&KeyDown==KEY_P17)
            {
                //P1.7 按键按下
                KeyCnt++;
                KeyZhi=KEY_P17;
            }
            else
            {
                KeyDown=KEY_NONE;
                GotimeDfA(0);              //关闭定时器
                GoKey(100);                //打开键盘中断
            }
        }
        else
        {
            KeyTime--;
        }
    }
}

/*********************************************************************
端口 1 中断函数
多中断中断源：P1IFG.0~P1IFG7，只用到 P1.5 P1.6 P1.7 三个中断源。
进入中断后应首先判断中断源，退出中断前应清除中断标志，否则将再次触发中断。
*********************************************************************/
#pragma vector=PORT1_VECTOR
__interrupt void Port1()
{
    unsigned char q0=0;
    if((KEYIFG&KEY0)==KEY0)
    {
        //处理 P1IN.5 中断
```

```
        KEYIFG &= ~KEY0;                              //清除中断标志
        KeyDown=KEY_P15;                              //记录按下的键值
        q0=1;
    }
    else if((KEYIFG&KEY1)==KEY1)
    {
        //处理 P1IN.6 中断
        KEYIFG &= ~KEY1;                              //清除中断标志
        KeyDown=KEY_P16;                              //记录按下的键值
        q0=1;
    }
    else if((KEYIFG&KEY2)==KEY2)
    {
        //处理 P1IN.7 中断
        KEYIFG &= ~KEY2;                              //清除中断标志
        KeyDown=KEY_P17;                              //记录按下的键值
        q0=1;
    }
    else
    {
        //其他干扰引起的中断。不对其进行处理，只清除中断标志
        KEYIFG=0;
    }

    if(q0==1)
    {
        GoKey(0);                                     //关闭键盘中断
        KeyTime=0;
        KeyCnt=0;
        GotimeDfA(100);                               //打开定时器 A
    }
}

df_timera.h
#ifndef __DF_TIMER_A
#define __DF_TIMER_A

void TimerAInit();
```

```
void SetTime(unsigned int ti);
void GotimeDfA(unsigned char doit);

#define TIME_20MS 655              //定时时间约 20ms
#endif

df_timera.c
#include <MSP430x14x.h>
#include "df_timera.h"
#include "key.h"

/**********************************************************************
初始化
**********************************************************************/
void TimerAInit()
{
    TACTL=TASSEL_1+TACLR+MC_0; //选择时钟源为 ACLK
    SetTime(TIME_20MS);        //设置定时时间
    TACCTL0=CCIE;              //允许定时器中断
}

/**********************************************************************
设置定时时间
ti：要定时的时间，与时钟源的频率有关
**********************************************************************/
void SetTime(unsigned int ti)
{
    TACCR0=ti;                 //定时时间
}

/**********************************************************************
打开、关闭定时器
doit：0：停止  100：运行
**********************************************************************/
void GotimeDfA(unsigned char doit)
{
    if(doit==100)
    {
```

```
        TACTL |= MC_1+TACLR;          //打开定时器
    }
    else if(doit==0)
    {
        TACTL &= ~MC0;                //关闭定时器
    }
}

#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0(void)       //定时器中断
{
    IsKey();
    LPM3_EXIT;
}

df_uart.h
#ifndef  __DF_UART
#define  __DF_UART
void UartInit();
void SendUart(unsigned char *pBuffer,unsigned char n_byte);
#endif

df_uart.c
#include <MSP430x14x.h>
#include "df_uart.h"

#define USART_DIR P3DIR
#define USART_SEL P3SEL
#define USART_IN  P3IN
#define USART_OUT P3OUT
#define UTXD0 BIT4
#define URXD0 BIT5

/**********************************************************************
初始化
**********************************************************************/
void UartInit()
{
```

```
    USART_SEL |= UTXD0+URXD0;          //设置管脚为第二功能
    UCTL0=CHAR+PENA+SWRST;             //8 位数据，1 位停止位，奇校验
    UTCTL0=SSEL0;                      //选择 UCLK = ACLK
    UBR00=0x3;                         //设置波特率 9600bit/s
    UBR10=0;
    UMCTL0=0x4A;
    UCTL0 &= ~SWRST;
    ME1 |= UTXE0;                      //打开模块 USART0
}

/*********************************************************************
发送函数。采用查询方式。
PBuffer：指向发送数据缓冲区的指针
n_byte：发送的字节数
*********************************************************************/
void SendUart(unsigned char *pBuffer,unsigned char n_byte)
{
    unsigned char q0;
    for(q0=0;q0<n_byte;q0++)
    {
        while((IFG1&UTXIFG0)==0);  //查询是否发送完毕
        TXBUF0=*pBuffer;
        pBuffer++;
    }
}

main.c
#include <MSP430x14x.h>
#include "key.h"
#include "df_timera.h"
#include "df_uart.h"

void InitSys();

int main( void )
{
    unsigned char key[2];
```

```
    WDTCTL=WDTPW+WDTHOLD;                    //关闭看门狗
    InitSys();                               //初始化
start:
    key[0]=GetKeyZhi(&key[1]);               //读键值和按键次数
    if(key[0]!=KEY_NONE)
    {
        //按键处理，本例中为通过 USART0 发送键值和按键次数
        SendUart(key,2);                     //若有键按下，则发送键值和按键次数
    }
    LPM3;
    goto start;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;                      //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                      //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);        //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!=0);               //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                     //选择 MCLK、SMCLK 为 XT2

    InitKey();                               //初始化按键端口
    TimerAInit();                            //初始化定时器 A
    UartInit();                              //初始化串行口
    _EINT();    //打开全局中断控制，若不需要打开，则可以屏蔽本句
}
```

8.2.2　行列式键盘

行列式键盘的连接方式如图 8-4 所示。P2.0~P2.3 为行端口，P2.4~P2.7 为列端口。初始

化时行端口设置为输入且能够触发中断，初始化时列端口设置为输出低电平。当没有键按下时，由于上拉电阻 R1~R4 的作用，行端口的输入全为高电平。当有键被按下时，按键所在行的电平就会成为低电平，触发中断。与非行列式键盘一样，在端口中断程序中关闭端口中断，打开定时器 A 进行延时，延时时间大约 20ms。延时时间到后读取按键的行值和列值，组合成为键值，行值为低 4 位，列值为高 4 位。在定时中断中检测到按键被释放后，关闭定时器 A，重新打开端口中断，以便响应下一次按键。

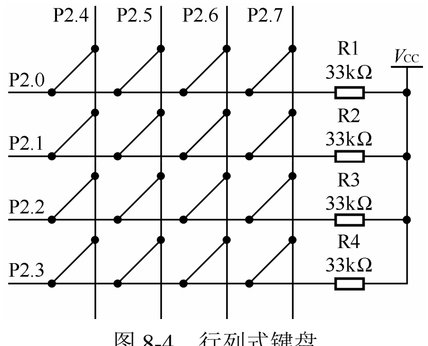


图 8-4　行列式键盘

读取行值和列值的方法为：当按键按下时将行端口读入的值取反，用位屏蔽的方法屏蔽掉列端口的 4 位，这样所得到的值为行值。如图 8-4 所示，右下角 P2.7 与 P2.3 交叉处的按键按下时，行端口读入的值为 0x07，取反并屏蔽掉高 4 位，所得行值为 0x08。

因为每一个行端口都对应着 4 个列端口，所以列值的读取要复杂一些。首先使列端口全部输出高电平，这样行端口读入的值必定为 0xFF，然后逐个使列端口输出低电平，当被按下键的列端口输出低电平时，行端口读入的电平就会为低电平，而那些没有按下键的列端口即使输出低电平也不会影响到行端口。由此可以判断出按键的列值。如当 P2.7 与 P2.3 交叉处的按键被按下时，当列端口输出为 0111（P2.7~P2.4）时，从行端口读入的值为 0111（P2.3~P2.0），将读入的列值左移 4 位，为 0x70，然后取反，得到列值为 0x80。最终，键值=行值+列值=0x88。图 8-4 的行列式键盘的键值如表 8-1 所示。

表 8-1　　图 8-4 所示行列式键盘的键值表

列端口 / 行端口	P2.4	P2.5	P2.6	P2.7
P2.0	0x11	0x21	0x41	0x81
P2.1	0x12	0x42	0x22	0x82
P2.2	0x14	0x24	0x44	0x84
P2.3	0x18	0x28	0x48	0x88

程序 8-3 的功能和工作环境都与程序 8-2 相同。键盘驱动程序的差别也不太大，主要是因为行列式键盘与非行列式键盘的结构不同，所以读键值的程序需要修改。非行列式读键值的方法比较简单，可直接读端口，并屏蔽掉不需要的位就行了。行列式键盘读键值的方法比较复杂，因此增加了一个函数 ReadKey 用来读取键值。只需要将程序 8-2 中读键值的语句更换为调用 ReadKey 函数就可以了，这是得益于结构化编程。在 hl_key.h 中有关端口的定义也需要重新定义。

函数 ReadKey 的作用为读取键值。变量 lie 是按键列值的掩码，它的值可以通过(KEY_MOD_L&(~KEY_L0))+KEY_MOD_H 计算得到。这里要说明的是，变量 lie 的值是在编译时就确定了的，算式中的宏定义的值都是已知的，没有不确定的值，所以，编译器可以在编译的时候计算出变量 lie 要赋的初值。这个算式实际上与 lie=0x1F 没有

区别。

程序 8-3 中只给出了 hl_key.c，其他的部分参见程序 8-2。程序 8-3 的结构图如图 8-5 所示。

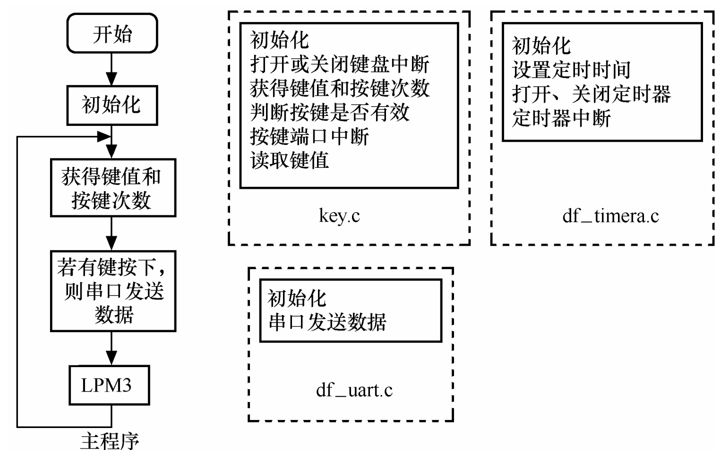


图 8-5　行列式键盘程序结构图

程序 8-3：

```
hl_key.h
#ifndef __HL_KEY
#define __HL_KEY

void InitKey();
void GoKey(unsigned char sw);
unsigned char GeiKeyZhi(unsigned char *key_cnt);
void IsKey();
unsigned char ReadKey();

//键值定义
#define KEY_NONE    0           //无键按下

#endif

hl_key.c
#include <MSP430x14x.h>
#include "hl_key.h"
#include "df_timera.h"

//选用端口定义
#define KEYDIR P2DIR
```

```
#define KEYIES P2IES
#define KEYIE P2IE
#define KEYIN P2IN
#define KEYOUT P2OUT
#define KEYIFG P2IFG

//定义键盘管脚
#define KEY_H0 BIT0
#define KEY_H1 BIT1
#define KEY_H2 BIT2
#define KEY_H3 BIT3
#define KEY_L0 BIT4
#define KEY_L1 BIT5
#define KEY_L2 BIT6
#define KEY_L3 BIT7
#define KEY_MOD_H  (KEY_H0+KEY_H1+KEY_H2+KEY_H3)   //行端口的模
#define KEY_MOD_L  (KEY_L0+KEY_L1+KEY_L2+KEY_L3)   //列段口的模
#define KEY_LIE  4                                 //列数

unsigned char KeyZhi=KEY_NONE;          //经过确认的键值
unsigned char KeyCnt;                   //某次连续按键的次数
unsigned char KeyDown=KEY_NONE;         //被按下的键

#define KEY_TIME 50         //连续按键经过此时间，按键次数加 1
unsigned char KeyTime;      //记录连续按键的时间
/***********************************************************************
初始化
***********************************************************************/
void InitKey()
{
    KEYDIR &= ~KEY_MOD_H;       //设置行端口为输入
    KEYDIR |= KEY_MOD_L;        //设置列端口为输出
    KEYOUT &= ~KEY_MOD_L;       //设置列端口输出低电平
    KEYIES |= KEY_MOD_H;        //设置行端口下降沿中断
    KEYIE |= KEY_MOD_H;         //打开行端口中断
    KeyCnt=0;                   //按键次数清零
}

/***********************************************************************
```

```
打开或者关闭键盘中断
sw:  0:关闭；100:打开
*********************************************************************/
void GoKey(unsigned char sw)
{
    if(sw==0)
        KEYIE &= ~KEY_MOD_H;        //关闭端口中断
    else
        KEYIE |= KEY_MOD_H;         //打开端口中断
}

/*********************************************************************
获得键值和连续按下的键的有效次数，并清除记录的键值和有效次数
key_cnt:返回某一键被连续按下的有效次数
返回值:按下的有效的键的键值
*********************************************************************/
unsigned char GeiKeyZhi(unsigned char *key_cnt)
{
    unsigned char q0;
    q0=KeyZhi;
    KeyZhi=KEY_NONE;                //清除键值
    *key_cnt=KeyCnt;
    KeyCnt=0;
    return q0;
}

/*********************************************************************
判断按键的有效性
*********************************************************************/
void IsKey()
{
    unsigned char key;

    key=ReadKey();
    if(key==KEY_NONE)
    {
        //没有键按下
        KeyDown=KEY_NONE;
        GotimeDfA(0);                   //关闭定时器
```

```
            GoKey(100);                     //打开键盘中断
        }
        else
        {
            if(KeyTime==0)                  //延时时间到
            {
                KeyTime=KEY_TIME;
                if(key==KeyDown)
                {
                    KeyCnt++;
                    KeyZhi=key;
                }
                else
                {
                    KeyDown=KEY_NONE;
                    GotimeDfA(0);           //关闭定时器
                    GoKey(100);             //打开键盘中断
                }
            }
            else
                KeyTime--;
        }
    }

    /**********************************************************************
    端口 1 中断函数
    多中断中断源：P1IFG.0~P1IFG.7，只响应行端口的中断
    进入中断后应首先判断中断源，退出中断前应清除中断标志，否则将再次触发中断
    **********************************************************************/
    #pragma vector=PORT2_VECTOR
    __interrupt void Port2()
    {
        if((KEYIFG&KEY_MOD_H)!=0)  //判断是否是按键触发的中断
        {
            KeyDown=ReadKey();
            if(KeyDown!=KEY_NONE)
            {
                GoKey(0);           //关闭键盘中断
                KeyTime=0;
```

```
            KeyCnt=0;
            GotimeDfA(100);       //打开定时器A
        }
    }
    KEYIFG=0;                     //清除中断标志
}

/*********************************************************************
行列式键盘读取键值，判断哪一个键被按下
返回值：读取的键值
*********************************************************************/
unsigned char ReadKey()
{
    unsigned char key=0,hang,q0=0,q1;
    //判断按键的列的掩码
    unsigned char lie=(KEY_MOD_L&(~KEY_L0))+KEY_MOD_H;

    //确定按键的行
    hang=(~KEYIN)&KEY_MOD_H;      //读入的数据取反，然后屏蔽不需要的位
    if(hang!= 0)
    {
        //确定按键的列
        for(q0=0;q0<KEY_LIE-1;q0++)
        {
            KEYOUT =0xFF;
            KEYOUT &= lie;        //某一列端口输出低电平，其他输出高电平
            q1=KEYIN&KEY_MOD_H;
            if((q1&hang)==0)
            {
                break;                    //确定了列值
            }
            lie <<=1;
        }
        if(q0!=KEY_LIE)
        {
            key=hang+((~lie)&KEY_MOD_L);  //键值为行值+列值
        }
        else
            key=KEY_NONE;
```

```
        }
        else
            key=KEY_NONE;
        KEYOUT &= ~KEY_MOD_L;                  //设置列端口输出低电平
        return key;
    }
```

第 9 章　DMA 数据传输与 IIC 总线

本章所讲述的是两种数据传输方式。MSP430 的 DMA 传输功能目前只存在于 15x、16x 中，DMA 传输方式的主要优点是提高数据传输的速度。本章 9.1 节介绍 MSP430 的 DMA 传输模块的使用方法。

IIC 总线是外部串行总线，用于和 CPU 外围的设备进行通信，目前使用的范围很广。15x、16x 中存在 IIC 总线控制模块，但本书的例子中没有使用，因为 430 系列大部分的 CPU 没有此模块。本章 9.2 节介绍通过软件模拟 IIC 总线的方法，软件模拟的方式在绝大多数情况下能够满足要求，而且便于移植到其他类型单片机的程序中。

9.1 DMA 数据传输

DMA（direct memory access）是直接存储器访问的意思。非 DMA 方式访问内存时，数据的读写由 CPU 控制，如果要将一个字节数据从 A 地址复制到 B 地址，那么 CPU 会首先将数据从 A 地址读到 CPU 内的寄存器中，然后再写入 B 地址。这种方式占用 CPU 的资源，而且速度比较慢。使用 DMA 方式时，不需要 CPU 的干预，而是在 DMA 控制器的控制下直接将数据从 A 地址复制到 B 地址，这样的数据传输速度比较快。因为 MSP430 中地址线和数据线都只有一条，所以进行 DMA 数据传输时，CPU 必须交出地址线和数据线的控制权，此时 CPU 由于无法从程序存储器中读取指令，所以处于停止状态，无法执行任何指令。

MSP430 的 DMA 控制器目前只存在于 15x\16x 系列中，具有如下特性：

（1）拥有 3 个独立的 DMA 通道。

（2）可以配置通道的优先权。

（3）每个字\字节传送只需要两个 MCLK 时钟周期。

（4）字节和字可以混合传送：字节到字节、字节到字、字到字节、字到字。

（5）块传送最多可达 65 535 字节。

（6）可配置多种触发源。

（7）可配置 DMA 触发方式：边沿触发或电平触发。

（8）4 种传输寻址模式：固定地址到固定地址、固定地址到块地址、块地址到固定地址、块地址到块地址。

（9）单个、块或突发传输模式。

MSP430 的 DMA 控制器设置时需要做的设置为：

（1）设置 DMA 通道优先权。

（2）选择触发源　以下为可选择的触发源：

- 软件触发　DMAREQ 位置位
- 定时器 A 的 CCIFG.2 置位触发
- 定时器 B 的 CCIFG.2 置位触发
- UART0 及 IIC 接收到数据触发
- UART0 及 IIC 发送数据完毕触发
- DAC12.0IFG 置位触发
- ADC12.0IFG 置位触发
- 定时器 A0 的 CCIFG 置位触发
- 定时器 B0 的 CCIFG 置位触发
- UART1 接收到数据触发
- UART1 发送数据完毕触发
- 乘法器准备好时触发

- DMA0IFG 触发 DMA1 通道，DMA1IFG 触发 DMA2 通道，DMA2IFG 触发 DMA0 通道
- 外部触发

（3）确定触发方式。

- 边沿触发方式
- 电平触发方式（只有外部触发式才需要采用电平触发方式）

（4）设置传输源地址和目的地址。

（5）选择传输模式：

- 单字/单字节传输
- 块传输
- 突发块传输
- 重复单字/单字节传输
- 重复块传输
- 重复突发块传输

程序 9-1 的功能为，使用 DMA 通道 0 将 WriteData 的数据传输到 r_data 中，DMA 传输触发方式为软件触发。

文件 dsp_dma.c 为 DMA 的驱动程序，包含 3 个 DMA 通道的初始化函数 InitDma0、InitDma1、InitDma2，打开或者 DMA 通道函数 OpenDma，设置 DMA 传输源、目的地址函数 DmaAdr，软件触发 DMA 传输函数 DmaSoftGo。

3 个通道的初始化函数 InitDma0、InitDma1、InitDma2 的初始化过程是一样的。首先选择触发源，本程序选择软件触发。选择传输模式为块传输，传输时源地址和目的地址都会自动增加。

函数 OpenDma 打开或者关闭某个 DMA 通道。3 个 DMA 通道各有独立的控制寄存器 DMAxCTL。这里使用了一个无符号整形的指针 pr 来对 DMAxCTL 操作。根据要使用的 DMA 通道，使 pr 指向相应的 DMAxCTL。因为 DMAxCTL 的数据类型是无符号整数，所以，必须进行强制类型转换。最后，根据打开还是关闭 DMA 通道，将 DMA 控制寄存器的 DMAEN 位复位或者置位。

程序 9-1 的结构比较简单，程序结构图省略。

程序 9-1：

```
    msp_dma.h
    #ifndef __MSP_DMA
    #define __MSP_DMA


    void InitDma0();
    void InitDma1();
    void InitDma2();
    void OpenDma(unsigned char doit,unsigned which);
    void DmaAdr(unsigned char which,unsigned int src_adr,unsigned int det_adr,
unsigned int size);
```

```
void DmaSoftGo(unsigned char which);
#endif

msp_dma.c
#include "msp430x16x.h"
#include "msp_dma.h"
/*********************************************************************
初始化 DMA，内容包括：设置通道优先权、选择触发源、确定触发方式、设置传输源
地址和目的地址、选择传输模式。
*********************************************************************/
/*********************************************************************
初始化通道 0
*********************************************************************/
void InitDma0()
{
   DMACTL0 &= DMA0TSEL_0;  //清除原来的触发源
   DMACTL0 |= DMA0TSEL_0;  //选择触发源：软件触发
   DMA0CTL  &= DMADT_0;     //清除原来的传输模式

   //选择传输模式：块传输。目的地址自动增加。源地址自动增加。
   //目的单元的存储单位为字节。源单元的存储单位为字节
   DMA0CTL |= MADT_1+DMADSTINCR_3+DMASRCINCR_3+DMADSTBYTE+DMASRCBYTE;
}

/*********************************************************************
初始化通道 1
*********************************************************************/
void InitDma1()
{
   DMACTL0 &= DMA1TSEL_0;  //清除原来的触发源
   DMACTL0 |= DMA1TSEL_0;  //选择触发源：软件触发
   DMA1CTL  &= DMADT_0;     //清除原来的传输模式

   //选择传输模式：块传输。目的地址自动增加。源地址自动增加。
   //目的单元的存储单位为字节。源单元的存储单位为字节
   DMA1CTL |= MADT_1+DMADSTINCR_3+DMASRCINCR_3+DMADSTBYTE+DMASRCBYTE;
}
```

```
/**********************************************************************
初始化通道 2
**********************************************************************/
void InitDma2()
{
    DMACTL0 &= DMA2TSEL_0;  //清除原来的触发源
    DMACTL0 |= DMA2TSEL_0;  //选择触发源：软件触发
    DMA2CTL &= DMADT_0;     //清除原来的传输模式

    //选择传输模式：块传输。目的地址自动增加。源地址自动增加。
    //目的单元的存储单位为字节。源单元的存储单位为字节
    DMA2CTL |= MADT_1+DMADSTINCR_3+DMASRCINCR_3+DMADSTBYTE+DMASRCBYTE;
}

/**********************************************************************
打开或关闭 DMA
doit：0：停止；100：运行；其他：什么都不做
which：操作的 DMA 的通道号 0～2
**********************************************************************/
void OpenDma(unsigned char doit,unsigned which)
{
    unsigned int *pr;
    switch(which)
    {
        case 0:
            pr=(unsigned int *)DMA0CTL_;
        break;
        case 1:
            pr=(unsigned int *)DMA1CTL_;
        break;
        case 2:
            pr=(unsigned int *)DMA2CTL_;
        break;
    }
    if(doit==0)
    {
        *pr &= ~DMAEN;
    }
```

```
        else if(doit==100)
        {
            *pr |= DMAEN;
        }
    }

    /**********************************************************************
    设置 DMA 传输的源、目标地址
    which：DMA 通道号 0～2
    src_adr：源地址
    det_adr：目标地址
    size：传送字或者字节数目。
    **********************************************************************/
    void DmaAdr(unsigned char which,unsigned int src_adr,unsigned int det_adr,unsigned
int size )
    {
        if(which==0)
        {
            DMA0SA=src_adr;
            DMA0DA=det_adr;
            DMA0SZ=size;
        }
        else if(which==1)
        {
            DMA1SA=src_adr;
            DMA1DA=det_adr;
            DMA1SZ=size;
        }
        else
        {
            DMA2SA=src_adr;
            DMA2DA=det_adr;
            DMA2SZ=size;
        }
    }

    /**********************************************************************
    软件触发 DMA
```

```
which：操作的DMA的通道号 0～2
*********************************************************************/
void DmaSoftGo(unsigned char which)
{
    switch(which)
    {
        case 0:
            DMA0CTL |= DMAREQ;
        break;
        case 1:
            DMA1CTL |= DMAREQ;
        break;
        case 2:
            DMA2CTL |= DMAREQ;
        break;
    }
}

main.c
#include  <msp430x16x.h>
#include "msp_dma.h"

#define N_DATA 7
const unsigned char WriteData[N_DATA]={15,1,9,3,4,5,6};

void InitSys();

int main( void )
{
    unsigned int src,det;
    unsigned char r_data[N_DATA]={0,0,0,0,0,0,0};

    WDTCTL=WDTPW+WDTHOLD;             //关闭看门狗
    InitSys();                        //初始化

    src=(unsigned int)WriteData;
    det=(unsigned int)r_data;
    DmaAdr(0,src,det,N_DATA);
```

```
    OpenDma(100,0);
    DmaSoftGo(0);
    LPM4;                              //进入低功耗模式 4
}

/************************************************************************
系统初始化
************************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;                    //打开 XT2 振荡器
    do
    {
        IFG1 &= ~OFIFG;                    //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);      //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!=0);             //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                   //选择 MCLK、SMCLK 为 XT2
    InitDma0();                            //初始化 DMA0
}
```

9.2 IIC 总线

IIC 总线是目前非常流行的一种串行总线，其特点为：实现简单、可靠性好、低成本、总线上任何设备都可以成为主设备、具有总线竞争和仲裁能力，包括 EEPROM、FALSH、RAM、AD 转换器、DA 转换器、LCD 驱动器、LED 驱动器、IO 接口、键盘接口等上百种。很多外围芯片都采用了 IIC 接口，典型的有 EEPROM 存储器 24Cxx 系列、键盘和 LED 接口芯片 ZLG7290、实时时钟 M41T0、16 位 AD 转换器 ADS1100，很多微处理器也集成了 IIC 总线的功能。

MSP430 系列中 F15x/F16x 集成了 IIC 模块，其他的型号没有集成。由于 IIC 总线协议非常简单，所以，可以用软件来模拟。程序 9-2 中的 iic.c 为 IIC 实现 iic 总线协议的程序模块，可以将 MSP430 作为主芯片，与各种拥有 IIC 总线功能的芯片进行通信。

IIC 总线的硬件接口电路如图 9-1 所示，总线由 SDA 和 SCL 两根线组成，每根线上都连

接一个上拉电阻，电阻的阻值通常在 3.3kΩ～10kΩ 之间。

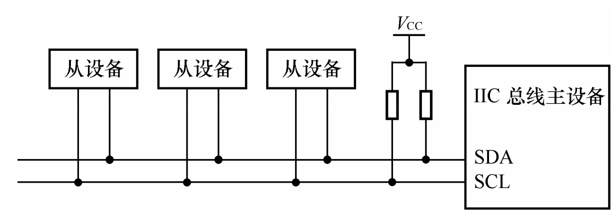


图 9-1　IIC 硬件接口电路

IIC 总线协议的基本内容为：SDA 为数据线，SCL 为同步时钟线。同步时钟信号 SCL 始终由主设备发送。IIC 协议中包含两种信号：控制信号和数据信号。SCL 为高电平时，SDA 上的电平如果发生跳变，则表明传送的是控制信号。SCL 为高电平时，SDA 上的信号稳定，表明传送的是数据。传送数据时，SDA 上的信号只能在 SCL 为低电平时改变。控制信号有：

- 启动（start）　SCL 为高电平时，SDA 出现一个下降沿。
- 停止（stop）　SCL 为高电平时，SDA 出现一个上升沿。

每一个传送的数据为 8 位，高位在前。每一位数据对应一个 SCL 周期，发送方在 SCL 为低电平时向 SDA 上写数据，接收方在 SCL 为高电平时读取 SDA 上的数据。每传送一个字节，后跟一个响应位，由接收方发送。响应位有两种：

- 应答（ACK）　SCL 为高电平时，SDA 为低电平。
- 无应答（NO ACK）　SCL 为高电平时，SDA 为高电平。在传送最后一个字节时使用。

IIC 总线中每一个设备都有自己的固定地址。主设备首先执行启动操作，然后发送地址字节，以确定要操作的从设备，其中地址为高 7 位，最低位表示对从设备进行何种操作：主设备向从设备写（W）为低电平，主设备从从设备中读（R）为高电平。从设备比较收到地址字节的高 7 位，如果收到的地址在自己的地址范围之内，则发送应答，并根据操作方向位决定是接收数据还是发送数据。

主设备向从设备写的情况为：主设备首先发出启动（start）信号，接着发出地址字节，地址字节的最低位为 0。从设备比较收到的地址，如果收到的地址在自己的地址范围之内，则成为被选中的设备，于是发送应答信号 ACK。主设备收到 ACK 信号后，发送数据字节。从设备每收到一个数据字节就返回一个 ACK 应答信号。主设备收到应答信号 ACK 后接着发送下一字节，直到所有数据发送完毕。最后由主设备发送停止（stop）信号。根据从设备的情况，一般从设备内部有多个寄存器，所以，主设备发送的第一个数据字节通常是从设备内部寄存器的地址，以确定即将写入数据的地址。具体的地址确定方式需查阅从设备的器件手册。

主设备读从设备的情况为，主设备首先发出启动（start）信号，接着发出地址字节，地址字节的最低位为 1。从设备比较收到的地址，如果收到的地址在自己的地址范围之内，则成为被选中的设备，于是发送应答信号 ACK。主设备收到从设备返回的 ACK 信号后，就开始发送 SCL 时钟信号，接收从设备发来的数据字节。主设备每收到一个字节，就发送一个 ACK 信号给从设备。在收完最后一个字节时，主设备发送 NO ACK 信号，然后发送停止（stop）信号。根据从设备的情况，一般从设备内部有多个寄存器，所以，在读数据之前

必须先确定从设备中要读出的寄存器的地址。通常的过程为：start 信号->从设备地址字节（写）->从设备内部寄存器地址字节->start 信号->从设备地址字节（读）->接收从设备数据1->接收从设备数据 2……->接收从设备数据 n->stop 信号。具体的地址确定方式需查阅从设备的器件手册。

24C02 是 24 系列 EEPROM 存储器中的一员，介绍 IIC 总线时很多都使用它作为例子，本书也针对 24C02 进行举例。24C02 存储容量为 256×8 字节，按照字节方式寻址。存储数据时，需要延时以便 24C02 完成内部编程，延时时间最长不超过 10ms。24 系列具有多种不同工作电压范围的芯片，注意选择与 MSP430 工作电压相配合的芯片。24C02 的管脚排列如图 9-2 所示。其管脚功能如下：

- A0、A1、A2　输入管脚，用来确定 24C02 的设备地址。由 24C02 的硬件决定其地址字节的 7～4 位固定为 1010(二进制)，A2、A1、A0 确定地址字节的 3～1 位，第 0 位为方向位。如：程序 9-2 的 24C02 中 A0、A1、A2 全部接低电平，其读地址为 0xa1，写地址为 0xa0。

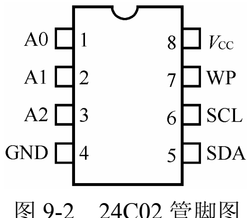


图 9-2　24C02 管脚图

- SCL、SDA　IIC 总线。
- WP　写保护。接高电平时，禁止对 24C02 存储数据。

24C02 的工作模式如下：

（1）单字节写　每次在 24C02 中的某个地址写一个字节。

（2）页写　每次最多允许向 24C02 中连续写 8 个字节的数据。从某一地址开始，24C02 每接收一字节的数据，地址就会自动加 1。这样就不用每次发送地址，可以大大加快写入的速度。

（3）当前地址读　24C02 内部有一个保存存储器地址的寄存器，每次读写后自动指向下一字节。使用当前地址读省去了发送存储器内部地址的过程，可以提高读出的速度。

（4）任意读　从 24C02 中任意地址读出一字节数据。每次必须发送存储器内部要读出的单元的地址。

（5）连续读　从某一首地址开始，连续读出数据。

程序 9-2 的功能为通过调用 24C02.c 中的函数读写 24C02 中的数据。程序 9-2 的工作环境为：MCLK 为 8MHz。

文件 24C02.c 是 24C02 的驱动模块，包括函数 Write24c02、Read24c02、Read24c02Seq。

函数 Write24c02 从某一地址开始向 24c02 中写入多个字节，这个函数没有使用 24C02 芯片中的页写工作模式，而是采用了循环调用单字节写的工作模式，原因是页写每次有写入字节数的限制。如果写入字节数超过了字节数的限制，就需要进行额外的计算。在时间允许的情况下，反而不如循环调用单字节写更简洁、可靠。

函数 Read24c02 从 24c02 中的某个地址读出一个字节。函数 Read24c02Seq 从 24c02 中的某个地址连续读出多个字节。

与其他使用 IIC 总线的芯片一样，24C02 除了遵守 IIC 总线协议，同时又拥有自己独特的部分。24C02.c 是在 IIC 软件模块的基础上进行二次封装，实现了针对 24C02 的读写模块。由于 24C02 在存储数据后必须经过延时，因此，又调用了 general.c 中的延时函数 DelayMs，延时函数不属于 IIC 总线协议的一部分。24C02 的地址在 24C02.h 中定义，分别为

W_DEVICE_ADR 和 R_DEVICE_ADR。其他 24 系列中的芯片读写模块只需要修改 24C02.c，很容易就能满足要求。

文件 iic.c 为 IIC 总写协议模块，包括函数 InitIIC、Start、Stop、Ack、NoAck、TestAck、Write8Bit、Read8Bit、delay。因为 IIC.c 不包含任何与操纵对象（24C02）硬件相关的内容，所以可以像 24C02 程序模块那样，通过二次封装应用到其他 IIC 芯片的驱动模块中。

函数 InitIIC 为 IIC 总线的初始化函数，功能为将 SCL 设置为输出，SDA 管脚设置为输入。然后初始化 IIC 总线上的电平，为发送和接收数据做好准备。函数 Start 在 IIC 总线上模拟发送 start 信号。函数 Stop 在 IIC 总线上模拟发送 stop 信号。函数 Ack 在 IIC 总线上模拟发送 Ack 信号。函数 NoAck 在 IIC 总线上模拟发送 NoAck 信号。函数 TestAck 在 IIC 总线上模拟检测从设备发来的 Ack 信号。函数 Write8Bit 将一字节的数据发送到 IIC 总线上。函数 Read8Bit 从 IIC 总线上接收一字节的数据。

以上几个函数实现了 IIC 总线协议所规定的几个基本操作。所有这些操作是建立在对 SCL 和 SDA 更基本的 7 种操作之上，这些操作为：SDA 置位、SCL 置位、SDA 复位、SCL 复位、设置 SDA 为输入、设置 SDA 为输出、读 SDA 的输入电平。程序在文件 iic.h 中将这 7 种操作分别定义为 7 个宏。在文件 iic.c 中，使用这些宏来完成这 7 种操作。这样做的好处是，如果更换了其他类型的 CPU，只需要将这 7 个宏重新定义就可以了，因为只有这一部分与 CPU 的硬件有关系。这样，这段代码就能够很容易地移植到其他类型 CPU 的系统中去。当然还要考虑 CPU 的运算速度，SCL 的频率不能太快，参见 delay 函数的解释。

函数 delay 为延时程序。主要用来满足 IIC 协议中 SCL 脉冲的最短时间特性，SCL 脉冲的频率必须符合总线上所有 IIC 器件的最低要求。由于函数 delay 使用 for 循环的方法进行延时，所以，当主时钟 MCLK 使用的频率改变时，应当调整循环的次数，以便保证延时时间处于合理的范围内。

文件 general.c 中的 DelayMs 函数是一个毫秒级的循环延时函数，利用 for 循环语句完成。MSP430 中一个 for 循环需要 8 个时钟周期。在 general.h 中首先定义 FREQUENCY 为 MCLK 的频率，单位为 kHz，定义 LOOPCNT 为 FREQUENCY/8，这样 LOOPCNT 就为 1 毫秒可以执行的 for 循环的次数。执行 n 个 for 循环就延时 n 毫秒。使用不同的 MCLK 频率，只需要修改 FREQUENCY 的定义就可以了。

文件 main.c 是主程序，其功能为对 24c02 进行读写，以验证 24c02 驱动程序和 IIC 总线程序的正确性。

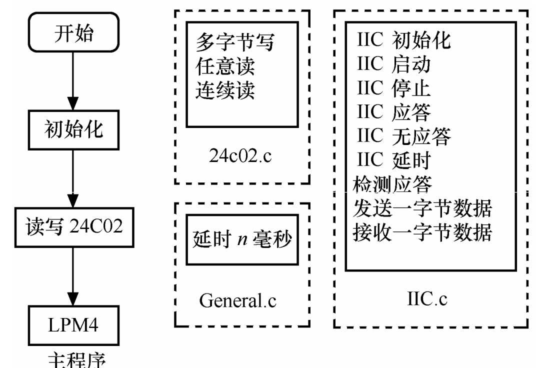


图 9-3　程序 9-2 结构图

程序 9-2 的程序结构框图如图 9-3 所示。

程序 9-2：

```
24c02.h
#ifndef __24C02
#define __24C02
unsigned char Write24c02(unsigned char*psrc_data,unsigned char adr,unsigned
```

```
char nbyte);
    unsigned char Read24c02(unsigned char *pdin_data,unsigned char adr);
    unsigned char Read24c02Seq(unsigned char *pdin_data,unsigned char adr,unsigned
char nbyte);

    #define W_DEVICE_ADR 0xa0        //24C02 的写地址   A0、A1、A2 均为 0
    #define R_DEVICE_ADR 0xa1        //24c02 的读地址   A0、A1、A2 均为 0
    #endif

    24c02.c
    #include  <msp430x14x.h>
    #include "24c02.h"
    #include "iic.h"

    /************************************************************************
    向 24c02 中写多个字节
    psrc_data：指向要写入数据数组的指针
    adr：24c02 中要写入数据的首地址
    nbyte：写入的字节数
    返回值： 0：执行完毕；  1：执行出现错误
    ***********************************************************************/
    unsigned char Write24c02(unsigned char* psrc_data,unsigned char adr,unsigned
char nbyte)
    {
        for(;nbyte!=0;nbyte--)
        {
            Start();                        //启动 IIC 总线
            Write8Bit(W_DEVICE_ADR);        //写 24C02 的芯片地址以及确定下面执行写操作
            if(TestAck()==1)                //检验应答
                return 1;                   //若应答错误，则退出函数，返回错误
            Write8Bit(adr);                 //写对 24C02 操作的地址
            if(TestAck()==1)
                return 1;

            Write8Bit(*psrc_data);          //向 24C02 中写数据
            if(TestAck()==1)
                return 1;
            psrc_data++;                    //指向待写数据的指针加 1
            adr++;                          //对 24C02 的操作地址加 1
```

```
        Stop();                         //停止 IIC 总线
        DelayMs(10);                    //写入延时
    }
    return 0;
}

/***********************************************************************
从 24c02 中读一个字节
pdin_data：指向要保存读出数据的变量的指针
adr：24c02 中要读出数据的地址
返回值：0：执行完毕；1：执行出现错误
***********************************************************************/
unsigned char Read24c02(unsigned char *pdin_data,unsigned char adr)
{
    Start();                        //启动 IIC 总线
    Write8Bit(W_DEVICE_ADR);        //写 24C02 的芯片地址以及确定下面执行写操作
    if(TestAck()==1)
        return 1;
    Write8Bit(adr);                 //写对 24C02 操作的地址
    if(TestAck()==1)
        return 1;

    Start();                        //再次启动 IIC 总线
    Write8Bit(R_DEVICE_ADR);        //写 24C02 的芯片地址以及确定下面执行读操作
    if(TestAck()==1)
        return 1;
    *pdin_data=Read8Bit();          //从 24C02 中读数据，存入 pdin_data 所指的存储器中
    NoAck();                        //IIC 无应答操作
    Stop();                         //停止 IIC 总线
    return 0;
}

/***********************************************************************
从 24c02 读多个字节
pdin_data：指向要保存读出数据的数组的指针
adr：24c02 中要读出数据的首地址
nbyte：读出的字节数
返回值： 0：执行完毕；  1：执行出现错误
***********************************************************************/
```

```
    unsigned char Read24c02Seq(unsigned char *pdin_data,unsigned char adr,
unsigned char nbyte)
    {
        Start();                        //启动 IIC 总线
        Write8Bit(W_DEVICE_ADR);        //写 24C02 的芯片地址以及确定下面执行写操作
        if(TestAck()==1)
            return 1;
        Write8Bit(adr);                 //写对 24C02 操作的地址
        if(TestAck()==1)
            return 1;

        Start();                        //再次启动 IIC 总线
        Write8Bit(R_DEVICE_ADR);        //写 24C02 的芯片地址以及确定下面执行读操作
        if(TestAck()==1)
            return 1;
        while(nbyte!=1)
        {
            *pdin_data=Read8Bit();      //循环从 24C02 中读数据，存入 pdin_data 所指的存储器中
            Ack();                      //IIC 应答
            pdin_data++;                //指向存储读入数据的存储器指针加 1
            nbyte--;                    //剩余要读入的字节减 1
        };

        *pdin_data=Read8Bit();          //读入最后一个字节
        NoAck();                        //IIC 无应答操作
        Stop();
        return 0;
    }

    iic.h
    #ifndef __IIC
    #define __IIC

    //定义 IIC 端口
    #define IIC_DIR P3DIR
    #define IIC_OUT P3OUT
    #define IIC_IN  P3IN
    #define IIC_SCL BIT3                        //SCL 定义
    #define IIC_SDA BIT1                        //SDA 定义
```

```
//定义 SDA 和 SCL 的操作方式
#define S_SDA   IIC_OUT |= IIC_SDA      //SDA 置位
#define S_SCL   IIC_OUT |= IIC_SCL      //SCL 置位
#define C_SDA   IIC_OUT &= ~IIC_SDA     //SDA 复位
#define C_SCL   IIC_OUT &= ~IIC_SCL     //SCL 复位

#define SDA_IN   IIC_DIR &= ~IIC_SDA    //设置 SDA 为输入
#define SDA_OUT  IIC_DIR |= IIC_SDA     //设置 SDA 为输出

#define READ_SDA (IIC_IN&IIC_SDA)       //读 SDA 电平

void InitIIC();
void Start();
void Stop();
void Ack();
void NoAck();
unsigned char TestAck();
void Write8Bit(unsigned char input);
unsigned char Read8Bit();
void delay();

#endif

iic.c
#include <msp430x14x.h>
#include "24c02.h"
#include "iic.h"
#include "general.h"
/*********************************************************************
初始化 IIC
*********************************************************************/
void InitIIC()
{
    IIC_DIR |= IIC_SCL;             //SCL 管脚为输出
   IIC_DIR &= ~IIC_SDA;             //SDA 管脚为输入
   C_SCL;
   Stop();
}
```

```
/**********************************************************************
进行短暂的延时，原因是MSP430的速度比较快。使用者可以根据时钟频率自行调节延时长短。
**********************************************************************/
void delay()
{
   unsigned char q0;
   for(q0=0;q0<20;q0++)
   {
       _NOP();
   }
}

/**********************************************************************
启动IIC总线
**********************************************************************/
void Start()
{
   SDA_OUT;
   S_SDA;
   delay();
   S_SCL;
   delay();
   C_SDA;
   delay();
   C_SCL;
   delay();
}

/**********************************************************************
停止操作，释放IIC总线
*********************************************************************/
void Stop()
{
   SDA_OUT;
   C_SCL;
   delay();
   C_SDA;
   delay();
   S_SCL;
```

```
    delay();
    S_SDA;
    delay();
}

/***********************************************************************
IIC 总线应答
***********************************************************************/
void Ack()
{
    SDA_OUT;
    C_SDA;
    delay();
    S_SCL;
    delay();
    C_SCL;
    delay();
    S_SDA;
}

/***********************************************************************
IIC 总线无应答
***********************************************************************/
void NoAck()
{
    SDA_OUT;
    S_SDA;
    delay();
    S_SCL;
    delay();
    C_SCL;
    delay();
}

/**********************************************************************
IIC 总线检验应答
返回值：IIC 应答位的值。0：应答；1：无应答
**********************************************************************/
unsigned char TestAck()
```

```
{
    unsigned char ack;

    S_SCL;
    delay();
    SDA_IN;
    delay();
    ack=READ_SDA;
    delay();
    C_SCL;
    delay();
    return(ack);
}

/**********************************************************************
IIC 总线写 8 位数据
input：要写的 8 位数据
**********************************************************************/
void Write8Bit(unsigned char input)
{
    unsigned char temp,q0;
    SDA_OUT;
    for(temp=8;temp!=0;temp--)
    {
        q0=input&0x80;
        if(q0==0x80)
            S_SDA;
        else
            C_SDA;
      delay();
        S_SCL;
        delay();
        C_SCL;
        delay();
        input=input<<1;
    }
}

/**********************************************************************
```

```
IIC 总线读 8 位数据
返回值：读出的 8 位数据
***********************************************************************/
unsigned char Read8Bit()
{
   unsigned char temp,q0,rbyte=0;
   SDA_IN;
   for(temp=8;temp!=0;temp--)
   {
      S_SCL;
      delay();
      rbyte=rbyte<<1;
      SDA_IN;
      q0=READ_SDA;
      if(q0==IIC_SDA)
         rbyte=rbyte|0x1;
      delay();
      C_SCL;
      delay();
   }
   return(rbyte);
}

general.h
#ifndef __GENERAL
#define __GENERAL

#define FREQUENCY   8000        //MCLK 的频率 8000kHz
#define LOOPBODY    8           //MSP430 中一个 for 循环体耗费 8 个周期
#define LOOPCNT (unsigned int)((FREQUENCY/LOOPBODY))

void DelayMs(unsigned int ms);
#endif

generla.c
#include "general.h"
/**********************************************************************
延时
ms：延时的毫秒数
```

```
***********************************************************************/
void DelayMs(unsigned int ms)
{
   unsigned int iq0, iq1;
   for (iq0=ms; iq0>0; iq0--)
   for (iq1=LOOPCNT; iq1>0; iq1--)
   ;
}

main.c
#include  <msp430x14x.h>
#include "iic.h"
#include "24c02.h"

#define N_DATA 7
const unsigned char WriteData[N_DATA]={15,1,2,3,4,5,6};

void InitSys();

int main( void )
{
   unsigned char q0,q1,q2;
   unsigned char *pdata;
   unsigned char w_data[N_DATA]={10,20,30,40,50,60,70};
   unsigned char r_data[N_DATA]={0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
   unsigned char r_data1[N_DATA]={0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};

   WDTCTL=WDTPW+WDTHOLD;            //关闭看门狗
   InitSys();                       //初始化

   q0=Write24c02(w_data,30,N_DATA);  //将 w_data 中的数据写入 24C02 中，首地址为 30
   //读 24C02 中的多个字节数据，首地址为 30，存入 r_data
   pdata=r_data;                    //存储数据的指针指向 r_data
   q2=30;                           //读数据首地址为 30
   for(q1=0;q1<N_DATA;q1++)
   {
      q0=Read24c02(pdata,q2);               //读取数据
      pdata++;                              //存储数据的指针加 1
      q2++;                                 //读数据的地址加 1
```

```
    }

    //将 WriteData 中的数据写入 24C02 中，首地址为 0
    q0=Write24c02((unsigned char*)WriteData,0,N_DATA);
    //读 24C02 中多个字节数据，首地址为 0，存入 data1
    q0=Read24c02Seq(r_data1,0,N_DATA);
    _NOP();
    LPM4;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
   //使用 XT2 振荡器
   BCSCTL1 &= ~XT2OFF;                        //打开 XT2 振荡器
   do
   {
        IFG1 &= ~OFIFG;                       //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);         //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!= 0);               //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                      //选择 MCLK、SMCLK 为 XT2
    InitIIC();                                //初始化 IIC 总线端口
}
```

第 10 章　FLL+锁频环与液晶屏驱动模块

MSP430F4xx 系列的 CPU 同时具有锁频环和液晶驱动模块。使用锁频环在不使用外部高频晶振的情况下也能得到高频率并且精确的时钟信号。本章第 10.1 节介绍 FLL+锁频环的使用方法。

以往的 CPU 驱动液晶屏时必须外接一个专门的驱动芯片来驱动液晶屏。MSP430 内部集成了液晶屏驱动模块。本章第 10.2 节介绍液晶屏驱动模块的使用方法。

集成锁频环和液晶屏驱动模块对于减小系统功耗、降低成本、缩小体积都非常有利。

10.1　FLL+锁频环

FLL+为增强型锁频环，存在于 MSP430F4xx 系列中。其功能为通过硬件自动调整 DCO 的频率。对于只有 XT1 时钟的 F41x/F42x，可以只接一个低频晶振，通过锁频环使 DCO 输出精确的高频时钟信号。对于有 XT2 的芯片，因为可以通过低频晶振产生精确的高频时钟信号，所以，多数情况下可以不在 XT2 上接晶振，这样可以降低系统的功耗和成本。MSP430F1xx 系列没有锁频环，因此只能通过软件调节 DCO 的频率，这样得到的时钟信号的精度有限。

FLL+调节 DCO 的输出频率受 3 个参数的影响：

（1）N　晶振频率的放大倍数。在 SCFQCTL 寄存器的 0～6 位。

（2）DCO+　决定是否允许按照 FLLDx 的值对 DCO 输出的频率再次放大。在 FLL_CTL0 的第 7 位。

（3）FLLDx　DCO+有效时，DCO 的输出频率再次放大的倍数。在 SCFI0 的第 7、6 两位。

由这 3 个参数所确定的 DCO 输出的频率为：

如果 DCO+=0，则 DCOCLK=(N+1) ×晶振频率

如果 DCO+=1，则 DCOCLK=FLLDx×(N+1)×晶振频率

还有一个需要说明的是调制器使能控制位 M，它在 SCFQCTL 寄存器的第 7 位。这一位被置位时，调制器通过混合相邻两个 DCLCLK 周期来克服累加周期误差。它不影响 DCO 输出频率，但对输出的每个周期的长短有影响。

程序 10-1 的功能为通过 FLL+调整 DCO 的输出频率。因为在默认的状态下 DCO 被用作 MCLK，所以，选择 P1.1 管脚使用第二功能输出 MCLK 信号，通过测量 P1.1 的输出信号频率就可以知道 MCLK 的频率，以此来了解 FLL+调整 DCO 的效果。

程序 10-1 的工作环境为：XT1 所接晶振为 32.768K。MCLK 选择 DCO。

fll.c 为锁频环 FLL+的驱动程序，只包含一个 FllInit 函数，为锁频环初始化函数。按照程序中设置的 N=31，FLLDx=2，DCO+允许，经过调制的频率输出值应该为 $32768\times(31+1)\times 2/1\times 10^6= 2.09$MHz，而在 P1.1 管脚上实际测量值为 2.1MHz。

main.c 为主程序。只在系统初始化函数 InitSys 中调用了锁频环初始化函数 FllInit，选择 P1.1 管脚工作在第二功能。最后进入低功耗模式。

程序 10-1 的结构比较简单，结构图省略。

程序 10-1：

```
fll.h
#ifndef __FLL
#define __FLL
void FllInit();
#endif
```

```
fll.c
#include <msp430x42x.h>
#include "fll.h"

/**********************************************************************
初始化
**********************************************************************/
void FllInit()
{
   SCFQCTL=SCFQ_1M;     //N=31，调制器使能
   SCFI0=FLLD_2;        //FLLDx=2，频率调整范围为：0.65～6.1MHz
   FLL_CTL0=DCOPLUS;    //DCO+有效：FDCOCLK=D*(N+1)*晶振频率
}

main.c
#include  <msp430x42x.h>
#include "fll.h"

void InitSys();

int main()
{
   WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
   InitSys();
start:
   _NOP();
   goto start;
}

/*********************************************************************
系统初始化
*********************************************************************/
void InitSys()
{
   P1DIR |= BIT1;       //使 MCLK 信号从管脚 P1.1 输出
   P1SEL |= BIT1;
   FllInit();
}
```

10.2 液晶屏驱动模块

目前越来越多的设备要求尽可能地减小能耗，在显示方面液晶显示屏已经在很多场合取代原来的 LED 数码管方式。

液晶屏一般分为段式和点阵式。段式和点阵式的原理是一样的，段式一般用于显示数字或者固定的几种图案。点阵式中的点实际上就相当于段式中的段，不过由于点很多，因此通常需要使用专用的集成电路来驱动。由于可以驱动的段比较有限，所以 MSP430 中的 LCD 驱动模块通常只能驱动段式液晶屏，如果不加说明，本节中的液晶屏都为段式液晶屏。

液晶屏必须使用交流驱动，否则会被损坏。驱动液晶屏的端口有两种：段极（SEGn）和公共极（COMn）。驱动方式有 4 种，如表 10-1 所示。

表 10-1　液晶模块驱动方式

方　式	公共极管脚数	每管脚驱动液晶段数	需要管脚总数
静态	1	1	1+需要驱动的段数
2MUX	2	2	2+需要驱动的段数/2
3MUX	3	3	3+需要驱动的段数/3
4MUX	4	4	4+需要驱动的段数/4

每种 MSP430 都有 4 个公共极 COM0～COM3。但段极的数目不同，所以可以驱动的液晶段也不同，驱动能力如表 10-2 所示。

表 10-2　MSP430 液晶驱动能力

型　　号	段极管脚数	可驱动最大段数
41x	S0～S23	96
42x	S0～S31	128
43x	S0～S31（80 脚封装）	128
	S0～S39(100 脚封装)	160
44x	S0～S39	160

MSP430 内部有显示缓存，显存的各个位与液晶的段一一对应。通过对显存中各位的置位和复位来使得液晶屏上对应的段显示或者消失。由于可驱动的段数不同，不同的型号中显存的大小也不同，但使用的方法是相同的。如图 10-1 所示，显存中的每个字节分为两组：高 4 位和低 4 位。每一个 4 位对应一个段的输出管脚（SEGn），4 位中的每一位又分别对应

COM3～COM0。按照段数最多的 MSP430F449 来计算，40 段管脚×4 公共极=160 段，MSP430F449 最多可驱动 160 段的液晶。

COM	3	2	1	0	3	2	1	0		
地址	7							0	n	段管脚
0A4h									38	39，38
0A3h									36	37，36
0A2h									34	35，34
0A1h									32	33，32
0A0h									30	31，30
09Fh									28	29，28
09Eh									26	27，26
09Dh									24	25，24
09Ch									22	23，22
09Bh									20	21，20
09Ah									18	19，18
099h									16	17，16
098h									14	15，14
097h									12	13，12
096h									10	11，10
095h									8	9，8
094h									6	7，6
093h									4	5，4
092h									2	3，2
091h									0	1，0
	S_{n+1}				S_n					

图 10-1　显示缓存器与液晶段的对应关系

液晶上的每一段都由一个公共极信号和一个段信号控制，从某种程度上来说，这与发光二极管的点亮方式类似，例如当公共极为低电平，段极为高电平，相应的段就显示出来。根据这个原理可以推断出显存中的内容与液晶屏上每一段的对应关系。

选用哪种液晶驱动方式由液晶屏的管脚和其内部电极排布决定，使用前必须搞清楚。通常段式液晶显示为数字“8”，各个段的定义如图 10-2 所示。以静态驱动模式为例，COM0 管脚与所有段的一端相连，每段的另一端与段极管脚相连。如 MSP430 段极管脚 S0 与液晶屏上的段 a（图 10-2）相连，那么可在显存（图 10-3）中找到与段极管脚 S0 和 COM0 对应的位置：地址为 0x91 的字节的最低位。向此位写入 1，液晶屏上的 a 段就会显示出来。由此可以看出，液晶屏上的每一段与 MSP430 的段极管脚和公共极管脚组合相对应，MSP430 的段极管脚和公共极管脚的组合又与显存中的某一位对应，当某一位发生变化时，显示内容也就随着变化了。如果改变 MSP430 段极管脚与液晶屏管脚的连接方式，同样的程序显示的内容也就不同了。

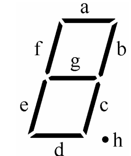


图 10-2　液晶段定义

通常连接 MSP430 与液晶管脚的时候遵循某一规律，这样程序比较容易移植，也便于理解。以下是几种驱动模式的常用接法。

图 10-3 为静态驱动模式的常用接法。可以看到左侧为 MSP430 的管脚与液晶屏管脚的对应关系。S0 对应 1a，意思是 S0 与液晶屏上显示的第一个“8”上的 a 段连在一起，其余以此类推。因为有了 MSP430 的管脚与液晶屏管脚这样的对应关系，显存与显示段的关系也就确定了。如图 10-3 的中间，地址为 0x91～0x94 的显存对应的段极管脚为 S0～S7，S0～S7 分别接在了液晶屏上第一个“8”的 8 个段上。又由于液晶屏只有一个公共极，连接在 COM0 上，所以，与这 8 段所对应的显存就是地址为 0x91～0x94 的这 4 个字节的第 0 位和第 4 位。右侧为液晶屏的段极和公共极的分配方式，上边是公共极的连接方式，可以看到只有一个公共极 COM0，它与所有的段对应。下面是段极，每个段极与一个段相对应。

图 10-4 为 2MUX 驱动方式。图 10-5 为 3MUX 驱动方式。图 10-6 为 4MUX 驱动方式。这 3 种驱动方式的工作原理与静态驱动方式相同。使用时读者可以根据所使用模式的图自行推断显存与显示内容的对应关系。

程序 10-2 的功能为：使用 MSP430 的液晶驱动模块在液晶屏上显示“9b:-H”，显示的内容按照一定的频率闪烁。

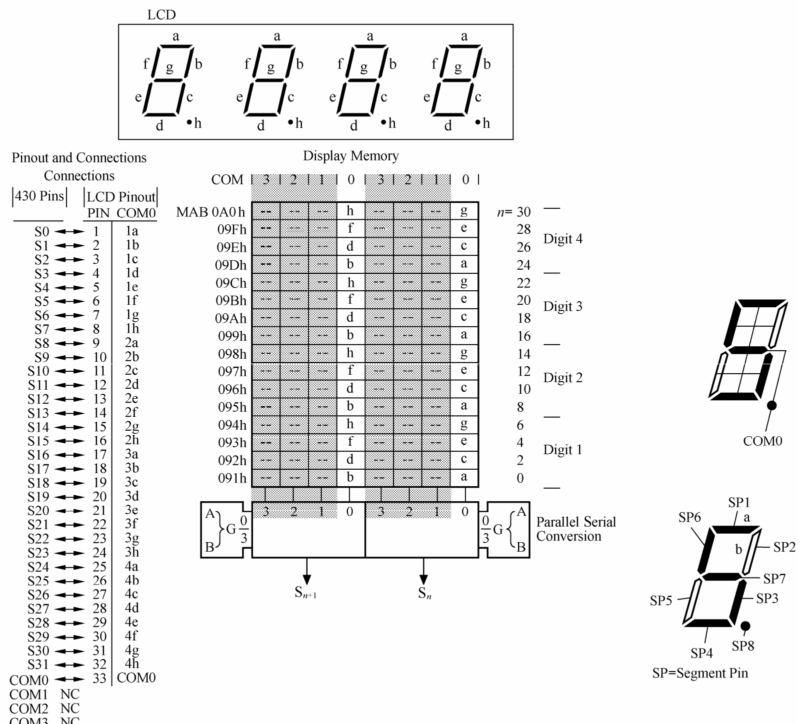


图 10-3　静态驱动模式

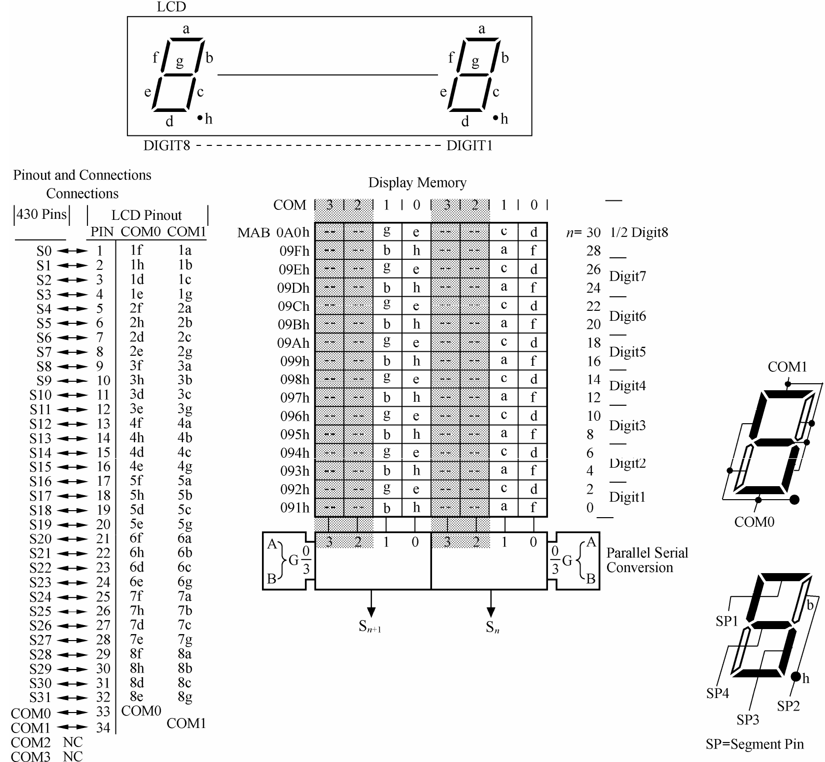


图 10-4　2MUX 驱动方式

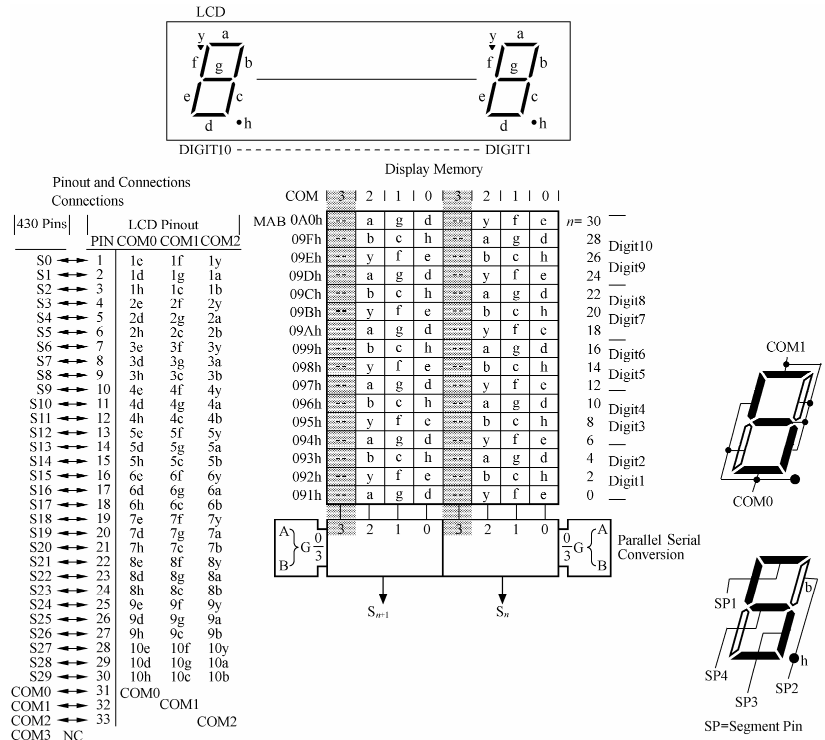


图 10-5　3MUX 驱动方式

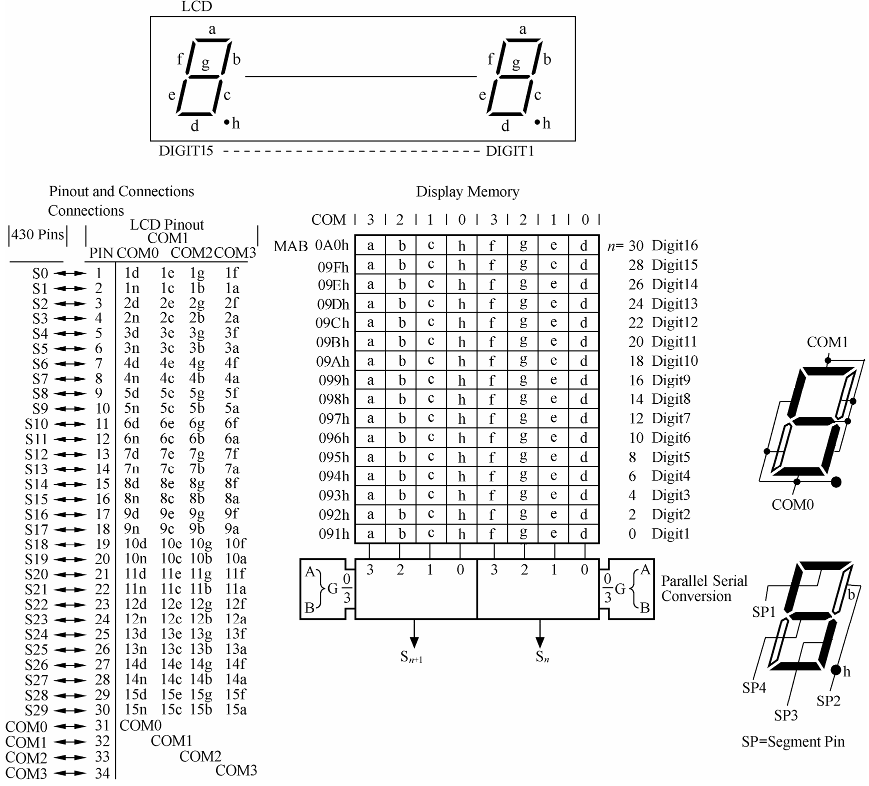


图 10-6　4MUX 驱动方式

程序 10-2 的工作环境为：CPU 使用 MSP430F425。MCLK 为 DCO，ACLK 为 32.768kHz。所使用液晶屏的型号为 LD-TD-10951，其显示段如图 10-7 所示，管脚的排列和功能如表 10-3 所示。液晶屏与 MSP430F425 的连接方式如附图 2 所示。

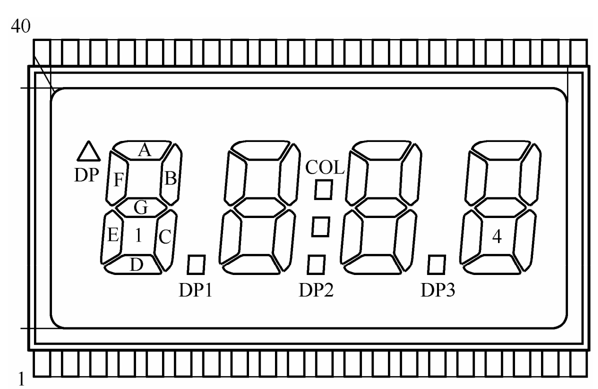


图 10-7 LD-TD-10951 液晶屏

表 10-3 LD-TD-10951 液晶屏管脚

管脚	1	2	3	4	5	6	7	8	9	10
段	COM	\	\	\	1E	1D	1C	DP1	2E	2D
管脚	11	12	13	14	15	16	17	18	19	20
段	2C	DP2	3E	3D	3C	DP3	4E	4D	4C	4B
管脚	21	22	23	24	25	26	27	28	29	30
段	4A	4F	4G	3B	3A	3F	3G	COL	2B	2A
管脚	31	32	33	34	35	36	37	38	39	40
段	2F	2G	\	1B	1A	1F	1G	DP	\	COM

从表 10-3 可以看出，LD-TD-10951 只有一个 COM 端口，所以，需要采用静态驱动方式。在 40 个管脚中，有两个功能相同的 COM 端，5 个标为“\”的空管脚，剩余 33 个管脚对应液晶屏上的 33 段显示。MSP430F425 工作在静态驱动模式时最多只能驱动 32 段液晶，因此，可以在 DP 段和 COL 段中选择一个来显示，数字和小数点全部保留。程序 10-2 选择显示 COL 段。

文件 lcd.c 和 bastimer.c 组成了 MSP430 液晶模块驱动程序。lcd.c 通过寄存器控制液晶模块的工作。bastimer.c 为液晶模块提供时钟信号。

函数 LcdInit 的功能为初始化液晶模块。通过写寄存器 LCDCTL 来设置液晶模块的工作方式。写入的内容中，LDCSG0_5 表示选择端口 s0～s31，LCDSTATIC 表示使用静态驱动方式。LDCSG0_5 和 LCDSTATIC 的定义在头文件 Msp430fx42x.h 中。

函数 LcdGo 的功能为打开或者关闭液晶显示。液晶的打开和关闭由寄存器 LCDCTL 中的 LCDON 位控制。

函数 LcdWrite 的功能为向显存中写数据。MSP430 的显存排列不是线性的，例如：静态显示时每一段所对应的位在显存中并不是紧密排列的，如图 10-3 所示，显存中每一字节只有第位 0 和第位 4 被用到。结合图 10-7 来看，地址 0x91 的第位 0 对应着数码 1 的 a 段，第 4

位对应着数码 1 的 b 段。而在 2MUX 方式时，显存中的位 0、1、4、5 位被用到。如地址 0x91 的位 0 对应着数码 1 的 f 段，位 1 对应着 a 段，位 4 对应着 h 段，位 5 对应着 b 段。因为考虑到有时会要求在不影响其他段的显示的情况下单独点亮或者熄灭一段，所以在向显存写数据时需要考虑覆盖写入、与写入、或写入 3 种情况。覆盖写入适用于全部刷新显存的情况，与写入适用于单独熄灭某一段的情况，或写入适用于单独点亮某一段的情况。函数中将指针 pmem 指向显存的基地址，计算出要写入的地址相对于基地址的偏移量 adr，将指针 pmem 加上偏移量 adr，指针就指向了要写入的地址，然后根据写入的方式对指针所指地址的内容进行操作。

函数 LcdBlink 的功能为显示或者全部消隐液晶屏上的内容。方法是控制寄存器 LCDCTL 中的 LCDSON 位，当 LCDSON 位为 1 时，液晶屏显示内容，当 LCDSON 为 0 时液晶屏不显示。

bastimer.c 使用程序 7-5 中的基本定时器模块，可以直接使用。这里可以再次看到结构化编程的优点。

文件 ping.c 为液晶屏 LD-TD-10951 的驱动程序。首先定义显示字模，由静态液晶驱动方式图 10-3 可以看出，每个字符的 8 段都由 4 个字节来控制，每个字节控制两段。定义 N_ZI 为 4，含义是每个字符用 4 个字节表示。如果是使用 2MUX 方式驱动，则应该定义 N_ZI 为 2，每个字符用 2 个字节表示。二维数组 LcdCh[19][N_ZI]为字模数组，包括全部熄灭的字符在内，共定义了 19 个不同的字符。4 个字符在显存中的地址偏移量分别是 12、8、4、0，储存在数组 LcdWei 中。LcdWei 是 const 类型，保存在 FLASH 中，不占用数据存储器 RAM。在显示某一位字符时可以根据 LcdWei 中的地址偏移量计算出要写入的起始地址，非常方便。

函数 PingInit 初始化与液晶显示有关的硬件。液晶屏调用 LcdInit 初始化液晶模块，调用 InitBasTimer 初始化基本定时器，调用 GoBasTimer 打开基本定时器为 LCD 提供时钟信号。

函数 ToLcd 将要显示的数码转换为对应的显示字模，并将字模写入显存。要显示的字符保存在 unsigned char 类型的数组中，显示时传递数组的指针给函数。从数组中循环读出要显示的字符，本程序中为 4 个字符。LcdCh[*pshow][0]为要显示字符对应字模的第一个字节，从此字节开始的 4 个字节为字符的字模。用&符号取此字节的地址，由于 LcdCh 为 const 类型，而 pmem 不是 const 类型，所以，必须进行强制类型转换，将从 LcdCh 中取到的地址转换为非 const 类型。调用 LcdWrite 将字模写入显存，字模的地址为 pmem，写入显存的位置由字符在液晶屏上的位决定，通过数组 LcdWei 查询出要写入的显存的首地址，写入字节为 N_ZI，4 个字节，写入方式为覆盖写入。每次写入完毕，将 pshow 加 1，指向要显示的下一个字符，如此循环 4 次，显示 4 位字符。

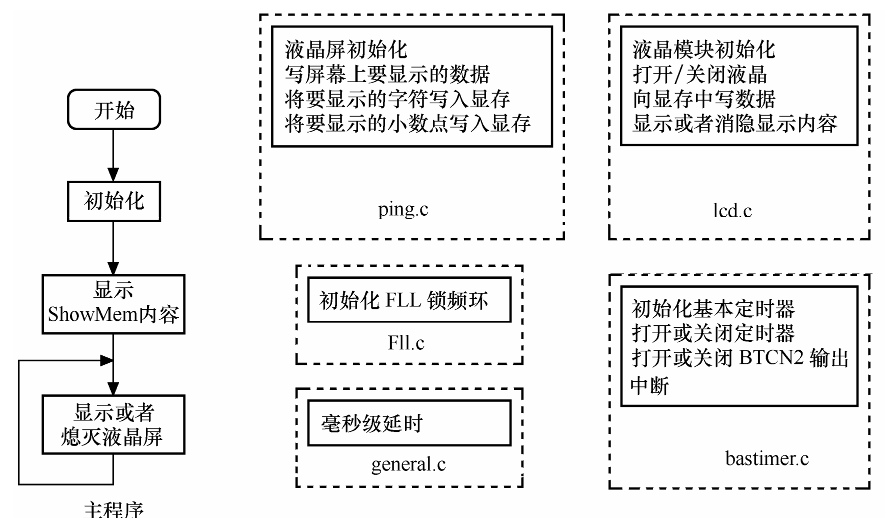
函数 ToLcdDian 的功能为显示或者消隐小数点。LcdCh 数组中字模对应的字符都不含小数点，因此，需要 ToLcdDian 函数来显示或者消隐小数点。函数 ToLcd 在向显存中写字模时使用的只有覆盖方式，原因是字符对应字模的 4 个字节总是以固定的组合出现的，不存在写入某个段时需要考虑另外的段是否要保留的情况。显示或者消隐小数点一定不能影响已经显示的其他的段，也就是除了小数点对应的位，不能改变显存中其他的位。小数点对应的位在 4 个字节中最后一个字节的第 4 位，因此，要写入的显存的偏移量为通过 LcdWei 查询到的值加 3。显示小数点时将写入的字节与 0x10 进行或运算，消隐小数点时将写入的字节与 0xEF 进行与运算。

函数 Show 刷新液晶屏上的显示内容。首先调用函数 ToLcd 显示 4 位字符，然后调用函数 ToLcdDian 显示小数点。这里只有一位小数点被点亮，没有考虑同时点亮多个小数点的情况。

程序 10-2 使用的 CPU 为 MSP430F425。MSP430F425 不能接外部晶振 XT2，但是具有锁频环。利用锁频环调整 DCO 的频率，只要调用函数 FllInit 对锁频环进行初始化即可。

main.c 是主程序。调用函数 Show 显示 ShowMem 中的内容“9b-H”。在每次循环中调用函数 LcdBlink，显示或者熄灭显示内容，造成闪烁的效果。

程序 10-2 的程序结构图如图 10-8 所示。



错误！

图 10-8 程序 10-2 结构图

程序 10-2

```
    lcd.h
    #ifndef __ LCD
    #define __ LCD
    void LcdInit();
    void LcdGo(unsigned char doit);
    void LcdWrite(unsigned char *pshow,unsigned char adr,unsigned char nchar,
unsigned char mod);
    void LcdBlink(unsigned char doit);
    #endif

    lcd.c
    #include  <msp430x42x.h>
    #include "lcd.h"
```

```
/*******************************************************************
初始化
*******************************************************************/
void LcdInit()
{
      LCDCTL=LCDSG0_5+LCDSTATIC;  //选择段端口 S0～S31，静态显示
}

/*******************************************************************
打开或关闭液晶
0：打开；100：关闭
*******************************************************************/
void LcdGo(unsigned char doit)
{
   if(doit==0)
   {
      //打开液晶显示
      LCDCTL |= LCDON;
   }
   else if(doit==100)
   {
      //关闭液晶显示
      LCDCTL &= ～LCDON;
   }
}

/******************************************************************
向显存中写数据
pshow：要写入数据的起始地址
adr：数据在显存中写入的起始位置 0～20
nchar：写入的字节数
mod： 0：覆盖写入；  1：与；  2：或
******************************************************************/
void LcdWrite(unsigned char *pshow,unsigned char adr,unsigned char nchar,
unsigned char mod)
{
   unsigned char *pmem=LCDMEM;
   pmem += adr;           //计算偏移写入地址
   for(;nchar!=0;nchar--)
```

```
    {
        if(mod==0)
            *pmem= *pshow;          //覆盖写入显存
        else if(mod==1)
            *pmem &= *pshow;        //“与”写入显存
        else
            *pmem |= *pshow;        //“或”写入显存
        pmem++;
        pshow++;
    }
}

/**********************************************************************
显示或者消隐显示内容
doit：0：消隐；  1：显示
**********************************************************************/
void LcdBlink(unsigned char doit)
{
    if(doit==0)
    {
        LCDCTL &= ~LCDSON;
    }
    else if(doit==1)
    {
        LCDCTL |= LCDSON;
    }
}

ping.c
#ifndef __PING
#define __PING
void PingInit();
void Show(unsigned char *pshow,unsigned char dian);
void ToLcd(unsigned char *pshow);
void ToLcdDian(unsigned char dian,unsigned char visable);
#endif

ping.c
#include "lcd.h"
```

```
#include "ping.h"
#include "general.h"
#include "bastimer.h"

//显示字模（与屏幕和MSP430的接法有关）
#define N_ZI  4                 //每个字符在显存中由4个字节组成
const unsigned char LcdCh[19][N_ZI] =
{
    {0x11,0x11,0x11,0x00},      //显示内容：0    数组下标：0
    {0x10,0x01,0x00,0x00},      //显示内容：1    数组下标：1
    {0x11,0x10,0x01,0x01},      //显示内容：2    数组下标：2
    {0x11,0x11,0x00,0x01},      //显示内容：3    数组下标：3
    {0x10,0x01,0x10,0x01},      //显示内容：4    数组下标：4
    {0x01,0x11,0x10,0x01},      //显示内容：5    数组下标：5
    {0x01,0x11,0x11,0x01},      //显示内容：6    数组下标：6
    {0x11,0x01,0x00,0x00},      //显示内容：7    数组下标：7
    {0x11,0x11,0x11,0x01},      //显示内容：8    数组下标：8
    {0x11,0x11,0x10,0x01},      //显示内容：9    数组下标：9
    {0x11,0x01,0x11,0x01},      //显示内容：A    数组下标：10
    {0x00,0x11,0x11,0x01},      //显示内容：b    数组下标：11
    {0x01,0x10,0x11,0x01},      //显示内容：C    数组下标：12
    {0x10,0x11,0x01,0x01},      //显示内容：d    数组下标：13
    {0x01,0x11,0x11,0x01},      //显示内容：E    数组下标：14
    {0x01,0x00,0x11,0x01},      //显示内容：F    数组下标：15
    {0x10,0x01,0x11,0x01},      //显示内容：H    数组下标：16
    {0x00,0x00,0x00,0x01},      //显示内容：-    数组下标：17
    {0x00,0x00,0x00,0x00}       //显示内容：熄灭  数组下标：18
};

//屏幕定义
#define N_SHOW 4            //屏幕显示的位数，最高位为3，最低位为0。
//小数点的位置，0在第0位（最右）上，依次类推。N_SHOW为没有小数点。
//显示数码各位在显存中的起始地址（与屏幕和MSP430的接法有关）
const unsigned char LcdWei[N_SHOW]={12,8,4,0};

/**********************************************************************
初始化
**********************************************************************/
void PingInit()
```

```
{
   LcdInit();              //初始化液晶模块
   InitBasTimer();         //初始化基本定时器
   GoBasTimer(0,2);        //打开基本定时器为 LCD 提供时钟
}

/**********************************************************************
写屏幕上要显示的数据
pshow：指向要写入的数据的指针
dian：显示小数点的位置
**********************************************************************/
void Show(unsigned char *pshow,unsigned char dian)
{
      ToLcd(pshow);           //刷新数码显示
      ToLcdDian(dian,1);      //刷新小数点显示
}

/*********************************************************************
将要显示的字符转换为显示用的字模，并写入显存。要显示的 4 个字符保存在一个数组中
pshow：指向要显示字符数组的指针
**********************************************************************/
void ToLcd(unsigned char *pshow)
{
   unsigned char *pmem;
   unsigned char q0;
   for(q0=0;q0<N_SHOW;q0++)
   {
      pmem=(unsigned char *)(&LcdCh[*pshow][0]);      //计算显示字模的首地址
      LcdWrite(pmem,LcdWei[q0],N_ZI,0);               //显示数码，覆盖写入
      pshow++;
   }
}

/**********************************************************************
显示小数点，并写入显存
dian：显示小数点的位置
visable:  0：消隐；  1：显示
**********************************************************************/
void ToLcdDian(unsigned char dian,unsigned char visable)
```

```
{
   unsigned char q0,q1,q2;
   if(dian<N_SHOW)
   {
      if(visable==0)
      {
         q1=1;                       //写入LCD模式：与
         q2=0xEF;
      }
      else
      {
         q1=2;                       //写入LCD模式：或
         q2=0x10;
      }
      q0=LcdWei[dian]+3;             //计算写入显存的起始地址
      LcdWrite(&q2,q0,1,q1);         //写入显存
   }
}

bastimer.h
#ifndef __BASIC_TIMER
#define __BASIC_TIMER
void InitBasTimer();
void GoBasTimer(unsigned char doit,unsigned char which);
void IntBtcn2(unsigned char doit);
#endif

bastimer.c
/***********************************************************************
初始化基本定时器
***********************************************************************/
void InitBasTimer()
{
    //确定液晶时钟信号和BTCN2的输入时钟源
    BTCTL=BT_fLCD_1K+BT_fCLK2_ACLK;
}

/***********************************************************************
打开或关闭定时器，BTCN2只能与BTCN1一起运行，单独打开BTCN2没有意义。
```

```
doti: 0: 打开;     100: 关闭
which: 操作定时器的哪一部分   0: BTCN1 和 BTCN2;  1: BTCN2;   2:BTCN1
**********************************************************************/
void GoBasTimer(unsigned char doit,unsigned char which)
{
   if(doit==0)
   {
      if(which==0)
         BTCTL &= ~BTHOLD;        //打开 BTCN1 和 BTCN2
      else if(which==2)
      {
         BTCTL |= BTHOLD;         //仅打开 BTCN1
         BTCTL &= ~BTDIV;
      }
   }
   else if(doit==100)
   {
      if(which==0)
         BTCTL |= BTHOLD+BTDIV;   //关闭 BTCN1 和 BTCN2
      else if(which==1)
         BTCTL |= BTHOLD;         //关闭 BTCN2
   }
}

/**********************************************************************
打开或关闭 BTCN2 输出中断
doit: 0: 打开;  100: 关闭
**********************************************************************/
void IntBtcn2(unsigned char doit)
{
   if(doit==0)
      IE2 |= BTIE;             //打开
   else if(doit==100)
      IE2 &= ~BITE;            //关闭
}

fll.h
#ifndef __FLL
#define __FLL
```

```
void FllInit();
#endif

fll.c
#include <msp430x42x.h>
#include "fll.h"

/**********************************************************************
初始化
**********************************************************************/
void FllInit()
{
   SCFQCTL=SCFQ_1M;             //N=31,调制器使能
   SCFI0=FLLD_2;                //D=2, 频率调整范围为：0.65～6.1MHz
   FLL_CTL0=DCOPLUS;            //DCO+有效：FDCOCLK=D*(N+1)*晶振频率
}

general.h
#ifndef __GENERAL
#define __GENERAL

#define FREQUENCY   8000        //MCLK 的频率 8000kHz
#define LOOPBODY    8           //MSP430 中一个 for 循环体耗费 8 个周期
#define LOOPCNT  (unsigned int)((FREQUENCY/LOOPBODY))

void DelayMs(unsigned int ms);
#endif

general.c
#include "general.h"
/**********************************************************************
延时
ms：延时的毫秒数
**********************************************************************/
void DelayMs(unsigned int ms)
{
   unsigned int iq0, iq1;
   for (iq0=ms; iq0>0; iq0--)
   for (iq1=LOOPCNT; iq1>0; iq1--)
```

```
    ;
}

main.c
#include  <msp430x42x.h>
#include "lcd.h"
#include "ping.h"
#include "general.h"

void InitSys();

unsigned char ShowMem[4]={16,17,11,9};     //显示“9b-H”
unsigned char ShowDian=0;
int main()
{
    unsigned char q0=0;
    WDTCTL=WDTPW+WDTHOLD;                   //关闭看门狗
    InitSys();
    Show(ShowMem,ShowDian);
start:
    DelayMs(500);       //延时
    if(q0==0)
        q0=1;
    else
        q0=0;
    ToLcdDian(ShowDian,q0);                 //小数点闪烁显示
    LcdBlink(q0);                           //全体闪烁显示
    goto start;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    PingInit();                             //初始化液晶屏
    LcdGo(0);                               //打开液晶模块
}
```

第11章　AD、DA 转换

微处理器处理的是数字量，而现实世界中的量值为模拟量，因此，无论是信号进入微处理器还是将微处理器的处理结果输出，都避免不了模拟量和数字量之间的相互转换。将模拟量转换为数字量称为模数转换，简称 AD 转换。将数字量转换为模拟量称为数模转换，简称 DA 转换。

MSP430 系列单片机内的 AD 转换器有 ADC10、ADC12、ADC14，转换精度分别为 10 位、12 位、14 位，具体集成的是哪一种转换器可以查阅芯片手册。11.1 节介绍 ADC12 的使用例程。

DA 转换器目前只存在于 15x、16x 系列中，称为 DAC12。11.2 节介绍 DAC12 的使用例程。

11.1　ADC12

ADC12 的寄存器比较多，功能比较复杂，分为 5 个功能模块：

（1）带有采样/保持功能的 ADC 内核。

（2）可控制的转换存储。

（3）可控制的参考电平发生器。

（4）可控制和选择的时钟源。

（5）可控制的采样及转换时序电路。

ADC12 的主要特性为：

（1）最大采样速率为 200 千次/秒。

（2）转换精度为 12 位，1 位差分非线性（DNL），1 位积分非线性（INL）。

（3）内装采样/保持电路，可选择软件、采样定时器或片内定时器来控制采样周期。

（4）内嵌专用的 RC 振荡器，用来产生采样时序。

（5）内嵌用于温度测量的热敏二极管。

（6）拥有 8 个可配置的外部模拟信号采样通道。

（7）拥有 4 个内部通道，用于 Vcc 电压值、温度、外部正负电压参考的测量。

（8）内部参考电压为 1.5V 或 2.5V，可由软件选择。

（9）可以为每个通道的正负参考电压选择内部或外部的电压源。

（10）可以选择转换时钟源。

（11）具有 4 种转换方式：单通道单次，单通道多次，多通道单次，多通道多次。

（12）拥有 16 个保存转换结果的寄存器，可以由软件进行独立访问和配置通道以及参考电压。

（13）ADC 内核与参考电压发生器可分别进入低功耗模式。

进行 AD 转换通常需要设置的内容有：转换通道、采样保持、参考电压、转换时钟、转换模式、结果缓存。

程序 11-1 的功能为进行单通道单次转换，使用内部热敏二极管测量温度。进行多通道单次转换，测量温度和$(AV_{CC}-AV_{SS})/2$。

程序 11-1 的工作环境为：CPU 为 MSP430F149，MCLK 选择 8MHz，ACLK 选择 32.768kHz。

Adc12.c 为 ADC12 的驱动程序。包括函数 Adc12Init、Adc12Open、AdcDo、SetTongDao、Adc12Go、Adc、AdcGet。

函数 Adc12Init 的功能为对 ADC12 进行初始化。控制寄存器 ADC12CTL0 中的 ENC 位为转换允许位，置位时允许 AD 转换，复位时不允许 AD 转换。控制寄存器 ADC12CTL0 和 ADC12CTL1 中的一些位和存储寄存器 ADCMCTLx 的所有位必须在 ENC 位复位时才能够修改。初始化的过程为，首先使 ENC 位复位，然后设置转换的启动方式，打开参考电压，选择参考电压为 2.5V，选择采样保持器的采样时间。

函数 Adc12Open 的功能为打开或关闭 ADC12。寄存器 ADC12CTL0 中的 ADC12ON 复位时关闭 ADC12 的内核，置位时打开 ADC12 的内核。内核关闭时无法进行 AD 转换。

函数 AdcDo 的功能为设置 AD 转换所使用的模式和 AD 转换的首地址。对于转换地址，寄存器 ADC12CTL1 中的 CSStarAdd 位分别对应转换存储器 ADC12MEM0～ADC12MEM15，取值范围为 0～15。每个 ADC12MEMx 寄存器对应一个转换控制寄存器 ADC12MCTLx。ADC12MCTLx 寄存器确定转换的采样通道和参考电压。通过参数 adr 可以选择 0～15 中任意一个转换存储器开始转换，每一个转换存储器都可以对应任意一个采样通道。这样，可以很灵活地选择要转换的通道并组织转换的顺序。参数 adr 对应 CSStarAdd 值，取值范围为 0～15，CSStarAdd 在寄存器 ADC12CTL1 的 12～15 位，所以写入寄存器 ADC12CTL1 时 adr 需要左移 12 位。参数 mod 对应转换模式 CONSEQ，所以写入寄存器 ADC12CTL1 时需要左移 1 位。因为所修改的寄存器 ADC12CTL1 中的位必须在 ENC 复位的情况下才有效，所以，本函数要在 ADC12 关闭的情况下使用。

函数 SetTongDao 的功能为设置转换通道。有关通道的设置是针对某一个转换存储器，需要设置以下内容：

- 选择转换通道
- 选择的通道是否是转换序列的最后一个通道
- 选择转换参考电压
- 转换完毕后是否触发中断

使用指针 pmem_ctl 来完成对 ADC12MCTLx 的操作。首先使指针变量 pmem_ctl 指向 ADC12MCTL（msp430x14x.h 中定义 ADC12MCTLx 的基地址）。参数 mem 的取值范围为 0～15，为转换存储器控制寄存器的号码。pmem_ctl 加上 mem，所指向的即为要操作的寄存器 ADC12MCTLx。在本模块中，应当至少使转换序列的最后一个通道转换完成时触发中断，以便置位等待转换结束的标志变量 Wait，Wait 为全局变量。

函数 Adc12Go 的功能为使用软件方式启动 AD 转换。某些情况下，AD 转换不需要由外部信号启动转换，而是由程序直接启动。全局数组 AdMem 用来存储转换结果，在启动 AD 转换前，将其元素全部赋值为 0，并复位转换完成标志 wait。启动转换的方法为将 ADC12CTL0 中的 ENC 位置位，从而允许 AD 转换。将寄存器 ADC12CTL0 中的 ADC12SC 位置位，ADC12 即开始采样，AD 转换开始。等待标志位在 AD 转换中断程序中被置位，表明转换结束。

函数 Adc 为 AD 转换的中断函数。本程序中针对两种情况进行编译：（1）单通道单次转换或者序列通道单次转换；（2）单通道多次转换或者序列通道多次转换。通过定义宏变量 DUO_CI，从而在编译的时候确定按照哪一种方式编译程序，这与程序中的分支不同。当 DUO_CI 为 0 时，按照情况（1）进行编译，DUO_CI 为 1 时，按照情况（2）进行编译。情况（1）将 16 个转换存储器中的值全部复制到 AdMem 中，并将转换结束标志变量 Wait 置位。情况（2）每次中断将测量的结果分别累加，当累加次数达到 AD_CI 时，对累加结果进行平均。最后，将转换结束标志变量 Wait 置位。程序中对这两种情况的处理只是一种范例，多数情况下可能不会用到所有的通道，读者可以自行修改。

函数 AdcGet 的功能为读出 AdMem 数组中保存的转换结果。将转换结果复制到指针 padc 所指向的数组中。

main.c 是主程序。首先进行单通道单次转换测量温度，使用内部热敏二极管测温度的模

拟通道为 10，测到的值通过一个算法转换为实际的温度值。接着进行序列通道单次转换，测量温度和(AVcc-AVss)/2，测量(AVcc-AVss)/2 的模拟通道为 11。

wendu.c 根据测量值计算华氏和摄氏温度。

程序 11-1 的结构比较简单，结构图省略。

程序 11-1：

```
adc12.h
#ifndef __ADC12
#define __ADC12

void Adc12Init();
void Adc12Open(unsigned char doit);
void AdcDo(unsigned int adr,unsigned char mod);
void SetTongDao(unsigned char tongdao,unsigned char eos,unsigned char mem,
unsigned char verf,unsigned char inter);
void Adc12Go();
void AdcGet(unsigned int *padc);
#endif

adc12.c
#include <msp430x14x.h>
#include "adc12.h"

unsigned char Wait=0;          //等待转换结束的标志位  0：没有结束；  1：结束
unsigned int AdMem[16];        //转换结果缓冲区
#define AD_CI 10
unsigned char AdCi=AD_CI;      //单通道或者序列多次转换的转换次数
#define DUO_CI  0          //0：单通道或者序列单次转换；1：单通道或者序列多次转换
/*********************************************************************
初始化
*********************************************************************/
void Adc12Init()
{
   ADC12CTL0 &= ~ENC;                //使 AD 模块处于初始状态
   //使用内部 2.5V 参考电压，使用采样保持器。
   ADC12CTL0=MSC+REFON+REF2_5V+SHT0_15+SHT1_15;
}

/*********************************************************************
```

```
打开或关闭ADC12模块
doit：  0：打开；   100：关闭
***********************************************************************/
void Adc12Open(unsigned char doit)
{
    if(doit==0)
    {
        ADC12CTL0 &= ～ENC;
        ADC12CTL0 |= ADC12ON;
        ADC12CTL0 |= ENC;            //允许转换
    }
    else if(doit==100)
    {
        ADC12CTL0 &= ～ENC;
        ADC12CTL0 &= ～ADC12ON;          //不允许转换
    }
}

/**********************************************************************
设置转换模式
adr：转换的首地址。取值：0～15
mod：转换模式。0：单通道单次；1：序列通道单次；2：单通道多次；3：序列通道多次
***********************************************************************/
void AdcDo(unsigned int adr,unsigned char mod)
{
    ADC12CTL1=(adr<<12)+SHP+(mod<<1);   //SHP表示由采样定时器控制采样
}

/**********************************************************************
设置通道
tongdao：选择的模拟输入通道，取值：0～15
eos：为0时序列没有结束；     为0x80时在序列转换时，表示本次转换结束后，本转换序列结束。
mem： 转换存储器控制寄存器，取值：0～15
verf：参考电压 0～7
inter：0：本通道转换完后不触发中断；  1：本通道转换完后触发中断
***********************************************************************/
void SetTongDao(unsigned char tongdao,unsigned char eos,unsigned char mem,
unsigned char verf,unsigned char inter)
{
```

```
    char *pmem_ctl=ADC12MCTL;
    pmem_ctl += mem;
    *pmem_ctl=tongdao+eos+(verf<<4);
    if(inter==0)
        ADC12IE &= ~(0x1<<mem);
    else
        ADC12IE |= (0x1<<mem);
}

/**********************************************************************
软件触发转换开始
**********************************************************************/
void Adc12Go()
{
    unsigned char q0;

    for(q0=0;q0<16;q0++)
        AdMem[q0]=0;
    Wait=0;
    ADC12CTL0 |= ENC+ADC12SC;            //转换开始
    while(Wait==0)                       //等待转换结束
    ;
}

/**********************************************************************
AD 转换器中断函数
**********************************************************************/
#pragma vector=ADC_VECTOR
__interrupt void Adc()
{
    #if DUO_CI==0                        //单通道或者序列单次转换
    unsigned char q0;
    int *pmem=ADC12MEM;

    Wait=1;                              //转换结束的标志位置位
    for(q0=0;q0<16;q0++)
    {
        AdMem[q0]= *pmem;
        pmem++;
```

```
    }
    #elif DUO_CI==1                    //单通道或者序列多次转换
    unsigned char q0;
    unsigned int iq0;
    int *pmem=ADC12MEM;

    if(Wait==0)
    {
        for(q0=0;q0<16;q0++)
        {
            AdMem[q0] += *pmem;
            pmem++;
        }

        AdCi--;
        if(AdCi==0)
        {
            Adc12Open(100);            //关闭 ADC12 模块
            Wait=1;                    //转换结束的标志位置位
            for(q0=0;q0<16;q0++)
            {
                  AdMem[q0]=AdMem[q0]/AD_CI;
            }
            AdCi=AD_CI;
        }
    }
    else
    {
        for(q0=0;q0<16;q0++)
        {
            iq0 += *pmem;
            pmem++;
        }
    }
    #endif
}

/**********************************************************************
```

```
从 AdMem 中读出转换结果，本函数读出了 AdMem 所有的 16 个寄存器，可根据
需要进行修改，不读出所有的寄存器。
padc：读出的结果保存在数组中，padc 为指向此数组的指针
***********************************************************************/
void AdcGet(unsigned int *padc)
{
   unsigned char q0;
   for(q0=0;q0<16;q0++)
   {
      *padc=AdMem[q0];
      padc++;
   }
}

wendu.h
#ifndef __WENDU
#define __WENDU
unsigned int SuanHuaShi(unsigned int adresult);
unsigned int SuanSheShi(unsigned int adresult);
#endif

wendu.c
#include "wendu.h"
/***********************************************************************
 计算华氏温度
***********************************************************************/
unsigned int SuanHuaShi(unsigned int adresult)
{
      return ((((((long)adresult-1615)*704)/4095)* 9/5)+32);
}

/************************************************************************
 计算摄氏温度
************************************************************************/
unsigned int SuanSheShi(unsigned int adresult)
{
      return ((((long)adresult-1615)*704)/4095);
}
```

```
main.c
#include <msp430x14x.h>
#include "adc12.h"
#include "wendu.h"

void InitSys();

int main()
{
   unsigned int ad_mem[16],wen_h,wen_s,iq0;
   unsigned char q0;
   float fq0;
   int *pmem=ADC12MEM;
   WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
   InitSys();

   //初始化寄存器
   for(q0=0;q0<16;q0++)
   {
      ad_mem[q0]=0xFF;
      *pmem=0xFF;
      pmem++;
   }

   //单通道单次转换
   AdcDo(0,0);                      //转换首地址为 0，单通道单次转换
   SetTongDao(10,0,0,1,1);          //模拟通道为 10，测温度
   Adc12Open(0);                    //打开 ADC12 模块
   Adc12Go();                       //转换开始
   Adc12Open(100);                  //关闭 ADC12 模块
   AdcGet(ad_mem);                  //读转换值
   wen_h=SuanHuaShi(ad_mem[0]);     //计算华氏温度
   wen_s=SuanSheShi(ad_mem[0]);     //计算摄氏温度

   //序列通道单次转换
   AdcDo(3,1);                      //转换首地址为 3
   SetTongDao(10,0,3,1,0);          //模拟通道为 10，测温度，不触发中断
   SetTongDao(11,0x80,4,1,1);       //模拟通道为 11，测(AVcc-AVss)/2，触发中断
```

```
    Adc12Open(0);                      //打开 ADC12 模块
    Adc12Go();                         //转换开始
    Adc12Open(100);                    //关闭 ADC12 模块
    AdcGet(ad_mem);                    //读转换值
    wen_h=SuanHuaShi(ad_mem[3]);       //计算华氏温度
    wen_s=SuanSheShi(ad_mem[3]);       //计算摄氏温度
    fq0=2.5*ad_mem[4]/4095;            //(AVcc-AVss)/2
    _NOP();                            //空操作
    LPM3;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
    Adc12Init();                   //初始化 AD 转换模块
    _EINT();                       //打开全局中断控制，若不需要打开，可以屏蔽本句
}
```

程序 11-2 的功能为进行单通道单次转换和多通道多次转换。编译时需要将宏定义变量 DUO_CI 修改为 1。其工作环境与程序 11-1 相同。程序的其他部分与程序 11-1 相同，不再重复。

程序 11-2：

```
main_m.c
#include <msp430x14x.h>
#include "adc12.h"
#include "wendu.h"

void InitSys();

int main()
{
    unsigned int ad_mem[16],wen_h,wen_s;
    unsigned char q0;
    float fq0;
    int *pmem=ADC12MEM;
    WDTCTL=WDTPW+WDTHOLD;              //关闭看门狗
    InitSys();
```

```
    //初始化寄存器
    for(q0=0;q0<16;q0++)
    {
        ad_mem[q0]=0xFF;
        *pmem=0xFF;
        pmem++;
    }

    //单通道多次转换
    AdcDo(5,2);                      //转换首地址为 5，单通道多次转换
    SetTongDao(10,0,5,1,1);          //模拟通道为 10，测温度
    Adc12Open(0);                    //打开 ADC12 模块
    Adc12Go();                       //转换开始
    AdcGet(ad_mem);                  //读转换值
    wen_h=SuanHuaShi(ad_mem[5]);     //计算华氏温度
    wen_s=SuanSheShi(ad_mem[5]);     //计算摄氏温度
    //多通道多次转换
    AdcDo(5,3);                      //转换首地址为 5，单通道多次转换
    SetTongDao(10,0,5,1,0);          //模拟通道为 10，测温度
    SetTongDao(11,0x80,6,1,1);       //模拟通道为 11，测(AVcc-AVss)/2，触发中断
    Adc12Open(0);                    //打开 ADC12 模块
    Adc12Go();                       //转换开始
    AdcGet(ad_mem);                  //读转换值
    wen_h=SuanHuaShi(ad_mem[5]);     //计算华氏温度
    wen_s=SuanSheShi(ad_mem[5]);     //计算摄氏温度
    fq0=2.5*ad_mem[6]/4095;          //(AVcc-AVss)/2
    _NOP();
    LPM3;
}

/************************************************************************
系统初始化
************************************************************************/
void InitSys()
{
    Adc12Init();     //初始化 AD 转换模块
    _EINT();         //打开全局中断控制，若不需要打开，可以屏蔽本句
}
```

11.2 DAC12

目前只有 MSP430F15/16x 系列有 DA 转换模块。MSP430 的 DA 转换模块是 12 位、电压输出的数模转换模块，称作 DAC12。

DAC12 的主要性能为：

（1）软件可选择 8 位、12 位输出。

（2）软件可选择转换速度和能量消耗的级别。

（3）软件可选择内部或者外部参考电压。

（4）支持有符号和无符号数据输入。

（5）具有自校准功能。

（6）多路 DA 转换器同步更新。

（7）可与 DMA 配合使用。

DAC12 的工作过程通常分为以下 4 步：

（1）设置 DAC12 的工作模式　DAC12 的工作模式比较简单。其中比较重要的一项为选择参考电压。参考电压是唯一影响 DAC12 输出的模拟量，参考电压的精度与稳定性直接影响 DA 转换的结果。可以选择内部或者外部的参考电压源。DAC12 的内部参考电压源来自 ADC12 模块。选择内部参考电压时，需要在 ADC12 模块中设置 V_{ref+}是 1.5V 或者 2.5V，然后 DAC12 通过寄存器 DAC12_Xctl 中的 DAC12SREFx 位选择参考电压源为 V_{ref+}。

（2）校准 DAC12 的输出　DAC12 存在一定的偏移误差，意思是当 DAC12 输入为 0 时，输出却不为 0，而是有一个小的误差，这个误差的值既有可能为正也有可能为负，因此，DAC12 的输出需要校准。首先要设置寄存器 DAC12_xCTL 中的 DAC12AMPx 位，其作用是选择输入和输出的稳定时间以及电流消耗。然后将 DAC12_xCTL 中的 DAC12CALON 位置位，DAC12 开始校准。校准完毕时 DAC12CALON 位自动复位。

（3）启动 DAC12　DAC12 的启动与停止由寄存器 DAC12_xCTL 中的 DAC12ENC 位控制。DAC12ENC 位置位时，DAC12 启动工作，复位时，DAC12 停止工作。有些工作模式位必须在 DAC12 停止工作时更改。

（4）更新 DAC12 的输出　将要输出的数据写入寄存器 DAC12_xDAT。当输出刷新的信号出现时，由硬件自动按照寄存器 DAC12_xDAT 中的数据进行输出。输出刷新信号共有 4 种，选择何种输出刷新信号由寄存器 DAC12_Xctl 的 DAC12LSELx 位确定。

程序 11-3 的功能为，从 DAC 通道 0（P6.6）输出锯齿波，从 DAC 通道 1（P6.7）输出正弦波。

程序 11-3 的工作环境为，CPU 选择 MSP430F16x，MCLK 为 8MHz，ACLK 为 32.768kHz。

文件 dac12.c 为 DAC12 的模块驱动程序，包括函数 InitDac12、GoDac12、Dac12Write。

函数 InitDac12 为 DAC12 的初始化函数。首先打开内部参考电压源，使用的参考电压为 2.5V。接着设置通道 0 和通道 1 的工作模式，并进行输出校准。

函数 GoDac12 的功能为打开或者关闭 DA 转换。

函数 Dac12Write 的功能为刷新 DAC12 的输出。本程序的输出刷新信号设置为只要寄存器 DAC12_xDAT 的值被更改，就立即刷新 DA 转换的输出。

文件 df_timera.c 为定时器 A 的驱动程序。其功能是产生定时中断，在中断程序中调用函数 Dac12Write 来修改 DAC 通道 0 和通道 1 的输出值。有关定时器工作原理的说明可参见 7.1.1 节。

文件 main.c 是主文件。调用函数 InitSys 进行系统初始化。函数 InitSys 中对定时器 A 和 DAC12 进行了初始化。

程序 11-3：

```
dac12.h
#ifndef _DAC12
#define _DAC12
void InitDac12();
void GoDac12(unsigned char doit,unsigned char which);
void  Dac12Write(unsigned int dat,unsigned char which);
#endif

dac12.c
#include <msp430x16x.h>
#include "dac12.h"

/***********************************************************************
初始化 DAC12
***********************************************************************/
void InitDac12()
{
    ADC12CTL0=REF2_5V+REFON;         //打开内部参考电压源，参考电压为 2.5V
    //设置 DAC12 通道 0
//满量程为参考电压，中速度/电流输出，12 位分辨率，Vref+为参考电压，立即刷新输出
    DAC12_0CTL=DAC12IR+DAC12AMP_5+DAC12ENC+DAC12LSEL_1 ;
    DAC12_0CTL |= DAC12CALON;        //自动校准 DA 输出
    //设置 DAC12 通道 1
//满量程为参考电压的 3 倍，中速度/电流输出，12 位分辨率，Vref+为参考电压，立即刷新输出
    DAC12_1CTL=DAC12AMP_5+DAC12ENC+DAC12LSEL_1 ;
    DAC12_1CTL |= DAC12CALON;        //自动校准 DA 输出
    //等待校准结束
    while(DAC12_0CTL & DAC12CALON==DAC12CALON);
    while(DAC12_1CTL & DAC12CALON==DAC12CALON);
}
```

```
/*********************************************************************
打开或关闭 DAC12
doit: 0: 关闭;   100: 运行
which: 打开或关闭。 0: DAC0;   1: DAC1
*********************************************************************/
void GoDac12(unsigned char doit,unsigned char which)
{
    if(doit==0)
    {
        //关闭 DAC12
        if(which==0)
            DAC12_0CTL &= ~DAC12ENC;
        else
            DAC12_1CTL &= ~DAC12ENC;
    }
    else if(doit==100)
    {
        //打开 DAC12
        if(which==0)
            DAC12_0CTL |= DAC12ENC;
        else
            DAC12_1CTL |= DAC12ENC;
    }
}

/*********************************************************************
写输出值
dat: 要输出的值，准备写入 DA 数据寄存器
which: 写入哪个通道。 0: DAC0;     1: DAC1
*********************************************************************/
void  Dac12Write(unsigned int dat,unsigned char which)
{
    if(which==0)
        DAC12_0DAT=dat;
    else
        DAC12_1DAT=dat;
}
```

```
df_timera.h
#ifndef __DF_TIMERA
#define __DF_TIMERA
void TimerAInit();
void SetTime(unsigned int ti);
void GotimeDfA(unsigned char doit);
#endif

df_timera.c
#include <MSP430x14x.h>
#include <math.h>
#include "df_timera.h"
#include "dac12.h"

unsigned int iOut0=0;
unsigned int iOut1=0;

#define TIME_10MS 328
/***********************************************************************
初始化
***********************************************************************/
void TimerAInit()
{
   TACTL=TASSEL_1+TACLR+MC_0;        //选择时钟源为 ACLK
   SetTime(TIME_10MS);               //设置定时时间
   TACCTL1=CCIE;                     //允许定时器中断
}

/***********************************************************************
设置定时时间
ti：要定时的时间，与时钟源的频率有关
***********************************************************************/
void SetTime(unsigned int ti)
{
   TACCR0=ti;                 //定时时间
}

/***********************************************************************
```

```
打开或关闭定时器
doit: 0: 停止; 100: 运行
*********************************************************************/
void GotimeDfA(unsigned char doit)
{
    if(doit==100)
    {
        TACTL |= MC_1+TACLR; //打开定时器
    }
    else if(doit==0)
    {
        TACTL &= ~MC0;       //关闭定时器
    }
}

/********************************************************************
定时器中断
*********************************************************************/
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1(void)
{
    unsigned int iq0;
    switch (__even_in_range(TAIV, 10))
    {
        case 2:
            //捕获/比较 1 中断
            //DAC 通道 0 输出锯齿波
            if(iOut0<0xFFF)
                iOut0++;
            else
                iOut0=0;
            Dac12Write(iOut0,0);
            //DAC 通道 1 输出正弦波
            if(iOut1<365)
                iOut1++;
            else
                iOut1=0;
            iq0=0xFFF*sin(iOut1);
```

```
          Dac12Write(iq0,1);
        break;
    }
    LPM3_EXIT;
}

main.c
#include <msp430x16x.h>
#include "dac12.h"
#include "df_timera.h"

void InitSys();

int main()
{
    WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
    InitSys();
    LPM3;
}

/**********************************************************************
系统初始化
***********************************************************************/
void InitSys()
{
     unsigned int iq0;

     //使用XT2振荡器
     BCSCTL1 &= ~XT2OFF;                       //打开XT2振荡器
     do
     {
        IFG1 &= ~OFIFG;                        //清除振荡器失效标志
        for (iq0=0xFF; iq0>0; iq0--);          //延时，等待XT2起振
     }
     while ((IFG1 & OFIFG)!= 0);               //判断XT2是否起振

    BCSCTL2=SELM_2+SELS;               //选择MCLK、SMCLK为XT2
       TimerAInit();                   //初始化定时器A
```

```
    InitDac12();                    //初始化 DAC12
    GotimeDfA(100);                 //打开定时器 A
    GoDac12(100,0);                 //打开 DAC12 通道 0
    GoDac12(100,1);                 //打开 DAC12 通道 1
    _EINT();
}
```

第12章 比较器A

MSP430系列中几乎所有的CPU都含有比较器A。比较器A的基本功能是进行模拟电压比较，并不复杂。但由于比较器是模拟电路中的一种基本电路，因此与其他外围元件和CPU内部模块相配合可以组成很多有实用价值的电路。

比较器A模块有两个输入端CA0和CA1，CA0和CA1分别接在比较器的正端和负端，具体的对应关系可由寄存器来控制。CA0和CA1可以来自外部信号输出，也可以来自内部基准电压。比较的结果输出到CAOUT、CCI1B和中断标志ACIFG。比较器A比较正端和负端的输入信号，如果正端>负端，则CAOUT=1，如果正端<负端，则CAOUT=0。CCI1B是定时器A的捕获事件信号输入端。比较结果输出给CCI1B这一特性很重要，它使得定时器可以直接捕获比较器的比较结果，很多应用设计都利用了这一特性。

本章12.1介绍如何利用定时器A实现AD转换，并给出例程。12.2介绍如何利用定时器A实现电阻值的测量，并给出例程。

12.1　斜边 AD 转换

很多系统中都会用到 AD 转换，将模拟量转换为数字量，供计算机进行处理。目前有些比较高档的微处理器中直接集成了 AD 转换器，如 MSP430 单片机中的 13、14、15、16 系列，但大部分低档的微处理器中则没有集成 AD 转换器。很多情况下都需要进行 AD 转换，一种方法是外接专用的 AD 转换芯片，但这种方法增加了成本和系统的体积以及功耗。对于 MSP430 单片机来说，还有另外一种方法，就是利用集成的比较器 A 进行 AD 转换。其外部电路非常简单，只需要一个电阻和一个电容，如图 12-1 所示，这种方式被称作斜边 AD 转换。

V_{in} 为输入的待测电压，通过 R1 和 R2 进行分压，分压后得到的电压值必须符合 MSP430 的指标要求，否则会损坏 MSP430。C 为充电电容，R_{ref} 为参考电阻。进行斜边 AD 转换的工作过程如下：

（1）将 CA0 的输入信号源设为内部参考电压 $0.25V_{cc}$，CA1 的输入信号源设为外部信号。定时器 A 工作在捕获模式，在上升沿捕获并触发中断，捕获信号来自 CCI1B。将端口 P1.0 设置为输出状态，输出高电平。通过 R_{ref} 为电容 C 充电。电容 C 充满电时，由于 CA1 端的电压值基本为 V_{cc}，大于 CA0 端的电压值，所以 CAOUT 输出 0。

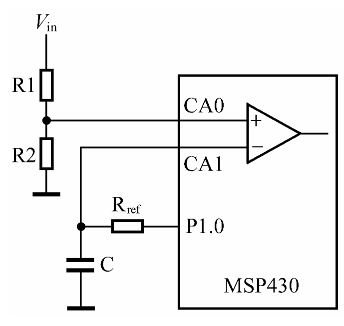


图 12-1　斜边 AD 转换电路

（2）读定时器 A 的计数值 time0。使 P1.0 输出低电平，通过电阻 R_{ref} 放电。当 CA1 端的电压下降至 $0.25V_{cc}$ 时，比较器的输出翻转，CAOUT 输出 1，信号 CCI1B 产生上升沿。触发定时器 A 捕获中断。在定时器 A 中断程序中读出捕获值 time1。计算通过电阻 R_{ref} 放电的时间：time_ref=time1−time0。

（3）将 CA0 的输入信号源设为外部信号。再次通过 R_{ref} 为电容 C 充电。电容充满电时，CA1 上的电压基本为 V_{cc}。为了保证斜边 AD 转换的进行，要求此时 CA0 输入的电压必须小于 CA1，以便使 CAOUT 输出 0。

（4）读定时器 A 的计数值 time0。再次通过 R_{ref} 放电，CA1 端电压逐渐下降，当下降至与 CA0 端输入的电压时，比较器的输出翻转，CAOUT 输出 1，信号 CCI1B 产生上升沿。触发定时器 A 捕获中断。在定时器 A 中断程序中读出捕获值 time1。计算通过电阻 R_{ref} 放电的时间：time_rse=time1−time0。

（5）计算 V_{in} 电压的公式为：

$$V_{in}=V_{CC}\times\frac{\mathrm{R1}+\mathrm{R2}}{\mathrm{R2}}\times e^{\frac{\text{time_ref}}{\text{time_rse}}\times\ln\frac{V_{CAREE}}{V_{cc}}}$$

其中 V_{CAREF} =$0.25V_{cc}$ 为第 1 步中所使用的内部参考电压。这里公式中求对数的部分可以直接计算出来，为 ln0.25=−1.3862944。

斜边 AD 转换方式的信号测量范围为：$V_{CAREF}<V_{CA0}<V_c$，V_c 为电容 C 充电所能够达到的最大电压值，可以认为等于 V_{cc}。考虑到电阻 R1、R2 分压的分压作用。测量范围为：

$$V_{CAREF}\times\frac{R1+R2}{R2}<V_{in}\leqslant V_{CC}\times\frac{R1+R2}{R2}$$

充电的时间与所要求的转换精度有关：5 τ 时的精度可以达到 1%，7 τ 时可以达到 0.1%，$\tau=R_{ref}\times C$。

工作过程中的第 1、2 步目的是测量 time_ref，这个值与被测的电压信号无关，只与 MSP430 工作的环境有关，如温度、电源电压、电容电阻的稳定程度。因此，不必每次测量，只有在工作环境发生变化的情况下，才需要重新测量。

程序 12-1 的功能为使用斜边 AD 转换的方法测量外部输入电压。

程序 12-1 的工作环境为 CPU 使用 MSP430F149，MCLK 选择 8MHz，ACLK 选择 32.768kHz。没有使用分压电阻 R1 和 R2，直接测量输入 CPU 的电压。电阻 R_{ref}=10K，电容 C=0.1μF。MSP430F149 本身内部具有 ADC12，实际不需要使用斜边 AD 转换，读者使用时可以自行更改，以适应其他型号的 CPU。

文件 comparator.c 是比较器 A 的驱动模块。其中有两个初始化函数 CompInitRef 和 CompInitVin。CompInitRef 在上述工作过程的第（1）步调用。CompInitVin 在上述工作过程的第（3）步调用。

文件 df_bh_timera.c 是定时器 A 的驱动程序。计数器的时钟源为 MCLK，为了延长计数时间，避免计数器很快溢出，进行了 8 分频，因此，计数频率为 1MHz。变量 StaBuHuo 为是否发生捕获事件的标志，在捕获中断中将其置位。通过调用函数 GetTime 读捕获的计数器值，函数 GetTime 的返回值表明捕获事件是否发生，如果没有发生，则读出的值是无效的。StaBuHuo 的值通过函数 ClearBuHuo 复位。

文件 slope.c 为进行斜边 AD 转换的模块。函数 CeRref 用来测量 time_ref。函数 Slope 首先测量 time_res，然后根据公式计算电压信号的值。

文件 main.c 为主程序。调用 CeRref 函数不在程序的主循环内，time_ref 只是在每次启动时测量一次。

程序 12-1 的结构比较简单，结构图省略。

程序 12-1：

```
comparator.h
#ifndef __COMPARATOR
#define __COMPARATOR
void CompInitRef();
void CompInitVin();
void CompOpen(unsigned char doit);
#endif

comparator.c
#include <msp430x14x.h>
#include "comparator.h"
```

```
#define CA_SEL P2SEL
#define CA0_IN BIT3          //将比较器输入端 CA0 定义到 P2.3
#define CA1_IN BIT4          //将比较器输入端 CA1 定义到 P2.4

/*********************************************************************
初始化为将内部参考源加到 CA0
*********************************************************************/
void CompInitRef()
{
   CACTL1=CAREF_1;           //内部参考源接正端，内部参考源为 0.25Vcc
   CACTL2=P2CA1;             //CA1 选择外部信号输入，CA0 不选择外部信号输入
   CAPD=CA0_IN+CA1_IN;       //关闭输入缓冲
}

/********************************************************************
初始化为将外部信号加到 CA0
********************************************************************/
void CompInitVin()
{
  CACTL1=0;
   CACTL2=P2CA1+P2CA0;       //CA0 选择外部信号输入，CA1 选择外部信号输入
   CAPD=CA0_IN+CA1_IN;       //关闭输入缓冲
}

/*********************************************************************
打开或关闭比较器模块
doit: 100：打开；   0：关闭
********************************************************************/
void CompOpen(unsigned char doit)
{
   if(doit==100)
   {
      CACTL1 |= CAON;
   }
   else if(doit==0)
   {
      CACTL1 &= ~CAON;
   }
```

```
}

df_bh_timera.h
#ifndef __DF_BH_TIMERA
#define __DF_BH_TIMERA

void InitBhTimerA();
void GoBhTimerA(unsigned char doit);
unsigned char GetTime(unsigned int *buhuo);
void ClearBuHuo();
#endif

df_bh_timera.c
#include <MSP430x14x.h>
#include "df_bh_timera.h"

#define FINISH 1
#define N_FINISH 0
unsigned char StaBuHuo=N_FINISH;   //捕获状态标志
/**********************************************************************
初始化
**********************************************************************/
void InitBhTimerA()
{
   TACTL=TASSEL_2+TACLR+ID_3;       //定时器A，时钟源：MCLK，连续计数模式，8分频
   CCTL1=CCIS_1+CM_1+CAP;           //选择上升沿捕获，CCI1B为信号源
}

/**********************************************************************
控制捕获运行或者停止，打开后进入休眠状态，等待捕获中断发生，再退出休眠状态。
doit：0：停止； 100：运行；  其他：什么都不做
**********************************************************************/
void GoBhTimerA(unsigned char doit)
{
   if(doit==0)
   {
      TACTL &= ~MC1;           //关闭计数器
      CCTL1 &= ~CCIE;          //关闭中断
   }
```

```
    else
    {
       TACCR1=0;
        TACTL |= MC_2+TACLR;     //打开计数器
        CCTL1 |= CCIE;           //捕获中断允许
    }
}

/*********************************************************************
定时器 A 中断函数
中断源：CC1
*********************************************************************/
#pragma vector=TIMERA1_VECTOR
__interrupt void TimerA1()
{
    switch (__even_in_range(TAIV, 10) )
    {
        case 2:
            //捕获/比较 1 中断
            StaBuHuo=FINISH;
        break;
    }
}

/*********************************************************************
读捕获状态和捕获值
buhuo：指向捕获值的指针
返回值：捕获状态。 0：捕获未完成； 1：捕获完成
*********************************************************************/
unsigned char GetTime(unsigned int *buhuo)
{
    unsigned int iq0=N_FINISH;
    if(StaBuHuo==FINISH)
    {
        iq0=FINISH;
    }
    *buhuo=CCR1;
    return iq0;
```

```
}

/*********************************************************************
清除捕获状态标志
*********************************************************************/
void ClearBuHuo()
{
    StaBuHuo=N_FINISH;
}

slope.h
#ifndef __SLOPE
#define __SLOPE
float Slope();
void CeRref();
#endif

slope.c
#include <msp430x14x.h>
#include <math.h>
#include "slope.h"
#include "general.h"
#include "comparator.h"
#include "df_bh_timera.h"

#define RDIR P1DIR
#define ROUT P1OUT
#define REF BIT0

#define REF_ZHI 10                   //参考电阻的阻值，单位为 kΩ
unsigned int time_ref;               //通过 time_ref 放电的时间
/*********************************************************************
斜边 AD 转换
返回值：电压值
*********************************************************************/
float Slope()
{
       float fvin;
```

```
    unsigned int time_rse,time0,time1;

    ROUT |= REF;                   //通过 REF 充电
    DelayMs(15);                   //延时 15ms
    ClearBuHuo();
    time0=TAR;
    ROUT &= ~REF;                  //通过 REF 放电
    while(GetTime(&time1)==0);  //读 REF 放电的时间
    time_rse=time1-time0;

    //计算电压值
    fvin=3.3*exp(((float)time_rse/time_ref)*(-1.3862944));
    return fvin;
}

/***********************************************************************
测量通过 Rref 放电的时间
***********************************************************************/
void CeRref()
{
    unsigned int time0,time1;
    //充电
    CompInitRef();                 //初始化比较器 A
    CompOpen(100);                 //打开比较器
    RDIR |= REF;                   //通过 REF 充电
    ROUT |= REF;
    DelayMs(15);                   //延时 15ms
    ClearBuHuo();
    time0=TAR;
    //测通过 REF 放电的时间
    ROUT &= ~REF;                  //通过 REF 放电
    while(GetTime(&time1)==0);  //读 REF 放电的时间
    time_ref=time1-time0;
    CompInitVin();                 //初始化比较器 A，准备进行 AD 转换
    CompOpen(100);                 //打开比较器
}

main.c
```

```
#include <msp430x14x.h>
#include "slope.h"
#include "df_bh_timera.h"
#include "comparator.h"

void InitSys();

int main()
{
    float fad;
    WDTCTL=WDTPW+WDTHOLD;         //关闭看门狗
    InitSys();
    CeRref();
st:
    fad=Slope();                  //斜边 ad 转换
    goto st;
}

/***********************************************************************
系统初始化
***********************************************************************/
void InitSys()
{
    unsigned int iq0;

    //使用 XT2 振荡器
   BCSCTL1 &= ~XT2OFF;            //打开 XT2 振荡器
   do
   {
       IFG1 &= ~OFIFG;                        //清除振荡器失效标志
       for (iq0=0xFF; iq0>0; iq0--);          //延时，等待 XT2 起振
    }
    while ((IFG1 & OFIFG)!= 0);               //判断 XT2 是否起振

    BCSCTL2=SELM_2+SELS;                      //选择 MCLK、SMCLK 为 XT2
    InitBhTimerA();                           //初始化定时器 A
    GoBhTimerA(100);                          //打开定时器捕获
    _EINT();
}
```

12.2　电阻值测量

在检测信号的时候，很多被测量都通过电阻值的形式表现出来，因此，经常会遇到需要检测电阻的情况。利用比较器 A，只需一个电阻和一个电容，就可以完成检测电阻值的功能，如图 12-2 所示。

R_{sens} 为待测的电阻。R_{ref} 为参考电阻。C 为充电电容。工作过程如下：

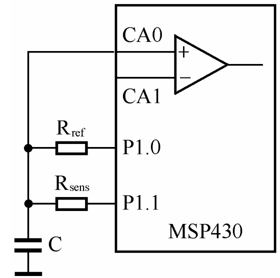


图 12-2　电阻检测电路

（1）将 CA0 的输入信号源设为外部信号，CA1 的输入信号源设为 $0.25V_{cc}$。定时器 A 工作在捕获模式，下降沿捕获并触发中断，捕获信号来自 CCI1B。将端口 P1.0、P1.1 设置为输入状态，由于端口的输入阻抗很大，所以，可以认为处于输入状态的端口为高阻状态。

（2）设置端口 P1.0 为输出状态，输出高电平，通过 R_{ref} 给电容 C 充电。充电完毕时，CA0 端的电压值为 V_{cc}，大于 CA1 端的电压值，因此，CAOUT 输出 1。

（3）读定时器 A 的计数值 time0。设置端口 P1.0 为输出状态，输出低电平，通过 R_{ref} 为电容 C 放电。当 CA0 上的电压降至 $0.25V_{cc}$ 时，比较器的输出翻转，CAOUT 输出 0，CCI1B 产生下降沿，触发定时器 A 捕获中断。在定时器 A 中断程序中读出捕获值 time1。计算通过电阻 R_{ref} 放电的时间：time_ref=time1−time0。

（4）设置端口 P1.0 为输出状态，输出高电平，通过电阻 R_{ref} 给电容 C 充电。充电完毕时，CA0 端的电压值为 V_{cc}，大于 CA1 端的电压值，因此，CAOUT 输出 1。

（5）读定时器 A 的计数值 time0。设置端口 P1.0 为输入状态，端口 P1.1 为输出状态，输出低电平，通过电阻 R_{sens} 为电容 C 放电。当 CA0 上的电压降至 $0.25V_{cc}$ 时，比较器的输出翻转，CAOUT 输出 0，产生下降沿，触发定时器 A 捕获中断。在定时器 A 中断程序中读出捕获值 time1。计算通过电阻 R_{sens} 放电的时间：time_sens=time1−time0。恢复端口 P1.1 为输入状态，为下一次测量做准备。

（6）计算电阻 R_{sens} 的公式为：

$$R_{sense} = R_{ref} \times \frac{\text{time_sens}}{\text{time_ref}}$$

充电的时间必须在 5τ 与 7τ 之间，并且影响测量精度，5τ 时测量精度为 1%，7τ 时为 0.1%，τ=R_{ref}×C。

程序 12-2 的功能为测量电阻 R_{sens} 的阻值。工作环境为，CPU 使用 MSP430F149，MCLK 选择 8MHz，ACLK 选择 32.768kHz。电阻 R_{ref}=10kΩ，电容 C=0.1μF。

文件 comparator.c 是比较器 A 的驱动模块。其工作过程与程序 12-1 中的 comparator.c 接近，可以参考程序 12-1 的相应解释。

文件 df_bh_timera.c 是定时器 A 的驱动程序。也可以参考程序 12-1 的相应解释，不同的

是程序 12-1 捕获的是 CCI1B 的上升沿，本程序捕获的是下降沿。

文件 zu.c 为测量电阻 R_{sens} 阻值的模块。函数首先测量 time_ref 和 time_sens，然后根据公式计算电阻值。

文件 main.c 为主程序，非常简单。主循环中只有对函数 MeasureR 的调用。

程序 12-2 的结构比较简单，程序结构图省略。

程序 12-2：

```
comparator.h
#ifndef __COMPARATOR
#define __COMPARATOR
void CompInit();
void CompOpen(unsigned char doit);
#endif

comparator.c
#include  <msp430x14x.h>
#include "comparator.h"

#define CA_SEL P2SEL
#define CA0_IN BIT3             //将比较器输入端 CA0 定义到 P2.3
#define CA1_IN BIT4             //将比较器输入端 CA1 定义到 P2.4

/**********************************************************************
初始化
**********************************************************************/
void CompInit()
{
   CACTL1=CARSEL+CAREF_1;      //内部参考源接负端，内部参考源为 0.25Vcc
   CACTL2=P2CA0;               //CA0 选择外部信号输入，CA1 不选择外部信号输入
   CAPD=CA0_IN+CA1_IN;         //关闭输入缓冲
}

/**********************************************************************
打开或关闭比较器模块
doit： 100：打开；   0：关闭
**********************************************************************/
void CompOpen(unsigned char doit)
{
   if(doit==100)
   {
```

```
        CACTL1 |= CAON;
    }
    else if(doit==0)
    {
        CACTL1 &= ~CAON;
    }
}

df_bh_timera.h
#ifndef __DF_BH_TIMERA
#define __DF_BH_TIMERA

void InitBhTimerA();
void GoBhTimerA(unsigned char doit);
unsigned char GetTime(unsigned int *buhuo);
void ClearBuHuo();

#endif

df_bh_timera.c
#include <MSP430x14x.h>
#include "df_bh_timera.h"

#define FINISH 1
#define N_FINISH 0
unsigned char StaBuHuo=N_FINISH;   //捕获状态标志
/***********************************************************************
初始化
***********************************************************************/
void InitBhTimerA()
{
    TACTL=TASSEL_2+TACLR+ID_3;      //定时器 A，时钟源：MCLK，连续计数模式，8 分频
    CCTL1=CCIS_1+CM_2+CAP;          //选择下降沿捕获，CCI1B 为信号源
}

/***********************************************************************
控制捕获运行或者停止，打开后进入休眠状态，等待捕获中断发生，再退出休眠状态
doit：0：停止；100：运行；其他：什么都不做
***********************************************************************/
void GoBhTimerA(unsigned char doit)
```

```
{
    if(doit==0)
    {
        TACTL &= ~MC1;          //关闭计数器
        CCTL1 &= ~CCIE;         //关闭中断
    }
    else
    {
        TACCR1=0;
        TACTL |= MC_2+TACLR;    //打开计数器
        CCTL1 |= CCIE;          //捕获中断允许
    }
}

/*********************************************************************
定时器A中断函数
中断源：CC1
*********************************************************************/
#pragma vector=TIMERA1_VECTOR
__interrupt void TimerA1()
{
    switch (__even_in_range(TAIV, 10) )
    {
        case 2:
            //捕获/比较1中断
            StaBuHuo=FINISH;
        break;
    }
}

/*********************************************************************
读捕获状态和捕获值
buhuo :指向捕获值的指针
返回值：捕获状态。 0：捕获未完成；  1：捕获完成
*********************************************************************/
unsigned char GetTime(unsigned int *buhuo)
{
    unsigned int iq0=N_FINISH;
    if(StaBuHuo==FINISH)
    {
```

```
        iq0=FINISH;
    }
    *buhuo=CCR1;
    return iq0;
}

/*********************************************************************
清除捕获状态标志
**********************************************************************/
void ClearBuHuo()
{
  StaBuHuo=N_FINISH;
}

zu.h
#ifndef __ZU
#define __ZU
float MeasureR();
#endif

zu.c
#include <msp430x14x.h>
#include "general.h"
#include "comparator.h"
#include "df_bh_timera.h"
#include "zu.h"

#define RDIR P1DIR
#define ROUT P1OUT
#define REF BIT0                //参考电阻接 P1.0
#define RSEN BIT1               //被测电阻接 P1.1

#define REF_ZHI 10              //参考电阻的阻值，单位为 kΩ

/*********************************************************************
测量电阻阻值
返回值：电阻值
*********************************************************************/
float MeasureR()
{
```

```
    unsigned int time_ref0,time_ref1,time_rsen0,time_rsen1,iq0,iq1;
    float fr;
    unsigned char t0,t1;

    //充电
    RDIR |= REF;                        //通过 REF 充电
    ROUT |= REF;
    DelayMs(15);                        //延时 15ms

    //测 REF 放电时间
    ClearBuHuo();
    time_ref0=TAR;
    ROUT &= ~REF;                       //通过 REF 放电
    while(GetTime(&time_ref1)==0);      //读 REF 放电的时间
    iq1=time_ref1-time_ref0;

    //充电
    RDIR |= REF;                        //通过 REF 充电
    ROUT |= REF;
    DelayMs(15);                        //延时 15ms

    //测 RSEN 放电时间
    ClearBuHuo();
    time_rsen0=TAR;
    RDIR &= ~REF;                       //停止通过 REF 充电
    ROUT &= ~RSEN;                      //通过 RSEN 放电
    RDIR |= RSEN;

    while(GetTime(&time_rsen1)==0);     //读 REF 放电的时间
    iq0=time_rsen1-time_rsen0;
    RDIR &= ~RSEN;                      //停止放电

    //计算电阻值
    fr=(float)REF_ZHI*iq0;
    fr=fr/iq1;
    return fr;
}

main.c
#include <msp430x14x.h>
```

```
#include "zu.h"
#include "df_bh_timera.h"
#include "comparator.h"

void InitSys();

int main()
{
   float fr;
   WDTCTL=WDTPW+WDTHOLD;            //关闭看门狗
   InitSys();
st:
   fr=MeasureR();                   //测量电阻值
   goto st;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
   unsigned int iq0;
   //使用 XT2 振荡器
    BCSCTL1 &= ~XT2OFF;         //打开 XT2 振荡器
   do
   {
      IFG1 &= ~OFIFG;                      //清除振荡器失效标志
      for (iq0=0xFF; iq0>0; iq0--);        //延时，等待 XT2 起振
   }
   while ((IFG1 & OFIFG)!= 0);             //判断 XT2 是否起振

   BCSCTL2=SELM_2+SELS;                    //选择 MCLK、SMCLK 为 XT2

   CompInit();                  //初始化比较器
   InitBhTimerA();              //初始化定时器 A
   CompOpen(100);               //打开比较器
   GoBhTimerA(100);             //打开定时器捕获
   _EINT();
}
```

第18章 特殊处理

本章讲述了两个问题：中断嵌套和程序异常处理。中断嵌套由于没有硬件的支持，因此尽量不要使用。但作为一种实现中断嵌套的方法，仍然给出，供读者参考。

程序异常处理实际上主要是对看门狗引起的复位进行处理。因为很多系统中都使用看门狗，所以，这里也给出一个处理看门狗的框架程序，供读者参考。

13.1 中 断 嵌 套

中断被触发时，MSP430 会将 SR 寄存器中的 GIE 位复位，GIE 为可屏蔽中断的全局控制位。当 GIE 位为 0 时，禁止响应中断，因此，中断触发的时候，新的中断就不会得到响应了，也就不会出现中断嵌套的情况。由于 CPU 的运算速度很快，可以很快处理完毕中断事件，然后响应新的中断，所以，不能实现中断嵌套的在大多数情况下对于完成应用目标没有影响。但在某些应用中，却需要进行中断嵌套。实现中断嵌套的方法是在中断程序中通过语句将 GIE 位置位。需要注意的是，编程者必须能够把握嵌套的层数，以防止嵌套层数过多而导致堆栈溢出，那样会导致程序执行混乱，出现无法预料的后果。

程序 13-1 的工作过程为，首次按下 KEY0 键，程序响应端口中断，置位 GIE，使静态变量 biaozhi 为 1，程序进入循环，直至变量 biaozhi 为 0 才能够退出循环。这样程序就无法从中断中返回。按下 KEY1 键，程序将再次响应中断，第二次进入中断程序，这样就实现了中断嵌套。第二次进入中断程序时，程序使变量 biaozhi 恢复为 0，然后从中断中返回。返回后，程序仍然执行首次进入中断后的循环语句，不过此时 biaozhi 为 0，所以可以退出循环。退出循环后，接着执行循环后面的语句，于是可以从中断中返回。

文件 key.c 的功能为处理按键，比较容易读懂，可以参考第 8 章的 I/O 端口一节。其中定义了宏 SET_GIE 来完成置位 GIE 的工作，所使用的是一个内部函数__bis_SR_register(GIE)，有关解释参见 3.3.2 节。

文件 main.c 为主程序，只是调用 InitSys 对系统进行初始化，然后进入低功耗模式。

程序 13-1：

```
key.h
#ifndef __KEY
#define __KEY
void InitKey();
#endif

key.c
#include <MSP430x14x.h>
#include "key.h"

//选用端口定义
#define KEYDIR P1DIR
#define KEYIES P1IES
#define KEYIE P1IE
#define KEYIN P1IN
#define KEYIFG P1IFG
```

```
//定义键盘管脚
#define KEY0 BIT5
#define KEY1 BIT6

#define SET_GIE  __bis_SR_register(GIE)//置位全局中断控制位
/**********************************************************************
初始化
**********************************************************************/
void InitKey()
{
   KEYDIR &= ~(KEY0+KEY1);     //设置端口为输入
   KEYIES |= KEY0+KEY1;        //设置下降沿中断
   KEYIE |= KEY0+KEY1;         //打开端口中断
}

/**********************************************************************
端口 1 中断函数
多中断中断源: P1IFG.0~P1IFG7
**********************************************************************/
#pragma vector=PORT1_VECTOR
__interrupt void Port1()
{
   static unsigned char biaozhi=0;
   if((P1IFG&BIT5)==BIT5)
   {
      //处理 P1IN.5 中断
      SET_GIE;
      P1IFG &= ~BIT5;          //清除中断标志
      biaozhi=1;
      while(biaozhi==1);
   }
   else if((P1IFG&BIT6)==BIT6)
   {
      //处理 P1IN.6 中断
      P1IFG &= ~BIT6;          //清除中断标志
      biaozhi=0;
   }
   else
   {
```

```
            P1IFG =0;
        }
        LPM3_EXIT;
    }

main.c
#include <msp430x14x.h>
#include "key.h"

void InitSys();

int main()
{
    WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
    InitSys();
    LPM3;
}

/*********************************************************************
系统初始化
*********************************************************************/
void InitSys()
{
     InitKey();
     _EINT();
}
```

13.2 程序异常处理

程序在执行过程中有时会发生意外的事件，使程序不能正常执行，严重时甚至会造成损失。程序首先应该尽量加强其容错能力，其次应该加强其对错误的处理能力，尽量避免或减少损失。MSP430 在硬件设计上对程序的几种重要错误进行监测，一旦发生异常，则触发复位中断，CPU 进行复位。但这样的复位发生时，触发复位中断的标志位不会被清除，所以可以通过判断标志位来获知错误的原因并进行处理。触发中断的错误有：看门狗和 FLASH 安全键值错误。

程序 13-2 的工作环境为，MCLK 选择 DCO。

程序 13-2 的工作过程为，系统复位后，首先关闭看门狗，等待初始化完毕再打开。函数 except 根据触发中断的标志位处理错误。本程序没有给出处理错误的代码，因为没有固定的

处理方式，必须根据具体情况确定。一般最简单的方式是：执行一段代码，这段代码可暂停系统的工作，给出报警信息，并使系统处于比较安全的状态，如在程序对电机进行控制的情况下，则使电机停止转动。比较复杂的方式则是从备份的数据中恢复中断前系统运行的状态，继续执行程序。

为了触发中断复位，在系统初始化函数 InitSys 中打开了看门狗，但并没有在程序主循环中刷新看门狗的定时器，因此，看门狗定时器的时间到时就会触发看门狗中断，引起程序复位。在函数 except 中处理引起复位中断的错误，本程序中只是将看门狗中断标志复位。

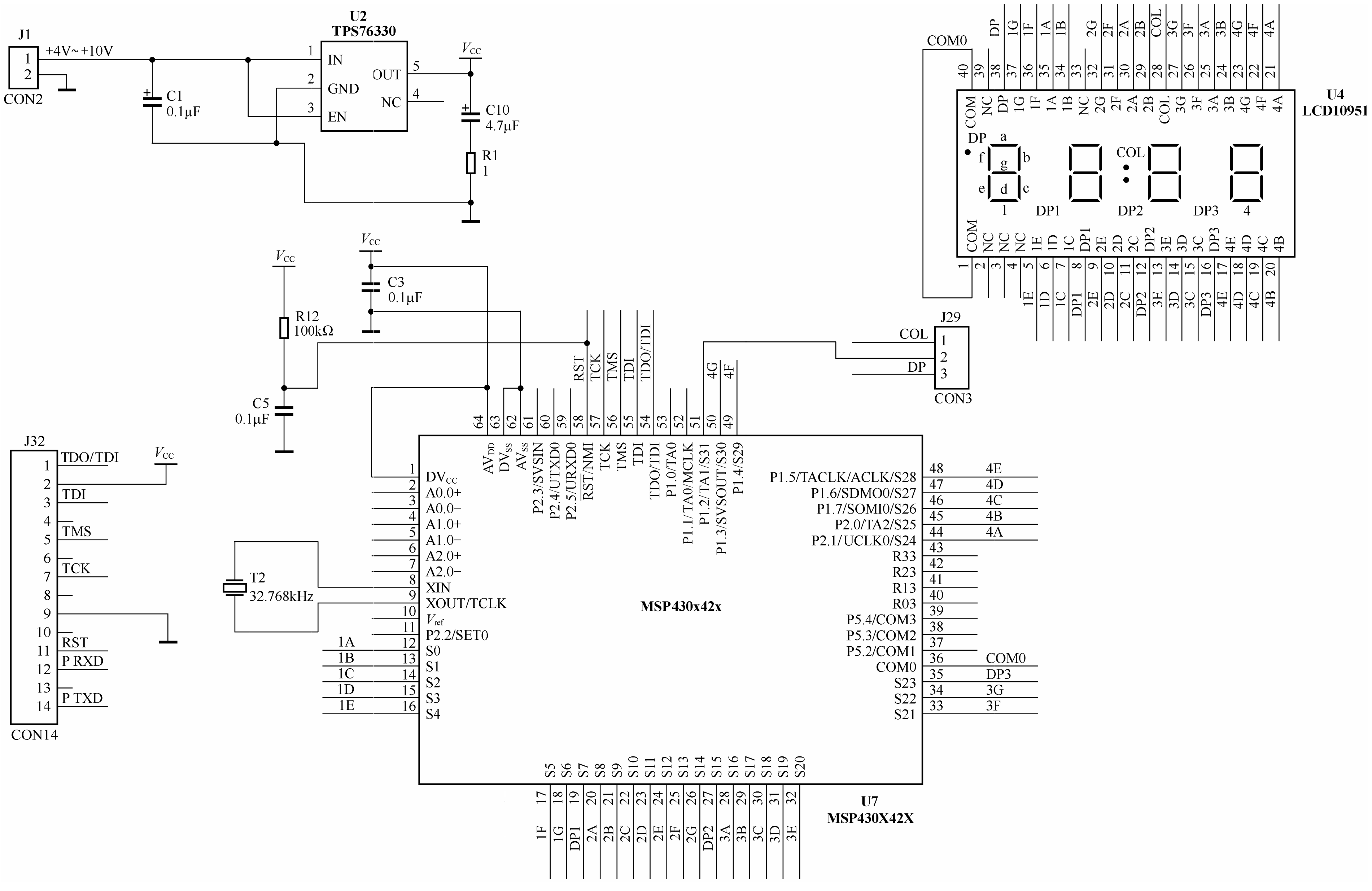
程序 13-2：

```
#include <msp430x14x.h>

void InitSys();
void except();

int main()
{
   WDTCTL=WDTPW+WDTHOLD;   //关闭看门狗
   except();
   InitSys();
   LPM4;
}

/**********************************************************************
系统初始化
**********************************************************************/
void InitSys()
{
   WDTCTL &= ~(WDTPW+WDTHOLD);    //打开看门狗
   _EINT();
}

/**********************************************************************
处理非正常复位
**********************************************************************/
void except()
{
   if(IFG1&WDTIFG==WDTIFG)
   {
      //看门狗中断或者安全键值错误，以下添加处理代码
```

```
            IFG1 &= ～WDTIFG;   //复位看门狗中断标志位
        }
        else if(FCTL3&KEYV==KEYV)
        {
            //FLASH 安全键值错误，以下添加处理代码

            FCTL3 &= ～KEYV;  //复位 FLASH 安全键值
        }
    }
```

附录 A　MSP430 基本电路图

J1 CON2 +4V~+10V
U2 TPS76030 IN GND EN OUT NC
C1 0.1μF
C10 4.7μF
R1 1Ω
R7 100kΩ
C2 0.1μF
T2 32.768kHz
J3 DB9
J2 CON14 TDO/TDI TDI TMS TCK RST
C3 0.1μF
C4 0.1μF
C5 0.1μF
C6 0.1μF
U4 MAX3232 R1 IN R2 IN T1 IN T2 IN C1+ C1− GND R1 OUT R2 OUT T1 OUT T2 OUT C2+ C2− $V+$ $V-$ V_{CC}
TXD0 RXD0
C12 0.1μF
C13 0.1μF
C14 12pF
C15 12pF
T1 8MHz X3 X4
U3 MSP430F13X/14X/15X/16X
MSP430x13x
MSP430x14x
MSP430x15x
MSP430x16x
DVcc P6.3/A3 P6.4/A4 P6.5/A5 P6.6/A6/DA0 P6.7/A7/DA1 V_{ref+} XIN XOUT/TCLK V_{eref+} V_{ref-}/V_{eref-} P1.0/TACLK P1.1/TA0 P1.2/TA1 P1.3/TA2 P1.4/SMCLK
P1.5/TA0 P1.6/TA1 P1.7/TA2 P2.0/ACLK P2.1/TAINCLK P2.2/CAOUT/TA0 P2.3/CA0/TA1 P2.4/CA1/TA2 P2.5/Rosc P2.6/ADC12CLK P2.7/TA0 P3.0/STE0 P3.1/SIMO0/SDA P3.2/SOMI0 P3.3/UCLK0/SCL P3.4/UTXD0
P3.5/URXD0 P3.6/UTXD1 P3.7/URXD1 P4.0/TB0 P4.1/TB1 P4.2/TB2 P4.3 P4.4 P4.5 P4.6 P4.7/TBCLK P5.0 P5.1 P5.2 P5.3 P5.4/MCLK
P5.5/SMCLK P5.6/ACLK P5.7/TBOUTH XT2OUT XT2IN TDO/TDI TDI TMS TCK RST/NMI P6.0/A0 P6.1/A1 P6.2/A2 AVss DVss AVcc
R8 33kΩ
R9 33kΩ
R10 33kΩ
S1 S2 S3
U7 24C02 A0 A1 A2 GND V_{CC} WP SCL SDA
R11 5.1kΩ
R12 5.1kΩ



J1
+4V~+10V
CON2
U2
TPS76330
IN
GND
EN
OUT
NC
C1
0.1μF
C10
4.7μF
R1
1
V_{CC}
U4
LCD10951
COM0
J29
COL
DP
CON3
R12
100kΩ
C3
0.1μF
C5
0.1μF
RST
TCK
TMS
TDI
TDO/TDI
J32
CON14
TDO/TDI
TDI
TMS
TCK
RST
P RXD
P TXD
T2
32.768kHz
MSP430x42x
U7
MSP430X42X