PIC单片机自学笔记

主　编　魏学海
副主编　陈义平　郑　爽

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS
CD-ROM INCLUDED

PIC 单片机自学笔记

主　编　魏学海
副主编　陈义平　郑　爽

北京航空航天大学出版社

内 容 简 介

本书以美国 Microchip 公司的 PIC16F877 单片机为主线，详细介绍其基本结构、工作原理及应用技术。全书共分 14 章，内容包括：集成开发环境、PIC 系列单片机的基本结构、存储器模块、PIC 指令系统及应用、I/O端口、同步串行通信、定时器、中断处理、A/D 转换以及应用实例等。

本书内容丰富而实用，通俗易懂，可作为高等工科院校相关专业的教材，也可供从事单片机开发应用的工程技术人员参考。

图书在版编目(CIP)数据

PIC 单片机自学笔记 / 魏学海主编. -- 北京 : 北京航空航天大学出版社，2011.2

ISBN 978-7-5124-0305-5

Ⅰ. ①P… Ⅱ. ①魏… Ⅲ. ①单片微型计算机 Ⅳ. ①TP368.1

中国版本图书馆 CIP 数据核字(2011)第 000033 号

PIC 单片机自学笔记

主　编　魏学海

副主编　陈义平　郑　爽

责任编辑　宋淑娟

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191)　http://www.buaapress.com.cn

发行部电话：(010)82317024　传真：(010)82328026

读者信箱：bhpress@263.net　邮购电话：(010)82316936

有限公司印装　各地书店经销

*

开本：787×960　1/16　印张：20.25　字数：454 千字

2011 年 2 月第 1 版　2011 年 2 月第 1 次印刷　印数：4 000 册

ISBN 978-7-5124-0305-5　定价：39.00 元(含光盘)

前　言

单片机是芯片级的小型计算机系统，可以嵌入到任何应用系统中，实现智能化控制。近20年来，8位单片机以其价格低、功耗低、指令简练、易于开发等优点，加上近几年嵌入式C语言的推广普及，指令执行速度的不断提升，片载Flash程序存储器及其在系统内编程ISP和在应用中编程IAP技术的广泛采纳，片内配置外设模块的不断增多，以及新型外围接口的不断扩充，使得单片机越来越受到广大电子工程师的欢迎。单片机的发展和性能的日益完善，开创了微控技术的新天地。现代控制理念的核心，就是嵌入式计算机应用系统。通过不断提高控制功能和拓展外围接口功能，使单片机成为最典型、最广泛、最普及的嵌入式微型控制单元。单片机拥有计算机的基本核心部件，将其嵌入到电子系统中，可以满足控制对象的要求，为电子系统的智能化奠定了基础。

单片机的实现方式比模拟控制思想更为简洁和方便得多；同时，可以跨越式地实现对外部模拟量的高速采集、逻辑分析处理以及对目标对象的智能控制。PIC系列单片机是美国Microchip(微芯)公司生产的单片机产品中的标志性产品。Microchip公司从10年前的默默无闻，到今天成为全世界8位单片机销量第一的公司，与其过硬的技术支持和设计完善的系统内核有着直接的关系。PIC系列单片机可以满足用户的各种需要。为了推广和普及PIC单片机的基础知识，提高系统开发及应用能力，特别是适应高校专业改革和教学内容更新的需要，近年来在美国Microchip公司卓有成效的推广之下，PIC单片机已逐渐为国内从事单片机开发应用的工程技术人员所认识和应用。在众多的PIC单片机家族成员中，PIC16F877型号具备在线调试功能和在线编程功能，并包含廉价的学习和开发工具套件。借助于这项独特的性能和优势，学习者可以边学边练、学用结合，既学习理论知识又掌握开发技能，而且在经济上还不需要投入太多。PIC系列单片机的硬件系统设计简洁，指令系统设计精练。在所有单片机品种中，PIC单片机具有性能完善、功能强大、学习容易、开发应用方便以及人机界面友好等突出优点。学好PIC单片机，掌握其核心技术内涵，拓展其应用范围，将具有划时代的意义。

本书以美国Microchip公司PIC16F877单片机为主线，详细介绍其基本组成、原理和实际应用。全书共分14章，第1、3、4、5、7、10、12、14章由魏学海编著，第2、11、13章由郑爽编著，第6、8、9章由陈义平编著。内容包括：第1章PIC单片机简介，主要介绍PIC单片机的硬件结构和开发所需的四件法宝；第2章PIC编译器的语法规则，对指令集系统及语法规则进行

分析和说明；第 3 章熟悉 PIC 开发环境，对 PIC 单片机 MPLAB 集成开发环境及使用方法进行详细介绍，并通过具体实例演示项目开发过程；第 4 章 I/O 端口实验，对 I/O 端口的基本功能及其内部结构、初始化设置进行介绍，并列举了很多 I/O 口应用开发实例；第 5 章按键及 B 口电平中断，介绍利用 PIC 单片机电平变化中断来实现一种新的按键扫描方法，并介绍电平变化中断在温度测量中的应用；第 6 章定时器/计数器的应用，重点讨论内部 2 个定时器/计数器的结构、配置及工作方式；第 7 章捕获/比较/脉宽调制 CCP 模块，介绍 CCP 模块的应用方法并给出具体实例；第 8 章 10 位模/数转换器模块，主要介绍 10 位 A/D 转换器的工作原理及其应用；第 9 章捕捉/比较/PWM(CCP)应用，2 个 CCP 模块与 TMR1 和 TMR2 配合，可实现捕捉外部输入脉冲，输出不同宽度的脉冲信号及输出脉冲宽度 PWM 调制；第 10 章休眠、看门狗和 EEPROM 应用，介绍看门狗、休眠和 EEPROM 的基本原理，并给出编程实例；第 11 章并行从动端口，介绍并行从动端口原理，并给出编程实例；第 12 章主从同步串行端口模块，介绍 SPI 总线和 I^2C 总线原理及应用实例；第 13 章通用同步/异步收发器，介绍 USART 异步模式；第 14 章GPS 应用实例，介绍当前比较流行的 GPS 的原理并给出开发实例。本书中各部分均给出了详细分析过程及参考程序，具有一定的实用价值。

本书内容丰富而实用，通俗而流畅，可作为高等工科院校相关专业的教材，也可供从事单片机开发应用的工程技术人员参考。

本书附带 1 张光盘，光盘中包含第 4～14 章详细的程序文件，程序文件均经过测试。光盘中的实例集以章节号命名，便于查询，供读者自学参考使用。

由于微芯公司不断推出新品，可查阅的中文新资料尚不十分丰富，需要撰写的内容不仅量大而且新颖，加之作者的水平有限，书中不妥之处敬请广大读友不吝赐教。

编　者

2010 年 9 月

目　录

第 1 章

PIC 单片机简介

首先恭喜大家找到了学习单片机的法宝。虽然我们学会了 51 单片机，但是距离嵌入式系统应用还有很大的差距。近年来随着信息技术的发展，嵌入式系统已经渗透到各个领域，如果现在不往嵌入式应用方向发展，今后会很难取得更大的成就。要想学好嵌入式系统的理论和应用，就必须先学好一款高级单片机，这里就推荐 PIC 系列单片机供大家学习参考。

1.1 PIC 单片机概述

由美国 Microchip 公司推出的 PIC 单片机系列产品，率先采用了精简指令集(RISC)结构的嵌入式微控制器，其高速度、低电压、低功耗、大电流 LCD 驱动能力和低价位 OTP 技术等都体现出单片机产业的新趋势。现在，PIC 系列单片机在世界单片机市场的份额排名中已逐年上升，尤其在8 位单片机市场上，据称已从 1990 年的第 20 位上升到目前的第 2 位。PIC 单片机从覆盖市场出发，已有 3 种(又称 3 层次)系列多个型号的产品问世，所以在全球都可以看到 PIC 单片机从计算机的外设、家电控制、电信通信、智能仪器、汽车电子到金融电子各个领域的广泛应用。现今的 PIC 单片机已经是世界上最有影响力的嵌入式微控制器之一。

据统计，我国的单片机年容量已达 1 亿～3 亿片，且每年以大约 16%的速度增长，但相对于世界市场，我国的占有率还不到 1%。这说明单片机应用在我国才刚刚起步，有着广阔的前景。因此，培养单片机应用人才，特别是在工程技术人员中普及单片机知识就更具有重要的现实意义。

当今单片机厂商繁多，产品性能各异。针对具体情况，应选择何种型号呢？首先，要弄清以下两个概念：集中指令集(CISC)和精简指令集(RISC)。采用 CISC 结构的单片机的数据线与指令线分时复用，即所谓冯・诺伊曼结构。它的指令丰富，功能较强；但取指令和取数据不能同时进行，速度受限，价格亦高。采用 RISC 结构的单片机的数据线与指令线分离，即所谓哈佛结构。它使得取指令和取数据可同时进行，且由于一般指令线宽于数据线，故使其指令较同类 CISC 单片机指令包含更多的处理信息，执行效率更高，速度亦更快。同时，这种单片机

指令多为单字节，程序存储器的空间利用率大大提高，有利于实现超小型化。属于 CISC 结构的单片机有 Intel 公司的 8051 系列、Motorola 公司的 M68HC 系列、Atmel 公司的 AT89 系列、中国台湾Winbond(华邦)公司的 W78 系列和荷兰 Philips 公司的 PCF80C51 系列等；属于 RISC 结构的单片机有 Microchip 公司的 PIC 系列、Zilog 公司的 Z86 系列、Atmel 公司的 AT90S 系列、韩国三星公司的 KS57C 系列 4 位单片机和中国台湾义隆公司的 EM—78 系列等。目前，两种结构的单片机共存，各有优势，CISC 单片机提供了更好的代码深度以及成熟的开发工具，而 RISC 单片机则有更高的时钟速度和广阔的市场前景。RISC 单片机在控制技术上不断完善，大有超过 CISC 单片机的趋势，是单片机发展的方向。

1.1.1 PIC 单片机的优势

自笔者开始进行单片机开发以来，不少朋友询问 PIC 单片机有哪些优势？现在就把笔者的使用心得与大家分享。

1. 高性能价格比

PIC 单片机不搞功能堆积，而从实际应用出发，设计各种类型和型号的单片机以适应不同场合的应用。PIC 单片机有 8 位、16 位和 32 位，每种类型又有很多型号供开发者选用。

2. 采用精简指令集

采用 RISC 结构，指令数量少，执行效率高。PIC 系列 8 位 CMOS 单片机具有独特的 RISC 结构，数据总线和指令总线分离的哈佛总线(Harvard)结构，使得指令具有单字长的特性，且允许指令码的位数多于 8 位的数据位数，这与传统的采用 CISC 结构的 8 位单片机相比，可以达到 2∶1 的代码压缩，速度提高 4 倍。

3. 优越的开发环境

PIC 在推出一款新型号的同时，也推出相应的仿真芯片，所有的开发系统都由专用的仿真芯片支持，实时性非常好。

4. 引脚具有瞬态抑制能力

PIC 单片机引脚可以直接驱动继电器，不需要加光电耦合器进行隔离，抗干扰能力强。

5. 安全保密性

PIC 以保密熔丝来保护代码，用户在烧入代码后熔断熔丝，别人再也无法读出，除非恢复熔丝。目前，PIC 采用熔丝深埋工艺，恢复熔丝的可能性极小。

6. 自带看门狗

看门狗不需要外接，提高了应用程序的可靠性。

7. 睡眠和低功耗模式

PIC 可以工作在睡眠和低功耗模式下，特别是在便携式设备中，满足电池供电场合应用。当然，具有这种方式的单片机很多，比较典型的是 MSP430，PIC 单片机虽然在低功耗方面无法与之比拟，但是也能满足一些低功耗场合的应用。

1.1.2 PIC 单片机的选型

面对那么多系列和那么多型号的 PIC 单片机，初学者应该选择哪一款比较合适呢？为了能与 51 单片机衔接，应该首先选择 8 位高档单片机进行学习。8 位单片机中也有很多系列，如 PIC10 MCU，PIC12 MCU，PIC16 MCU 和 PIC18 MCU，鉴于网上 PIC16 MCU 的资料丰富，且价格易接受，故推荐选择 PIC16 MCU 系列中的 PIC16F877 型号单片机进行学习。

PIC16F877 有 40 个引脚，3 种封装形式。这 3 种封装形式分别是 DIP，PLCC 和 QFP，各种封装图形如图 1－1～图 1－3 所示。

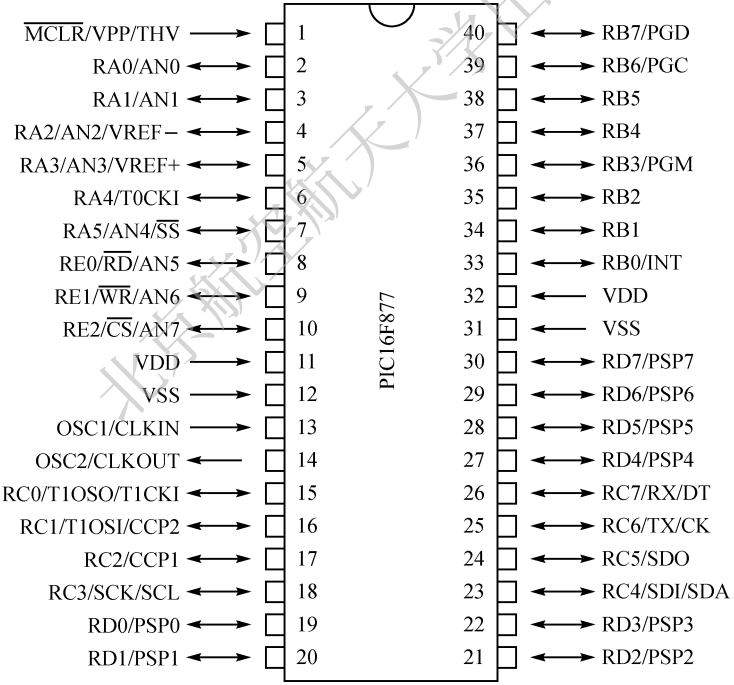


图 1－1　DIP 封装的 PIC16F877

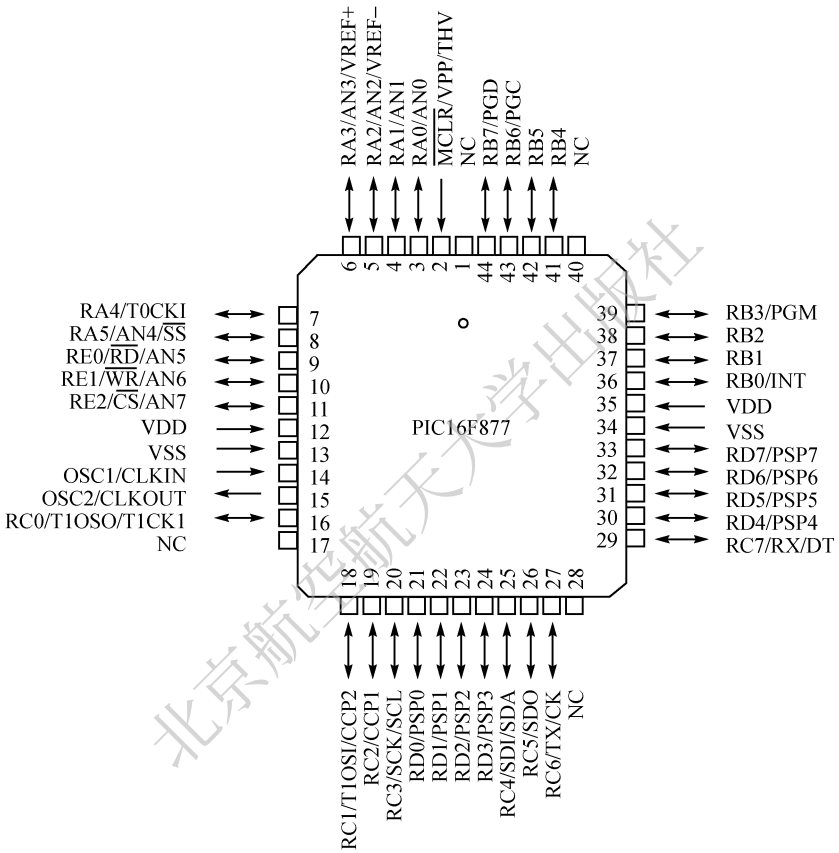


图 1－2　PLCC 封装的 PIC16F877

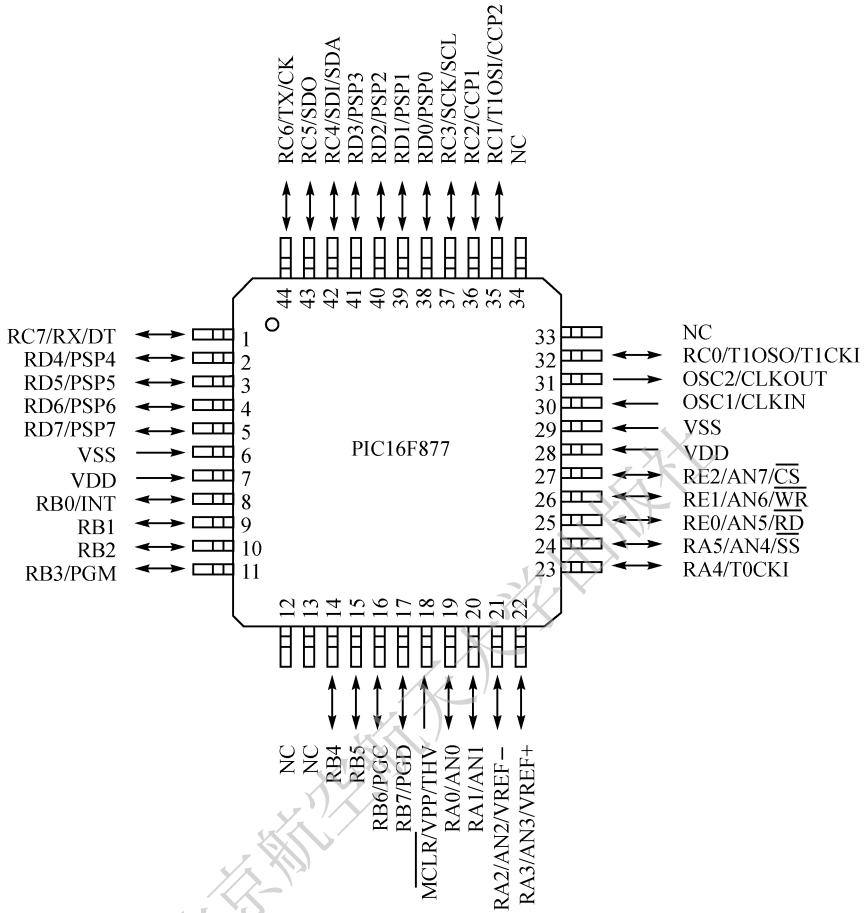


图 1-3　QFP 封装的 PIC16F877

1.2　硬件结构和引脚定义

想必大家通过 1.1 节的学习对 PIC16F877 这款单片机已有了初步的了解，为了能够更加顺利地开发单片机程序，有必要对其内部硬件结构和引脚定义进行探讨。

1.2.1　内部结构

从图 1-4 所示的结构框图可以看出，PIC16F877 采用了哈佛结构。下面介绍其内核特性：

◆ 高性能的精简指令集 CPU。

◆ 16 位字长的 35 条指令。

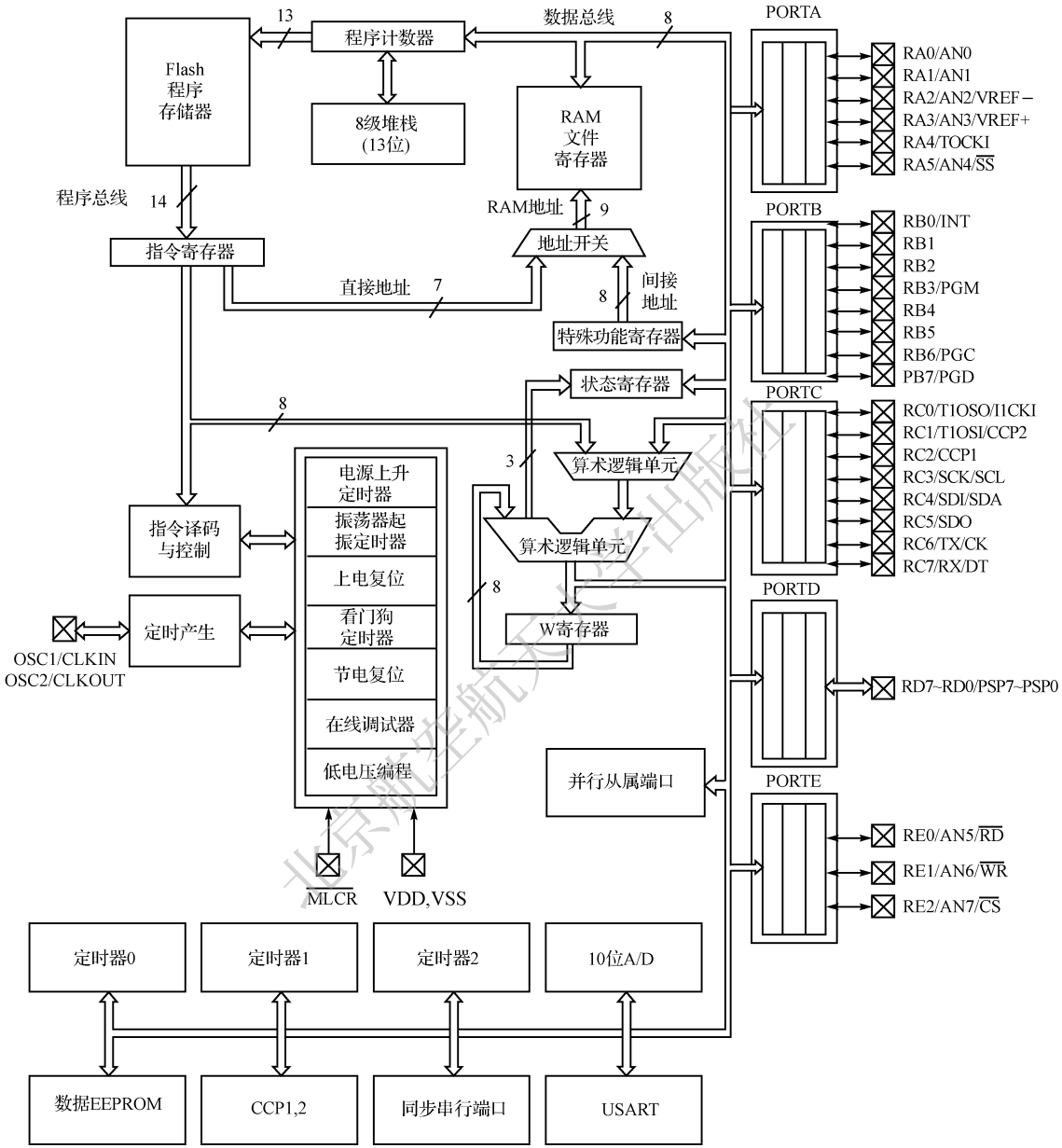


图 1－4　PIC16F877 内部结构框图

◆ 除了程序分支指令外都是单周期指令。
◆ 工作速度是 DC—20 MHz 时钟输入，DC—200 ns 指令周期。
◆ Flash 程序存储器为 8K × 14 W，数据存储器 RAM 为 368 × 8 B，EEPROM 为256×8 B。
◆ 引脚与 PIC16F874 兼容。
◆ 8 级深度的硬件堆栈，即能放 8 个程序返回地址的堆栈。
◆ 直接、间接、相对三种寻址方式。
◆ 具有上电复位功能 POR(Power-on Reset)。
◆ 具有电源上升定时器 PWRT(Power-up Timer)和振荡器起振定时器 OST(Oscillator Start-up Timer)。
◆ 带有片内的看门狗定时器 WDT(Watchdog Timer)。
◆ 可编程的代码保护，保护程序防止窃密。
◆ 具有节电休眠 SLEEP 模式，适合电池供电场合。
◆ 可以随意选择振荡器。
◆ 低电源、高速度 CMOS 闪速 Flash/EEPROM 技术。
◆ 全静态设计。
◆ 可经两个引脚在线串行编程 ICSP(In-Circuit Serial Programming)。
◆ 具有单电源 5 V 在线串行编程能力。
◆ 可经两个引脚进行在线调试。
◆ 处理器可读/写程序存储器。
◆ 工作电压范围为 2.0～5.5 V。
◆ 高漏/源电流为 25 mA。
◆ 具有商业和工业级温度范围。
◆ 耗电低，即典型值为 5 V、4 MHz 时，耗电小于 2 mA；典型值为 3 V、32 kHz 时，耗电小于 20 mA；典型值为等待电流时，耗电小于 1 μA。

PIC16F877 除了内核之外，外围还集成了很多电路，它们的电路特性是：

◆ 集成了定时器 0。带有 8 位预分频器的 8 位定时器/计数器，预分频器可以通过程序改变定时器的时钟频率。
◆ 集成了定时器 1。带有预分频器的 16 位定时器/计数器，可以外接晶振/时钟，也可以采用系统时钟。如果外接晶振，则可以在休眠期间工作。
◆ 集成了定时器 2。带有 8 位周期寄存器 PR2、预分频器和后分频器的 8 位定时器/计数器。
◆ 有两个捕获(capture)、比较(compare)、脉冲宽度调制 PWM 模式。捕捉 16 位时，最大分辨力为 12.5 ns；比较 16 位时，最大分辨力为 200 ns；PWM 的最大分辨力是 10 位。

◆ 带有 8 通道 10 位 A/D 转换器。
◆ 带有 SPI(主模式)和 I^2C(主/从)的同步串行端口 SSP(Synchronous Serial Port)。
◆ 带有 9 位地址检测的通用同步异步接收发送器 USART(USART/SCI)。
◆ 8 位并行从属端口 PSP(Parallel Slave Port),配有外部$\overline{RD}$、$\overline{WR}$、$\overline{CS}$控制引脚。
◆ 具有节电锁定复位 BOR(Brown-Out Reset)的节电检测电路。

1.2.2 引脚定义

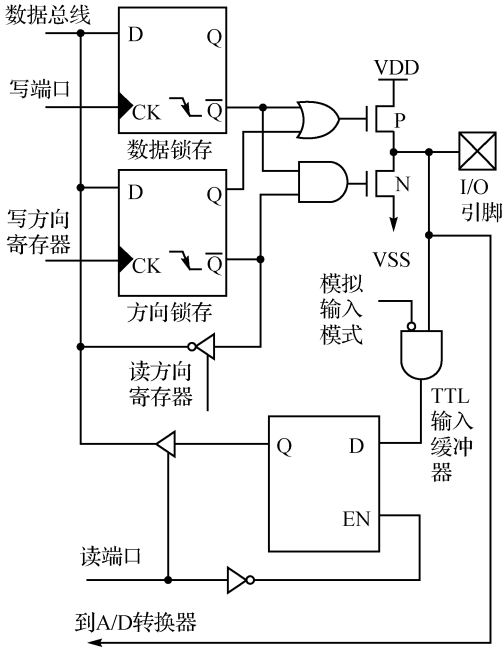
DIP 封装的 PIC16F877 共有 40 个引脚,如图 1-1 所示,其中 11、32 脚是电源引脚,12、31 脚是电源地,分别接到 5 V 电源的正负极上;1 脚是复位引脚,低电平复位;13、14 引脚是时钟输入引脚和时钟输出引脚,可由 RC 振荡电路或晶振振荡电路构成时钟源;其余引脚分别是 RA0～RA5、RB0～RB7、RC0～RC7、RD0～RD7、RE0～RE2。这些引脚除了当做普通端口使用外,很多引脚还有其他功能。笔者在第 1 次使用 RA4 口控制液晶屏使能端口时,无论如何液晶屏都不能正常显示,后来详细了解端口后发现,RA4 是漏极开路的,需要接上拉电阻。所以对于初学者来说,有必要详细了解端口的特性,做到知己知彼。

PIC16F877 的每个 I/O 端口都配有 2 个相关的寄存器:一个是寄存器 TRISx;另一个是端口寄存器 PORTx。TRISx 是一个方向寄存器,用来设置端口的方向,也就是将端口设置为输入口或者输出口:逻辑 1 代表端口是输入口,逻辑 0 代表端口是输出口。需要注意在上电复位后,这些 I/O 端口的默认状态都是输入。PORTx 是端口寄存器,在作为输出口时,该寄存器中存放的是要输出的数据;在作为输入口时,该寄存器中存放的是要读取的数据。这两个寄存器都是 8 位寄存器,分别与每个端口的 8 个引脚相对应。

1. PORTA 和 TRISA 寄存器

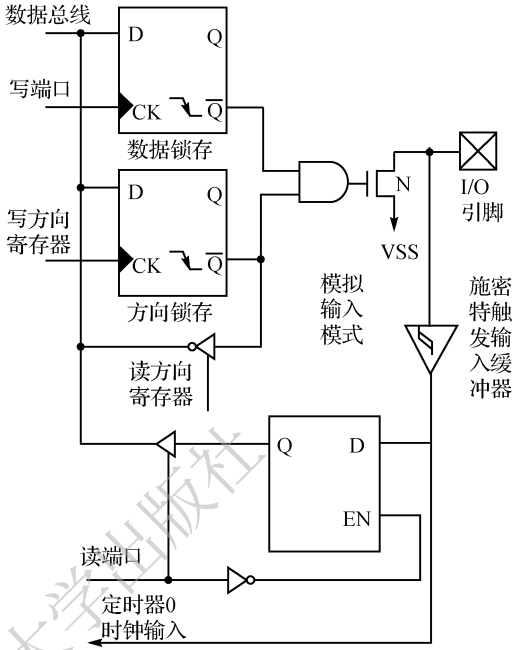
PORTA 是一个 6 位长度的双向端口,相对应的方向寄存器是 TRISA。设置相应的位为 1 将使相对应的 PORTA 端口设置为输入口,设置相应的位为 0 将使相对应的 PORTA 端口设置为输出口。当将端口设置为输入口时,从 PORTA 寄存器读取的数据表示引脚的状态;当将端口设置为输出口时,则将数据写到端口锁存器中。所有写操作都经过读—修改—写的过程,因此对端口的“写”意味着先读引脚,然后修改值,最后再写入到端口锁存器,其内部电路如图 1-5 所示。

RA4 引脚同时被复用为 TMR0 的时钟输入引脚,作为输入时,其内部是一个施密特触发器;作为输出时,则是一个漏极开路输出,其内部电路如图 1-6 所示。所有其他 RA 端口引脚的输入信号都是 TTL 电平,输出信号都是 CMOS 驱动输出。需要注意的是,将 RA4 作为输出口时一定要接上拉电阻。当其他引脚复用为模拟输入或者模拟参考电压输入时,由 ADCON1 寄存器进行设置。注意,在上电复位后,这些引脚都将配置为输入。端口 A 各个引脚的功能如表 1-1 所列。



注：I/O 脚对 VDD 和 VSS 有保护二极管。

图 1-5　RA3～RA0 和 RA5 引脚内部电路



注：I/O 脚只对 VSS 有保护二极管。

图 1-6　RA4/T0CKI 引脚内部电路

表 1-1　端口 A 功能表

引脚名称	位	缓　冲	功　能
RA0/AN0	第 0 位	TTL	通用输入/输出口，模拟输入口
RA1/AN1	第 1 位	TTL	通用输入/输出口，模拟输入口
RA2/AN2/VREF−	第 2 位	TTL	通用输入/输出口，模拟输入口，负参考电压
RA3/AN3/VREF+	第 3 位	TTL	通用输入/输出口，模拟输入口，正参考电压
RA4/T0CKI	第 4 位	ST	通用输入/输出口，定时器 0 外部时钟输入口，漏极开路输出
RA5/$\overline{SS}$/AN4	第 5 位	TTL	通用输入/输出口，模拟输入口，同步串口从模式选择输入

注：TTL＝TTL 输入，ST＝施密特触发输入。

通过上面的学习，相信对端口 A 已经比较了解了。下面为便于学习，总结归纳出与端口 A 相关的寄存器如表 1-2 所列。表 1-3 所列为 ADCON1 寄存器中 PCFG3～PCFG0 的作用，用来设置 A 口和 E 口是作为模拟输入口还是数字 I/O 口使用，以及如何配置模拟输入的正参考电压和负参考电压。

表 1-2　端口 A 相关寄存器

地　址	名　称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	上电复位和节电锁定复位时的值	其他复位时的值
05H	PORTA	—	—	RA5	RA4	RA3	RA2	RA1	RA0	--0X0000	--0U0000
85H	TRISA	—	—	端口 A 方向寄存器						--111111	--111111
9FH	ADCON1	ADFM	—	—	—	PCFG3	PCFG2	PCFG1	PCFG0	--0-0000	--0-0000

注：1　X＝未知；U＝未改变；-＝未占用位，读做“0”。

2　端口 A 不使用阴影单元。

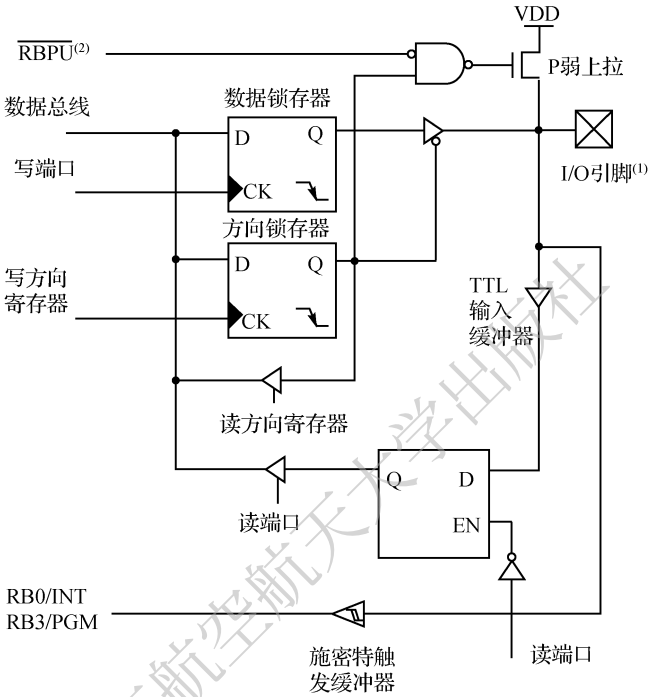
表 1-3　ADCON1 中 PCFG3～PCFG0 的作用

PCFG3～PCFG0	AN7 RE2	AN6 RE1	AN5 RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF＋	VREF－
0000	A	A	A	A	A	A	A	A	VDD	VSS
0001	A	A	A	A	VREF＋	A	A	A	RA3	VSS
0010	D	D	D	A	A	A	A	A	VDD	VSS
0011	D	D	D	A	VREF＋	A	A	A	RA3	VSS
0100	D	D	D	D	A	D	A	A	VDD	VSS
0101	D	D	D	D	VREF＋	D	A	A	VDD	VSS
011X	D	D	D	D	D	D	D	D	VDD	VSS
1000	A	A	A	A	VREF＋	VREF－	A	A	RA3	RA2
1001	D	D	A	A	A	A	A	A	VDD	VSS
1010	D	D	A	A	VREF＋	A	A	A	RA3	VSS
1011	D	D	A	A	VREF＋	VREF－	A	A	RA3	RA2
1100	D	D	D	A	VREF＋	VREF－	A	A	RA3	RA2
1101	D	D	D	D	VREF＋	VREF－	A	A	RA3	RA2
1110	D	D	D	D	D	D	D	A	VDD	VSS
1111	D	D	D	D	VREF＋	VREF－	D	A	RA3	RA2

2. PORTB 和 TRISB 寄存器

PORTB 是一个 8 位长度的双向端口，相对应的方向寄存器是 TRISB。设置 TRISB 相应的位为 1 将使相对应的 PORTB 端口设置为输入口，设置相应的位为 0 将使相对应的 PORTB 端口设置为输出口。PORTB 端口中的三个端口 RB3/PGM，RB6/PGC 和 RB7/PGD 复用为低电压编程功能。

端口 B 中的每一个引脚都有内部的弱上拉电阻，由一个控制位设定，即 OPTION_REG 寄存器中的第 7 位$\overline{\text{RBPU}}$。当$\overline{\text{RBPU}}$=0 时打开所有的内部弱上拉电阻。当引脚配置为输出时，自动关闭所有的上拉电阻。在电源上电复位时所有的弱上拉电阻被禁止。RB3～RB0 引脚的内部电路如图 1－7 所示。



注：(1) I/O 脚对 VDD 和 VSS 有保护二极管。

(2) 为了使能弱上拉电阻，应对 TRISB 的合适位进行设置，并清除$\overline{\text{RBPU}}$(OPTION_REG〈7〉)。

图 1－7　RB3～RB0 引脚内部电路

RB7～RB4 这四个引脚在电平变化时产生中断，因此可以利用这个特点构成电平变化中断键盘。当这些引脚配置成输入引脚后，可以自动将锁存器中的数值与当前引脚电平数值进行比较。若不一致，称为“失配”，则会产生一个中断，该中断可将单片机从睡眠状态中唤醒。任何一个中断的产生都有一个中断标志位，电平变化中断标志位为 RBIF，该位是中断控制寄存器 INTCON 的第 0 位。如果有中断产生，标志位 RBIF=1，那么处理完中断后将该位清零。在中断程序中可以采用如下方法清除中断：

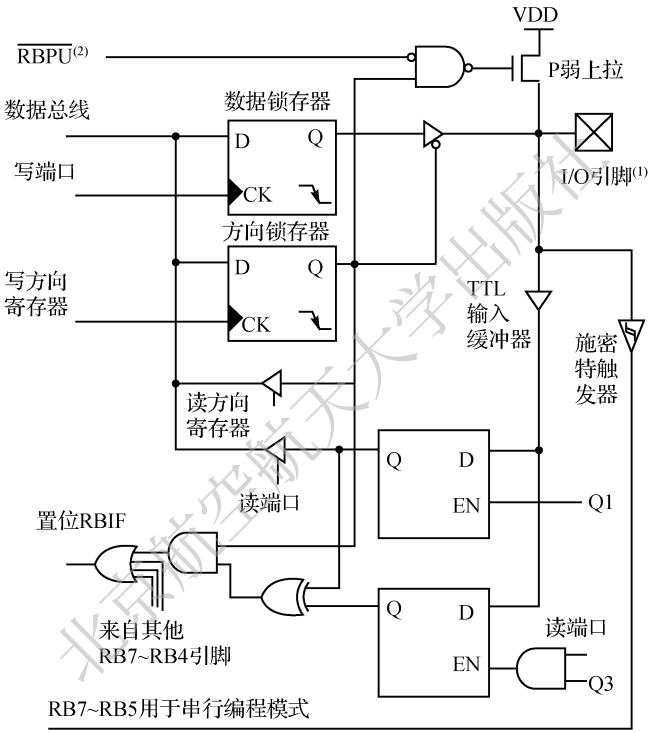
◆ 对 PORTB 的任何读或写。这将结束不一致的条件。

◆ 清除标志位 RBIF。

失配的条件将会继续设置标志位 RBIF。读 PORTB 将结束失配的条件，并允许清除标志位 RBIF。

“电平变化中断”推荐用于键盘唤醒操作和电平变化时的中断操作。当使用电平变化中断时，不建议用 PORTB 查询。图 1－8 是 RB7～RB4 引脚内部电路图。

RB0/INT 是一个外部中断输入引脚，使用 INTEDG 位进行配置，INTEDG 位是 OPTION_REG 寄存器的第 6 位。表 1－4 是端口 B 的功能表，表 1－5 列出了与端口 B 有关的寄存器。



注：(1) I/O 脚对 VDD 和 VSS 有保护二极管。

(2) 为了使能弱上拉电阻，应对 TRISB 的合适位进行设置，并清除$\overline{\text{RBPU}}$(OPTION_REG〈7〉)。

图 1－8　RB7～RB4 引脚内部电路

表 1－4　端口 B 功能表

引脚名称	位	缓　冲	功　能
RB0/INT	第 0 位	TTL/ST(1)	通用输入/输出口或外部中断输入，内部软件可编程弱上拉电阻
RB1	第 1 位	TTL	通用输入/输出口，内部软件可编程弱上拉电阻
RB2	第 2 位	TTL	通用输入/输出口，内部软件可编程弱上拉电阻

引脚名称	位	缓　冲	功　能
RB3/PGM	第 3 位	TTL	通用输入/输出口或 LVP 模式可编程引脚,内部软件可编程弱上拉电阻
RB4	第 4 位	TTL	通用输入/输出口(电平变化中断),内部软件可编程弱上拉电阻
RB5	第 5 位	TTL	通用输入/输出口(电平变化中断),内部软件可编程弱上拉电阻
RB6/PGC	第 6 位	TTL/ST(2)	通用输入/输出口(电平变化中断)或在电路调试引脚,内部软件可编程弱上拉电阻和串行可编程时钟
RB7/PGD	第 7 位	TTL/ST(2)	通用输入/输出口(电平变化中断)或在电路调试引脚,内部软件可编程弱上拉电阻和串行数据口

注:TTL=TTL 输入,ST=施密特触发输入。

(1) 当为外部中断时,该缓冲器为施密特触发器。

(2) 当用于串行可编程模式时,该缓冲器为施密特触发器。

表 1-5　端口 B 相关寄存器

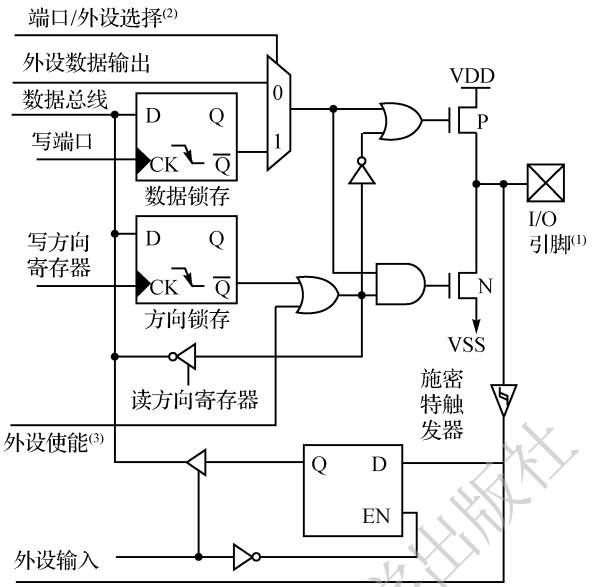
地　址	名　称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	上电复位和节电锁定复位时的值	其他复位时的值
06H,106H	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	XXXX XXXX	UUUU UUUU
86H,186H	TRISB	—	—	端口 B 方向寄存器						1111 1111	1111 1111
81H,181H	OPTION_REG	$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

注:1　X=未知,U=未改变。

2　PORTB 不使用阴影单元。

3. PORTC 和 TRISC 寄存器

PORTC 是一个 8 位的双向端口。相对应的方向寄存器是 TRISC。设置 TRISC 相应位为 1 将使该端口设置成输入端口。将 TRISC 相应位清零则将使该端口设置成输出端口。RORTC 与几个外设的功能是复用的(表 1-6)。PORTC 具有施密特触发输入缓冲器。当 I^2C 模块使能时,PORTC 的 3、4 引脚可以通过 CKE 位(SSPSTAT〈6〉)配置为标准的 I^2C 电平和 SMBUS 电平。当外设功能使能时,特别要注意 PORTC 的方向寄存器 TRISC 的设置。某些外设忽略 TRISC 中相应位将一个引脚设置为输出引脚,而其他外设忽略 TRISC 中相应位将一个引脚设置为输入引脚。当外设使能时,由于忽视 TRISC 中相应位的影响,故要避免使用以 TRISC 为目标的读出—修改—写入指令。图 1-9 是 RC0～RC2 和 RC5～RC7 的引脚内部电路(不考虑外设输出)。图 1-10 是 RC3～RC4 的引脚内部电路(不考虑外设输出)。表 1-6 是端口 C 的功能表,表 1-7 列出了与端口有关的寄存器。



注：(1) I/O 脚对 VDD 和 VSS 有保护二极管。

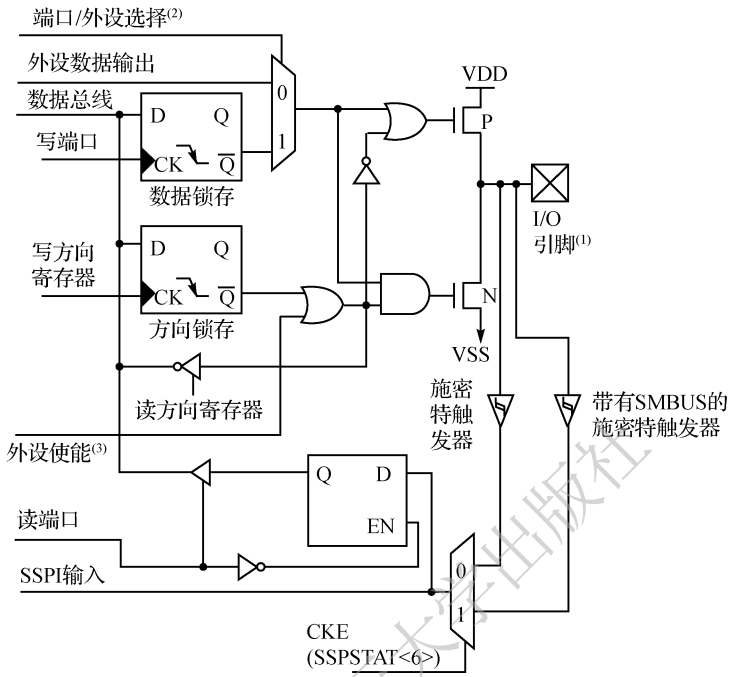
(2) 端口外设选择信号在端口数据和外设输出之间进行选择。

(3) 如果外设选择是激活的，则只能激活外设输出使能 OE(Out Enable)。

图 1-9　RC0～RC2 和 RC5～RC7 引脚内部电路

表 1-6　端口 C 功能表

引脚名称	位	缓　冲	功　能
RC0/T1OSO/T1CK1	第 0 位	ST	通用输入/输出口或定时器 1 振荡器输出/定时器 1 时钟输入
RC1/T1OSI/CCP2	第 1 位	ST	通用输入/输出口或定时器 1 振荡器输入或 Capture2 输入/Compare2 输出/PWM2 输出
RC2/CCP1	第 2 位	ST	通用输入/输出口或 Capture1 输入/Compare1 输出/PWM1 输出
RC3/SCK/SCL	第 3 位	ST	通用输入/输出口或作为 SPI 和 I^2C 模式的同步串行时钟
RC4/SDI/SDA	第 4 位	ST	通用输入/输出口或同步串行数据输入端口或 I^2C 数据口
RC5/SDO	第 5 位	ST	通用输入/输出口或同步串行数据输出端口
RC6/TX/CK	第 6 位	ST	通用输入/输出口或 USART 异步发送或同步时钟
RC7/RX/DT	第 7 位	ST	通用输入/输出口或 USART 异步接收或同步数据



注：(1) I/O 脚对 VDD 和 VSS 有保护二极管。

(2) 端口外设选择信号在端口数据和外设输出之间进行选择。

(3) 如果外设选择是激活的，则只能激活外设输出使能 OE(Out Enable)。

图 1-10　RC3～RC4 引脚内部电路

表 1-7　端口 C 相关寄存器

地　址	名　称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	上电复位和节电锁定复位时的值	其他复位时的值
07H	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	XXXX XXXX	UUUU UUUU
87H	TRISC	端口 C 方向寄存器								1111 1111	1111 1111

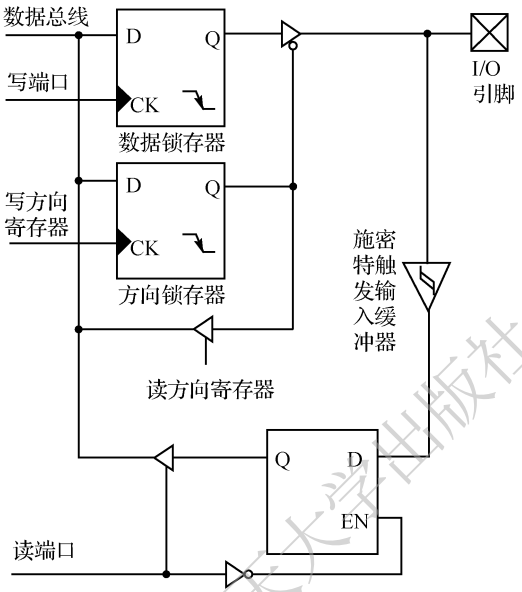
注：X＝未知，U＝未改变。

看到这里，大家可能对 PIC16F877 的端口引脚有所了解了，其实只要了解就已经足够了，因为在开发时很少关注其内部结构，而主要研究如何应用它来开发实际的产品，不过对芯片内部结构有所了解还是很必要的。

4. PORTD 和 TRISD 寄存器

PORTD 也是一个具有施密特触发输入缓冲器的 8 位端口。每个端口可以配置成输入或者输出引脚。通过设置控制位 PSPMODE(TRISE〈4〉)，也可以将 PORTD 配置为 8 位宽的微

处理器端口(并行从端口)。在这种模式下,输入缓冲器是 TTL。图 1-11 是 PORTD 口引脚内部电路。表 1-8 为端口 D 的功能表,表 1-9 列出了与端口 D 有关的寄存器。



注:I/O 脚对 VDD 和 VSS 有保护二极管。

图 1-11　PORTD 口引脚内部电路

表 1-8　端口 D 功能表

引脚名称	位	缓　冲	功　能
RD0/PSP0	第 0 位	ST/TTL(1)	通用输入/输出口或并行从端口第 0 位
RD1/PSP1	第 1 位	ST/TTL(1)	通用输入/输出口或并行从端口第 1 位
RD2/PSP2	第 2 位	ST/TTL(1)	通用输入/输出口或并行从端口第 2 位
RD3/PSP3	第 3 位	ST/TTL(1)	通用输入/输出口或并行从端口第 3 位
RD4/PSP4	第 4 位	ST/TTL(1)	通用输入/输出口或并行从端口第 4 位
RD5/PSP5	第 5 位	ST/TTL(1)	通用输入/输出口或并行从端口第 5 位
RD6/PSP6	第 6 位	ST/TTL(1)	通用输入/输出口或并行从端口第 6 位
RD7/PSP7	第 7 位	ST/TTL(1)	通用输入/输出口或并行从端口第 7 位

注:ST=施密特触发输入,TTL=TTL 输入。

(1) 输入缓冲器 I/O 模式下为施密特触发器,并行从端口模式下为 TTL 缓冲器。

表 1-9　端口 D 相关寄存器

地　址	名　称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	上电复位和节电锁定复位时的值	其他复位时的值
08H	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	XXXX XXXX	UUUU UUUU
88H	TRISD	端口 D 方向寄存器								1111 1111	1111 1111
89H	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE 数据方向位			0000 -111	0000 -111

注：1　X＝未知；U＝未改变；-＝未占用，读做“0”。

2　PORTD 不使用阴影单元。

5. PORTE 和 TRISE 寄存器

PORTE 端口有 3 个引脚：RE0/$\overline{RD}$/AN5、RE1/$\overline{WR}$/AN6 和 RE2/$\overline{CS}$/AN7。它们都可以独立地配置成输入或输出引脚。这些引脚都具有施密特触发器输入缓冲器。当控制位 PSPMODE(TRISE〈4〉)置位时，PORTE 变成微处理器端口的控制输入。在这种模式下，用户必须确保 TRISE 寄存器的第 2～0 位被置位(引脚配置成数字输入)，并通过 ADCON1 寄存器将 PORTE 配置成数字 I/O 口。在这种模式下，输入缓冲器为 TTL。表 1-10 为端口 E 寄存器。

表 1-10　端口 E 寄存器

地　址	名　称	R	R	R/W	R/W	U	R/W	R/W	R/W
89H	TRISE	IBF	OBF	IBOV	PSPMODE	—	Bit2	Bit1	Bit0

注：R＝可读；R/W＝可读可写；U＝上电复位值。

TRISE 寄存器各位含义如下：

IBF　输入缓冲器满状态位(Input Buffer Full status bit)：

1＝接收到一个字等待 CPU 读取。

0＝没有接收到字。

OBF　输出缓冲器满状态位(Output Buffer Full status bit)：

1＝输出缓冲器仍然保持以前写的字。

0＝输出缓冲器已经读取。

IBOV　输入缓冲器溢出检测位(Input Buffer Overflow detect bit)：

1＝当以前写的字还没有读取时，发生写溢出(软件中必须清除)。

0＝没有溢出发生。

PSPMODE　并行从端口模式选择位(Parallel Slave Port Mode select bit)：

1＝并行从端口模式。

0＝通常 I/O 口模式。

—　未占用，读做“0”。

Bit2　引脚 RE2/$\overline{CS}$/AN7 的方向控制位：

1＝输入引脚。

0＝输出引脚。

Bit1　引脚 RE1/$\overline{WR}$/AN6 的方向控制位：

1＝输入引脚。

0＝输出引脚。

Bit0　引脚 RE0/$\overline{RD}$/AN5 的方向控制位：

1＝输入引脚。

0＝输出引脚。

6. 并行从端口(Parallel Slave Port)

当控制位 PSPMODE(TRISE〈4〉)置位时，PORTD 也可作为 8 位宽的并行从端口或微处理器端口。在从模式下，8 位并行从端口由外界通过引脚$\overline{RD}$控制输入引脚 RE0/$\overline{RD}$和通过引脚$\overline{WR}$控制输入引脚 RE1/WR 来进行异步读/写。

8 位并行从端口可以直接与 8 位微处理器接口，外部微处理器可以读和写作为 8 位的 PORTD 锁存器。设置 PSPMODE 可以使 RE0/$\overline{RD}$作为$\overline{RD}$输入，使 RE1/$\overline{WR}$作为$\overline{WR}$输入，使 RE2/$\overline{CS}$作为$\overline{CS}$输入。当 RE2～RE0 作为控制端口使用时，必须将 TRISE 寄存器的第 2～0 位设置成输入，再通过 A/D 端口配置位 PCFG3～PCFG0(ADCON1〈3:0〉)将引脚 RE2～RE0 配置成数字 I/O 口。

当$\overline{CS}$和$\overline{WR}$为低电平时，对并行从端口进行写操作；当$\overline{CS}$和$\overline{RD}$为低电平时，对并行从端口进行读操作。

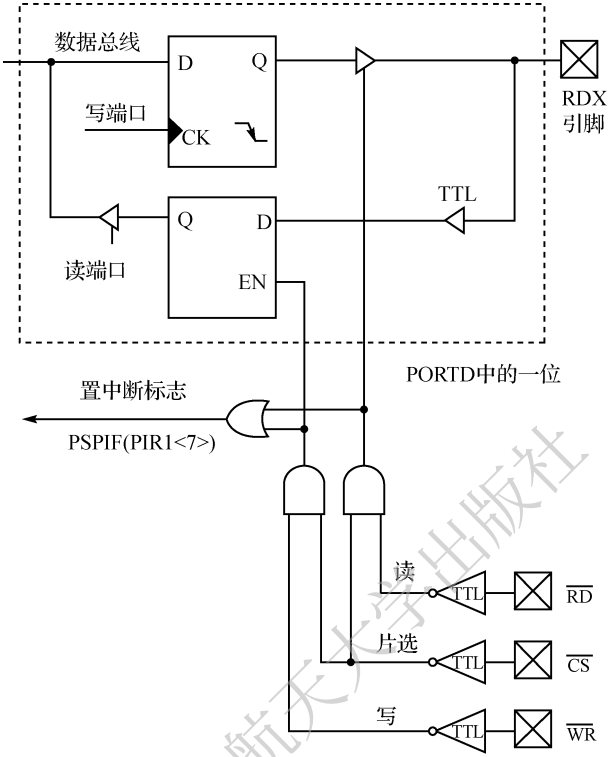
图 1－12 是并行从属操作时 PORTD 和 PORTE 引脚内部电路。表 1－11 为与并行从端口相关的寄存器。

表 1－11　与并行从端口相关的寄存器

地　址	名　称	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	上电复位和节电锁定复位时的值	其他复位时的值
08H	PORTD	写时为端口数据锁存器，读时为端口引脚								XXXX XXXX	UUUU UUUU
09H	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -XXX	---- -UUU
89H	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE 数据方向位			0000 -111	0000 -111
0CH	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF			0000 0000	0000 0000
8CH	PIE1	PSPIE	ADIE	RCIE	TXIF	SSPIF	CCP1IE			0000 0000	0000 0000
9FH	ADCON1	ADFM	—	—	—	PCFG3	PCFG2	PCFG1	PCFG0	--0- 0000	--0- 0000

注：1　X＝未知；U＝未变化；-＝未占用，读做“0”。

2　并行从端口不使用阴影单元。



注：RDX 脚对 VDD 和 VSS 有保护二极管。

图 1-12　并行从属操作时 PORTD 和 PORTE 引脚内部电路

1.3　PIC 单片机开发中的四件法宝

其实 51 单片机开发中的四件法宝同样适用 PIC 单片机，只要掌握了这四件法宝，学习 PIC 单片机就轻车熟路了。

1.3.1　实验开发板

此处自己设计的 PIC 开发板采用了《51 单片机自学笔记》一书中的 51 单片机外围电路，这样做可以节省了解单片机外围电路的时间，使大家快速踏上学习嵌入式开发的道路。开发板上的资源都是一些比较基础且应用广泛的设备，包括 8 个共阳极数码管、4 个独立按键、4×4 矩阵键盘、8 个 LED、串行接口、继电器、蜂鸣器、DS18B20 数字温度传感器、93C46 存储器、DS1302 时钟日历芯片、12864 液晶显示屏和 1602 液晶显示屏等。电路板如图 1-13 所示。

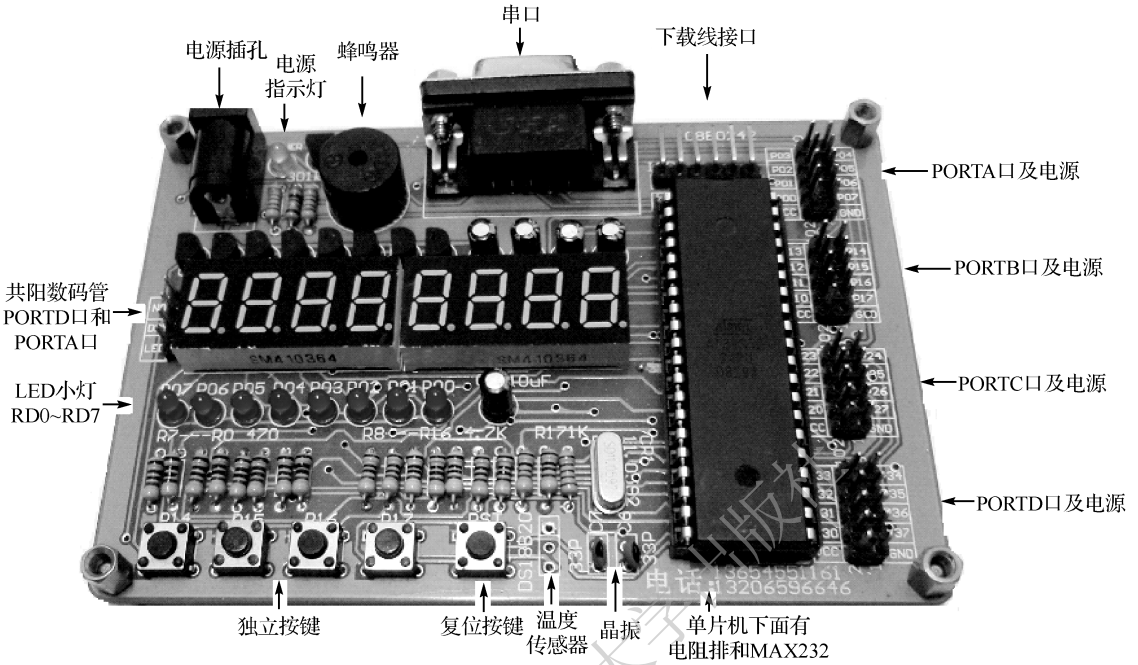


图 1-13　实验开发板

1.3.2　下载线

这里介绍两款下载线，一款是 USB 口下载线 PICkit2，外观如图 1-14 所示，这种下载线在 PIC 网站上提供了相应的原理图和固件，适合笔记本电脑使用。其内部电路如图 1-15 和图 1-16 所示。

1—状态指示灯；2—复位按钮；3—挂绳连接；4—USB 端口连接；
5—图 1-16 中 J3 的引脚 1 标记；6—编程连接器

图 1-14　PICkit2 下载线外观图

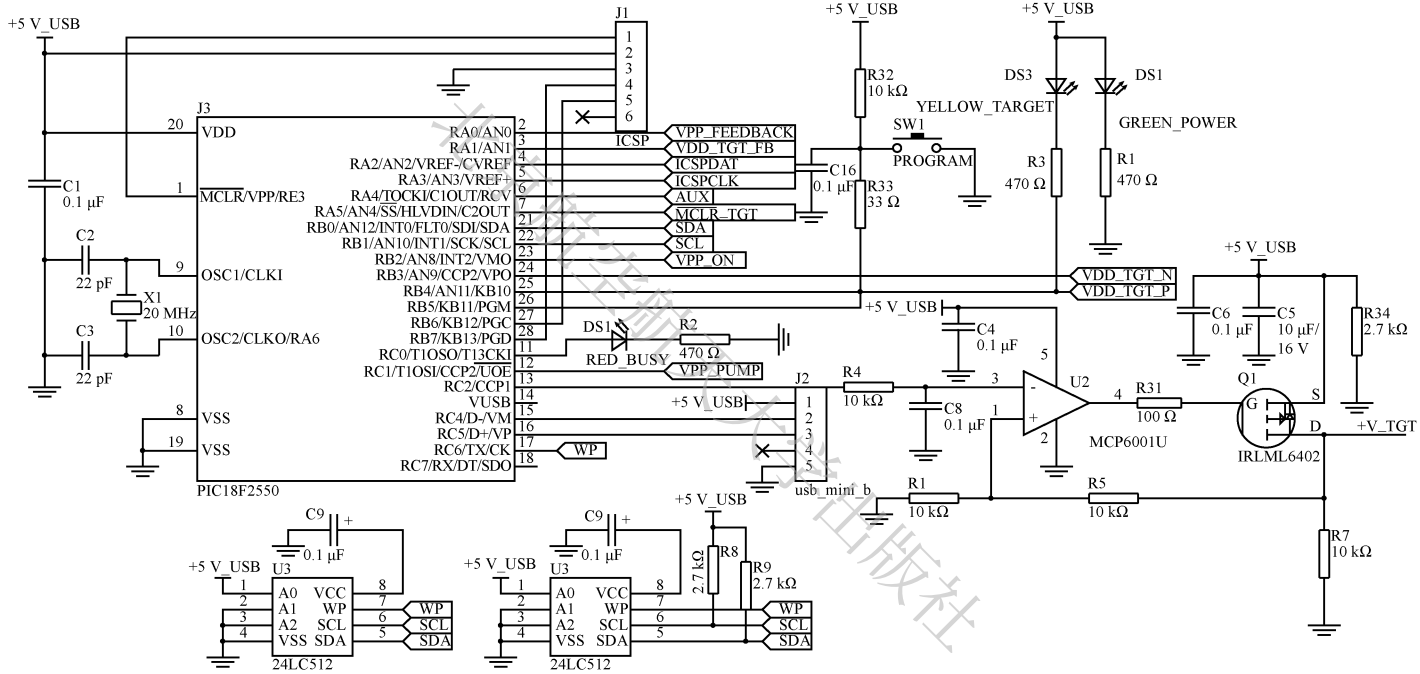


图 1–15　PICkit2 下载线内部电路图1

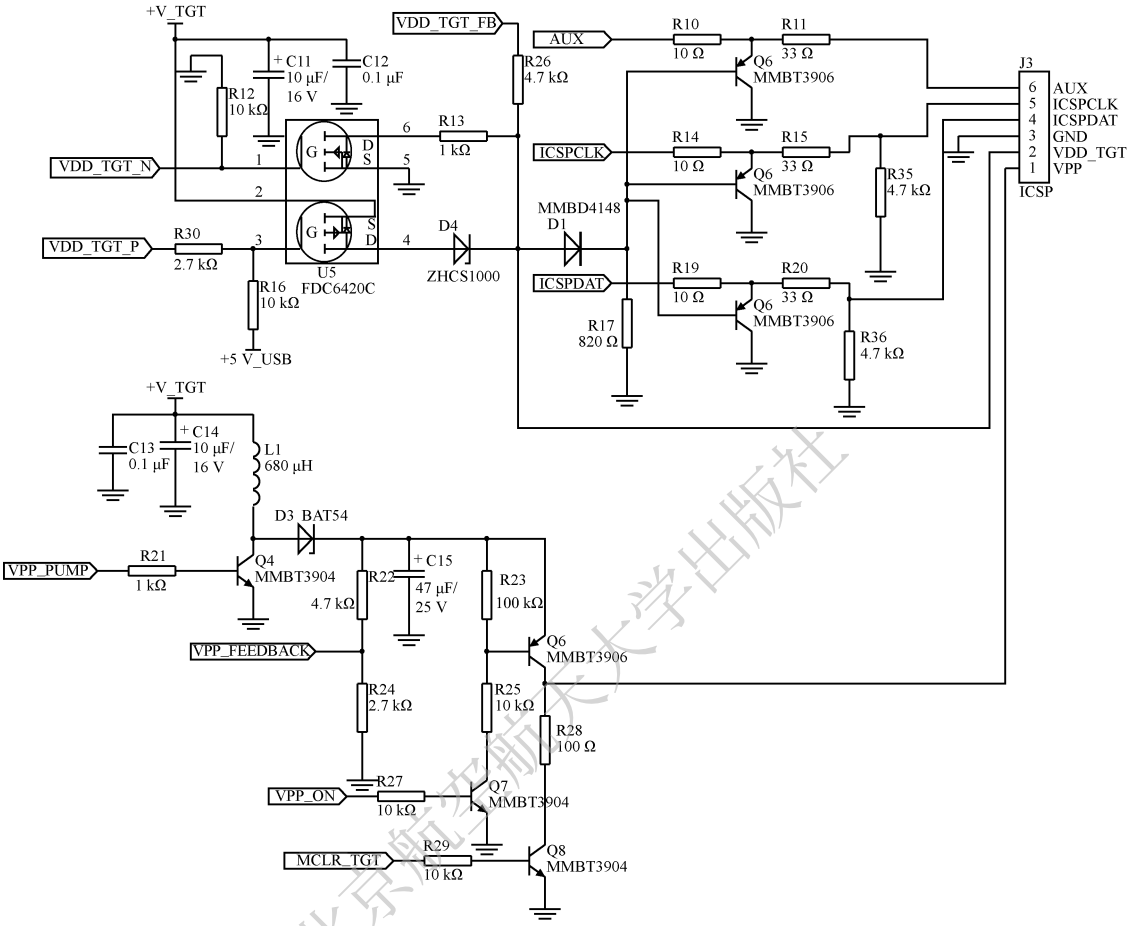


图 1－16　PICkit2 下载线内部电路图 2

这款下载线的电路图比较复杂，成本较高，目前价格在 100 元以上，不过原理图和固件都是开放的，大家可以尝试制作。除了上款下载线外，还有其他的下载线，如 K150 USB 口下载线，如图 1－17 所示。该款下载线电路简单，特别适合初学者制作，相应的固件也可在网上找到。本书中以后的编程将全部采用 PICkit2 下载线。

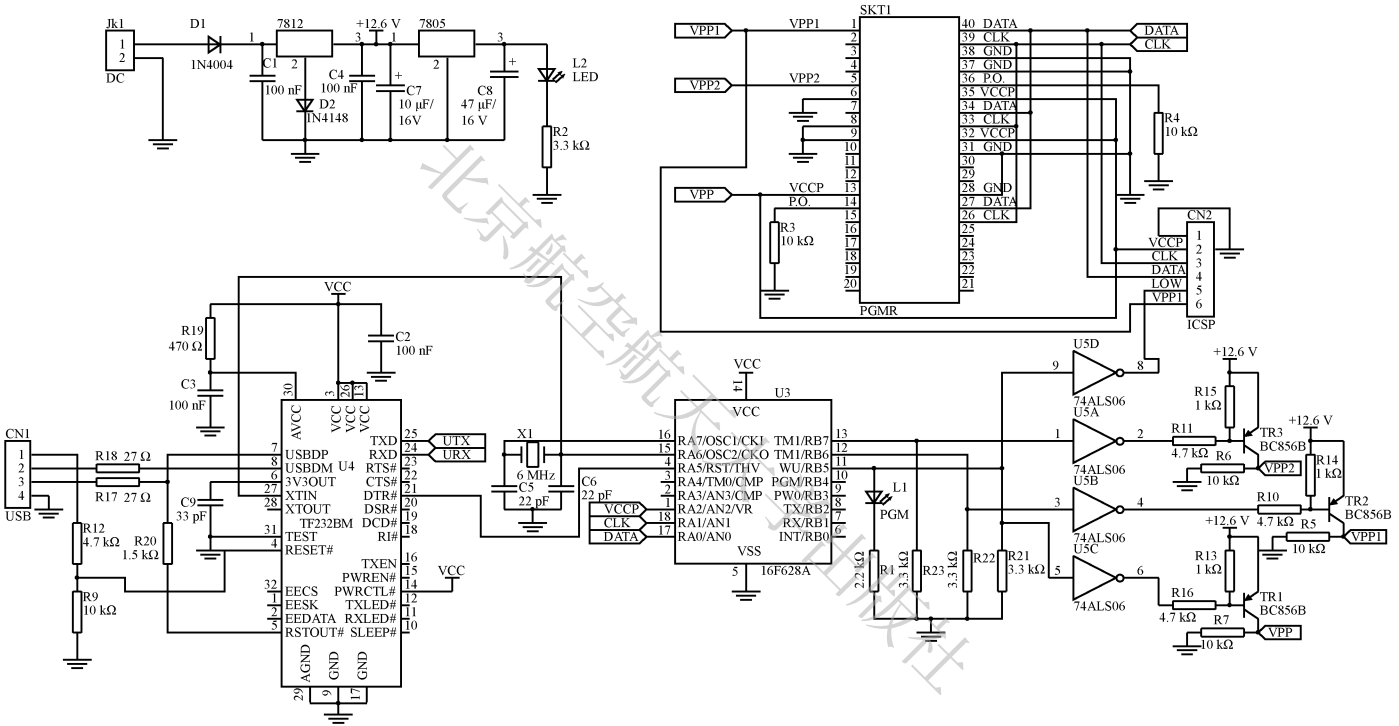


图 1–17　K150 USB口下载线内部电路图

1.3.3 编程软件

应用最广的编程软件是 MPLAB IDE，目前该软件的版本很多，本书用的是 MPLAB IDE V8.33，如果想用最新的版本，可以到 PIC 网站上下载。将 MPLAB IDE V8.33 软件压缩包(图 1-18)解压缩到文件夹中，双击 Install_MPLAB_8_33.exe 文件进行安装，安装界面如图 1-19 所示。单击 Next 按钮进入下一个界面，如图 1-20 所示，单击接受注册协议单选按钮后单击 Next 按钮进入下一步，显示选择安装类型界面，如图 1-21 所示，这里为了修改安装路径，选择自定义安装，单击 Next 按钮进入下一步安装过程。在图 1-22 界面中，单击 Browse 按钮选择安装的具体路径，这里安装在 d:/Program Files/Microchip 目录下，单击 Next 按钮接着进行安装。进入图 1-23 界面出现编程器件类别、C 语言编译器和编译工具选项按钮，这里根据具体应用情况进行选择，然后单击 Next 按钮进入图 1-24 界面，询问是否安装 C 编译器，单击“是”按钮进入图 1-25 安装界面 1，单击 Next 按钮进入图 1-26 安装界面 2，接着单击 Next 按钮即可完成软件安装。

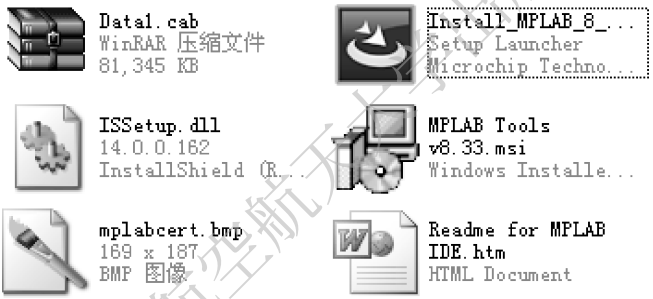
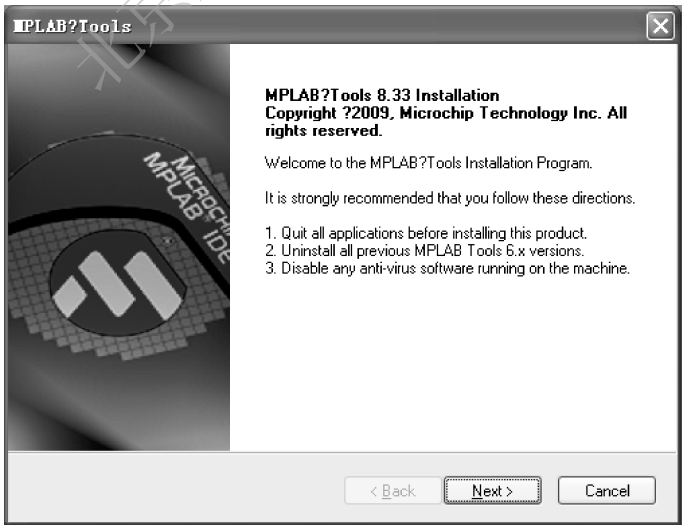


图 1-18 MPLAB IDE V8.33 压缩包



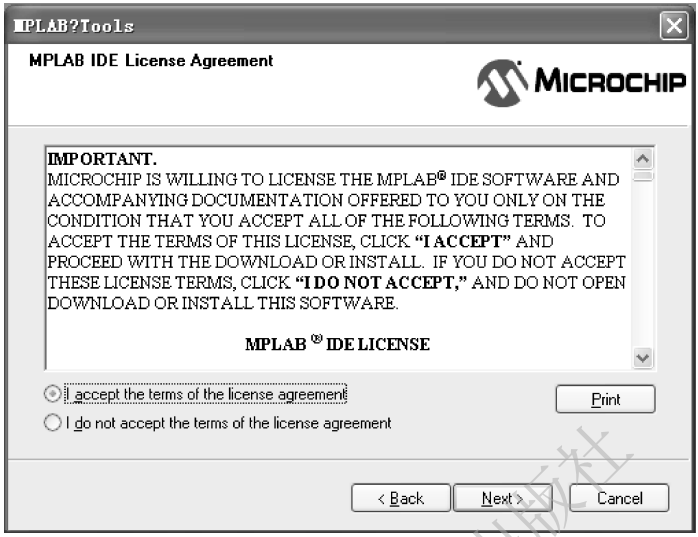


图 1－20　MPLAB IDE V8.33 接受协议界面

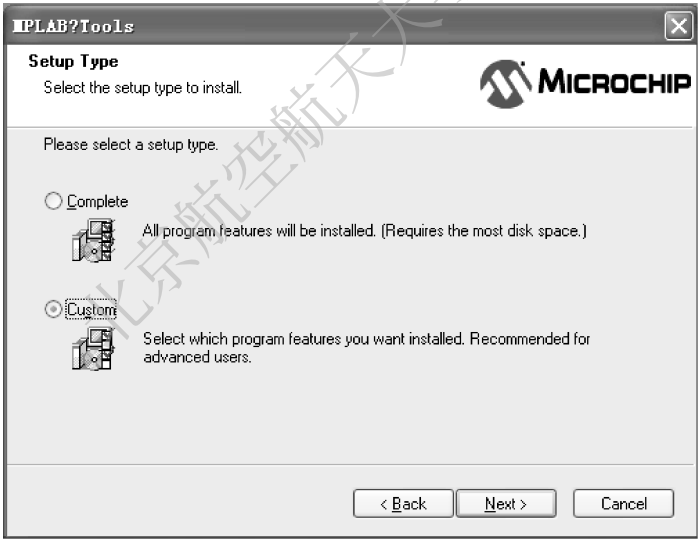


图 1－21　MPLAB IDE V8.33 选择安装类型

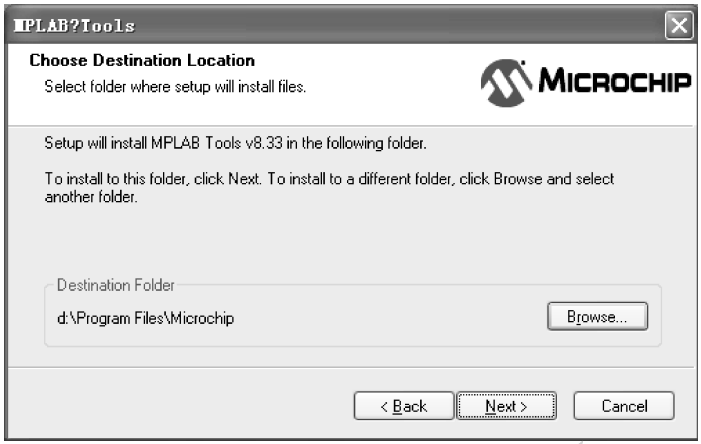


图 1－22　MPLAB IDE V8.33 选择安装路径

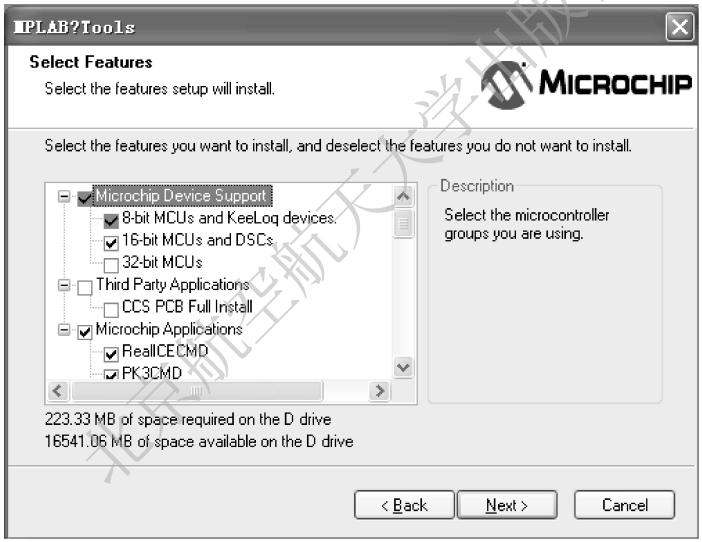


图 1－23　选择编程器件类别、C 语言编译器和编译工具

图 1－24　询问是否安装 HI-TECH C 编译器

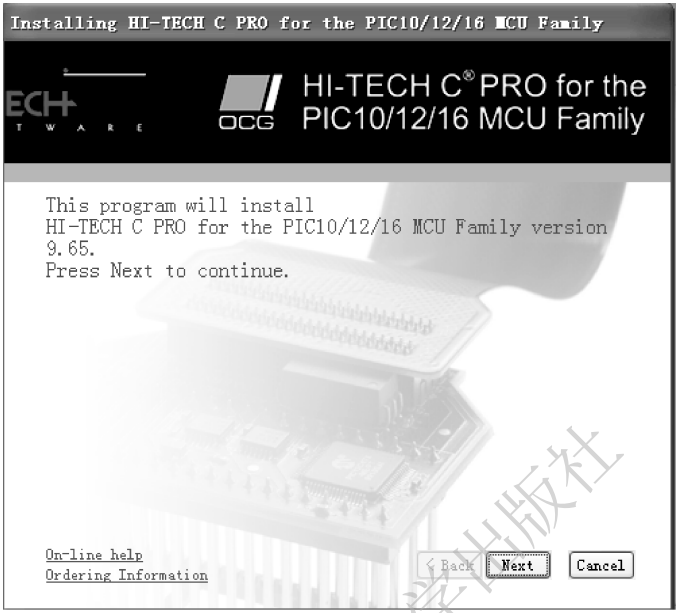


图 1－25　C 编译器安装界面 1

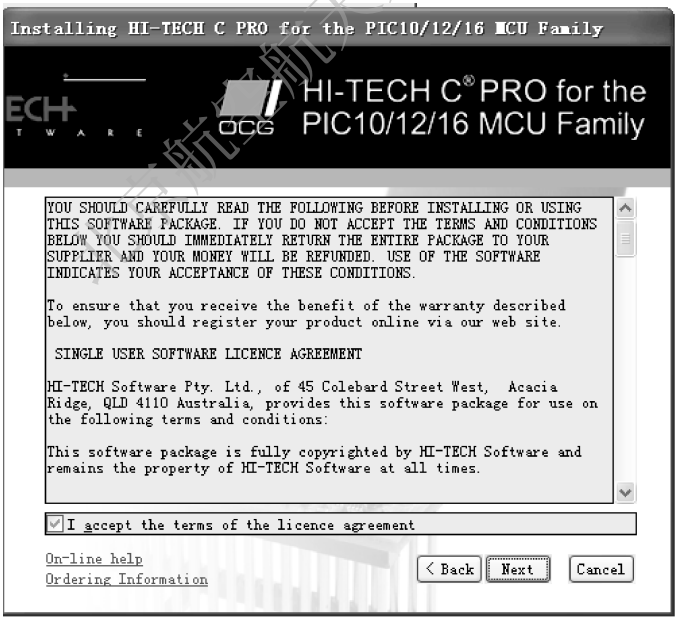


图 1－26　C 编译器安装界面 2

1.3.4 下载软件

可以完成下载任务的软件很多，这里介绍一款下载软件 PICkit2，该软件是由 PIC 公司提供的，支持 USB 口下载。解压软件如图 1－27 所示，双击 setup.exe 文件即可进行安装，出现安装界面 1 如图 1－28 所示，然后单击 Accept 按钮进入安装界面 2，如图 1－29 所示。接着单击 Next 按钮选择安装路径，如图 1－30 所示，这里安装在默认路径下。然后单击 Next 按钮，出现图 1－31 界面，单击接受安装协议单选按钮，再单击 Next 按钮，出现图 1－32 界面，单击 Close 按钮，安装过程结束。

图 1－27　PICkit2 软件安装包

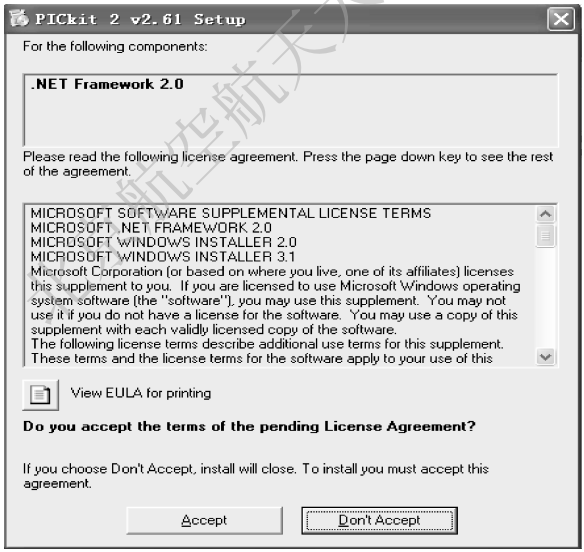


图 1－28　PICkit2 安装界面 1

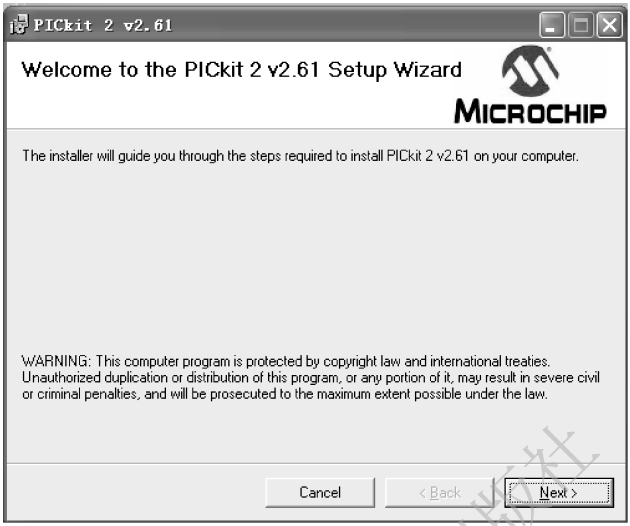


图 1－29　PICkit2 安装界面 2

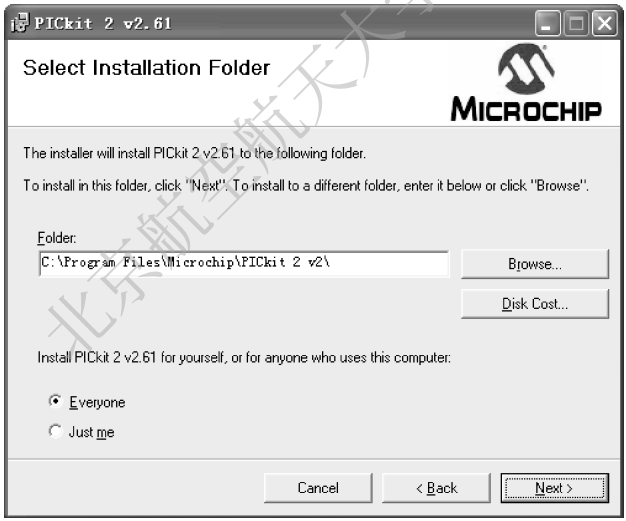


图 1－30　选择安装路径

图 1－31　接受安装协议

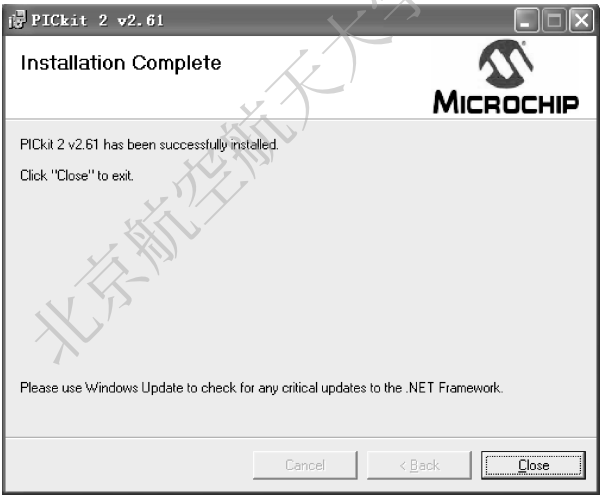


图 1－32　安装完成界面

第2章

PIC 编译器的语法规则

用 C 语言开发单片机系统软件的最大好处是编写代码效率高、软件调试直观、维护升级方便、代码的重复利用率高和便于跨平台的代码移植等，因此用 C 语言编程在单片机系统设计中已得到越来越广泛的运用。针对 PIC 单片机的软件开发，同样可以用 C 语言实现。PIC 单片机的 C 语言编译器有着自己的语法规则，因此在开始软件开发前有必要进行了解。

单片机内的资源非常有限，对控制的实时性要求很高，因此，如果对单片机体系结构和硬件资源没有进行详尽的了解，就无法写出高质量实用的 C 语言程序。Microchip 公司自己没有针对中低档系列单片机的 C 语言编译器，但很多专业的第三方公司却提供了众多支持 PIC 单片机的 C 语言编译器，常见的有 HI-TECH、CCS、IAR、ByteCraft 等公司。这些第三方公司提供的编译器可以在 MPLAB IDE 集成开发环境中通过选择 Project→Set Language Tool Location 菜单项，在所弹出的对话窗口中查询到。PICC 和 PICC18 编译器就是优化了 PIC 8 位系列单片机的 C 语言编译器，由 HI-TECH 公司研制。

PICC 和 PICC18 编译器除了不支持函数的递归调用外，基本上符合 ANSI 标准。不支持函数的递归调用是因 PIC 单片机特殊的堆栈结构所致。PIC 单片机中的堆栈是用硬件实现的，其深度已随芯片而固定，从而无法实现需要大量堆栈操作的递归算法；另外，在 PIC 单片机中实现软件堆栈的效率也不是很高。为此，PICC 编译器采用一种叫做“静态覆盖”的技术，以便实现对 C 语言函数中的局部变量分配固定的地址空间。这样可最大限度地利用 PIC 单片机有限的 RAM 数据存储器。另外，PICC 编译器还可直接挂接在 MPLAB IDE 集成开发平台下，实现一体化的编译、链接和源代码调试，非常方便。

2.1 数据类型

PICC 和 PICC18 中的基本数据类型可分为整型、浮点型和字符型三类，每一类中又有常量和变量之分。所谓常量指在程序运行过程中，其值始终保持不变的量。变量则是在程序运行过程中，其值可以改变的量。

2.1.1 PICC 中的常量

PICC 和 PICC18 的常量分为整型、浮点型、字符型和字符串型四种。

1. 整型常量

PICC 和 PICC18 支持二进制、八进制、十进制和十六进制的整型常量，同时也支持 ANSI 标准进制。不同的进制，其表示方法也不同，如要表示十进制数 120，则不同进制的表示方法如表 2-1 所列。

表 2-1 不同进制的表示方法

进 制	格 式	举 例
二进制	0b+数值或 0B+数值	0b1111000
八进制	O+数值	O170
十进制	数值	120
十六进制	0x+数值或 0X+数值	0x78

在整型常量后加后缀 L(或 l)，可使整型常量变为有符号或无符号的长整型；若加后缀 U(或u)，可使整型常量变为无符号数据类型；若加后缀 L(或 l)和 U(或 u)，可使整型常量变为无符号的长整型。所以一个整型常量应该是符合整型类型且最小的整型类型。

2. 浮点型常量

浮点型常量通常是用一个十进制数表示的有符号实数，可以通过加后缀 F(或 f)加以区分。

3. 字符型常量

字符型常量是用一对单引号括起来的一个字符，这与标准 C 语言的表达格式一样，如‘a’，这里单引号只起定界作用。表示方法为：

```
char x='a';
```

在 C 语言中，字符型数据是按照 ASCII 码存储的，一个字符占一字节(8 位二进制码)，所以字符型变量可以与其他整型数据混合使用。下面程序中的 x 值完全相同。

```
int x=97;          //定义一个整型变量 x,初始值是 97
char x='a';        //定义一个字符型变量 x,初始值是字符 a,a 的 ASCII 值也是 97
```

所以对于 PIC 单片机来说，字符型数据不仅可以作为 8 位整型数据使用，还可用来存储 ASCII 码值。在使用上，字符型数据是最有效的数据，也是最常用的数据类型。

注意：能用 8 位定义的整型变量，最好采用字符型，这样生成的代码最简单，相应运行速度

也最快。

4. 字符串型常量

字符串型常量是用一对双引号括起来的一串字符，如“how are you”，双引号内只能是西文字符。字符串型的常量初始化时被存储在程序存储器中，只能通过常数指针访问，表示方法为：

```
const char * zheng = "how are you";  //定义字符串型常量 zheng,值为 how are you,存储在程序存储器中
```

当把字符串型常量初始化为非常数数组时，将被存储在数据存储器中，如：

```
char zheng[] = "how are you";        //定义字符串型常量 zheng,值为 how are you,存储在数据存储器中
```

在这种情况下，将产生一个数据存储器数组。它在启动运行时，将字符串“how are you”从程序存储器中复制出来。当使用由 const 定义的数组表示字符串时，必须直接访问程序存储器。

若将字符常数作为函数参数，如为指针类型，那么指针必须是常数字符指针(const char *)，如：

```
void sendBuff(const char * ptr);
```

这样，就可以使指针指向程序存储器或数据存储器，以便正确地从适当地方读取数据。中级系列的单片机就是按照这种方式工作的，基本系列的单片机则总是指向程序存储器。通常情况下，PICC 和 PICC18 编译器将字符串、常数以及被定义为 const 的数据存储在程序存储器中。

2.1.2 PICC 中的变量

1. 整型变量

整型变量的数据类型参见表 2-2。从表中可以看出，PICC 和 PICC18 编译器支持 8 位、16 位和 32 位整型数据。有符号整型数都用二进制补码表示，所有 16 位和 32 位数据类型都遵循 Little-endian 标准，即多字节变量的低字节放在存储空间的低地址，高字节放在高地址。

表 2-2　PICC 和 PICC18 中的基本数据类型

类　型	长度/位	数学表达
bit	1	布尔型位变量，有 0 或 1 两种取值
char	8	有符号或无符号字符变量。PICC 默认 char 型变量为无符号数，但可通过编译选项改为有符号字节变量

续表 2－2

类　型	长度/位	数学表达
unsigned char	8	无符号字符变量
short	16	有符号整型数
unsigned short	16	无符号整型数
int	16	有符号整型数
unsigned int	16	无符号整型数
long	32	有符号长整型数
unsigned long	32	无符号长整型数
float	24	浮点数
double	24 或 32	浮点数。PICC 默认 double 型变量为 24 位长，但可通过编译选项改为 32 位长

2. 字符型变量

由表 2－2 可以看出，PICC 和 PICC18 支持有符号字符和无符号字符的 8 位整型数据，也就是把字符当做 8 位整型数据处理，这是与其他高级语言有差别的地方。字符型变量的默认值为无符号字符，无符号字符的取值范围是 0～255。若用 PICC 和 PICC18 编译时采用－SIGNED_CHAR 选项，则该变量为有符号字符，有符号字符用 8 位二进制补码表示，其取值范围是－128～＋127。C 语言中的字符型数据是为 ASCII 码设置的。在使用上，字符型数据是 4 种整型变量中取值范围最小的一种，除此之外，其他方面都与整型数据完全相同。

3. 浮点型变量

PICC 中描述浮点数是以 IEEE－754 标准格式实现的。在此标准下定义的浮点数为 32 位长，在单片机中使用 4 字节存储。为了节约单片机的数据空间和程序空间，PICC 专门提供了一种长度为 24 位的缩短型浮点数，它损失了浮点数的一点精度，但浮点运算的效率得以提高。在程序中定义的 float 型标准浮点数的长度固定为 24 位，双精度 double 型浮点数一般也是 24 位长，但可以在程序编译选项中选择 double 型浮点数为 32 位长，以提高计算的精度。一般控制系统中关心的是单片机的运行效率，因此在精度能够满足的前提下尽量选择 24 位的浮点数运算。

2.2　位指令

1. 位变量

除上述基本数据类型外，PICC 和 PICC18 还包括位型数据，这种类型变量的取值只能是 1

或 0，初始值均为 0，若要使某一位初始值为 1，则应在程序开始部分将其置 1。指针不能指向位变量，位变量也不能作为函数参数，这种位变量只能是全局或者静态的，PICC 将把定位在同一 bank 内的 8 个位变量合并成一字节存放于一个固定地址。因此，所有针对位变量的操作都将直接使用 PIC 单片机的位操作汇编指令高效实现。基于此，位变量不能是局部自动型变量，而且也无法将其组合成复合型高级变量。

PICC 对整个数据存储空间实行位编址，0x000 单元的第 0 位是位地址 0x0000，以此后推，每字节有 8 个位地址，所以整个数据存储空间都可以按位寻址。编制位地址的意义纯粹是为了编译器最后产生汇编级位操作指令而用，对编程人员来说基本可以不考虑。但若能了解位变量的位地址编址方式，则可在最后程序调试时很方便地查找自己所定义的位变量，如果一个位变量 flag1 被编址为 0x12，那么其实际的存储空间位于：

$$字节地址 = 0x12/8 = 0x02$$

$$位偏移 = 0x12\%8 = 2$$

即 flag1 位变量位于地址为 0x02 字节的第 2 位。在程序调试时如果要观察 flag1 的变化，就必须观察地址为 0x02 的字节，而不是地址 0x12。

2. 位操作指令

PIC 单片机的位操作指令是非常高效的。因此，PICC 在编译源代码时只要有可能，即使对普通变量的操作也将以最简单的位操作指令来实现。如假设一字节变量 current 最后被定位于地址 0x20，现要改变这个变量中某一位的数值，那么可进行如下操作：

```
current  |= 0x80;                    //地址 0x20 中第 7 位置 1
current  &= 0xf7;                    //地址 0x20 中第 3 位清 0
```

即所有只对变量中某一位操作的 C 语句代码都将被直接编译成汇编的位操作指令。

若要把 current 变量的第 1 位和第 2 位置 1，还可按如下编写 C 语言程序：

```
uchar current;                       //定义要进行位操作的变量
bitset(current,1);                   //current 变量的第 1 位置 1
bitset(current,2);                   //current 变量的第 2 位置 1
```

在有些应用中需要将一组位变量放在同一字节中，以便需要时一次性地进行读写，这一功能可以通过定义一个位域结构和一个字节变量的联合来实现，例如：

```
union {
    struct {
        unsigned b0:1;
        unsigned b1:1;
        unsigned b2:1;
```

```
        unsigned b3:1;
        unsigned b4:1;
        unsigned :3;          //最高三位保留
    }oneBit;
    unsigned char allBits;
}myDate;
```

当需要存取其中某一位时，可以按如下操作：

```
myDate.oneBit.b1 = 1;        //b1 位置 1
```

当一次性将全部位清零时可以按如下操作：

```
myDate.allBits = 0;        //全部位变量清 0
```

当程序中将非位变量进行强制类型转换而转换成位变量时，要注意编译器只对普通变量的最低位做判别：如果最低位是 0，则转换成位变量 0；如果最低位是 1，则转换成位变量 1。而标准 ANSI C 的做法是判别整个变量值是否为 0。另外，函数可以返回一个位变量，实际上，此返回的位变量将存放于单片机的进位位中带出返回。

2.3 变量的绝对定位

在用 C 语言编写程序时，变量一般由编译器和链接器最后定位，因此在写程序之时无须知道所定义的变量具体被放在哪个地址（除了 bank 必须声明以外）。但有些时候，若采用@address的结构，则可把全局或静态变量固定于某一绝对地址中，例如：

```
unsigned char currentData @0x20;      //currentData 定位在 0x20 数据存储器中
```

千万注意，PICC 和 PICC18 编译器对绝对定位的变量不保留地址空间。换句话说，上面变量 currentData 的地址是 0x20；但最后，地址 0x20 完全有可能又被分配给其他变量使用，这样就发生了地址冲突。因此，针对变量的绝对定位要特别小心，需要编程人员自己确保绝对地址的正确性。PICC 和 PICC18 编译器并不检查绝对变量的地址是否与其他变量的地址重复。

其实，只有单片机中的特殊功能寄存器需要绝对定位，而这些寄存器的地址定位又在 PICC 编译环境所提供的头文件中已经实现，无须用户操心。因此，编程员所要了解的也就是 PICC 是如何定义这些特殊功能寄存器，以及其中相关控制位的名称。

如果需要，位变量也可以绝对定位；但必须遵循上面介绍的位变量编址的方式。如果一个普通变量已经被绝对定位，那么此变量中的每个数据位就都可以用下面的计算方式实现位变量指派：

```
unsigned char currentData @0x20;        //currentData 定位在地址 0x20
bit currentBit0 @currentData * 8 + 0;   //currentBit0 对应于 tmpData 第 0 位
bit currentBit1 @currentData * 8 + 1;   //currentBit1 对应于 tmpData 第 1 位
bit currentBit2 @currentData * 8 + 2;   //currentBit2 对应于 tmpData 第 2 位
```

如果 currentData 事先没有被绝对定位，那么就不能使用上面的位变量定位方式了。

2.4 结构和联合

PICC18 支持结构和联合类型，与标准的 C 语言相同，如果要表示个人信息，如：姓名、年龄、身份证号码及家庭住址等，这些数据显然是不同的数据类型，但它们之间相互又有联系，可用来描述一个人的个人属性。如果用独立的若干变量表示各个属性，则难以表示各个数据的联系。所以，可用一种特殊的数据类型将这些属性变量进行统一描述，这种数据类型就是结构类型。在 C 语言中，还有一种叫做联合的特殊数据类型，也是将不同的数据变量组织成一个整体，在内存中占有共同的存储区域。结构和联合既有相同又有不同的地方，在标准 C 语言中都有说明。PICC18 支持的结构和联合与标准 C 语言中支持的没有什么区别，只是结构和联合中的成员在存储器中的偏移位置不同。结构和联合中的成员不可以是位类型的变量；但结构和联合可以作为函数参数和返回值，也可以定义指向结构和联合的指针。

2.4.1 结构和联合的定义

1. 结构的定义

结构类型变量的定义与其他类型变量的定义是一样的，有 3 种方法：

① 先定义类型，再定义变量，其形式如下：

```
struct  结构名
{
    结构成员列表；
};
struct  结构名  变量列表；
```

例：

```
struct stu                  //定义学生结构类型
{
    char name[25];          //姓名
    int age;                //年龄
    long int id;            //身份证号
    char address[50];       //家庭住址
```

```
};
struct stu stu_1,stu_2;        //定义 stu_1 和 stu_2 为结构类型变量
```

② 定义结构类型的同时定义结构变量,其形式如下:

```
struct  结构名
{
    结构成员列表;
}结构变量列表;
```

例:

```
struct stu                    //定义学生结构类型
{
    ……                        //同上
}stu_1,stu_2;                 //定义 stu_1 和 stu_2 为结构类型变量
```

也可以在后面再定义更多的 struct stu 类型结构变量,如:

```
struct stu stu_3;             //stu_3 也是结构变量
```

③ 直接定义结构变量,其形式如下:

```
struct
{
    结构成员列表;
}结构变量列表;
```

例:

```
struct
{
    ……                        //同上
}stu_1,stu_2;                 //定义 stu_1 和 stu_2 为结构类型变量
```

采用这种方式定义的结构可以不定义结构的名称。如果定义的结构没有定义结构名,那么在其他地方就不能定义这种结构类型的结构变量了。

在定义结构变量时,可以根据需要进行选择。如果采用定义方式①和②,那么,由于在定义结构变量的同时已经定义了结构的名称,因此,就可以在程序的其他地方定义具有相同结构的其他变量;如果采用定义方式③,那么,由于在定义结构变量时没有定义结构的名称,因此,就不便在程序的其他地方定义具有这种结构的其他变量了。

2. 联合的定义

联合的定义与结构的定义相似,也分 3 种定义方法。所不同的是,定义联合时并不为其分

配具体的存储空间，而只是说明该类型的联合变量将要使用的存储模式。联合与结构的最大区别在于，结构类型使用的存储空间为所有成员变量之和，而联合类型使用的存储空间为所有成员中变量空间最大的一个成员的空间。例如：

```
struct message
{
    int x;
    long y;
    char z;
}
```

此结构所占有的存储空间是 int 类型、long 类型和 char 类型的空间之和，所占有的存储单元长度共为 7 字节。

```
union test
{
    int x;
    long y;
    char z;
}a;
```

此联合所占有的存储空间为占有字节最多的 long 类型的空间，所占有的存储单元长度为 4 字节。

2.4.2 结构和联合的引用

结构与联合是含有若干成员的整体，所以在引用时，结构和联合变量一般不能进行整体操作，而只能操作他的成员，二者的引用格式如下：

结构/联合变量名.成员名

值得注意的是：一个联合变量不能存放多个元素的值（这是因为其所有的成员共用一个地址单元），而只能存放一个成员的值，即联合变量最后赋予的值。例如：

```
union test key;
key.a = 200;
key.b = 30000;
```

联合变量 key 中只有一个值，即 key.b 的值。

2.4.3 结构和联合的限定词

结构和联合是高级变量，当一个变量被定义为结构类型时，其所有成员都可以通过该变量被访问。例如：

```
struct{
    int num;
    int * ptr;
}record;
```

在上例中,结构体和它的所有成员存储在 RAM 中。

PICC 和 PICC18 支持使用限定词。如果在一个结构前使用了限定词,则其所有成员会受到与限定词相对应的限定。下面的结构就使用了限定词 const,例如:

```
const struct{
    int num;
    int * ptr;
}record;
```

在上例中,结构存储在 ROM 中,每个成员都是只读的。如果在结构前使用限定词 const,则必须对每个成员初始化。

但如果分别对结构的成员使用 const,而结构本身不使用该限定词,则此结构仍将被存储在 RAM 中,但只有其成员是只读的。

在 PICC 中,联合变量也可使用限定词,与结构相同。当联合体被定义为常数时,联合中的变量将被存储在 ROM 中,否则将被存储在 RAM 中。

2.4.4 结构中的 bit 域

PICC18 支持结构中的 bit 域。bit 域总是存储在一个 8 位宽的单元中,其第一个字符表示字节的最低位。在定义一个 bit 域时,如果当前地址中有合适的 8 位宽单元,则分配到该单元中;否则将分配到一个新的 8 位宽单元中。bit 域绝不会跨字节单元存储。例如:

```
struct{                    //定义结构体类型
    unsigned low:1;        //定义 bit 域 low
    unsigned mid:6;        //定义一个整型成员也位于该结构 bit 域中
    unsigned hig:1;        //定义 bit 域 hig
}foo@0x10;
```

上例中将生成一个结构,该结构占有一字节。如果结构 foo 链接到地址 10H,则 low 将为 10H 的第 0 位,hig 将为 10H 的第 7 位。mid 的最低有效位为 10H 的第 1 位,最高有效位为 10H 的第 6 位。

如果在定义的结构中有未命名的 bit 域,则会占据更多的控制寄存器空间,而这些空间并没有使用。例如,不使用结构中的成员 mid,则以上结构可以采用如下定义方式:

```
struct{
    unsigned low:1;
    unsigned:6;
    unsigned hig:1;
}foo@0x10;
```

如果在结构中定义了一个 bit 域，并且要为该 bit 域指定绝对地址值，则不会再为该结构分配存储单元。如果想更方便地访问寄存器中的任一位，则应先将结构映射到寄存器，再为结构指定绝对地址值。

对于含有 bit 域成员变量的结构，也可对其进行初始化，初始值用逗号隔开，例如：

```
struct{
    unsigned low:1;
    unsigned mid:6;
    unsigned hig:1;
}foo{1,8,0};
```

2.5 PICC 对数据寄存器 bank 的管理

为了使编译器产生最高效的机器码，PICC 把单片机中数据寄存器的 bank 问题交由编程员自己管理，因此在定义用户变量时，编程员必须自己决定这些变量具体放在哪一个 bank 中。如果没有特别指明，所定义的变量将被定位在 bank0，例如：

```
unsigned char buffer[32];
bit flag1,flag2;
float val[8];
```

除了 bank0 内的变量声明不需要做特殊处理外，定义在其他 bank 内的变量前面必须加上相应的 bank 序号，例如：

```
bank1 unsigned char buffer[32];     //变量定位在 bank1 中
bank2 bit flag1,flag2;              //变量定位在 bank2 中
bank3 float val[8];                 //变量定位在 bank3 中
```

中档系列 PIC 单片机数据寄存器的一个 bank 大小为 128 字节，其中包括前面若干字节的特殊功能寄存器区域。在 C 语言中，某一 bank 内定义的变量字节总数不能超过可用 RAM 的字节数。如果超过 bank 容量，在最后链接时会报错，大致信息如下：

```
Error[000] : Can't find 0x12C words for psect rbss_1 in segment BANK1
```

链接器显示出总共有 0x12C(300)字节准备放到 bank1 中，但 bank1 容量不够。显然，只

有把一部分原本定位在 bank1 中的变量改放到其他 bank 中才能解决此问题。

虽然变量所在的 bank 定位必须由编程员自己决定,但在编写源程序时,在进行变量存取操作之前无须再特意编写设定 bank 的指令。C 编译器会根据所操作的对象自动生成对应 bank 设定的汇编指令。为了避免频繁的 bank 切换以提高代码效率,应尽量把实现同一任务的变量定位在同一个 bank 内;在对不同 bank 内的变量进行读/写操作时,也应尽量把位于相同 bank 内的变量归并在一起进行连续操作。

2.6 局部变量和全局变量

俗话说:“国有国法,家有家规”。国家的法律可以管理到每一个人,可家中的家规只适合于各自的家庭。那么,对于 PICC 和 PICC18 编译器中的变量而言,全局变量与局部变量的关系就如同国法与家规的关系,全局变量可以在不同文件中使用,而局部变量只能在函数内部使用。

局部变量分为自动变量和静态变量两种。自动变量通常分布在函数的自动变量区;静态变量则分配到一个固定的存储单元内。

2.6.1 自动变量

PICC 把所有函数内部定义的 auto 型局部变量放在 bank0。为节约宝贵的存储空间,它采用了一种叫做“静态覆盖”的技术来实现局部变量的地址分配。其大致的原理是:在编译器编译源代码时扫描整个程序中函数调用的嵌套关系和层次,算出每个函数中的局部变量字节数,然后为每个局部变量分配一个固定的地址,且按调用嵌套的层次关系,各变量的地址可以相互重叠。利用这一技术后,所有的动态局部变量都可以按已知的固定地址进行直接寻址,用 PIC 汇编指令实现的效率最高,但这时不能出现函数递归调用。PICC 在编译时会严格检查递归调用的问题,并认为这是一个严重错误而立即终止编译过程。

既然所有局部变量将占用 bank0 的存储空间,因此用户自己定位在 bank0 内的变量字节数将受到一定限制,在实际使用时须注意。

2.6.2 静态变量

未初始化的静态变量分配在 rbss_n 程序块中,占用 1 个固定的存储单元,而且该存储单元不会被其他函数调用。静态变量只在声明它的函数范围内有效,但其他函数可以通过指针访问它。除非通过指针明确修改静态变量的值,否则该值在两次函数调用之间保持不变。此外,静态变量不受 PIC 任何结构的限制。

静态变量在程序执行期间只初始化一次,而程序每次运行到自动变量初始化语句时都会对自动变量进行初始化。

2.6.3 全局变量

全局变量要通过关键词 extern 进行外部变量声明。如果要在一个 C 程序文件中使用一些变量,但其原型写在了其他文件中,那么在本文件中必须用关键词 extern 将这些变量声明成外部类型。

例如程序文件 code1.c 中有如下定义:

```
bank1 unsigned char var1, var2; //定义了 bank1 中的两个变量
```

在另外一个程序文件 code2.c 中要对上面定义的变量进行操作,则必须在程序的开头有如下定义:

```
extern bank1 unsigned char var1, var2; //声明位于 bank1 的外部变量
```

2.7 特殊类型限定词

大家都知道,车牌号前面的第一个字母是用来区分车的归属地的,比如说在黑龙江省,开头字母为 A 表示车属于哈尔滨,B 表示车属于齐齐哈尔,诸如此类。那么,在 PICC 和 PICC18 中也有一些特殊变量修饰词,只要在变量前看到这些修饰词就能判断出变量的类型及其存储地址。

1. volatile——易变型变量声明

PICC 中有一个变量修饰词在普通 C 语言介绍中一般是看不到的,这就是关键词 volatile。顾名思义,它说明了一个变量的值是会随机变化的,即使程序并没有刻意对它进行任何赋值操作。在单片机中,作为输入的 I/O 端口,其内容将是随意变化的;在中断内被修改的变量,相对主程序流程来讲也是随意变化的;很多特殊功能寄存器的值也将随着指令的执行而动态改变。所有这种类型的变量必须明确定义成 volatile 类型,例如:

```
volatile unsigned char STATUS @ 0x03;
volatile bit commFlag;
```

volatile 类型定义在单片机的 C 语言编程中之所以如此重要,是因为它可以告诉编译器的优化处理器,这些变量是实实在在存在的,在优化过程中不能无故消除。假定程序中定义了一个变量,并对其进行了一次赋值,但随后就再也没有对其进行任何读写操作,如果是非 volatile 型变量,则优化后的结果是该变量将有可能被彻底删除以便节约存储空间。另外一种情形是在使用某一个变量进行连续的运算操作时,该变量的值将在第一次操作时被复制到中间临时变量中,如果它是非 volatile 型变量,则紧接其后的其他操作将有可能直接从临时变量中取数以提高运行效率,显然这样做之后,对于那些随机变化的参数来说就会出问题。只要将变量定

义成 volatile 类型后，编译后的代码就可以保证每次操作时直接从变量地址处取数。

2. const——常数型变量声明

如果变量定义语句前冠以 const 类型修饰词，那么所有这些变量就将成为常数，且在程序运行过程中不能对其修改。除了位变量之外，其他所有基本类型的变量或高级组合变量都将被存放在程序空间(ROM 区)中以节约数据存储空间。显然，被定义在 ROM 区中的变量是不能再在程序中对其进行赋值修改的，这也是 const 的本来意义。实际上，这些数据最终都将以 retlw 的指令形式存放在程序空间，但 PICC 会自动编译生成相关的附加代码，以便从程序空间读取这些常数，编程员无须太多操心。例如：

```
const unsigned char name[] = "how are you"; //定义一个常量字符串
```

如果定义了 const 类型的位变量，那么这些位变量还是被放置在 RAM 中，但程序不能对其赋值修改。其实，不能修改的位变量没有太多的实际意义，相信大家在实际编程时不会大量用到。

3. persistent——非初始化型变量声明

按照标准 C 语言的做法，程序在开始运行前首先要把所有定义的但没有预置初值的变量全部清零。PICC 会在最后生成的机器码中加入一小段初始化代码来实现这一变量清零操作，且这一操作将在 main 函数被调用之前执行。问题是作为一个单片机的控制系统，有很多变量是不允许在程序复位后被清零的。为了达到这一目的，PICC 提供了 persistent 修饰词以声明此类变量无须在复位时自动清零。编程员应该自己决定程序中的哪些变量必须声明成 persistent 类型，而且必须自己判断何时需要对其进行初始化赋值。例如：

```
persistent unsigned char hour,minute,second; //定义时分秒变量
```

经常用到的是，如果程序经上电复位后开始运行，那么需要将 persistent 类型的变量初始化；如果是其他形式的复位，例如由看门狗引发的复位，则无须对 persistent 类型变量做任何修改。PIC 单片机内提供了各种复位的判别标志，用户程序可依具体设计灵活处理不同的复位情形。

2.8 指　针

PICC 中指针的基本概念与标准 C 语法中的没有太大差别。但是在 PIC 单片机这一特定的架构上，指针的定义方式还是有几点需要特别注意。

1. 指向 RAM 的指针

如果是汇编语言编程，实现指针寻址的方法肯定就是用 FSR 寄存器，PICC 也不例外。为

了生成高效的代码，PICC 在编译 C 源程序时将指向 RAM 的指针操作最终用 FSR 来实现间接寻址。这样就势必产生一个问题：FSR 能够直接连续寻址的范围是 256 字节(bank0/1 或 bank2/3)，要想覆盖最大 512 字节的内部数据存储空间，又该如何定义指针呢？PICC 还是将这一问题留给编程员自己解决，即在定义指针时必须明确指定该指针所适用的寻址区域，例如：

```
unsigned char * ptr0;                    //①定义覆盖 bank0/1 的指针
bank2 unsigned char * ptr1;              //②定义覆盖 bank2/3 的指针
bank3 unsigned char * ptr2;              //③定义覆盖 bank2/3 的指针
```

上面定义了三个指针变量，其中指针①没有任何 bank 限定，默认就是指向 bank0 和 bank1；指针②和③一个指明了 bank2，另一个指明了 bank3，但实际上两者是一样的，因为一个指针可以同时覆盖两个 bank 的存储区域。另外，上面三个指针变量自身都存放在 bank0 中。稍后将介绍如何在其他 bank 中存放指针变量。

既然定义的指针有明确的 bank 适用区域，那么在对指针变量赋值时就必须实现类型匹配。

下面的指针赋值将产生一个致命错误：

```
unsigned char * ptr0;                    //定义指向 bank0/1 的指针
bank2 unsigned char buff[8];             //定义 bank2 中的一个缓冲区
```

程序语句是：

```
ptr0 = buff;                             //错误！试图将 bank2 内的变量地址赋给指向 bank0/1 的指针
```

若出现此类错误的指针操作，PICC 在最后链接时会告知类似于下面的信息：

```
Fixup overflow in expression (...)
```

同样的道理，若函数调用时使用指针作为传递参数，也必须注意 bank 作用域的匹配，而这点往往容易被忽视。假定下面的函数实现发送一个字符串的功能：

```
void SendMessage(unsigned char * );
```

那么被发送的字符串必须位于 bank0 或 bank1 中。如果还要发送位于 bank2 或 bank3 内的字符串，那么必须再另外单独写一个如下函数：

```
void SendMessage_2(bank2 unsigned char * );
```

这两个函数从内部代码的实现来看可以一模一样，但传递的参数类型不同。根据笔者的应用经验和体会，当看到“Fixup overflow”的错误指示时，几乎可以肯定是指针类型不匹配的赋值所至，这时，应重点检查程序中有关指针的操作。

2. 指向 ROM 常数的指针

如果一组变量是已经被定义在 ROM 区的常数，那么指向它的指针可以这样定义：

```
const unsigned char company[] = "Microchip";      //定义 ROM 中的常数
const unsigned char * romPtr;                     //定义指向 ROM 的指针
```

程序中可以对上面的指针变量赋值和实现取数操作：

```
romPtr = company;                                 //指针赋初值
data = * romPtr ++ ;                              //取指针指向的一个数，然后指针加 1
```

反过来，下面的操作将是一个错误，因为该指针指向的是常数型变量，不能赋值。

```
* romPtr = data;                                  //往指针指向的地址写一个数
```

3. 指向函数的指针

在对单片机编程时，函数指针的应用相对较少，但作为标准 C 语法的一部分，PICC 同样支持函数指针调用。如果对编译原理有一定的了解，就应该明白在 PIC 单片机这一特定的架构上实现函数指针调用的效率是不高的：PICC 将在 RAM 中建立一个调用返回表，而真正的调用和返回过程是靠直接修改 PC 指针来实现的。因此，除非特殊算法的需要，建议大家尽量不要使用函数指针。

4. 指针的类型修饰

前面介绍的指针定义都是最基本的形式。与普通变量一样，指针定义也可以在前面加上特殊类型的修饰关键词，例如 persistent、volatile 等。考虑指针本身还要限定其作用域，因此 PICC 中的指针定义初看起来显得有点复杂；但只要了解了各部分的具体含义，理解了一个指针的实际用途，指针的定义就变得很直接了。

(1) bank 修饰词的位置含义

前面介绍的一些指针有的作用于 bank0/1，有的作用于 bank2/3，但它们本身的存放位置全部在 bank0。显然，在一个程序设计中指针变量将有可能定位在任何可用的地址空间，这时，bank 修饰词出现的位置就是一个关键，例如：

```
//定义指向 bank0/1 的指针，指针变量位于 bank0 中
unsigned char * ptr0;
//定义指向 bank2/3 的指针，指针变量位于 bank0 中
bank2 unsigned char * ptr0;
//定义指向 bank2/3 的指针，指针变量位于 bank1 中
bank2 unsigned char * bank1 ptr0;
```

从上例可以看出规律:前面的 bank 修饰词指明了此指针的作用域;后面的 bank 修饰词定义了此指针变量自身的存放位置。只要掌握了这一法则,就可以定义任何作用域的指针,且可以将指针变量放于任何 bank 中了。

(2) volatile、persistent 和 const 修饰词的位置含义

实际上,volatile、persistent 和 const 关键词出现在前后不同位置上的含义规律与 bank 修饰词的含义是一致的。例如:

```
//定义指向 bank0/1 易变型字符变量的指针,指针变量位于 bank0 中,且自身为非易变型
volatile unsigned char * ptr0;
//定义指向 bank2/3 非易变型字符变量的指针,指针变量位于 bank1 中,且自身为易变型
bank2 unsigned char * volatile bank1 ptr0;
//定义指向 ROM 区的指针,指针变量本身也是存放于 ROM 区的常数
const unsigned char * const ptr0;
```

亦即出现在前面的修饰词,其作用对象是指针所指处的变量;出现在后面的修饰词,其作用对象是指针变量自己。

2.9 函 数

当编写较大程序时,一般分为若干个小模块程序,每个模块实现一个特定的功能。在C 语言中,模块的功能是用函数来实现的,就是相当于一个小程序模块。一般来说,一个 C 语言程序可由一个主函数和若干子函数构成,主函数调用子函数,子函数之间可以相互调用,同一个函数能被调用多次。

PICC 和 PICC18 函数的语法和调用方法与标准 C 语言的基本一致。

2.9.1 函数的参数传递

与标准 C 语言一致,参数传递过程如下:

① 如果函数只有一个参数,并且是单字节,那么它通过 W 寄存器实现参数传递。

② 如果函数只有一个参数,长度大于 1 字节,那么它会通过被调函数的参数区域传递参数;如果参数是连续的,那么它也是通过被调函数的参数区域传递参数。

③ 如果函数参数多于 1 个,而且函数的第一个参数只有 1 字节,那么它将经被调函数的自动变量区传递参数,余下的连续参数将经由被调函数的参数区域传递。

④ 在使用省略号"…"这一可变参数列表时,调用函数会建立可变参数列表,然后把指向可变参数列表的指针传过去。

2.9.2 函数返回值

与标准 C 语言一致，通常通过调用函数从调用函数中得到一个返回值，该值主要分为 8 位、16 位和 32 位返回值及结构体返回值。

1. 8 位返回值

8 位返回值(字符型、无符号字符型及指针)通过 W 寄存器返回，例如函数：

```
char return_8(void)
{
    return 1;          //返回 0 给 W 寄存器
}
```

将会生成以下代码：

```
movlw 1
return
```

2. 16 位和 32 位返回值

16 位和 32 位返回值(整型、无符号整型、无符号短整型、一些指针、长整型、无符号长整型、浮点型及双精度型)都以最少的有效字保存在最低的存储单元中，例如函数：

```
char return_16(void)
{
    return 0x1234;     //返回 0x1234
}
```

将会生成以下代码：

```
movlw low1234h
movwf btemp            //低字节 0x34 返回 btemp
movlw high1234h
movwf btemp + 1        //高字节 0x12 返回 btemp + 1
return
```

3. 结构体返回值

4 字节及更小的合成返回值(结构体和联合体)与 16 位和 32 位返回值的返回方式一致。但大于 4 字节长度的结构体或联合体的返回值会被复制到结构块(struct psect)中。例如函数：

```
struct fred
{
     int ace[4];
}
struct fred return_struct(void)
{
     struct fred wow;
     return wow;
}
```

将会生成以下代码：

```
movlw ? a_return_struct + 0
movwf 4
movlw structret
movwf btemp
movwf 8
GLOBAL structcopy
lcall structcopy
return
```

2.9.3 调用层次的控制

基本级系列 PIC 单片机的硬件堆栈深度只有 2 级，中档系列 PIC 单片机的硬件堆栈深度为 8 级，考虑中断响应需占用一级堆栈，所以函数调用嵌套的最大深度不要超过 7 级，编程员必须自己控制子程序调用时的嵌套深度以符合这一限制要求。

PICC 在最后编译链接成功后生成一个链接定位映射文件（*.map），在此文件中有详细的函数调用嵌套指示图“call graph”，例如：

```
Call graph:
 * _main size 0,0 offset 0
_RightShift_C
 *  _Task size 0,1 offset 0
lwtoft
ftmul size 0,0 offset 0
ftunpack1
ftunpack2
ftadd size 0,0 offset 0
ftunpack1
ftunpack2
ftdenorm
```

上面所举的信息表明，整个程序在正常调用子程序时嵌套最多为 2 级（没有考虑中断）。因为 main 函数不可能返回，故其不用计算在嵌套级数中。其中有些函数调用是编译代码时自动加入的库函数，这些函数调用从 C 源程序中无法直接看出，但在此嵌套指示图上则一目了然。

基本级的 PIC 单片机通过汇编跳转指令、查表及转移表等方式实现函数的调用。PICC 和 PICC18 通常通过 CALL 指令实现 C 函数的调用。由于中级和高级 PIC 单片机有多级硬件堆栈，所以允许更深的函数嵌套调用。在调用函数时，都不使用查表方式。

任何函数调用时，编译器都将自动选择存储区 bank0。为了达到这一目的，它会在必要时嵌入适当的指令。所以，任何可以被 C 调用的汇编函数，在返回时都必须确保选择了存储区 bank0。

PICC 在编译时将严格进行函数调用时的类型检查。一个良好的编程习惯是在编写程序代码前先声明所有用到的函数类型。例如：

```
void Task(void);
unsigned char Temperature(void);
void BIN2BCD(unsigned char);
void TimeDisplay(unsigned char, unsigned char);
```

这些类型声明确定了函数的入口参数和返回值类型，这样编译器在编译代码时就能保证生成正确的机器码。建议大家在编写一个函数的源代码时，立即将此函数的类型声明复制到源文件的起始处或者专门的包含头文件中，然后再在每个源程序模块中引用。

2.9.4 中断函数的实现

PICC 可以实现 C 语言的中断服务程序。中断服务程序有一个特殊的定义方法，即：

```
void interrupt ISR(void);
```

其中的函数名 ISR 可以改成任意合法的字母或数字组合，但其入口参数和返回参数类型必须是 void 型，亦即没有入口参数和返回参数，且中间必须有一个关键词 interrupt。中断函数可以放置在源程序的任意位置。因为已有关键词 interrupt 声明，所以 PICC 在最后进行代码链接时会自动将其定位到 0x0004 中断入口处，实现中断服务响应。编译器也会实现中断函数的返回指令 retfie。一个简单的中断服务示范函数如下：

```
void interrupt ISR(void)                    //中断服务程序
{
    if (T0IE && T0IF)                       //判 TMR0 中断
    {
        T0IF = 0;                           //清除 TMR0 中断标志
                                            //在此加入 TMR0 中断服务
```

```
    }
    if (TMR1IE && TMR1IF)                    //判 TMR1 中断
    {
        TMR1IF = 0;                          //清除 TMR1 中断标志
                                             //在此加入 TMR1 中断服务
    }
}                                            //中断结束并返回
```

PICC 会自动加入代码以实现中断现场的保护，并在中断结束时自动恢复现场，所以编程员无须像编写汇编程序那样加入中断现场保护和恢复现场的额外指令语句。但当在中断服务程序中需要修改某些全局变量时，是否要保护这些变量的初值将由编程员自己决定和实施。

用 C 语言编写中断服务程序必须遵循高效的原则：

① 代码尽量简短，中断服务强调的是一个“快”字。

② 避免在中断服务程序内使用函数调用。虽然 PICC 允许在中断里调用其他函数，但为了解决递归调用的问题，此函数必须为中断服务独家专用。既如此，不妨把原本要写在其他函数内的代码直接写在中断服务程序中。

③ 避免在中断服务程序内进行数学运算。数学运算将很有可能用到库函数和许多中间变量，就算不出现递归调用的问题，仅在中断入口和出口处为了保护和恢复这些中间临时变量就需要大量开销，这样会严重影响中断服务的效率。

中档系列 PIC 单片机的中断入口只有一个，因此整个程序中只能有一个中断服务程序。

2.9.5 标准库函数

PICC 提供了较完整的 C 标准库函数支持，其中包括数学运算函数和字符串操作函数。在程序中使用这些现成的库函数时需要注意的是入口参数必须在 bank0 中。

如果需要用到数学函数，则应在程序前添加“#include〈math.h〉”文字以包含数学运算函数的头文件；如果要使用字符串操作函数，就需要添加“#include〈string.h〉”文字以包含字符串操作函数的头文件。在这些头文件中提供了函数类型的声明。通过直接查看这些头文件就可以知道 PICC 提供了哪些标准库函数。

C 语言中常用的格式化打印函数“printf/sprintf”用在单片机的程序中时需要特别谨慎。printf/sprintf 是一个非常大的函数，一旦使用，程序代码长度就会增加很多。除非是在编写试验性质的代码时，才可以考虑使用格式化打印函数以简化测试程序；而在最终产品设计中都是自己编写最精简的代码来实现特定格式的数据显示和输出。其实，在单片机应用中输出的数据格式都相对简单而且固定，实现起来应该很容易。

对于标准 C 语言的控制台输入(scanf)/输出(printf)函数，PICC 需要用户自己编写其底层函数 getch()和 putch()。在单片机系统中实现 scanf/printf 本来就没有太多意义，如果一

定要实现，只要编写好特定的 getch()和 putch()函数，就可以通过任何接口来输入或输出格式化的数据了。

2.10 ＃pragma 伪指令

在阅读一些 PIC 单片机的源程序开始段时，会发现有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常将这些特殊指令助记符称为伪指令。其中，可以用一些特定编译时间指令修改编译器的运行特性，PICC 和 PICC18 中可以使用 ANSI 标准指令中的＃pragma 语句。＃pragma 语句的格式如下：

```
＃pragma keyword options
```

其中，keyword 是关键字，options 是选项式中的关键词，参见表 2－3，其中，有些关键词后带有选项。

表 2－3 ＃pragma 伪指令

关键字	含 义	例 子
interrupt_level	允许主程序调用中断函数	＃pragma interrupt_level 1
jis	允许对 JIS 字符进行操作	＃pragma jis
nojis	禁止对 JIS 字符进行操作(默认状态)	＃pragma nojis
printf_check	允许打印格式符校验	＃pragma printf_check const
psect	重命名编译器定义的程序块	＃pragma psect text＝mytext
regsused	指定在中断中使用的寄存器	＃pragma regsused w
switch	指定根据开关状态生成代码	＃pragma switch direct

1. ＃pragma interrupt_level 伪指令

当中断程序和主程序或另一个中断程序都在调用同一个函数时，链接器就会报错。但是，在编译时，使用＃pragma interrupt_level 伪指令就可以将中断函数分级。PICC 提供了 2 级中断，而且编译器会认为同一级的任何中断函数都是独立的。但由于中级 PIC 只支持一个中断级，所以，这种独立性必须由用户保证，因此编译器并不能控制它们的优先级别。每一个中断程序可以指定一个中断级别，即 0 或 1。

当任何非中断函数分别被中断函数和主函数调用时，可以使用＃pragma interrupt_level 伪指令来规定它们绝不会被多级中断函数调用。同时，也可避免函数被多个调用所包含而引起的链接器报错。通常，可以通过在调用函数前屏蔽总中断来达到上述目的，在被调用函数内屏蔽中断是不可行的。例如：

```
/*非中断函数分别被中断函数和主函数调用*/
#pragma interrupt_level 1
void bill(){
    int i;
    i=12;
}
/*2个中断函数调用同一个非中断函数*/
#pragma interrupt_level 1
void interrupt fred(void)
{
    bill();
}
#pragma interrupt_level 1
void interrupt joh()
{
    bill();
}
main()
{
    bill();
}
```


2. #pragma jis 和 #pragma nojis 伪指令

当程序中包含用 JIS 编码的一些日语字符串或者其他语种的双字节字符串时，可以使用 #pragma jis 伪指令来处理这些字符串；如果字符串的前半字节和后半字节间不使用续行符“\”，则使用 #pragma jis 伪指令的效果会更好。#pragma nojis 伪指令则禁止使用这种操作，即不处理 JIS 字符，默认方式下 JIS 字符是禁止操作的。

3. #pragma printf_check 伪指令

某些库函数可以接受在一个字符串后跟几个参数变量，这一点与 printf()函数相同。虽然这种格式的字符串在运行时才说明，但在编译时编译器将检查它是否冲突。这个伪指令可实现对已命名的函数进行检查。例如，系统头文件〈stdio.h〉中包括 #pragma printf_check (printf)const 伪指令，就实现了对 printf()函数的检查。可以使用该伪指令对任何用户定义的可以接受打印格式的字符串的函数进行检查。在函数名后放置限定词可以自动转换指针类型指向变量参数列表中的变量。

4. #pragma psect 伪指令

通常，编译器生成的目标代码被分解成标准程序块，并生成相应的文档。这对大多数应用

来说是最好的;但是,当要求配置一些特定的存储器时,就必须在不同的程序块中重新定位变量和代码。使用#pragma psect 伪指令可以使编译器重新分配任何标准 C 程序块的代码和数据。例如,如果希望让一 C 源文件的所有未初始化全局变量放置于名为 otherram 的程序块中,则可以使用以下伪指令来实现:

```
#pragma psect bass = otherram
```

这个语句告诉编译器,在通常情况下放置在 bass 程序块里的任何内容,此时都应该放置在 otherram 程序块中。

若将 text 程序块放置在另一个代码块中,则可以使用以下预处理伪指令重新指定放置代码的程序块:

```
#pragma psect text = othercode
```

其中,othercode 是新建并填充的程序块名字。

如果希望将某模块中的多个函数放置在各自程序块中,则可以在每个函数前使用这个伪指令,即:

```
#pragma psect text = othercode0
void function(void)
{
//函数定义等
}
#pragma psect text = othercode1
void another(void)
{
//函数定义等
}
```

例中为函数 function()定义了一个程序块 othercode0,同时也为函数 another()定义了另一个程序块 othercode1。

任何特定的程序块在特定的源文件中都应该仅重新定位一次,而且所有程序块的重定位都应该放置在源文件的顶部,位于所有#include 指令的下面和其他一些定义的上面。例如,为说明未初始化的变量组放置在 otherram 程序块中,应该按以下方法定义:

```
//file OTHERRAM.C
#pragma psect bss = otherram
char buffer[5];
int var1,var2,var3;
```

凡需要存取定义在 otherram.c 中的任何变量,程序文件都应该包含以下头文件: