PIC16F87X单片机实用软件与接口技术

——C语言及其应用

刘和平　等编著

北京航空航天大学出版社
http://www.buaapress.com.cn

PIC16F87X 单片机实用软件与接口技术

——C 语言及其应用

刘和平　等编著

北京航空航天大学出版社

http://www.buaapress.com.cn

内 容 简 介

本书讨论了 PIC16F87X 系列单片机的 C 语言程序编程方法。书中介绍了大量的 C 语言程序例程，涉及到 PIC16F87X 单片机的各个功能模块的编程应用；给出了应用实例的电路原理图和源程序清单，所有程序均在实验板上调试通过，并配有光盘。

本书与北京航空航天大学出版社出版的《PIC16F87X 单片机实用软件与接口技术——汇编语言及其应用》构成姊妹篇。该姊妹篇是单片机开发者和初学者的很好的参考书，也可作为大学本科学生单片机原理及应用课程的实验指导书。

图书在版编目(CIP)数据

PIC16F87X 单片机实用软件与接口技术——C 语言及其应用 刘和平等编著. —北京：北京航空航天大学出版社，2002.1

ISBN 7-81077-169-8

Ⅰ.P… Ⅱ.刘… Ⅲ.C 语言-程序设计

Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2002)第 019673 号

PIC16F87X 单片机实用软件与接口技术

——C 语言及其应用

刘和平 等编著

责任编辑 王 瑛

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(100083) 发行部电话：(010)82317024 传真：(010)82328026

http://www.buaapress.com.cn

E-mail:pressell@publica.bj.cninfo.net

河北省涿州市新华印刷厂印装 各地书店经销

开本：787×960 1/16 印张：17 字数：380 千字

2002 年 1 月第 1 版 2002 年 1 月第 1 次印刷 印数：5 000 册

ISBN 7-81077-169-8/TP·093 定价：32.00 元

本书编委会

主编：刘和平

编委：沈成瑜　杨利晖　杨立勇

崔　晶　吴　俊　黄开长

前　言

目前市场上应用广泛的是8位单片机。美国微芯公司作为世界上8位单片机的第2大生产商，推出了CMOS 8位PIC系列单片机。该系列芯片采用精简指令集(RISC)、哈佛总线结构、2级流水线取指令方式，具有实用、低价、指令集小、简单易学、低功耗、高速度、体积小、功能强等特点，体现了单片机发展的一种新趋势，深受用户的欢迎，已经逐渐成为世界单片机的新潮流。

本书内容以微芯公司的采用14位RISC指令集的中级产品PIC16F87X为主。由于芯片内含A/D、内部EEPROM存储器、比较输出、捕捉输入、PWM输出、I^2C和SPI接口、异步串行通信(USART)接口、LCD驱动、FLASH程序存储器读写等许多功能，对初学者来说有一定的难度；而且这方面的参考资料又很少，更没有介绍应用实例和应用程序库的书，给广大的使用者带来了困难。

对于一般涉及单片机开发和应用工作的人员，使用汇编语言编制短小程序较容易；但要编写大程序或者多人合作编程，以及编程的后期维护，将是较困难的事情。

针对微芯公司的采用14位RISC指令集的中级产品PIC16F87X，微芯公司的第3方提供了几种C语言开发工具，HI_TECH PICC就是其中的一种。本书以HI_TECH PICC为基础，介绍PIC中级产品的C语言基本知识、软件开发环境、C语言函数库及C语言编程实例。

本书以重庆大学-美国微芯公司PIC单片机实验室开发的实验板为对象，以单片机的各种功能模块为线索，通过给出实验板的硬件连接方式和C语言编程实例进行讲解，从易到难，循序渐进，逐步深入。

全书共分15章。第1章，PICC C语言基础和特点；第2章，PICC的使用；第3章，PICC的库函数；第4章，PIC16F877单片机实验板介绍；第5章，PIC16F877的外围功能模块；第6章，模拟量输入与输出；第7章，秒表；第8章，通用同步/异步通信的应用；第9章，PIC16F87X在CAN通信中的应用；第10章，利用CCP模块设计频率计；第11章，交流电压测量；第12章，与PLC接口的4位LED数字显

示表；第 13 章，数控步进直流稳压电源；第 14 章，单片机控制的电动自行车驱动系统；第 15 章，液晶显示模块编程。

其中第 9,12 及 14 章与本书介绍的实验板联系不太紧密；其余章节都是在实验板的基础上开发出来的，只需少量外围电路即可构成实验系统。

编者还在北京航空航天大学出版社出版了《PIC16F87X 单片机实用软件与接口技术 · 汇编语言及其应用》、《PIC16F87X 数据手册》和在重庆大学出版社出版了《单片机原理及应用》。它们各有侧重，本书侧重于 C 语言编程方法和 C 语言应用编程方面。书中介绍了大量的程序例子，涉及到 PIC16F87X 单片机的各个功能模块的编程应用；给出了多个应用实例的电路原理图和源程序清单，所有程序均调试通过。本书可作为大学本科学生单片机原理及应用课程的实验指导书，对单片机开发者来说也是一本很好的软件开发参考书。

在成书过程中，得到了电力电子与电力传动系李远树、郑连清、郑群英、巫宣文等老师的大力协助和支持，他们编写了部分章节，并做了校对录入以及实验板制作工作。在此表示感谢。

在这里还要感谢微芯公司提供的大力支持。

限于编者的水平，书中难免存在错误和不当之处，恳请读者批评指正。

编　者

2002 年 1 月于重庆大学

目　　录

第 2 章 PICC 的使用

第 3 章 PICC 的库函数

第 15 章 液晶显示模块编程

第 1 章　PICC C 语言基础和特点

PICC 具有许多特殊的性质，并且进行了 C 语言的扩展，从而可以更轻松地完成编程任务。本章提供了编辑器选项和一些有用的特点，读完本章之后，可以做到：

- 配置 I/O 接口程序，从而可利用<stdio.h>程序对硬件进行操作。
- 用 C 语言建立中断。
- 利用 C 语言进行 I/O 口编程。
- 通过内嵌或外部汇编语言程序建立起 C 和汇编之间的联系。

1.1　PICC 与 ANSI C 标准的区别

PICC 与 ANSI C 标准的区别在于函数的递归调用。这是因为 PIC 受硬件的限制，没有堆栈，内存的数量也有限；所以不支持递归调用。

1.2　处理器支持

PICC 支持的处理器范围如表 1.1 所列。只需在 LIB 目录下编写 picinfo.ini 文件，其他处理器就可以加进来。picinfo.ini 文件分为低等、中等及高等 3 部分，用户定义的处理器可以放到文件的最后。文件的开头部分解释怎样定义处理器。

表 1.1　PICC 支持的处理器范围

Baseline Processors	Midrange Processors			High - End Processors
PIC12C508	PIC12C671	PIC16C711	PIC16F81	PIC17C42
PIC12C508A	PIC12C672	PIC16C712	PIC16F84A	PIC17C42A
PIC12C509	PIC12CE673	PIC16C715	PIC16F627	PIC17C43
PIC12C509A	PIC12CE674	PIC16C716	PIC16F628	PIC17C44
PIC12CE518	PIC14000	PIC16C717	PIC16F870	PIC17C752
PIC12CE519	PIC16C554	PIC16C72	PIC16F871	PIC17C756
PIC12CR509A	PIC16C554A	PIC16C72A	PIC16F872	PIC17C756A
PIC16C505	PIC16C556	PIC16C73	PIC16F873	PIC17C762
PIC16C52	PIC16C556A	PIC16C73A	PIC16F874	PIC17C766
PIC16C54	PIC16C558	PIC16C73B	PIC16F876	PIC17CR42
PIC16C54A	PIC16C558A	PIC16C74	PIC16F877	PIC17CR43

续表 1.1

Baseline Processors	Midrange Processors			High – End Processors
PIC16C54B	PIC16C61	PIC16C71A		
PIC16C54C	PIC16C62	PIC16C74B		
PIC16C55	PIC16C62A	PIC16C745		
PIC16C55A	PIC16C62B	PIC16C76		
PIC16C56	PIC16C620	PIC16C77		
PIC16C56A	PIC16C620A	PIC16C770		
PIC16C57	PIC16C621	PIC16C771		
PIC16C57C	PIC16C621A	PIC16C773		
PIC16C58	PIC16C622	PIC16C774		
PIC16C58A	PIC16C622A	PIC16C765		
PIC16C58B	PIC16C63	PIC16C84		
PIC16CR54A	PIC16C63A	PIC16C923		
PIC16CR54B	PIC16C64	PIC16C924		
PIC16CR54C	PIC16C64A	PIC16C99		
PIC16CR56A	PIC16C641	PIC16CE623		
PIC16CR57B	PIC16C642	PIC16CE624		
PIC16CR57C	PIC16C65	PIC16CE625		
PIC16CR58A	PIC16C65A	PIC16CR62		
PIC16CR58B	PIC16C65B	PIC16CR63		
PIC16HV540	PIC16C66	PIC16CR64		
	PIC16C661	PIC16CR65		
	PIC16C662	PIC16CR72		
	PIC16C67	PIC16CR83		
	PIC16C71	PIC16CR84		
	PIC167C710	PIC16F83		

1.3 配置设置

PIC 处理器的配置可以用_config 宏进行设置，形式如下：

```
#include <pic.h>
__CONFIG( x);
```

其中 x 是配置语句。在头文件中适当地定义了不同的标号，可以利用这些标号使用芯片的某一特性。下面是一个 PIC16C5X 的例子：

```
__CONFIG(WDTDIS & XT & UNPROTECT);
```

这条语句的作用是关闭看门狗定时器，使用 XT 晶振和不保护程序代码。在下载程序之前，一定要先检查__CONFIG 宏中所有的配置位以及头文件是否正确；也可以通过 ICD 的“OPTION”对话框配置。

注意，所有的个人选项都被加到一起，所有未被宏选择的位将保持不编程状态。应该在下载时，保证所有选择的位都正确，从而使各部分合理地工作。更详细的资料可以查看有关的 PIC 数据手册。

宏__CONFIG 不产生可执行代码，使用时可以将其放在函数的外部。

1.4　ID 区域

一些 PIC 芯片在可编程存储区间外，还有一些其他区域。这些区域可以用来存储下载信息，例如 ID 号。宏__IDLOC 可以用来将所需数据放到这一区域，它使用的格式如下：

```
#include <pic.h>
__IDLOC(x);
```

这里 x 为一串十六进制数，指向相应的 ID 区域。其中只有低 4 位 ID 数被下载编程；因此，对于

```
__IDLOC(15F0);
```

将有 4 位十进制数进入 ID 区域，即 1，5，15(F)及 0。最基本的 ID 区域的确定，是由指令 idloc 完成的。它将自动根据所选择的芯片安排适当的地址。

1.5　EEPROM 数据

有些芯片支持外部 EEPROM 数据。这可以通过宏__EEPROM_DATA 以十六进制文件的形式初始化 EEPROM 数据。用法如下：

```
#include <pic.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

这个宏需要 8 个参数，分别表示 8 个数据值。每个数值为 1 B。不需要使用的数据则应该用参数 0 代替。这个宏可以被多次调用，从而定义所要求数量的 EEPROM 数据。宏应该放到所有函数的外部。每个数据存储时，都占用 2 B，并且第一个字节为 0，存储为十六进制的形式。

这个宏在程序运行时，不能将数据写入 EEPROM 存储区；在下载编程时，宏 EEPROM_READ()，EEPROM_WRITE()及其他这类宏可以用来读或者写 EEPROM。

1.6　位指令

有时，PICC 要用到 PIC 的位操作指令。例如，当需要位取反操作以及要改变一个整型数中的某一位时，编译器将检测屏蔽值，以决定是否可以采用一条位指令实现同样的功能。

```
int foo;
foo  = 0x40;
```

将产生下面的指令：

```
bsf _foo,6
```

为了将一个整型数中的某一位置 1 或清 0，可以使用下面的宏定义：

```
#define bitset(var,bitno) ((var) |= 1 <<< (bitno))
#define bitclr(var,bitno) ((var) &= ~(1 <<< (bitno)))
```

为了执行与上面一样的操作，也可以采用如下的位设置宏：

```
bitset(foo,6);
```

1.7　支持数据类型

PICC 编辑器支持基本的 1，2 及 4 B 数据。所有多字节数据遵守的格式为低字节在前、高字节在后的原则，即一个数中低位字节存储在地址低的存储单元中。表 1.2 列出了 PICC 支持的数据类型及对应的大小和数学类型。其中，字符型的缺省值为无符号型；如果 PICC 采用了 SIGNED_CHAR 选项，则为有符号型。Double 型变量的缺省值为 24 位；如果选择 PIC - D32，则为 32 位。

表 1.2　PICC 数据类型

类　型	大小/bit	数学类型
Bit	1	二进制
Char	8	符号或无符号整型
Unsigned char	8	无符号整型
Short	16	符号整型
Unsigned short	16	无符号整型
Int	16	符号整型
Unsigned int	16	无符号整型
Long	32	符号整型
Unsigned long	32	无符号整型
Float	24	实　型
double	24	实　型

1.7.1　数进制及常数

整数格式具有不同的数进制。PICC 支持 ANSI 标准码制，同样的，C 代码也支持二进制常数。不同格式的表示方法如表 1.3 所列。其指明二进制和十六进制的字母有时不敏感，就像用字母指明十六进制数一样。

表 1.3　数据格式

进　制	格　式	例　子
二进制	0b+数值或 0B+数值	0b10101100
八进制	0+数值	0234
十进制	数　值	432
十六进制	0x+数值或 0X+数值	0x2F

一个整型常量的类型应该是可以装得下这个值、使之不溢出的、最小的数据类型。在常数后面加 L 或 l，可以使常数的数据类型为有符号或无符号长整型；如加后缀 u 或 U，则可使常数为无符号数据类型；如同时加 l 或 L 和 u 或 U，则表示无符号长整型。浮点常数加后缀 f 或 F 时，为浮点数据类型；否则，认为是双精度数据类型。在双精度数据类型后加后缀 l 或 L，则表示 PICC 中的双精度数据类型。

字符常数应加单引号(注意：此处的单引号为西文符号，不能用中文符号)表示，如'a'。字符常数的数据类型为字符型。PICC 不支持多字节字符常数。

字符串常数或常量需要加双引号(**注意**：此处的双引号为西文符号，不能用中文符号)表示，例如"hello world"。字符常量的数据类型为字符常量指针类型(const char *)，被存储在 ROM 中。如指定字符串常量为非常量字符指针，编辑器将产生一个警告。例如：

```
char * cp = "one";                  /* "one" in ROM, 产生警告
const char * ccp = "two";           // "two" in ROM
char ca[] = "two";                  // "two" 与上面不同
```

(**注意**：HI_TECH PICC 的注释可以用"//"号，也可以用"/* 注释内容 */")一个由字符串数组初始化的非常数数组，如上面最后一句所示，将产生一个 RAM 数组，这个数组在程序开始运行前，被字符串常数"two"(从 ROM 中复制而来)初始化；而其他方法表示的字符串常数将产生一个未命名的常数数组，由直接访问 ROM 获得。

除像上面最后一条语句所示的字符串将在 RAM 中被初始化外，对具有相同字母顺序的字符串，PICC 将采用相同的存储空间和标号；对于邻近的字符串(也就是 2 个字符串之间只有一个空格)，编辑器将自动将它们连接起来。也就有：

```
const char * cp = "hello world";
```

编辑器将使指针指向字符串"hello world"。

1.7.2　位数据类型

PICC 允许利用 bit 指令定义位变量。一个变量被定义为位变量，例如：

```
static bit init_flag;
```

它将被分配在可寻址位区域 rbit_n(n 表示存储区的序号)，并且只在这个模式或函数下有效；当在所有的函数外部使用下面的语句进行位定义时：

```
bit init_flag;
```

则 init_flag 全局有效。

位变量不能被定义为自变量，也不能作为函数参数；但函数的返回值可以是位变量。

位变量很多地方和无符号整型量一样，但位变量的值只能是 0 或 1。因此，提供了一个方便而有效的方法，以存储二进制位变量，而不需要浪费 RAM 空间。指针不能指向位变量，位变量也不能被静态初始化。需要对位进行操作时，只需采用简单的位指令即可，由此产生的用来访问位变量的代码是十分高效的。

所有位区间启动时都被清 0，但不是初始化。为了使一个位具有不为 0 的值，最好是在程序开始部分明确地将其初始化。

注意将一个很大的整型数赋给一个位时，只有最低位被使用。如果想根据其他值为 0 或非 0 来设置一个位为 0 或 1 时，应该采用下面的形式：

```
bitvar = other_var ! = 0;
```

位程序段是通过程序段指令标志位定义的。8 bit 将占用 1 B 的存储空间，在 map 文件中可以看到位程序段的长度为 8。这个长度的单位为 bit。

如果 PICC 采用了-STRICT 标志，则 bit 指令无效。

1.7.3 使用可位寻址寄存器

位变量可以很容易地和绝对变量(下一节介绍)声明结合使用，以访问某一特定地址的位。绝对位从 0(第 1 个字节的最低位)开始计数；因此，第 5 字节的第 3 位的绝对地址为 43(也就是 8 bit/B×5 B+3 bit)。

例如，为了访问 STATUS 寄存器中的低压保护位，STATUS 寄存器的绝对地址定义为 03h；因此应该定义位变量的绝对地址为 27：

```
static unsigned char STATUS @ 0x03;
static bit PD @ (unsigned)&STATUS * 8+3;
```

注意所有的标准寄存器和位都在头文件中被定义。只需将<pic. h>头文件包含进来，就可以访问 PIC 的寄存器；但在汇编时，需根据所选芯片包含适当的头文件。

1.7.4 8 bit 整型数据

PICC 支持有符号字符和无符号字符的 8 bit 整型数据。字符型的缺省值为无符号字符。如果 PICC 采用了-SIGNED_CHAR，则为有符号字符。有符号数为 8 bit 二进制补码，其范围是-128～+127；无符号字符为 8 bit 无符号整型，可表示 0～255 之间的整数。容易引起误解的是，C 语言的字符型仅仅为 ASCII 字符而设计。这是不对的，实际上，C 语言并没有保证字符型缺省设置时代表的是 ASCII 码。字符型是 4 种整型变量中最小的一种，其他方面和其他整型数据完全一样。

命名为字符型并不意味着只能代表字符。在 C 语言的表达式中，字符型变量可以和短整

型、整型及长整型混合使用。对于 PIC 来说，字符型有不同的用途，可作为 8 bit 整型数，用来存储 ASCII 字符；也可以用来访问输入/输出(I/O)口。无符号整型是 PIC 芯片最有效的数据类型，并且可以通过 PIC 指令直接访问。为了使程序最有效地执行并使生成的代码最少，建议尽量使用字符型变量。

1.7.5　16 bit 整型数据

PICC 支持 16 bit 整型数据。Int 和 short 为 16 bit 二进制补码表示的有符号整型，可表示 32 768～＋32 767 之间的整数。Unsigned int 和 unsigned short 为 16 bit 无符号整型，可表示 0～65 535 之间的整数。所有的 16 bit 整型数都采用 little endian 格式，即低位字节存储在低地址内存单元。Int 和 short 都为 16 bit 宽，因为这是 ANSC 标准允许的最小尺寸的整型数。这样选择 int 整型数的大小，是为了使之不违反 ANSC 标准。如果允许更小的 int 整型数，如 8 bit，将导致与 C 标准严重不一致。实际上字符型完全支持 8 bit 整型数，如果可能，尽量用字符型代替 int 整型数。

1.7.6　32 bit 整型数据

PICC 支持 32 bit 整型数据。Long 型为 32 bit 二进制有符号整型数，可以表示 －2 147 483 648～＋2 147 483 647之间的整数。Unsigned long 是 32 bit 无符号整型，可以表示 0～4 294 967 295 之间的整数。所有 32 bit 整型数都采用 little endian 格式，即低位字节存储在低地址内存单元。Long 和 unsigned long 为 32 bit 宽，这是 C 的 ANSI 标准所允许的最小的长整型。

1.7.7　浮点型

浮点型采用的是 IEEE 754 的 32 bit 格式或被修改的 IEEE 754 格式(缩短的)，即 24 bit 格式。

缩短的 24 bit 格式用于所有的浮点数。对于 double 型数据，其缺省值为 24 bit 浮点；但也可以选择 PICC－D24 选项，将之明确。如选择 PICC－D32，double 型数据用 32 bit 格式表示。

所有这些格式如表 1.4 所列，这里：

- 符号表示符号位。
- 指数为 8 bit，存储为≥127 的数(指数 0 存储为 127)。
- 尾数表示小数点右边的尾数。小数点左边有一个隐含位，一般情况为 1；当指数为 0 时，隐含位为 0。

浮点数的大小为：$(-1)^{sign} \times 2^{(exp-127)} \times 1.mantissa$。其中 mantissa 为尾数，sign 为符号位，exp 为指数。

表 1.4 浮点格式

格 式	符 号	指 数	尾 数
IEEE 754 32 bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
被修改的 IEEE 754 24 bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

下面是 IEEE 754 32 bit 格式和被修改的 IEEE 754 24 bit 浮点格式的例子，如表 1.5 所列。

表 1.5 浮点数举例

格 式	数 据	指 数	1.mantissa	十进制值
IEEE 754 32 bit	7DA6B69Bh	11111011b (251)	1.01001101011010011101b (1.302 447 676 659)	2.77000e+37
被修改的 IEEE 754 24 bit	42123h	10000100b	1.00100100011101b (1.142 395 019 531)	36.557

注意，表 1.5 中尾数的最高位为隐含位(即小数点左边一位)，如果指数不为 0，这一位都为 1。表 1.5 中的 32 bit 浮点数可以用如下方法计算：符号位为 0，基本指数为 251；因此实际指数为 251－127＝124。取尾数中小数点右边的二进制数，将它转换为十进制数，除以 2^{23} 得到 0.302 447 676 659，其中 23 为尾数的位数；再将尾数加 1，即得浮点数如下：

$$(-1)^0 \times 2^{124} \times 1.302\ 447\ 676\ 659 =$$

$$1 \times 2.126\ 764\ 793\ 256e+37 \times 1.302\ 447\ 676\ 659 \approx$$

$$2.77000e+37$$

1.8 绝对变量

采用@address 结构定义时，可以使全局或静态变量定位于绝对地址。例如：

```
volatile unsigned char Portvar @ 0x06;
```

定义了一个称为 Portvar 的变量，并使其定位于单元 06h。**注意**，汇编器并不保留任何存储空间，只是将变量等于地址。编辑器汇编之后，将产生下面的形式：

```
_Portvar  equ  06h
```

编辑器和连接器并不检测绝对变量是否和其他变量重复；所以程序员必须保证绝对变量被分配了唯一的地址，而地址没有作其他用途。

这种结构最主要的目的是，将一个微处理器寄存器等于一个 C 标识符的地址。为了将定义的绝对变量分配绝对地址，应该将绝对地址定义到一个程序区间段，并通知连接器将程序区间段放到规定的地址。

1.9 结构和联合

PICC 支持 1 B 以上的结构和联合数据类型。结构和联合可以自由地成为函数参数及返

回值。PICC 完全支持结构及联合指针。

1.9.1　结构变量

PICC 支持结构类型变量。当一个变量定义为结构类型时，它的所有成员都将访问这一变量。例如：

```
bank1 struct {
int number;
int * ptr;
} record;
```

在这个例子中，结构和它的成员及 ptr 将存放在存储区 1(bank1)中。类似的下面的结构被定义为常数型：

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

在这种情况下，结构将放到 ROM 中；但如果结构的个别成员被定义为常数型，而结构不是常数型，那么结构还是被放到 RAM 中，且这个结构为只读结构。

1.9.2　结构中的位段

PICC 支持结构位段。位段从位段的最低位开始存储。位段以 8 bit 为单位进行分配，第一个分配的字符为字节的最低位，位段总是分配 8 bit 单元。当一个位段被定义时，如果长度合适，将被分配到当前的 8 bit 的存储单元中；否则，将被分配到结构中的一个新的 8 bit 单元中。一个位段必须存储在同一个存储单元中，不能跨 2 个单元。例如，定义：

```
struct {
unsigned hi : 1;
unsigned dummy : 6;
unsigned lo : 1;
} foo @ 0x10;
```

将产生一个从地址 10h 开始占用 1 B 的结构变量。位段 hi 被定义为地址 10h 的第 0 位，lo 将被定义为 10h 单元的第 7 位。dummy 的最低位为 10h 单元的第 1 位，而 dummy 的最高位为 10h 单元的第 6 位。如果一个位段被定义，只是被安排了一个绝对地址，而并没有分配存储空间。

未命名的位段可用来增加可控寄存器中 2 个有用的位段之间的空间。例如，上面的 dum-

my位段不使用，则上面的结构可以这样定义：

```
struct {
unsigned hi : 1;
unsigned : 6;
unsigned lo : 1;
} foo @ 0x10;
```

1.10 ROM和RAM中的字符串

一个未命名的常数字符串总是被放在ROM中，而且只能通过常数指针访问。在下面的例子中，字符串“Hello world”为字符串常量，被存储在ROM中；因此要通过常数指针访问。

```
#define HELLO "Hello world"
SendBuff(HELLO);
```

当用字符串常数初始化非常数数组时，例如：

```
char fred[] = "Hello world";
```

产生一个RAM数组。它在启动运行时，被字符串“Hello world”初始化（从ROM中复制而来）；而用没有采用const定义的数组表示字符串时，必须直接访问ROM。

如果想将字符常数作为函数参数，或使之为指针类型，那么指针必须为一个常数字符指针（const char *），例如：

```
void SendBuff(const char * ptr)
```

或其他类似用法。这样就可以将指针指向ROM或RAM（只有中等系列单片机是这样，而低等系列单片机总是指向ROM），并且能够正确地从适当的地方取得数据。

1.11 常数型和可变型变量

PICC支持ANSI标准的const（常数）和volatie（可变）型变量。

常数型变量用来通知编辑器一个目标具有常数值，不能被修改。对任何试图修改被定义为const型的变量，编辑器都将产生警告。定义的const型变量将被放在ROM中的某一程序段。显然，const型变量只能在定义时被初始化，因为不允许在接下来的任何地方、采用任何代码对它进行修改。例如：

```
const int version = 3;
```

可变型变量用来通知编辑器，某一目标不能被保证在连续访问的条件下，其值不被改变。这种用法能够防止程序优化时，将认为明显多余的、定义为可变型的变量删除，因为这将影响程序的下载。所有的I/O口以及任何有可能在中断时被改变的变量，应该被定义为可变型变

量，例如：

```
volatile unsigned char P_A @ 0x05;
```

可变型变量与其他非可变型变量被访问的路径不同。例如，当 1 赋给非可变型变量时，这个变量先是被清 0，然后再依次增加；但对可变型变量进行相同的操作时，可变型变量将直接从 W 寄存器中取出 1，然后将它放到适当的地址中。

1.12　ROM 对象的存放及访问

ROM 对象包括字符串、常数及其他被定义为 const 的变量。不同的芯片在进行汇编时，ROM 对象的存储方式不一样。下面只对中等系列单片机进行说明。

中等系列单片机将 ROM 对象存储为 retlw 指令。这些对象包含在称为字符串(strings)或常数 n(constn)的程序存储段，其中 n 表示数。小于 256 B 的常数混合体(例如结构或数组)或基本的常数对象(例如 int 型变量)都被存放在 constn 程序存储段；其他对象存储在 strings 程序存储段。strings 程序存储段被明确地定位；而 constn 程序存储段必须利用 const 将其归类。

constn 程序存储段除了需要 retlw 指令外，还需在前面加上 addwf pc 指令(这也是为什么不能在 constn 存储段存放刚好 256 B 的原因)。这条指令被程序用来实现目标的间接访问。

1.13　特殊类型限定词

PICC 支持在 static 和 extern 型变量前加下列特殊的限定词：persistent，bank1，bank2 及 bank3，从而使这些变量存放到特定的地址空间。如果使用了 PICC 的-STRICT 选项，这些限定词变为__persistent，__bank1，__bank2 及__bank3。这些类型限定词同样适用于指针，而不能用于 auto 型变量。如果需要将它们用于函数的局部变量，必须和关键词 static 结合使用。例如，不能像下面这样写：

```
void test(void)
{
/* 错误!   */
persistent int intvar;
.. 其他代码 ..
}
```

因为 intvar 为 auto 型变量。为了将函数 test()中的局部变量 intvar 定义为 *persistent* 型，应该这样写：

```
static persistent int intvar;
```

1.13.1 persistent 类型限定词

作为缺省值，所有的没有明确初始化的 C 变量都在启动时被清 0。这和 C 语言定义是一致的；但也有例外，PICC 设计允许一些数据在复位或供电电压循环变化(开--关- 开)时，其值保持不变。

persistent 类型限定词用来使变量在启动时不被清 0。任何 persistent 变量与其他变量存储在不同的存储空间(例如 nvram 或 nvram_1 存储段)。

有一些库函数可用来检查和初始化 persistent 数据(见库函数)，并提供了 persistent 数据应用例子。

1.13.2 bank1，bank2 及 bank3 类型限定词

类型限定词 bank1，bank2 及 bank3 可将静态变量分别放到 RAM 的 bank1，bank2 及 bank3 存储区。低等系列芯片不支持这些类型限定词(因为低等系列芯片的 bank 只有 1 个分区)。

没有 bank0 类型限定词。如果不使用其他类型限定词，对象默认存放到 RAM bank0 中。所有动态(auto)对象都和函数参数一起放到 bank0 RAM 中。

定位于 bank3 RAM 中的无符号字符定义如下：

```
static bank3 unsigned char fred;
```

指向定位于 bank3 RAM 中的无符号字符的指针定义如下：

```
bank3 unsigned char * ptrfred;
```

指向定位于 bank3 RAM 中的无符号字符的指针，且指针放在 bank2 RAM 中的定义如下：

```
static bank 3 unsigned char * bank2 ptrfred;
```

1.14 指 针

指针的格式和使用方法根据处理器的系列而定。下面对中等和低等系列单片机的指针进行说明。

1.14.1 低等系列单片机指针

所有低等系列单片机指针都为 8 bit。它们的类型和特点如下：

- RAM 指针　通过 FSR 指针寄存器指向 RAM。
- const 指针　通过 RETLW 指令表指向 ROM。
- 函数指针　指向函数。函数是通过跳转而不是调用来执行的。一个跳转表被用来返

回调用函数。

1.14.2　中等系列单片机指针

中等系列单片机指针和低等系列单片机指针除以下几方面外，其余完全相同。

● RAM 指针　因为 8 bit 指针最多访问 256 B，所以 RAM 指针只能访问 bank0 和bank1 中的对象。

● bank2 指针和 bank3 指针　分别用来访问 bank2 和 bank3 中的 RAM。注意，中等系列单片机中的一个指针不可能访问 3 个或更多个 bank 中的 RAM，也不可能访问像 bank0 和 bank2，bank1 和 bank3 这样的组合。

● const 指针　中等系列单片机的 const 指针为 16 bit。它们既可以用来访问 ROM，又可以用来访问 RAM。如果 const 指针的高位不为 0，则为指向任意 bank 的指针；如果高位为 0，则指针可以用来访问整个 ROM 空间。一个 const 指针可以用来读 RAM，但不允许向 RAM 写数据。这种既可以访问 ROM 又可以访问 RAM 的指针，对与字符串有关的函数特别有用。这样作为函数参数的指针既可以用来访问 ROM，又可以访问 RAM。

● 函数指针　指向函数。函数是通过指针指向的地址被调用的。

1.14.3　类型限定词和指针的结合

const，volatile 及 persistent 同样可以用于指针，控制指针指向的对象性质。当使用这些限定词定义指针时，必须区分它们是用来修饰指针，还是用来修饰指针指向的变量。规则为：如果定义时修饰词在 * 的左边，它用来修饰指针指向的变量；如果修饰词在 * 的右边，则它用来修饰指针变量本身。以 volatile 为例进行说明：

```
volatile char * nptr;
```

定义了一个指向可变型变量的指针。因为限定词在 * 的左边，故限定词 volatile 修饰的是指针指向的对象。

如定义为：

```
char * volatile ptr;
```

则与上面完全不同。修饰词 volatile 在 * 的右边，因而它用来修饰指针变量 ptr，而不是指针指向的变量。最后，定义：

```
volatile char * volatile nnptr;
```

将产生一个指向可变型变量的可变型指针。

1.14.4　常数指针

当需要间接访问采用 const 限定词定义的对象时，应该用指向常数的指针。常数指针和一般的指针在每一个存储器模式下几乎相同，唯一的差别是，编辑器禁止通过常数指针写数。

那么，给出定义为：

```
const char * cptr;
```

语句：

```
ch = *cptr;
```

是合法的。而语句：

```
*cptr = ch;
```

是不合法的。在低等系列芯片中，由于常数型对象被存储在 ROM 中，因而只能访问 ROM；而中等系列芯片中，常数指针既可访问 ROM，又可访问 RAM。

1.15 工具定义特性

ANSI 标准有些部分具有工具定义特性。本节说明 PICC 编辑器在这方面的特性。

1.15.1 整型的移位

ANSI 标准中，将有符号整数右移(操作符为 >>)定义为工具。典型的，当变量右移一位时，移入结果最高位的值可以是 0，也可以与移位前最高位相同。后者相当于有符号扩展。

PICC 对所有有符号整型进行有符号扩展(如有符号字符型、有符号整型、有符号长整型)。这样，有符号整型数 0124h 右移一位得到 0012h，而 8024 右移一位得到 C012h。

右移无符号整型数总是将结果的最高位清 0。

左移(操作符为<<)有符号和无符号数总是将结果的最低位清 0。

1.15.2 整型数的除法和取模运算

当除法的任意一个操作数为负时，结果都有特殊性。表 1.6 列出了 PICC 根据操作数 1 和操作数 2 确定的整数除法结果的符号。

表 1.6 整型除法

操作数 1	操作数 2	商	余 数
+	+	+	+
−	+	−	−
+	−	−	+
−	−	+	−

当第 2 个操作数为 0 时(除数为 0)，结果总是 0。

1.16 C 的中断处理

编辑器允许 PIC 单片机不写任何汇编指令就进行中断处理。硬件中断可以直接调用采

用限定词 interrupt 的函数。编辑器将产生不同于其他函数的中断函数，生成保护和恢复任何寄存器的代码，通过在函数最后采用 retfie 指令，而不是 retlw 或 return 指令退出。（如果 PICC 采用了-STRICT 选项，则关键词 interrupt 变为_interrupt。如果 PICC 采用了-STRICT 选项，本书中的关键词 interrupt 都认为是_interrupt。）

一个中断函数必须被定义为 interrupt void 型，并且不能带参数，不能被 C 代码直接调用；但在一定条件下，可以调用其他的函数。

1.16.1　中等系列单片机中断函数

下面是一个中等系列单片机处理器的中断函数例子：

```
long tick_count;
void interrupt tc_int(void)
{
++tick_count;
}
```

由于中等系列单片机最多只有一个中断矢量，因此只能定义一个中断函数。中断矢量自动被设置为指向这一函数。

1.16.2　中断现场保护

无论中断什么时候发生，PIC 处理器只是将其 PC 值放到堆栈中保存，其他的寄存器和对象必须用软件保存。PICC 编辑器决定哪些寄存器和目标被中断函数使用，并将其适当地保存起来。

如果中断程序调用了其他函数，并且这些函数在中断函数前定义为相同的形式，那么这些函数用到的任何寄存器都将被保存起来。如果被中断调用的函数在调用前没有被编辑器发现，那么可能将出现最坏的情况，即所用的寄存器和目标都被保存起来。

PICC 不检查内嵌在中断函数中汇编代码的寄存器的使用；因此，如果中断函数中包含汇编代码，必须加上相应的汇编代码中将用到的寄存器单元，保存和恢复。

下面具体讲述中等系列单片机的现场保护。

中断函数中不需要用到的寄存器或目标代码被直接放到中断矢量中称为 intcode 的程序存储段中。

如果要求现场保护，那么执行现场保护的代码将被存放到中断矢量中称为 intenry 的程序存储段中。所有的需要保存的寄存器和目标都被放到特定的用来保存的内存中。

如果 W 寄存器要求被存放，将被存放到 bank0 中称为 intsave_0 的程序存储段中。如果处理器中的代码用到不止一个 RAM bank，要不影响 W 寄存器的交换 bank 0 是不可能的；因此每一个 RAM bank 中必须有一个 intsave_n 程序存储段（这里 n 表示 bank 的序号）。这些

内存的地址可以通过低 7 位区别开来。当中断发生时，W 寄存器将根据中断发生前处理器所在的 bank 被存取到相应的内存区域中。

如果 STATUS 寄存器要被存储，将被保存到位于 bank 0 的 intsave 程序存储段。

一些 C 代码，例如除法，可以调用使用了暂时 RAM 单元的汇编程序。如果在中断程序中使用了这些代码，它们将被独立的自动连接的程序保护起来。

1.16.3 恢复现场

编译器会在中断返回前自动恢复现场。中档 PIC 的恢复现场代码放置在 int_ret 程序段中，而高档 PIC 的恢复现场代码直接与中断程序放在一起。

1.16.4 中断等级

通常，编译器认为中断会在任意时刻出现；所以当中断程序和主程序或另一个中断程序都调用同一个函数时，链接器就会报错。但是，在编程时往往可以保证以上的事情不会发生，所以编译器支持中断等级功能。

编程时，使用 # pragma interrupt_level 伪指令，可以将中断函数分级。PICC 可提供 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(注意，由于中档 PIC 只支持一个中断级，所以在选用这些处理器时，使用多级中断是无意义的)；但是，这种独立性必须由用户保证，因为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——0 或者 1。

任何分别被中断函数和主函数调用的非中断函数，可以使用 # pragma interrupt_level 伪指令，规定它们决不会被 1 级或多级的中断函数调用；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。值得注意的是，此时必须保证主函数和中断函数不会在同一时刻调用同一个非中断函数。通常，可以通过在调用函数前屏蔽总中断，以达到上述目的；在被调函数内屏蔽中断是不可行的。

以下例程将说明中断级别的使用。其中，# pragma 伪指令只作用于紧跟其后的函数。在非中断函数前放置多个 # pragma interrupt_level 伪指令，可以规定该函数不会被多级中断调用。

```
/ * 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 = 23;
}
/ * 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( );
}
```

1.16.5　中断使能

PICC 提供 2 个直接控制全局中断使能位的函数，被定义在头文件<pic. h>中。其中，ei( )函数用于允许全局中断，di( )函数用于屏蔽全局中断。在中档 PIC 器件中，它们将影响 INTCON 寄存器的 GIE 位。例如：

```
ADIE = 1;       // A/D 中断允许
PEIE = 1;       //所有外设中断使能
ei( );          //全局中断使能
di( );          //全局中断屏蔽
```

1.17　C 语言和汇编语言的混合编程

通过使用 3 种不同的技术，可以实现 C 语言和汇编语言的混合编程。

1.17.1　外部汇编函数

PICC 中，函数可以完全用汇编语言编写，作为独立的. as 文件由汇编器(ASPIC)汇编后，再使用链接器将它合并成二进制映像。这种技术允许函数参数和返回值在 C 程序和汇编程序之间传递。为了访问外部函数，在调用 C 程序中首先要对外部函数进行 C extern 声明。例如，假设需调用由 PIC 的移位指令编写的汇编函数实现左移功能，必须首先声明：

```
extern char rotate_left(char);
```

该语句声明了一个外部函数，函数名为 rotate_left( )，返回值为 char 型，只有一个 char 型的函

数参数，代码由经过 ASPIC 独立汇编后的外部.as 文件提供。以下是该函数的全部汇编代码：

```
processor 16C84
PSECT text0,class=CODE,local,delta=2
GLOBAL _rotate_left
SIGNAT _rotate_left,4201
_rotate left
;Fred is passed in the W register — assign it
;to ? a_rotate_left.
movwf ? a_rotate_left
;Rotate left. The result is placed in the W register.
rlf ? a_rotate_left,w
;The return is already in the W register as required.
return
FNSIZE _rotate_left,1,0
GLOBAL ? a_rotate_left
END
```

在以上程序段中，汇编函数名由 C 中声明的函数名加下划线“_”构成。GLOBAL 伪操作符等同于 C 中的关键字 extern，SIGNAT 伪操作符用于迫使链接器在链接时进行例行的检查。

为了使汇编函数能正确运行，必须严格遵守函数参数传递和返回规则。此外，可被 C 调用的汇编函数在返回前必须确保选择存储区 bank0。

1.17.2 在汇编程序内访问 C 变量

通过在变量名前加下划线“_”，汇编程序可以直接访问 C 全局变量。例如，C 程序中定义的全局变量：

```
int foo;
```

在汇编程序中可以通过以下形式访问它：

```
GLOBAL  _foo
movwf   _foo
```

如果汇编程序包含在不同模块中，则在汇编程序中必须使用 GLOBAL 伪指令。如果从另一个模块的嵌入汇编程序中访问 C 变量，则 C 程序中必须使用 extern 对 C 变量进行声明。例如：

```
extern int foo;
```

只有当 C 程序中也需要访问 foo 变量，以上声明才有效；否则，必须使用 GLOBAL 伪指令。注

意，在访问不在存储区 bank0 的变量前，必须给出它们的存储区信息；否则，链接器将认为这是一个固定错误。例如，在 PIC16C77 中定义一个变量 fred 如下：

```
bank2 int fred;
```

用汇编语言对 fred 进行写操作，如下：

```
movlw 055h
bcf 3,5
bsf 3,6                    ;选择存储区 2
movwf _fred&07Fh
```

PIC16C77 的每个存储区共有 0x80 字节；因此在访问 RAM 时，它的指令是一个 7 bit 的字段地址。

如果对使用汇编程序访问 C 变量还有难解的地方，可以先用 C 语言编一条与所要实现的功能类似的程序，编译后再仔细研究编译器产生的程序代码，将会有所帮助。

1.17.3　#asm，#endasm 及 asm( )指令

除以上 2 种方式外，还可以使用 #asm，#endasm 及 asm( )，将 PIC 汇编指令嵌入到 C 程序中。#asm 和 #endasm 指令分别用于嵌入汇编程序块的开头和结尾；asm( )用于将单条汇编指令嵌入到编译器生成的代码中。例如，可以使用下例中的任何一种方法，将一条左移指令嵌入到 C 程序中：

```
#include <stdio.h>
unsigned char var;
void main(void)
{
var = 1;
#asm
rlf _var,f
#endasm
asm("rlf _var,f");
}
```

当使用内嵌汇编程序时，必须注意它们与编译器生成代码之间的相互影响。如果有疑问，可以选择 PICC-S 选项对程序进行编译，然后用编译器测试生成的汇编代码。

注意，#asm 和 #endasm 结构不是 C 程序的语法部分，所以不服从 C 的流程规则。例如，在 if 语句中使用 #asm 块，并试图让它正常工作是不可能的。只有 asm("")命令才可以在任何 C 结构(如 if，while，do 等)中嵌入汇编程序。因为它被解释成 C 语句，并且可以和所有的 C 控制流程相符。

1.18 程序链接

汇编代码(PICC-S 选项)和目标代码(PICC C 选项)生成完毕后,除非编译器接收到停止信号,它会自动调用链接器。

PICC 和 HPDPIC 的缺省值生成 Intel HEX 文件和 Bytecraft COD。如果使用-BIN 选项或者使用 PICC-O 选项规定输出文件格式为.bin 型,则编译器生成二进制映像。链接完成后,编译器会自动生成存储器使用映射表,显示被使用的地址、存储器总空间及编译代码使用的内存区域等;而位变量将会被独立显示出来。例如:

```
Memory Usage Map:
Program ROM  $0000 - $001A $001B (27) words
Program ROM  $07EE - $07FF $0012 (18) words
$002D ( 45) words total Program ROM
Bank 0 RAM  $0020    $0022 $0003 ( 3) bytes total Bank 0 RAM
Bank 1 RAM  $00A0 - $00A2 $0003 ( 3) bytes total Bank 1 RAM
Bank 0 Bits  $0118 - $0119 $0002 ( 2) bits total Bank 0 Bits
```

1.19 函数参数传递

函数参数的传递方式取决于被传参数的长度。

● 如果函数只有 1 个参数,而且是单字节,则它通过 W 寄存器实现传递参数。

● 如果函数只有 1 个参数,但是,它的长度大于 1 B,那么会通过被调函数的参数区域传递参数;如果参数是连续的,也是通过被调函数的参数区域传递。

● 如果函数参数多于 1 个,而且第 1 个函数参数只有 1 B,那么它将经被调函数的自动变量区传递,余下的连续参数将经由被调函数的参数区域传递。

● 在用缺省符号"…"定义可变参数列表的情况下,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。

例如以下 ANSI 格式函数:

```
void test(int a,int b,int c)
{ }
```

函数 test( )将从它的函数参数块中获取所有参数。按如下方式调用函数:

```
test( 0x65af,0x7288,0x080c);
```

会产生如下代码:

```
movlw 0AFh
```

```
movwf (((? _test))&7fh)
movlw 065h
movwf (((? _test+1))&7fh)
movlw 088h
movwf ((0+((? _test)+02h))&7fh)
movlw 072h
movwf ((1+((? _test)|02h))&7fh)
movlw 0C1h
movwf ((0+((? _test)+04h))&7fh)
movlw 08h
movwf ((1+((? _test)-04h))&7fh)
lcall (_test)
```

传递给函数的参数可以通过一个由问号“?”、下划线“_”及函数名加一个偏移量构成的标号获取。例如，在上面的代码中，传递给函数的第 1 个参数(值为 0x65af)保存在地址为“? _test”和“? _test+1”的内存单元中。

建议用 C 语言编写一条和汇编程序具有相同参数类型的空函数，选择 PICC S 选项，编译成汇编程序，然后再检查生成的入口和出口代码。

1.20　函数返回值

函数返回值按以下方式传递给调用函数。

1.20.1　8 bit 返回值

对于低档 PIC，8 bit 返回值(字符型、无符号字符型及指针)通过内存临时程序块返回；对于中档 PIC，8 bit 返回值通过 W 寄存器返回。例如函数：

```
char return_8(void)
{
return 0;
}
```

将会生成以下代码：

```
movlw  0
return
```

1.20.2　16 bit 和 32 bit 返回值

16 bit 和 32 bit 返回值(整型、无符号整型、短整型、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。例如以下函数:

```
int return_16(void)
{
return 0x1234;
}
```

将会生成以下代码:

```
movlw low 01234h
movwf btemp
movlw high 01234h
movwf btemp+1
return
```

1.20.3 结构返回值

4 B和更小的合成返回值(结构和联合)与16 bit和32 bit返回值具有相同的返回方式。大于4 B长度的结构或联合返回值会被复制到结构块(struct psect)中。例如:

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

可以达到以下代码:

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

1.21　函数调用规则

低档 PIC 只有 2 级深度硬件堆栈可用于保存子程序调用的返回地址。通常，PICC 使用 CALL 指令实现 C 函数的调用；但是，对于低档 PIC，硬件堆栈深度严重限制了 C 函数嵌套调用的个数。

缺省情况下，低档 PIC 通过汇编跳转指令、查表及转移表等方式实现函数的调用。在这些方式下，为了实现函数调用的返回，函数的“调用”必须在保存转移表地址后，才直接跳至函数所在的地址。当函数调用执行完毕后，单片机将会进行查表操作。这些方式允许单片机在堆栈不溢出的情况下，实现函数的嵌套调用；但代价是消耗大量的存储空间和程序执行速度。

在函数定义时，使用 fastcall 限定词可以禁止查表操作。在这种情况下，函数调用是通过 CALL 指令实现的；但是，由于 fastcall 函数的调用必须占用 1 级硬件堆栈，所以在 fastcall 函数嵌套调用时，必须注意堆栈的溢出。应用时，应检查映射表文件的调用图，确保堆栈不会溢出。

低档 PIC 的 fastcall 函数原型如下所示：

```
fastcall void my_function(int a);
```

由于中档和高档 PIC 有多级硬件堆栈，所以都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。

任何函数返回时，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，必须确保选择了存储区 0。

1.22　局部变量

在函数中，C 提供 2 种局部变量：自动变量和静态变量。其中，自动变量通常分布在函数的自动变量区；静态变量则都分配有一个固定的存储单元。

1.22.1　自动变量

局部变量的缺省类型是自动变量，除非明确地将它声明为静态变量。自动变量保存在自动变量区，并通过函数变量区起始地址加上偏移量来引用。值得注意的是，由于自动变量的存储单元是不固定的，所以除 const 和 volatile 外，大多数限定词都不适用于它们。此外，所有的自动变量都被分配在存储区 0，所以 bank 限定词也不能用于自动变量。编译时，链接器会自动把一些不可能同时被调用的函数的自动变量区重叠在一起，以达到内存的高效利用。

自动变量可以通过问号“?”、下划线“a_”及函数名加某一偏移量来引用。例如，设某 int 变量是定义在 main( )函数中的第 1 个局部变量，则可以通过地址“? a_main”和“? a_main+

1”访问它们。

1.22.2 静态变量

未初始化的静态变量将分配在 rbss_n 程序块中，占用 1 个固定的存储单元，而且这个存储单元不会再被别的函数使用。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。除非通过指针明确修改静态变量的值，否则它在 2 次函数调用之间将保持不变。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 1 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；所以，从某种意义上讲，静态变量比自动变量更具优越性。

1.23 pragma 伪指令

某些编译伪指令可修改汇编器的运行，可以通过 ANSI 标准 # pragma 语句实现。pragma 语句的格式如下：

pragma 关键词 选项；

以上式子中，关键词是一系列关键词中的一个，如表 1.7 所列。其中，有些关键词后带有选项。

表 1.7 pragma 伪指令

伪指令	含 意	例 子
Interrupt_level	允许主程序调用中断函数中的函数	# 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	# pragma nojis
Printf_check	允许打印格式符校验	# pragma printf_check const
Psect	重新命名编译器定义程序块	# pragma psect text = mytext
Regsused	规定中断中使用的寄存器	# pragma regsused

1.23.1 # pragma jis 和 # pragma nojis 伪指令

如果代码中包含日本 JIS 和其他国家字符的双字节字符串，则 # pragma jis 伪指令允许对这些字符进行适当的操作；而 # pragma nojis 伪指令则会禁止这项操作。在缺省方式下，JIS 字符操作是禁止的。

1.23.2 # 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，并做成现成的文档。这对大部分应用是很有好处的；但是，当希望得到特定的存储器配置时，必须在不同的程序块中重新分配数据和代码。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，如果希望让某一 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，可以使用以下伪指令实现：

```
#pragma psect rbss_0=otherram
```

以上伪指令将告诉编译器，把存放在 rbss_0 程序块的任何数据都转存到 otherram 程序块中。

重新配置程序代码与重新配置数据所用伪指令是有区别的。由于程序代码会放置在多个程序块中，如 text0，text1，text2 等等，所以根本不可能精确知道某代码将会被放置在哪个程序块中。为了实现代码的转存功能，可以使用以下伪指令：

```
#pragma psect text%%u=othercode
```

其中，%u 序列与 text 程序段标号数相对应，附加的“%”用于保证编译器能准确地扫描到程序块的名字。

如果希望将某模块中多个函数放置到独立的程序块中，可以在每个函数前放置 pragma 伪指令，如下所示：

```
#pragma psect text%%u=othercode%%u
void function(void)
{……
//函数定义等
}
#pragma psect text%%u=othercode%%u
void another(void)
{……
//函数定义等
}
```

上例为函数 function( )的代码定义了一个程序块 othercode0，同时也为函数 another( )的代码定义了另一个程序块 another0。每放置一个函数，%u 加 1。

1.23.3　#pragma regsused 伪指令

当中断出现时，PICC 会自动保护现场；但是，缺省条件下，编译器会保护中断函数需要用到的所有存储单元。#pragma regsused 伪指令允许程序员在中断函数中对寄存器进行有选择的保护。

表 1.8 给出了与这条伪指令一起使用的寄存器名称。这些寄存器名对大小写不敏感，而且如果编译器不认识寄存器名，就会发出警告。

该伪指令只影响紧跟其后的中断函数。对于高档 PIC，如果程序中包含多个中断函数，则每个中断函数前都需加一个 #pragma 伪指令。

例如，以下例子将限制编译器在中断函数中只保护 W 寄存器和 FSR 寄存器：

```
#pragma regsused w fsr
```

表 1.8 寄存器名

寄存器名	说 明
W	W 寄存器
Btemp,btemp+1…btemp+11	btemp 临时寄存器
Fsr	间接数据指针
Tablreg	表寄存器:数据指针和数据寄存器的高、低字节

1.24 标准 I/O 函数

PICC 库函数为编译器提供了大量的标准 I/O 函数,将对规格化文本进行读写操作。表 1.9 列出了 2 个函数,如果想得到更详细的信息,请参阅函数库一章。

表 1.9 标准输入输出函数

函数名	目 的
Printf(const char * s,…)	标准输出文件的格式化打印
Sprintf(char * buf,const char * s,…)	向缓冲区写入格式化文本

使用这些函数对字符串进行读、写操作前,必须首先使用 putch( )和 getch( )函数。其他程序还必须包含 getche( )和 kbhit( )函数。

1.25 MPLAB 的特殊调试信息

某些选项和编译器特征可以帮助 MPLAB 实现符号调试。MPLAB 不会读取编译器定义的局部变量的信息;但是,使用-FAKELOCAL 选项可以产生额外的全局符号变量。其名由函数名和变量名组成,用以取代程序中的局部符号变量,从而可以使用 MPLAB 对局部变量进行监控。例如,main( )函数内定义了一个变量 foo,则在 MPLAB 中可以访问到一个名为 main.foo 的全局变量;但是,这种变量格式在汇编代码中是无效的。在汇编程序中可以通过符号_main$foo 引用这个变量。虽然,-FAKELOCAL 选项允许 MPLAB 访问大多数的局部变量;但当同一函数中有多个同名的局部变量时,MPLAB 只能对其中的一个进行监控。

-FAKELOCAL 选项还会改变编译器生成的加行号信息,从而使 MPLAB 在单步执行时可以更好地跟随 C 源代码。

-ICD 选项生成的代码适用于 MPLAB 在线调试器。这种调试器可用于某些芯片,而且需要保留特定的 ROM 和 RAM 地址。编译器将从 picinfo.ini 文件中读取有关信息,用于确认该芯片是否支持调试器,然后相应地调节链接器的选项。当使用这种调试器时,单片机将不会执行复位矢量的第 1 条指令。该指令是 powerup.as 文件的第 1 条指令,并且由编译器自动链接;但这不会改变程序和调试器的动作,并且不一定是 NOP 指令。

第 2 章　PICC 的使用

有很多第 3 方计算机语言开发工具可在 MPLAB IDE 环境下工作，HI-TECH PICC 就是其中的一种。

本章将一步步地引导使用 HI-TECH PICC 生成单个源文件项目和多个源文件项目。可学到：

- 如何生成源文件。
- 如何设置 MPLAB IDE 开发模式。
- 如何生成新项目。
- 如何设置项目节点属性(PICC 汇编器)。
- 如何加载源文件。
- 如何编译和调试项目。

2.1　生成单源文件项目

2.1.1　生成源文件

选择 File ＞ New 下拉菜单，打开一个空白的编译窗口；同时会弹出生成项目(Create Project)对话框，单击 No 按钮。

在窗口中输入以下程序，并保存在 c:\proj1 路径下：

```
#include <pic.h>
void main(void);
unsigned char Add(unsigned char a,unsigned char b);
unsigned char x,y,z;
void main()
{
x = 2;
y = 5;
t = Add(x,y);
}
unsigned char Add(unsigned char a,unsigned char b)
```

```
{ return a | b;}
```

注意,程序中有错;但是,请读者不要立即更改。

2.1.2 开发模式设置

选择 Options > Development Mode,打开开发模式(Development Mode)对话框,如图 2.1所示。

① 单击 Tools 标签。

② 选择 MPLAB-SIM 仿真器。

③ 在 Processor 框中选择 PIC16F84。

④ 单击 OK 按钮。

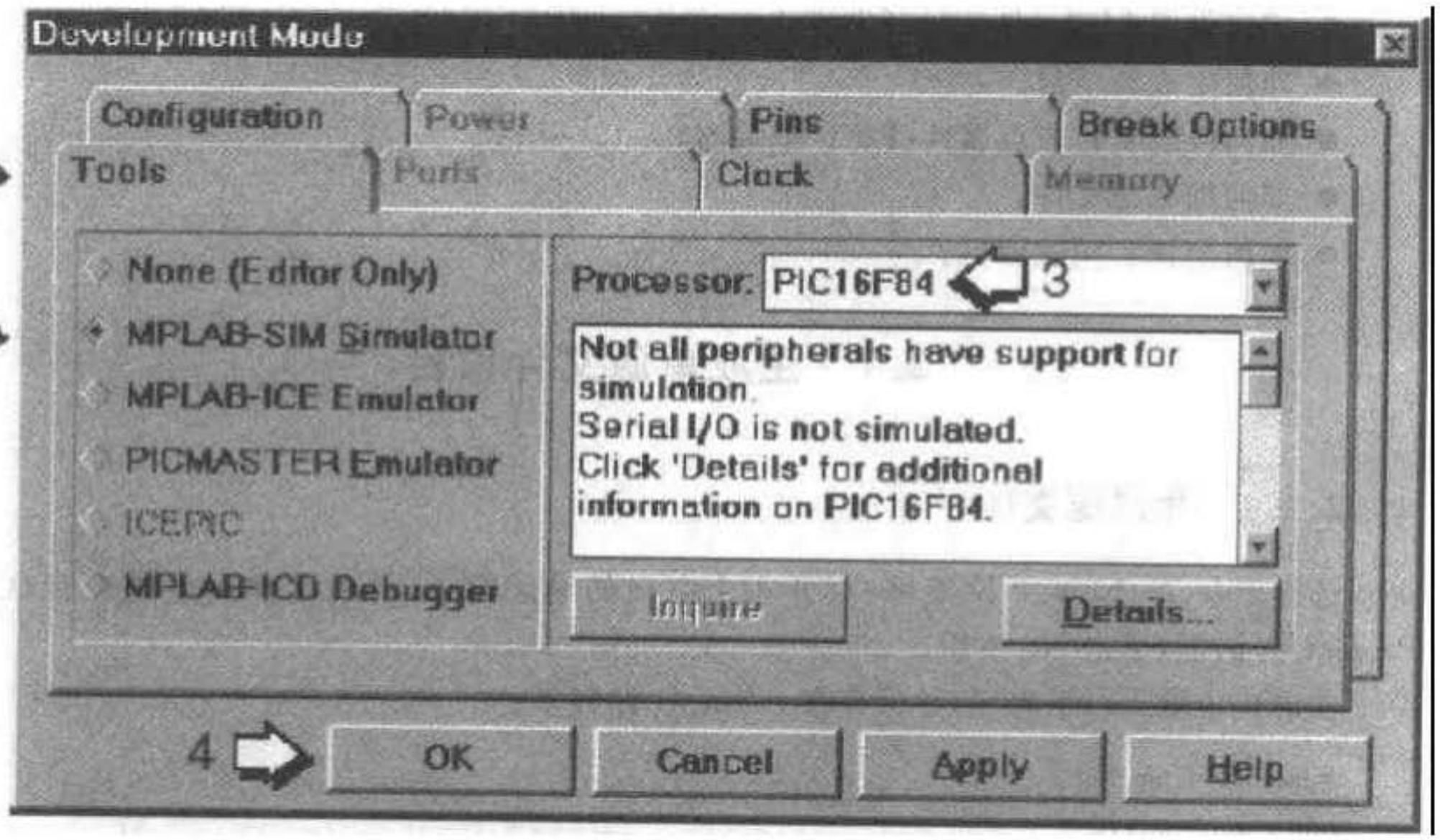

图 2.1 开发模式(Development Mode)对话框

2.1.3 生成新项目

选择 Project > New Project,打开新项目(New Project)对话框,如图 2.2 所示。

① 选择新项目的路径,并使之与源文件所在子目录的路径一致。

② 将新项目命名为 ex1.pjt。

③ 单击 OK 按钮。

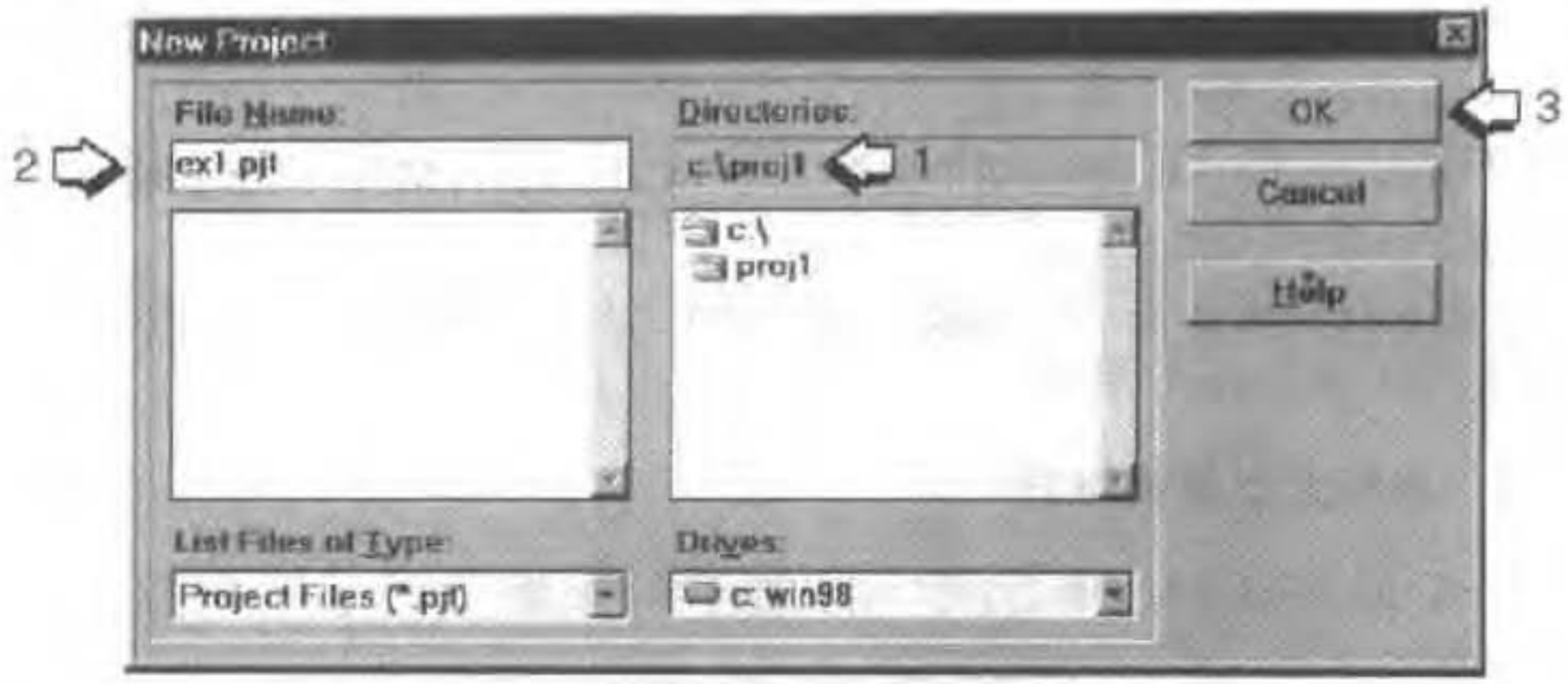

图 2.2　新项目(New Project)对话框——ex1. pjt

2.1.4　编辑项目

编辑(Edit Project)对话框如图 2.3 所示。

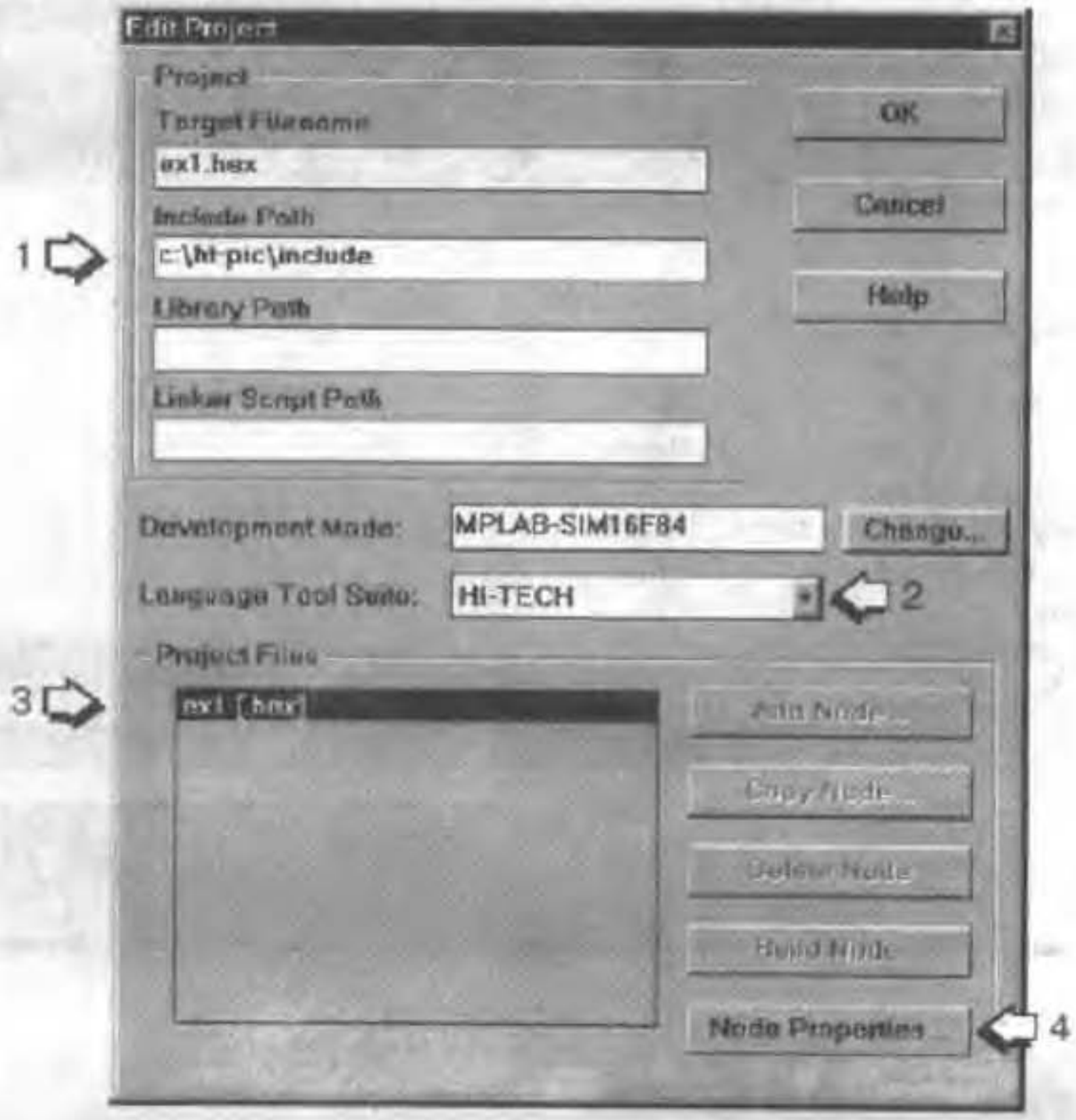

图 2.3　编辑项目(Edit Project)对话框——ex1. hex

① 在 Project 区输入包含路径(如 c:\ht-pic\include)。这个信息将告诉 MPLAB IDE 如何查找 HI-TECH 包含文件——*.h。

② 在 Language Tool Suite 选项中选择 HI-TECH。同时,MPLAB IDE 将显示 Change Suite Warning 对话框,单击 OK。

③ 在 Project File 区单击 ex1[.hex],点亮 HEX 文件名,同时激活 Node Properties 按钮。

① 单击 Node Properties 按钮。

2.1.5 目标节点属性设置

打开 Node Properties 对话框后,需要对其进行以下设置:

① 选择 PIC-C Compiler 作为语言工具。

② Options 区的命令行(Command Line)将对被选语言工具进行叙述性说明。当第 1 次打开本对话框时,复选框将出现语言工具的缺省值。如果想得到更多关于每个选项的信息,请查阅 HI-TECH 资料。就本例程而言,有几个设置是需要更改的,如图 2.4 所示。

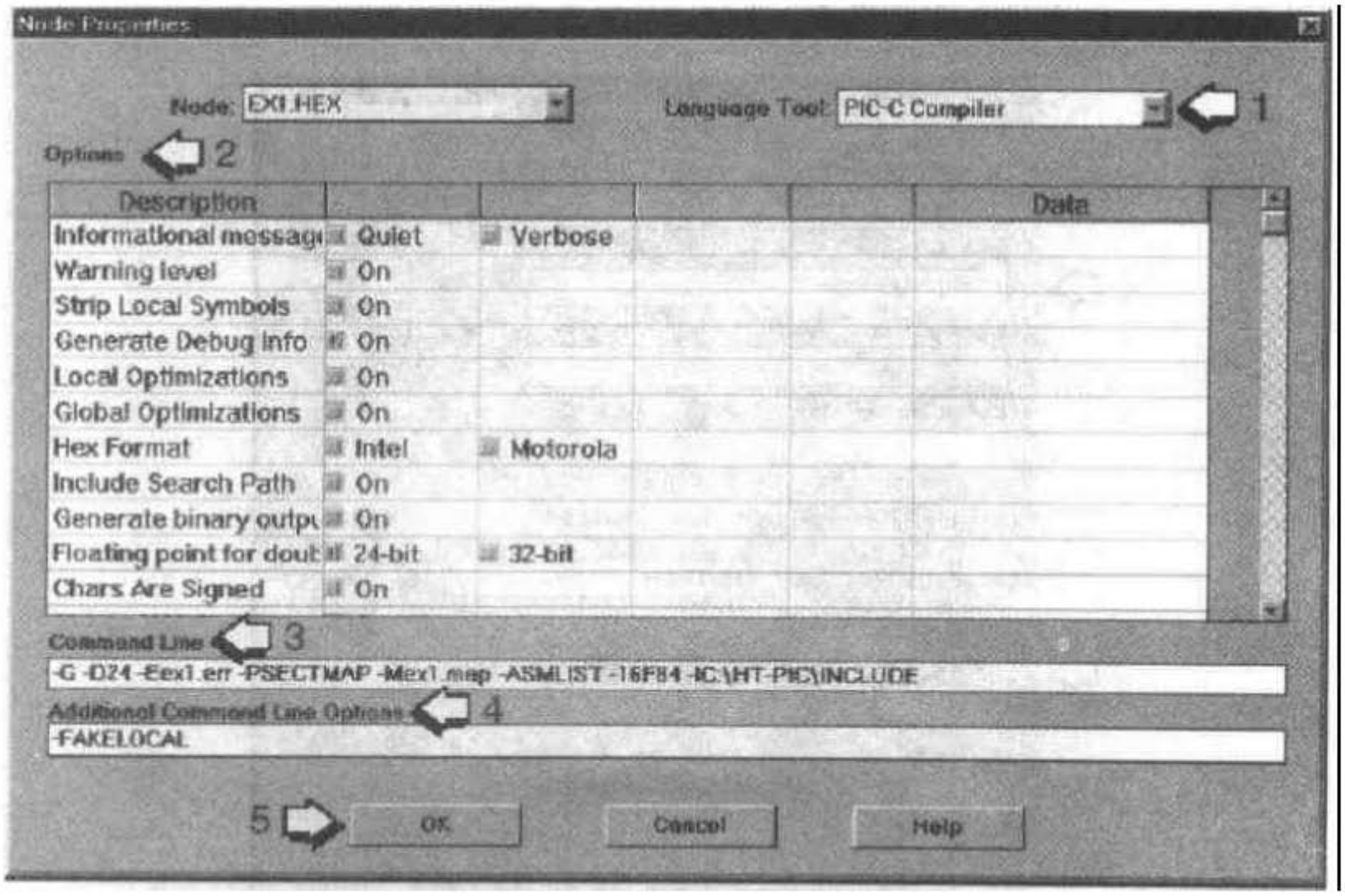

图 2.4 节点属性(Node Properties)对话框——ex1.hex

Options 区需要更改的信息如下:

- 选择 Generate Debug Info 复选框。

- 选择 Error File 复选框，并在 Data 一栏键入文件名，如 ex1.err。
- 选择 Display Memory Usage 复选框。
- 选择 Map File 复选框，并在 Data 一栏中键入文件名，如 ex1.map。
- 选择 Assembler List File 复选框，用于产生 ex1.lst 文件。

③ Command Line 中的内容将随着复选框的不同选择而改变。

④ 在 Additional Command Line Option 选项中键入-FAKELOCAL 字样(7.84 及以上版本的 PICC 工具均支持该命令)。

⑤ 单击 OK，返回编辑项目(Edit Project)对话框。

2.1.6　加载源文件

编辑完节点属性后，编辑项目(Edit Project)对话框中的 Add Node 按钮将被激活。此时单击按钮，打开 Add Node 对话框，如图 2.5 所示。

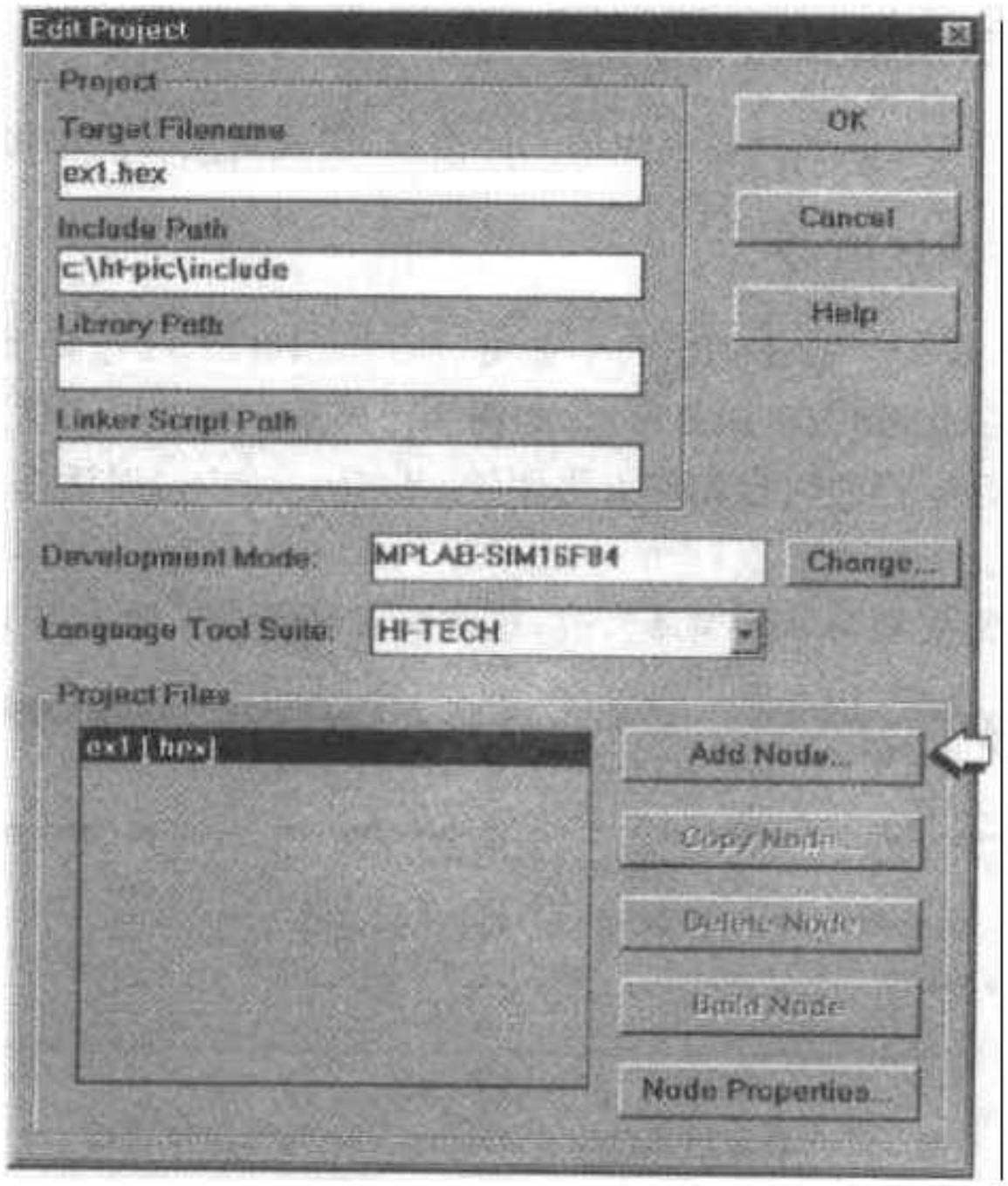

图 2.5　编辑项目(Edit Project)——加载节点对话框

加载节点的步骤(如图 2.6 所示)：

① 从源文件目录中选中要加载的源文件 ex1.c。

② 单击 OK,返回编辑项目(Edit Project)对话框。

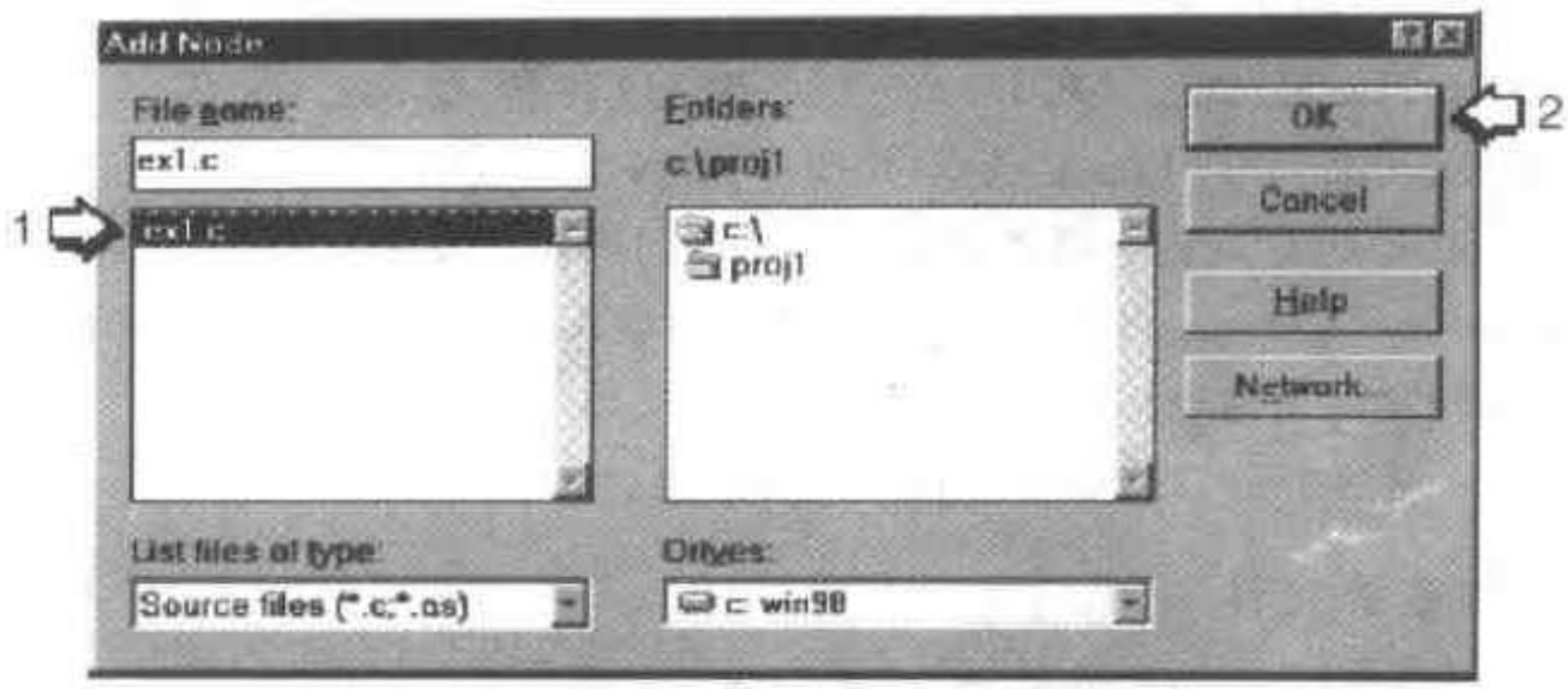

图 2.6 加载节点(Add Node)对话框

2.1.7 完成项目编辑

单击编辑项目(Edit Project)对话框中的 OK 按钮,完成对项目的编辑。

2.1.8 编译和调试项目

从菜单中选择 Project > Make Project 选项,使用 HI-TECH 编译器编译应用程序。编译器将会产生编译结果(Build Results)窗口,如图 2.7 所示。包括以下信息:

① 传给编译器的命令行的内容和节点属性(Node Properties)对话框中规定的其他信息。

② 编译错误信息。

③ 任意一个编译错误都会导致编译失败,这意味着编译器不能生成 HEX 文件。

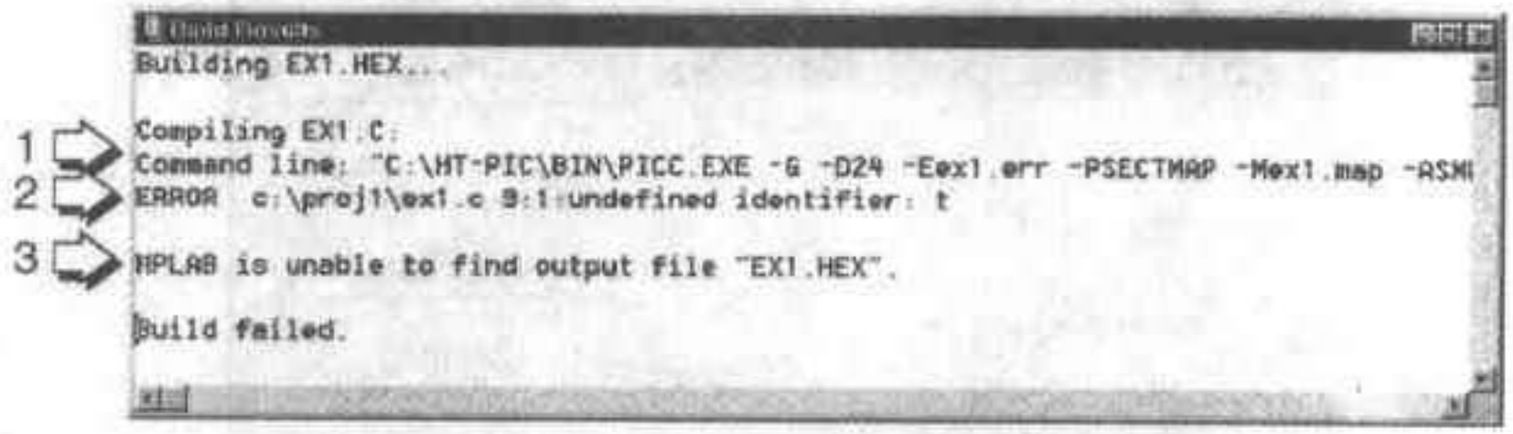

图 2.7 编译结果(Build Results)——编译失败窗口

如果程序编译有错,双击编译结果(Build Results)窗口中的错误信息,将会使光标移至源文件中错误所在的行。如果此时没打开源文件,则 MPLAB IDE 会自动打开它,并把光标移至错误所在的行。更详细的错误描述可在相同目录下的错误文件 ex1.err 中查找。

也可以双击错误文档(如 ex1.err)中的错误,达到以上目的。在节点属性(Node Proper-

ties)对话框中选择 Error File 复选框，MPLAB IDE 就会根据 Data 栏中给定的文件名生成错误文档，如 ex1.err，如图 2.8 所示。

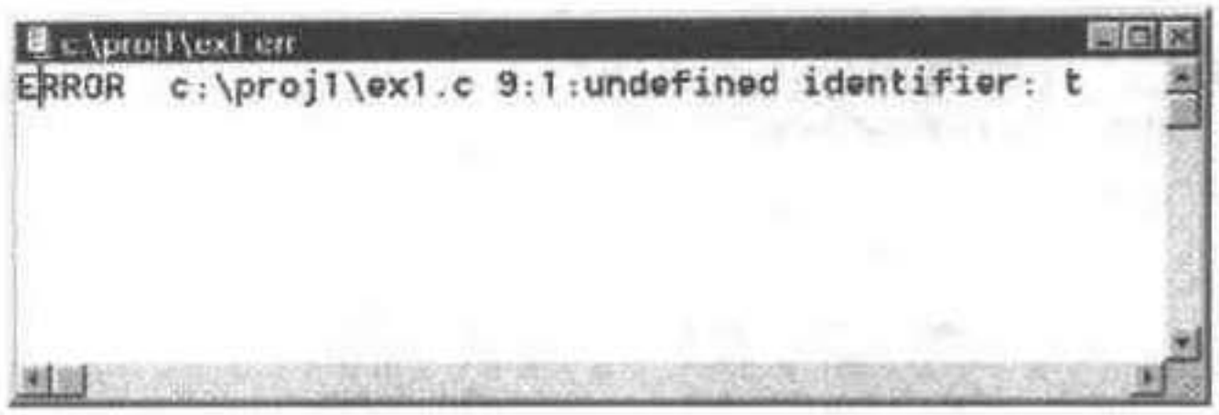

图 2.8　汇编器文档列表窗口——ex1.err

选择 File ＞ Open 选项，从文档类型列表中选择 All Files (* . *)，然后选择. crr 文档，再单击 OK，就可打开错误文档。

将源程序中的“t”改成“z”，修改程序错误，如图 2.9 所示。

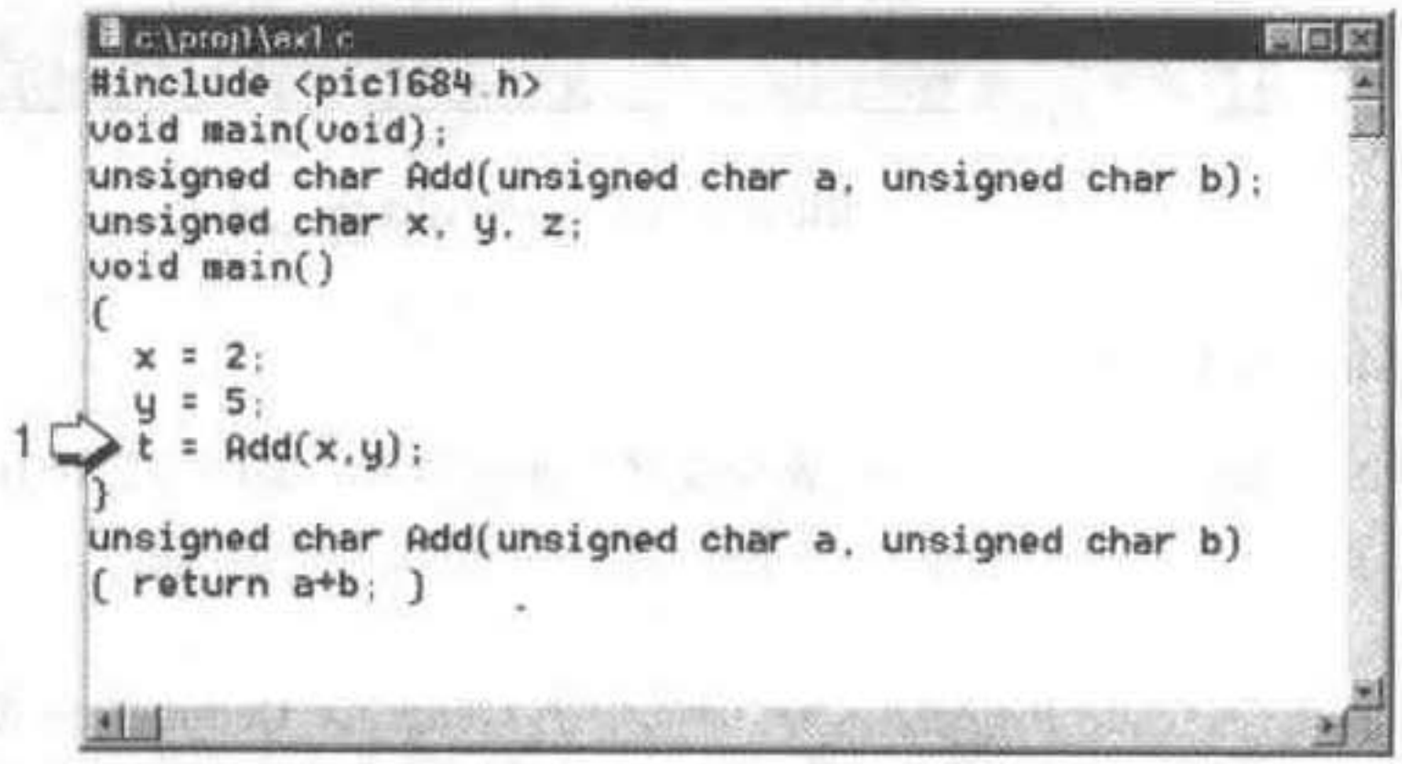

图 2.9　源程序错误的修改

选择 File ＞ Save，保存修改后的源程序(这是一种好习惯；但在此不是必须的，因为 MPLAB IDE 将在编译时自动保存源程序)。然后选择 Project ＞ Make Project，重新编译项目，将会得到以下编译结果。

① 传给编译器的命令行的内容和节点属性(Node Properties)对话框中规定的其他信息。

② 由于选择了节点属性(Node Properties)对话框中的 Display Memory Usage 复选框，编译结果窗口中将会给出程序段使用映射表(Psect Usage Map)和内存使用映射表(Memory Usage Map)。这些信息将会显示链接后存储器使用区段。

就 HI-TECH 工具而言，编译器会自动调用汇编器(也就是说，对于单节点(源文件只有一个)项目，不需要指定它用于链接)。

③ 编译结果正确。如图 2.10 所示。

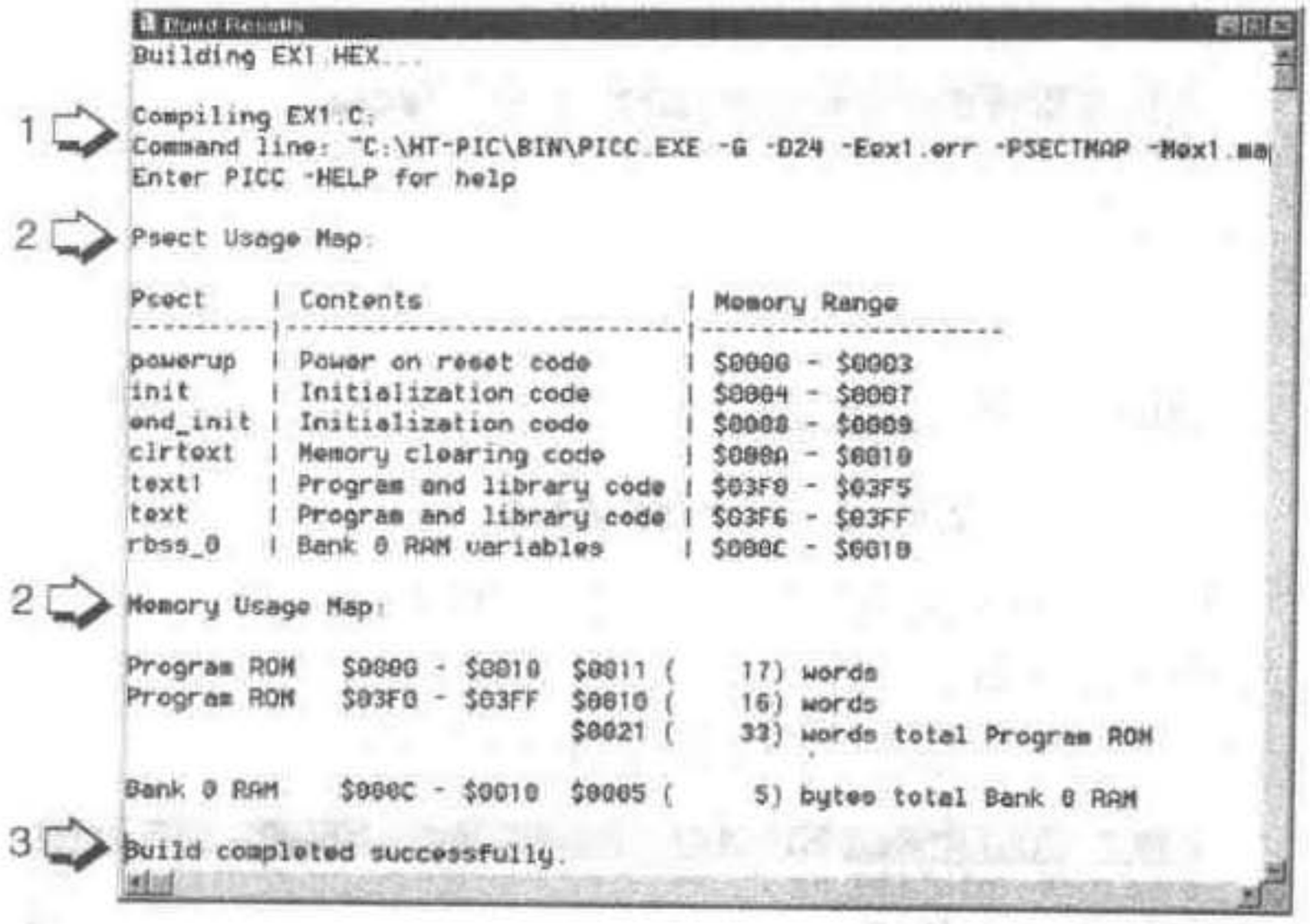

图 2.10 编译结果窗口——编译成功

2.1.9 其他项目信息

由于本例程的源程序只有一个错误，故修改起来非常简单。实际中错误有可能比较多，这时可以通过修改节点属性得到一些信息，帮助查找错误原因。

1. 调试信息

通过选择节点属性(Node Properties)对话框中的 Generate Debug Info 复选框，MPLAB IDE 可以获得必要的信息，有助于调试源程序；同时可以在源程序中直接设置断点调试或单步调试。

2. 映射表文件

通过选择节点属性(Node Properties)对话框中的 Map File 复选框，MPLAB IDE 可以根据 Data 一栏中给定的文件名生成映射表文件。通过选择 Window > Map File 选项，可以打开映射表文件(如图 2.11 所示)。如果想得到关于映射表文件的更多信息，请参阅 HI-TECH 资料。

3. 完全列表

通过选择节点属性(Node Properties)对话框中的 Assembler List File 复选框，MPLAB IDE 可以生成完全列表。该列表文件将会显示 C 源代码、生成的汇编代码及相应的二进制代码。如图 2.12 所示。

```
c:\proj1\ex1.map
Linker command line:

-z -Mex1.map -oC:\WINDOWS\TEMP\1.obj \
  -ppowerup=0,intentry=4,intcode,intret,stringtable,strings,init,end_init,clrtext \
  -pintsave_0=0Ch -ABANK0=0Ch-04Fh \
  -prbss_0=BANK0,rdata_0=BANK0,idata_0=CODE -ACOMBANK=0Ch-04Fh \
  -ptemp=COMBANK -ACODE=0-03FFh -pconfig=2007h \
  -pfloat_text0=CODE,float_text1=CODE,float_text2=CODE \
  -pfloat_text3=CODE,float_text4=CODE -Q16F84 -h+ex1.sym \
  C:\HT-PIC\LIB\picrt400.obj ex1.obj C:\HT-PIC\LIB\pic400-c.lib

Object code version is 3.7

Machine type is 16F84

Call graph:

*_main->_Add size 1,1 offset 0

                Name            Link     Load   Length Selector Space Scale
C:\HT-PIC\LIB\picrt400.obj
```

图 2.11　映射表文件窗口——ex1.map

```
c:\proj1\ex1.lst

HI-TECH Software PIC Macro Assembler                              Page 1
                                                  Mon Aug 28 13:10:33 2008

     1                                    processor       16F84
     2  0000                      indf    equ     0
     3  0000                      rtcc    equ     1
     4  0000                      pc      equ     2
     5  0000                      status  equ     3
     6  0000                      fsr     equ     4
     7  0000                      porta   equ     5
     8  0000                      portb   equ     6
     9  0000                      portc   equ     7
    10  0000                      pclath  equ     10
    11                                    global  _main
    12                                    signat  _main,88
    13                                    psect   text0,class=CODE,local,delta=2
    14                                    psect   text0
    15                                    file    "c:\proj1\ex1.c"
```

图 2.12　汇编列表文件窗口——ex1.lst

4. 标号列表

通过在节点属性(Node Properties)对话框的附加命令行选项中键入-FAKELOCAL 命令，可在单步执行程序过程中，通过窗口查看函数中局部动态变量的值。

选择 Window > Show Symbol List 选项，打开符号列表。滚动变量列表的滚动条，可找到需要观察的变量名及地址，如图 2.13 所示。

5. 项目窗口

单击 Window > Project，打开项目窗口(Project Window)，如图 2.14 所示。这里将列出各种项目信息，这有助于调试过程中对项目进行检查。

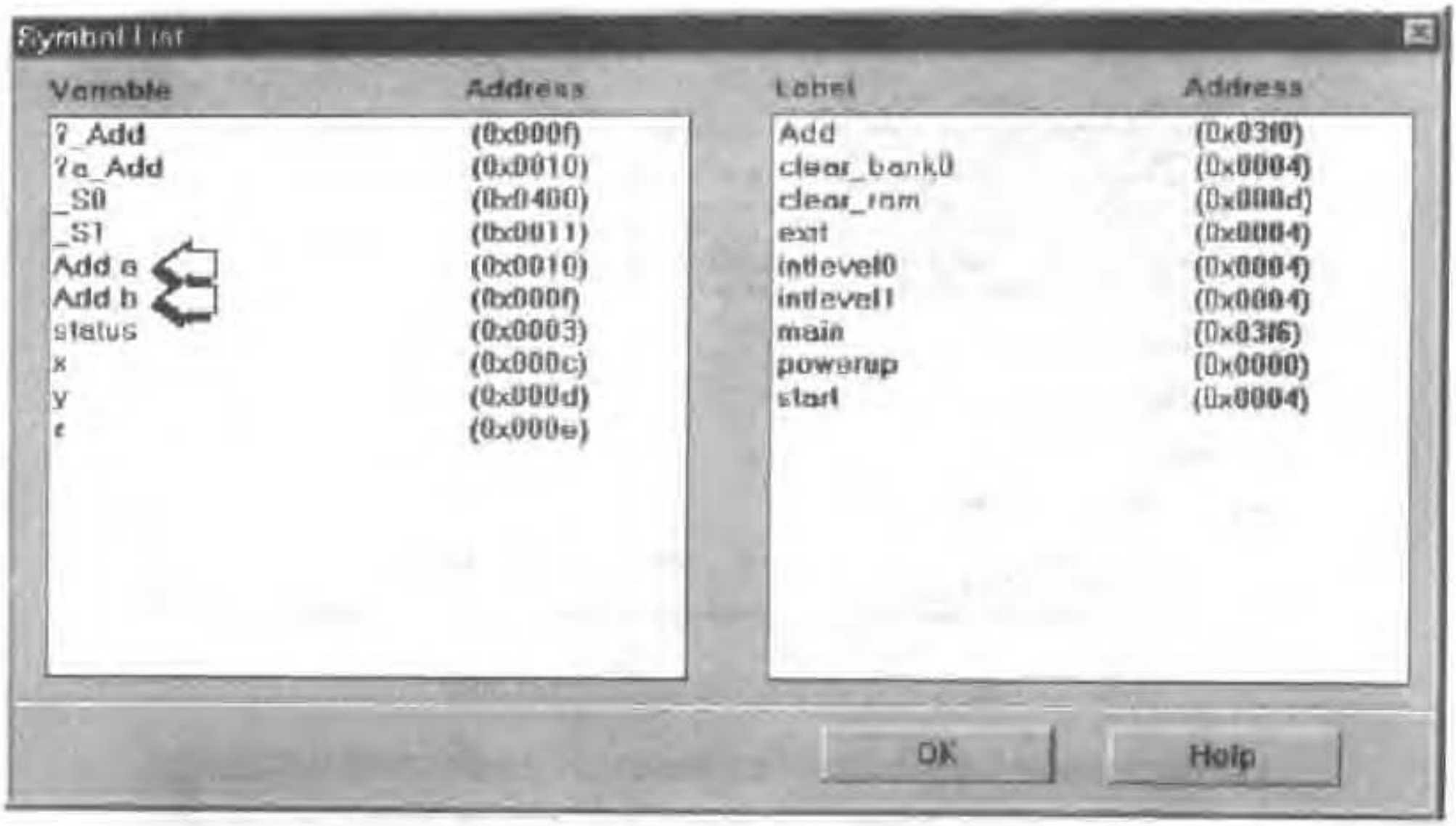

图 2.13　变量列表窗口

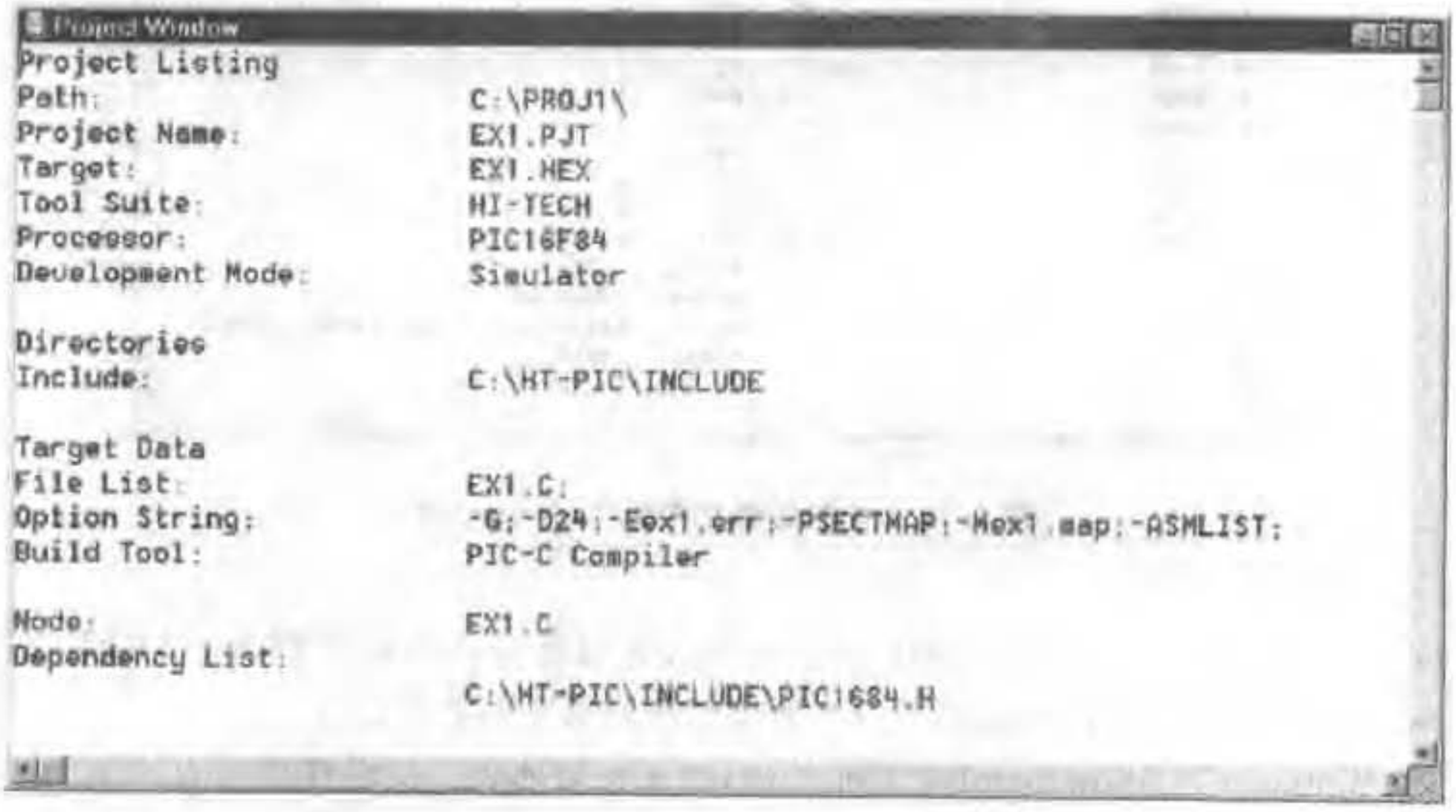

图 2.14　项目窗口——ex1. pjt

单击窗口中任一蓝色的文件名，可以在 MPLAB IDE 中打开该文件。这对于打开不在项目所在目录下的文件是非常有用的(比如头文件)；否则就必须从其他路径查找需要的文件。

2.2　生成多源文件项目

利用 PICC 生成多源文件项目与生成单源文件项目的步骤基本是一致的。本节只详细说明其不同点，相同之处请参阅“生成单源文件项目”一节。

2.2.1　生成源文件

在多源文件项目中，源文件的生成方式可参阅 2.1.1 节。本节只给出 2 个例程，请将它们保存到同一目录下，如 c:\proj2。

生成一个新源文件文本，输入以下程序，命名为 ex1.c，并保存到 c:\proj2 目录下。

```
#include <pic.h>
void main(void);
unsigned char Add(unsigned char a,unsigned char b);
unsigned char x,y,z;
void main()
{
x = 2;
y = 5;
t = Add(x,y);}
```

重新生成另外一个源文件文本，输入以下程序，命名为 add.c，并保存到 c:\proj2 目录下。

```
#include <pic.h>
unsigned char Add(unsigned char a,unsigned char b)
{ return a+b; }
```

注意，例程 1 中有错误，请读者不要更改。

2.2.2　开发模式设置

请参阅 2.1.2 节。

2.2.3　生成新项目

请参阅 2.1.3 节。注意，项目的存放路径必须与源文件的存放路径相同。

2.2.4　编辑项目

请参阅 2.1.4 节。

2.2.5 设置目标节点的属性

打开节点属性(Node Properties)对话框后,进行以下选择:

① 将 PIC-C Linker 设置为语言工具。

② Options 区的命令行(Command Line)将对被选语言工具进行叙述性说明。当第 1 次打开本对话框时,复选框将出现语言工具的缺省值。如果想得到更多关于每个选项的信息,请查阅 HI-TECH 资料。就本例程而言,有几个设置是需要更改的,如图 2.15 所示。

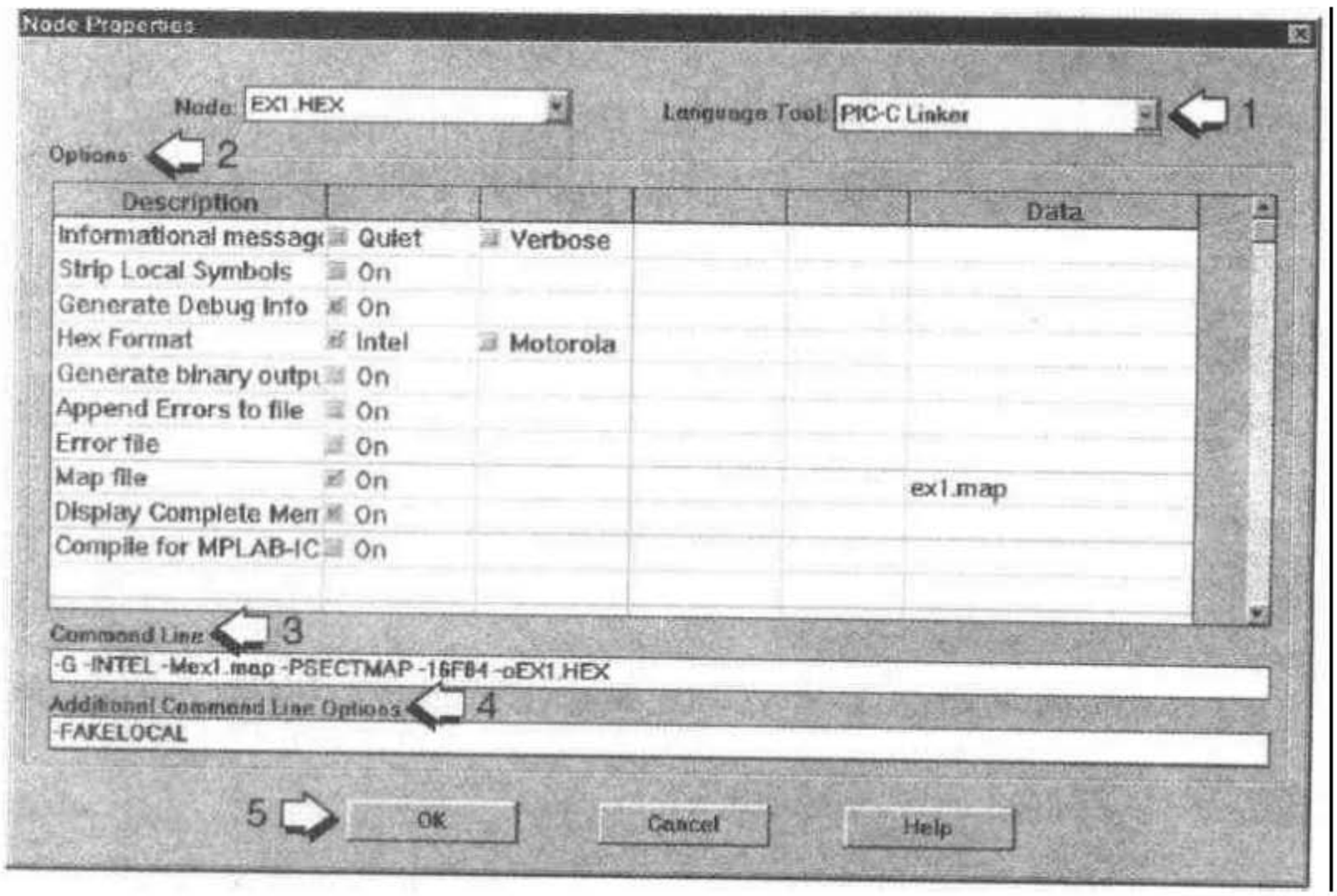

图 2.15 节点属性(Node Properties)对话框——ex1.c

Options 区需要更改设置如下:

- 选择 Generate Debug Info 复选框。
- 选择 Error file 复选框,并在 Data 一栏键入文件名,如 ex1.err。
- 选择 Display Complete Memory Usage 复选框。
- 选择 Map file 复选框,并在 Data 一栏中键入文件名,如 ex1.map。

③ 命令行将随选项不同改变,详细情况请参阅 HI-TECH 文档。

④ 单击 Additional Command Line Options 文本框,输入-FAKELOCAL 命令(7.84 及以上版本 PICC 工具支持该命令)。

⑤ 单击 OK，返回编辑项目(Edit Project)对话框。

2.2.6　加载第 1 个源文件

编辑完项目后，Add Node 按钮将被激活。单击按钮，打开加载节点(Add Node)对话框，如图 2.16 所示。

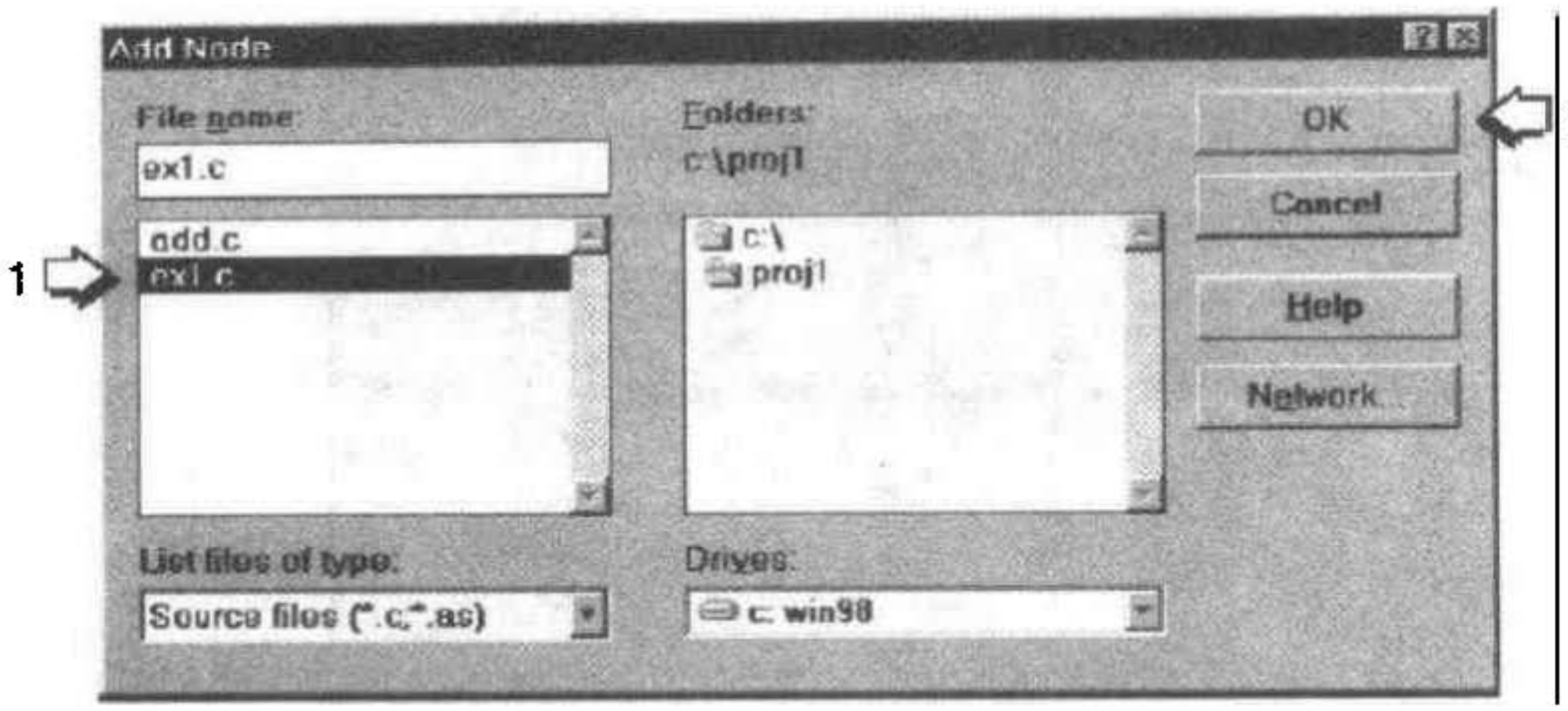

图 2.16　加载节点(Add Node)对话框——ex1.c

在该对话框中进行以下操作：

① 从源文件目录中加载源文件 ex1.c。

② 单击 OK，返回编辑项目(Edit Project)对话框。

2.2.7　设置第 1 个源文件的节点属性

在编辑项目对话框中按以下步骤进行设置，如图 2.17 所示。

① 单击源文件。

② 单击节点属性(Node Properties)按钮。

此时，将打开节点属性(Node Properties)对话框。在该对话框中进行以下操作：

① 将语言工具设置为 PIC-C Compiler。

② 命令行将随选择的语言工具改变而改变。如果是第 1 次打开该对话框，其中选项为缺省值。在本例程中将要改变如下几个设置，如图 2.18 所示。

在 Option 区中选择以下设置：

- 选择 Generate Debug Info 复选框。
- 选择 Error File 复选框，在 Data 一栏中键入文件名，例如 ex1.err。
- 选择 Assembler List File 复选框，用于生成 ex1.lst。

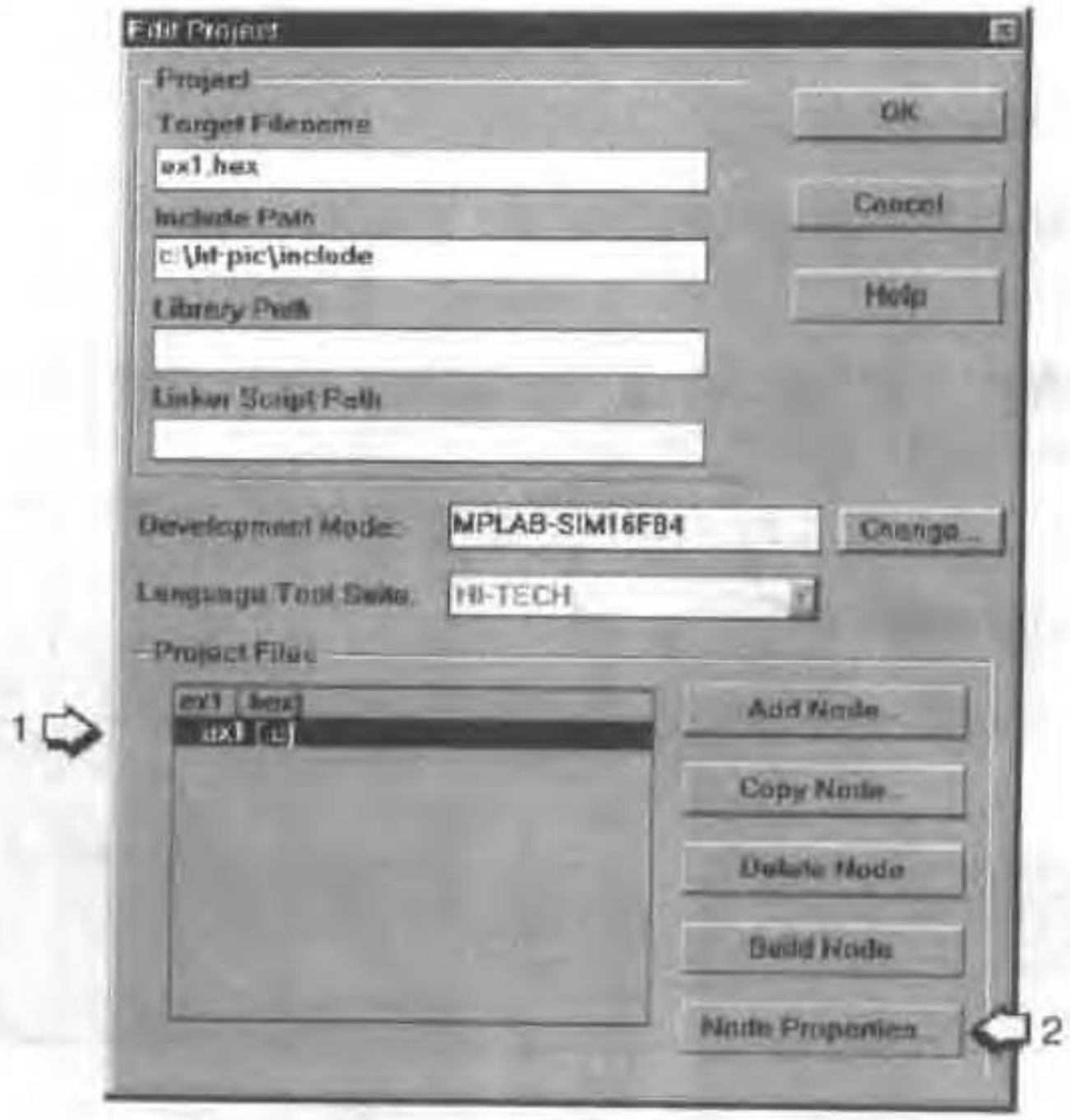

图 2.17 编辑项目(Edit Project)对话框——ex1.c

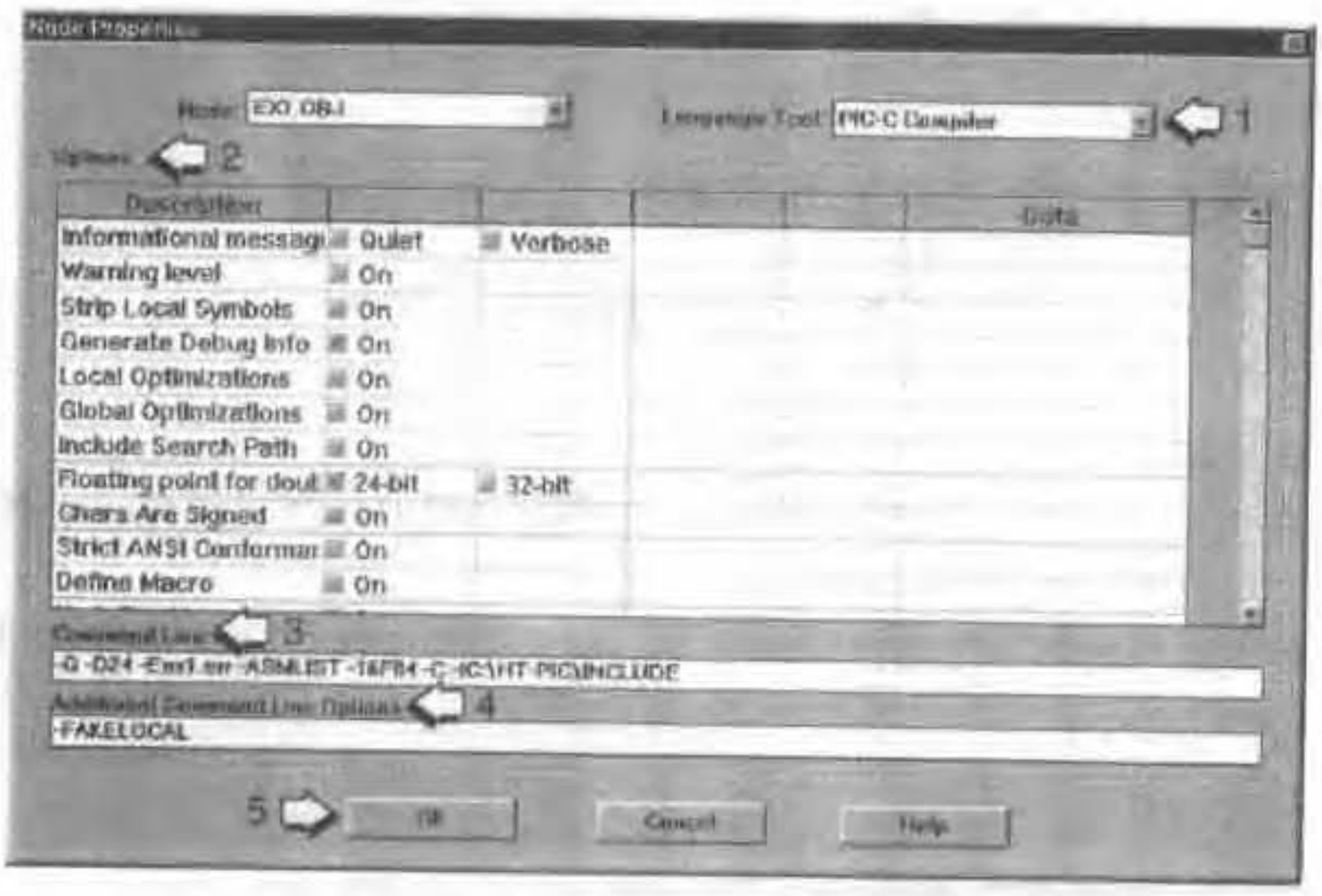

图 2.18 节点属性(Node Properties)对话框——ex1.obj

③ 命令行将随选项不同而改变，详细情况请参阅 HI-TECH 文档。

④ 单击 Additional Command Line Options 文本框，输入-FAKELOCAL 命令(7.84 及以上版本 PICC 工具支持该命令)。

⑤ 单击 OK，返回编辑项目(Edit Project)对话框。

2.2.8　加载第 2 个源文件

此时，编辑项目(Edit Project)对话框中所有的按钮均被激活。单击 Copy Node 按钮，打开复制节点(Copy Node)对话框，如图 2.19 所示。

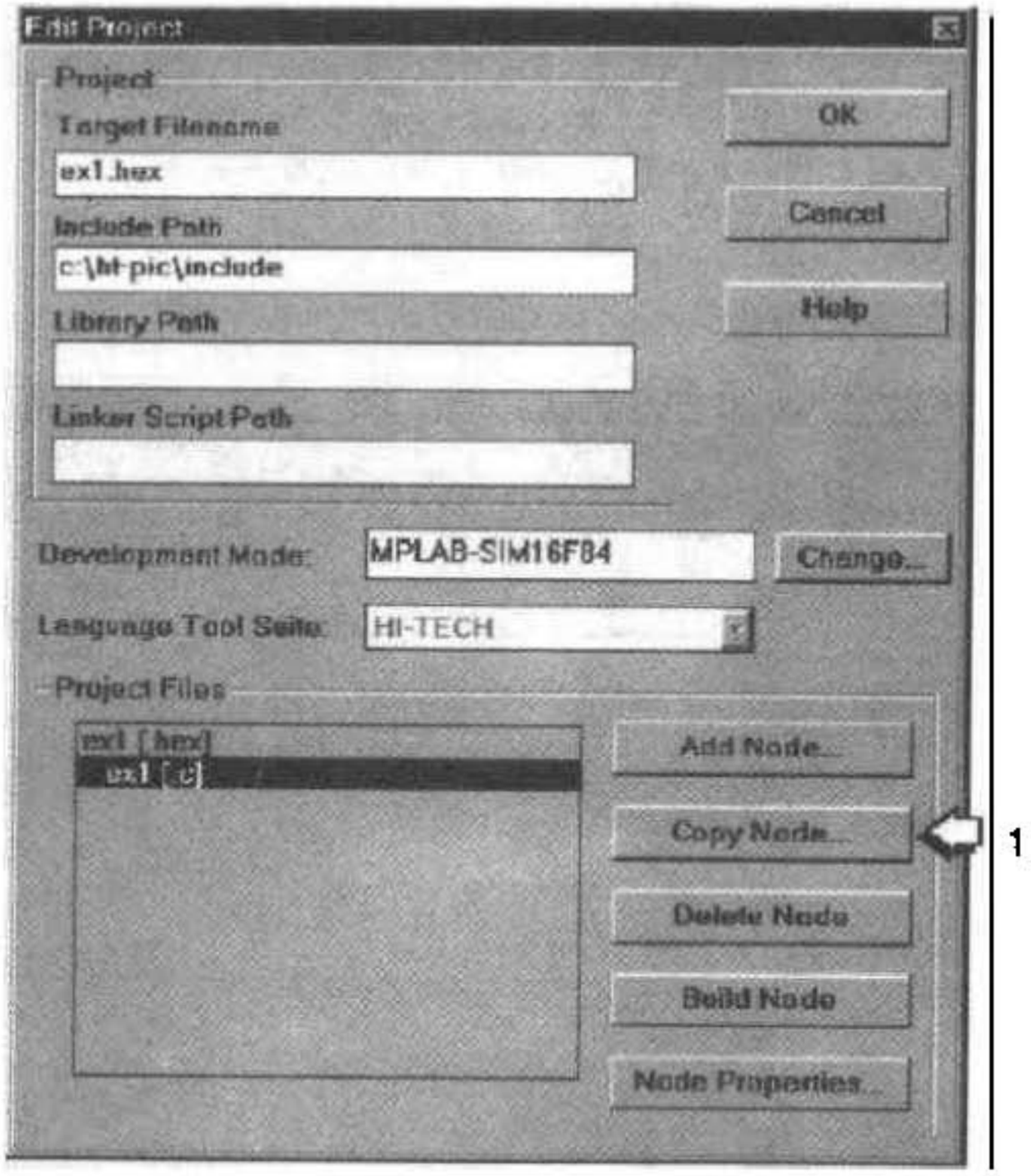

图 2.19　编辑项目(Edit Project)——复制节点对话框

打开复制节点(Copy Node)对话框(如图 2.20 所示)后，按以下步骤操作。

① 从源文件路径找到源文件 add.c，并单击。该文件将会与 ex1.c 的节点属性一起加载到项目中。

② 单击 OK，返回编辑项目(Edit Project)对话框。

在编辑项目(Edit Project)对话框中：

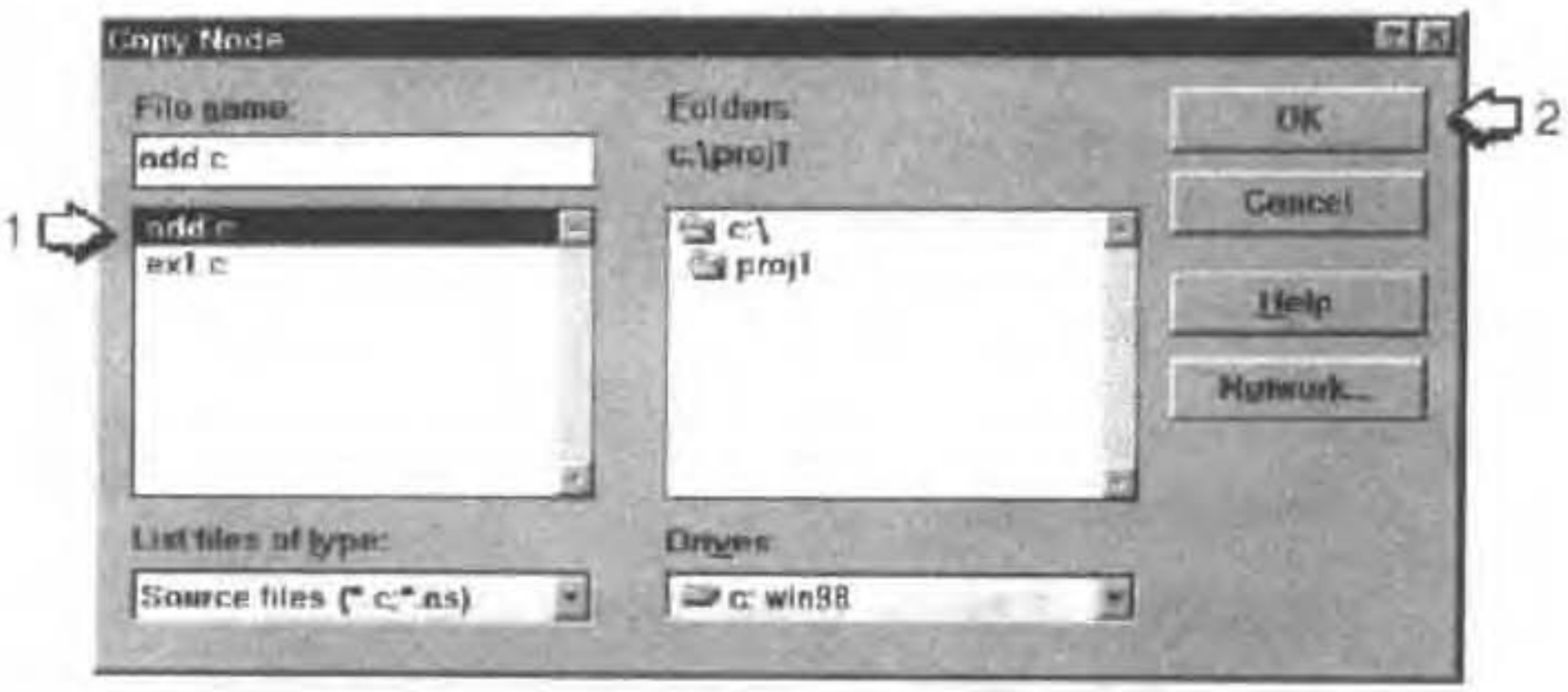

图 2.20 复制节点(Copy Node)对话框——add.c

① 单击 add [.c],如图 2.21 所示。

② 单击 Node Properties,编辑从 ex1.c 复制过来的节点属性。

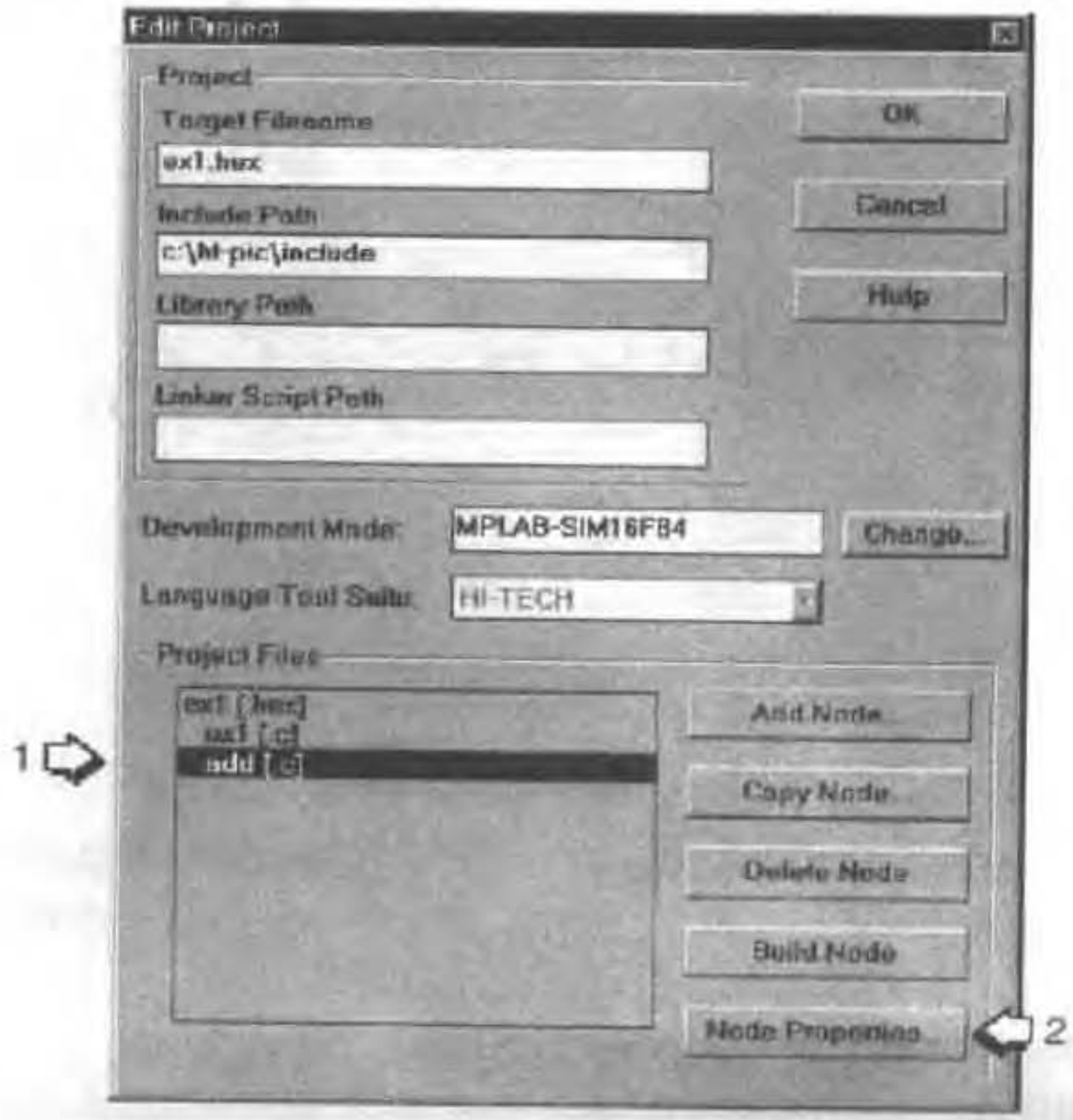

图 2.21 编辑项目(Edit Project)对话框——add.c

在 Option 选项下 Data 一栏中更改错误文件(Error File)的文件名为 add. err，然后单击 OK，返回编辑项目(Edit Project)对话框。

2.2.9　完成项目编辑

单击编辑项目对话框的 OK 按钮，完成项目的编辑。

2.2.10　调试和编译项目

请参阅 2.1.8 节。

第 3 章　PICC 的库函数

本章将详细列出 PICC 编译器的库函数。每个函数均从函数名开始，然后按照以下几个标题给出详细解释。

提要 —— 函数的 C 语言定义以及定义函数的头文件。

描述—— 对函数及其目的进行叙述性描述。

例程 —— 给出一个能说明该函数的应用例子。

数据类型 —— 列出函数中使用的一些特殊的数据类型(如结构体等)的 C 语言定义。这些数据类型的定义包含在提要标题下列出的头文件中。

参阅——给出相关联的函数。

返回值——如果函数有返回值，则在本标题下将给出返回值的类型和性质，同时还包括错误返回的信息。

3.1　ABS 函数

1. 提　要

```
#include <stdlib.h>
int abs (int j)
```

2. 描　述

abs( )函数返回变量 j 的绝对值。

3. 例　程

```
#include <stdio.h>
#include <stdlib.h>
void
main (void)
{
int a =    5;
printf("The absolute value of %d is %d\n",a,abs(a));
}
```

4. 返回值

j 的绝对值。

3.2　ACOS 函数

1. 提　要

#include <math.h>

double acos (double f)

2. 描　述

acos( )函数是 cos( )的反函数。函数参数在[－1,1]区间内,返回值是一个用弧度表示的角度,而且该返回值的余弦值等于函数参数。

3. 例　程

```
#include <math.h>
#include <stdio.h>
/*以度为单位,打印[-1,1]区间内的反余弦值*/
void
main (void)
{
float i,a;
for(i = -1.0,i < 1.0;i += 0.1) {
a = acos(i) * 180.0/3.141592;
printf("acos(%f) = %f degrees\n",i,a);
}
}
```

4. 参　阅

sin( ),cos( ),tan( ),asin( ),atan( ),atan2( )。

5. 返回值

返回值是一个用弧度表示的角度,区间是[0,π]。如果函数参数超出区间[－1,1],则返回值将为 0。

3.3　ASCTIME 函数

1. 提　要

#include <time.h>

char * asctime (struct tm * t)

2. 描　述

asctime( )函数通过指针 t 从上 struct tm 结构体中获得时间,返回描述当前日期和时间

的 26 个字符串，其格式如下：

Sun Sep 16 01:03:52 1973\n\0

值得注意的是，在字符串的末尾有换行符。字符串中的每个字长是固定的。以下例程得到当前时间，通过 localtime( )函数将其转换成一个 struct tm 指针，最后转换成 ASCII 码，并打印出来。其中，time( )函数需要用户提供(详情请参阅 time( )函数)。

3. 例　程

```
#include <stdio.h>
#include <time.h>
void
main (void)
{
time_t clock;
struct tm * tp;
time(&clock);
tp = localtime(&clock);
printf("%s",asctime(tp));
}
```

4. 参　阅

ctime( ),gmtime( ),localtime( ),time( )。

5. 返回值

指向字符串的指针。

由于编译器不提供 time( )例行程序，故在本例程中它需要由用户提供。详情请参阅 time( )函数。

6. 数据类型

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

3.4　ASIN 函数

1. 提　要

#include <math.h>

double asin (double f)

2. 描　述

asin( )函数是 sin( )的反函数。它的函数参数在[－1,1]区间内,返回一个用弧度表示的角度值,而且这个返回值的正弦等于函数参数。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
float i,a;
for(i = -1.0;i < 1.0  ;i += 0.1) {
a = asin(i) * 180.0/3.141592;
printf("asin(%f) = %f degrees\n",i,a);
}
}
```

4. 参　阅

sin( ),cos( ),tan( ),acos( ),atan( ),atan2( )。

5. 返回值

本函数返回一个用弧度表示的角度值,其区间为[－π/2,π/2]。如果函数参数的值超出区间[－1,1],则函数返回值将为 0。

3.5　ATAN 函数

1. 提　要

#include <math.h>

double atan (double x)

2. 描　述

函数返回参数的反正切值。也就是说,本函数将返回一个在区间[－π/2,π/2]的角度 e,而且有 tan(e)=x(x 为函数参数)。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
printf("%f\n",atan(1.5));
}
```

4. 参　阅

sin( ),cos( ),tan( ),asin( ),acos( ),atan2( )。

5. 返回值

返回函数参数的反正切值。

3.6　ATAN2 函数

1. 提　要

```
#include <math.h>
double atan2 (double y,double x)
```

2. 描　述

本函数返回 y/x 的反正切值,并由 2 个函数参数的符号决定返回值的象限。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
printf("%f\n",atan2(1.5,1));
}
```

4. 参　阅

sin( ),cos( ),tan( ),asin( ),acos( ),atan( )。

5. 返回值

返回 y/x 的反正切值(用弧度表示),区间为[$-\pi$,π]。如果 y 和 x 均为 0,将出现定义域错误,并返回 0。

3.7　ATOF 函数

1. 提　要

```
# include <stdlib. h>
double atof (const char * s)
```

2. 描　述

atof( )函数将扫描由函数参数传递过来的字符串，并跳过字符串开头的空格；然后将一个数的 ASCII 表达式转换成双精度数。这个数可以用十进制数、浮点数或者科学记数法表示。

3. 例　程

```
# include <stdlib. h>
# include <stdio. h>
void
main (void)
{
char buf[80];
double i;
gets(buf);
i = atof(buf);
printf("Read %s: converted to %f\n",buf,i);
}
```

4. 参　阅

atoi( ),atol( )。

5. 返回值

本函数返回一个双精度浮点数。如果字符串中没有发现任何数字，则返回 0.0。

3.8　ATOI 函数

1. 提　要

```
# include <stdlib. h>
int atoi (const char * s)
```

2. 描　述

atoi( )函数扫描传递过来的字符串，并跳过开头的空格读取其符号；然后将一个十进制数的 ASCII 表达式转换成整数。

3. 例　程

```
#include <stdlib.h>
#include <stdio.h>
void
main (void)
{
char buf[80];
int i;
gets(buf);
i = atoi(buf);
printf("Read %s: converted to %d\n",buf,i);
}
```

4. 参　阅

xtoi( ),atof( ),atol( )。

5. 返回值

返回一个有符号的整数。如果在字符串中没有发现任何数字,则返回 0。

3.9　ATOL 函数

1. 提　要

```
#include <stdlib.h>
long atol (const char * s)
```

2. 描　述

atol( )函数扫描传递过来的字符串,并跳过字符串开头的空格;然后将十进制数的 ASCII 表达式转换成长整型。

3. 例　程

```
#include <stdlib.h>
#include <stdio.h>
void
main (void)
{
char buf[80];
long i;
gets(buf);
i = atol(buf);
printf("Read %s: converted to %ld\n",buf,i);
```

}

4. 参　阅

atoi( ),atof( )。

5. 返回值

返回一个长整型数。如果字符串中没有发现任何数字,返回值为 0。

3.10　CEIL 函数

1. 提　要

```
#include <math.h>
double ceil (double f)
```

2. 描　述

本函数对函数参数 f 取整。取整后的返回值为≥f 的最小整数。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
double j;
scanf("%lf",&j);
printf("The ceiling of %lf is %lf\n",j,ceil(j));
}
```

3.11　COS 函数

1. 提　要

```
#include <math.h>
double cos (double f)
```

2. 描　述

本函数将计算函数参数的余弦值。其中,函数参数用弧度表示。余弦值通过多项式级数近似值展开式算得。

3. 例　程

```
#include <math.h>
```

```
#include <stdio.h>
#define C 3.141592/180.0
void
main (void)
{
double i;
for(i = 0;i <= 180.0;i + = 10)
printf("sin(%3.0f) = %f,cos = %f\n",i,sin(i * C),cos(i * C));
}
```

4. 参　阅

sin( ),tan( ),asin( ),acos( ),atan( ),atan2( )。

5. 返回值

返回一个双精度数,区间为[-1,1]。

3.12　COSH,SINH,TANH函数

1. 提　要

```
#include <math.h>
double cosh (double f)
double sinh (double f)
double tanh (double f)
```

2. 描　述

这些函数是cos( ),sin( )及tan( )的双曲函数。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
printf("%f\n",cosh(1.5));
printf("%f\n",sinh(1.5));
printf("%f\n",tanh(1.5));
}
```

4. 返回值

cosh( )函数返回双曲余弦值,sinh( )函数返回双曲正弦值,tanh( )函数返回双曲正切值。

3.13　CTIME 函数

1. 提　要

#include <time.h>

char * ctime (time_t * t)

2. 描　述

ctime( )函数将函数参数所指的时间转换成字符串，其结构与 asctime( )函数所描述的相同，并且精确到 s(秒)。以下例程将打印出当前的时间和日期。

3. 例　程

```
#include <stdio.h>
#include <time.h>
void
main (void)
{
time_t clock;
time(&clock);
printf("%s",ctime(&clock));
}
```

4. 参　阅

gmtime( ),localtime( ),asctime( ),time( )。

5. 返回值

本函数返回一个指向该字符串的指针。

由于编译器不会提供 time( )程序，故它需要由用户给定。详情请参阅 time( )函数。

6. 数据类型

typedef long time_t

3.14　DI,EI 函数

1. 提　要

#include <pic.h>

void ei(void)

void di(void)

2. 描　述

ei( )和 di( )函数分别实现全局中断使能和中断屏蔽，其定义在 pic.h 头文件中。它们将

被扩展为一条内嵌的汇编指令，分别对中断使能位进行置位和清0。

以下例程将说明ei( )函数和di( )函数在访问一个长整型变量时的应用。由于中断服务程序将修改该变量，所以，如果访问该变量不按照本例程的结构编程，一旦在访问变量值的连续字期间出现中断，则函数getticks( )将返回错误的值。

3. 例　程

```
#include <pic.h>
long count;
void interrupt tick(void)
{
count++;
}
long getticks(void)
{
long val; /*在访问 count 变量前禁止中断，保证访问的连续性*/
di( );
val    count;
ei( );
return val;
}
```

3.15　DIV函数

1. 提　要

```
#include <stdlib.h>
div_t div (int numer,int demon)
```

2. 描　述

div( )函数实现分子除以分母，得到商和余数。

3. 例　程

```
#include <stdlib.h>
#include <stdio.h>
void
main (void)
{
div_t x;
x = div(12345,66);
```

```
printf("quotient = %d,remainder = %d\n",x.quot,x.rem);
}
```

4. 返回值

返回一个包括商和余数的结构体 div_t。

5. 数据类型

```
typedef struct
{
int quot;
int rem;
} div_t;
```

3.16　EEPROM_READ,EEPROM_WRITE 函数

1. 提　要

```
#include <pic.h>
unsigned char eeprom_read (unsigned char addr);
void eeprom_write (unsigned char addr,unsigned char value);
```

2. 描　述

这些函数允许访问片内 EEPROM(如果片内有 EEPROM)。EEPROM 不是可直接寻址的寄存器空间,当需要访问 EEPROM 时,需要将一些特定的字节序列加载到 EEPROM 控制寄存器中。写 EEPROM 是一个缓慢的过程;故 eeprom_write( )函数在写入下一个数据前,会通过查询恰当的寄存器,以确保前一个数据已经写入完毕。另外,读 EEPROM 可以在一个指令周期内完成;所以没有必要查询读操作是否完成。

3. 例　程

```
#include <pic.h>
void
main (void)
{
unsigned char data;
unsigned char address;
address = 0x10;
data = eeprom_read(address);
}
```

注意,如果调用 eeprom_write( )函数后需即刻调用 eeprom_read( )函数,则必须查询

EEPROM寄存器，以确保写入完毕。全局中断使能位(GIE)在eeprom_write( )程序中重新恢复(写eeprom时需要关闭总中断)；而且，本函数不会清EEIF标志位。

3.17 EVAL_POLY函数

1. 提 要

```
#include <math.h>
double eval_poly (double x,const double * d,int n)
```

2. 描 述

eval_poly( )函数将求解一个多项式的值。这个多项式的系数分别包含在x和数组d中，例如：

$$y = x * x * d2 + x * d1 + d0$$

该多项式的阶数由参数n传递过来。

3. 例 程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
double x,y;
double d[3] = {1.1,3.5,2.7};
x = 2.2;
y = eval_poly(x,d,2);
printf("The polynomial evaluated at %f is %f\n",x,y);
}
```

4. 返回值

本函数返回一个双精度数，该数是自变量x对应的多项式值。

3.18 EXP函数

1. 提 要

```
#include <math.h>
double exp (double f)
```

2. 描 述

exp( )函数返回参数的指数函数值，即e^f(f为函数参数)。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
double f;
for(f = 0.0;f <= 5;f += 1.0)
printf("e to %1.0f = %f\n",f,exp(f));
}
```

4. 参　阅

log( ),log10( ),pow( )。

3.19　FABS 函数

1. 提　要

```
#include <math.h>
double fabs (double f)
```

2. 描　述

本函数返回双精度函数参数的绝对值。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
printf("%f %f\n",fabs(1.5),fabs(-1.5));
}
```

4. 参　阅

abs( )。

3.20　FLOOR 函数

1. 提　要

```
#include <math.h>
```

double floor (double f)

2. 描　述

本函数对函数参数取整，取整后的返回值不大于函数参数 f。

3. 例　程

```
#include <stdio.h>
#include <math.h>
void
main (void)
{
printf("%f\n",floor( 1.5 ));
printf("%f\n",floor( -1.5));
}
```

3.21　FREXP 函数

1. 提　要

```
#include <math.h>
double frexp (double f,int * p)
```

2. 描　述

frexp( )函数将一个浮点数分解成规格化小数和 2 的整数次幂 2 部分，整数幂部分存于指针 p 所指的 int 单元中。本函数的返回值 x 或在区间(0.5,1.0)内，或为 0；而且有 $f=x\times 2^p$。如果 f 为 0，则分解出来的 2 部分均为 0。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
double f;
int i;
f = frexp(23456.34,&i);
printf("23456.34 = %f * 2^%d\n",f,i);
}
```

4. 参　阅

ldexp( )。

3.22　GET_CAL_DATA 函数

1. 提　要

```
#include <pic.h>
double get_cal_data (const unsigned char * code_ptr)
```

2. 描　述

本函数从 PIC 14000 标定空间返回一个 32 bit 的浮点标定数据。只有利用这个函数，才能访问 KREF，KBG，BHTHERM 及 KTC 单元(32 bit 浮点参数)。由于 FOSC 和 TWDT 均是 1 B 长度，故可以直接访问它们。

3. 例　程

```
#include <pic.h>
void
main (void)
{
double x;
unsigned char y;
x = get_cal_data(KREF);/ * 获得参考斜率(slope reference ratio) * /
y =TWDT;                    / * 获得 WDT 溢出时间 * /
}
```

4. 返回值

返回定标参数值。

本函数仅用于 PIC 14000

3.23　GMTIME 函数

1. 提　要

```
#include <time.h>
struct tm * gmtime (time_t * t)
```

2. 描　述

本函数把指针 t 所指的时间分解，并存于结构体中，精确度为 s。其中，t 所指的时间必须自 1970 年 1 月 1 日 0 时 0 分 0 秒起。本函数所用的结构体被定义在 time.h 文件中，可参照本节“数据类型”部分。

3. 例　程

```
#include <stdio.h>
#include <time.h>
void
main (void)
{
time_t clock;
struct tm * tp;
time(&clock);
tp = gmtime(&clock);
printf("It's %d in London\n",tp->tm_year+1900);
}
```

4. 参 阅

ctime( ),asctime( ),time( ),localtime( )。

5. 返回值

返回 tm 类型的结构体。

由于编译器不会提供 time( )程序,故它需由用户给定。详情请参阅 time( )函数。

6. 数据类型

```
typedef long time_t;
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

3.24 ISALNUM,ISALPHA,ISDIGIT,ISLOWER 等函数

1. 提 要

```
#include <ctype.h>
int isalnum (char c)
```

```
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

2. 描　述

以上函数都被定义在 ctype.h 文件中。它们将测试给定的字符，看该字符是否为已知的几组字符中的成员。

isalnum (c)　c 在 0～9，a～z 或者 A～Z 范围内；

isalpha (c)　c 在 A～Z 或 a～z 范围内；

isascii (c)　c 为 7 bit ASCII 字符；

iscntrl (c)　c 为控制字符；

isdigit (c)　c 为十进制阿拉伯数字；

islower (c)　c 在 a～z 范围内；

isprint (c)　c 为打印字符；

isgraph (c)　c 为非空格可打印字符；

ispunct (c)　c 不是字母数字混合的；

isspace (c)　c 是空格键、Tab 键或换行符；

isupper (c)　c 在 A～Z 范围内；

isxdigit (c)　c 在 0～9，a～f 或 A～F 范围内。

3. 例　程

```
#include <ctype.h>
#include <stdio.h>
void
main (void)
{
char buf[80];
int i;
gets(buf);
```

```
i = 0;
while(isalnum(buf[i]))
i++;
buf[i] = 0;
printf("'%s' is the word\n",buf);
}
```

4. 参　阅

toupper( ),tolower( ),toascii( )。

3.25　KBHIT 函数

1. 提　要

```
#include <conio.h>
bit kbhit (void)
```

2. 描　述

如果键盘上的字符被按下,函数返回 1;否则返回 0。通常,该字符可通过 getch( )函数读取。

3. 例　程

```
#include <conio.h>
void
main (void)
{
int i;
while(! kbhit()) {
cputs("I'm waiting..");
for(i = 0 ; i ! = 1000 ; i++)
continue;
}
}
```

4. 参　阅

getch( ),getche( )。

5. 返回值

如果有键被按下,函数将返回 1;否则返回 0。此外,返回值为 1 bit。

注意,程序的主体需由用户实现,其主要框架可以从 sources 目录下直接获得。

3.26　LDEXP 函数

1. 提　要

#include <math.h>

double ldexp (double f,int i)

2. 描　述

ldexp( )函数是 frexp( )的反函数。它先进行浮点数 f 的指数部分与整数 i 的求和运算，然后返回合成结果。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
double f;
f = ldexp(1.0,10);
printf("1.0 * 2^10 = %f\n",f);
}
```

4. 参　阅

frexp( )。

5. 返回值

本函数返回浮点数 f 指数部分加上整数 i 后得到的新浮点数。

3.27　LDIV 函数

1. 提　要

#include <stdlib.h>

ldiv_t ldiv (long number,long denom)

2. 描　述

ldiv( )函数实现分子除以分母，得到商和余数。商的符号与精确商的符号一致，绝对值是一个小于精确商绝对值的最大整数。

ldiv( )函数与 div( )函数类似；不同点在于，前者的函数参数和返回值(结构体 ldiv_t)的成员都是长整型数据。

3. 例　程

```
#include <stdlib.h>
#include <stdio.h>
void
main (void)
{
ldiv_t lt;
lt = ldiv(1234567,12345);
printf("Quotient = %ld,remainder = %ld\n",lt.quot,lt.rem);
}
```

4. 参　阅

div( )。

5. 返回值

返回值是结构体 ldiv_t。

6. 数据结构

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

3.28　LOCALTIME 函数

1. 提　要

```
#include <time.h>
struct tm * localtime (time_t * t)
```

2. 描　述

本函数把指针 t 所指的时间分解，并存于结构体中，精确度为 s。其中，t 所指的时间必须自 1970 年 1 月 1 日 0 时 0 分 0 秒起，所用的结构体被定义在 time.h 文件中。localtime( )函数需考虑全局整型变量 time_zone 中的内容，因为它包含本地时区位于格林威治以西的时区数值。由于在 MS-DOS 环境下无法预先确定这个值，所以，在缺省的条件下，localtime( )函数的返回值将与 gmtime( )的相同。

3. 例　程

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

```
char * wday[] = {
"Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday"
};
void
main (void)
{
time_t clock;
struct tm * tp;
time(&clock);
tp = localtime(&clock);
printf("Today is %s\n",wday[tp->tm_wday]);
}
```

4. 参　阅

ctime( ),asctime( ),time( )。

5. 返回值

本函数返回 tm 结构体型数据。

注意,由于编译器不会提供 time( )程序,故它需由用户给定。详情请参阅 time( )函数。

6. 数据结构

```
typedef long time_t;
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

3.29　LOG,LOG10 函数

1. 提　要

```
#include <math.h>
```

```
double log (double f)
double log10 (double f)
```

2. 描　述

Log( )函数返回 f 的自然对数值。Log10( )函数返回 f 以 10 为底的对数值。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
double f;
for(f = 1.0;f <= 10.0;f += 1.0)
printf("log(%1.0f) = %f\n",f,log(f));
}
```

4. 参　阅

exp( ),pow( )。

5. 返回值

如果函数参数为负,返回值为 0。

3.30　MEMCHR 函数

1. 提　要

```
#include <string.h>
/* 初级和中级系列单片机 */
const void * memchr (const void * block,int val,size_t length)
/* 高级系列单片机 */
void * memchr (const void * block,int val,size_t length)
```

2. 描　述

memchr( )函数与 strchr( )函数在功能上类似;但前者没有在字符串中寻找 null(空)终止字符的功能。memchr( )函数实现在一段规定了长度的内存区域中寻找特定的字节。它的函数参数包括指向被寻内存区域的指针、被寻字节的值及被寻内存区域的长度。函数将返回一个指针,该指针指向被寻内存区域中被寻字节首次出现的单元。

3. 例　程

```
#include <string.h>
```

```
#include <stdio.h>
unsigned int ary[ ] = {1,5,0x6789,0x23};
void
main (void)
{
char * cp;
cp = memchr(ary,0x89,sizeof ary);
if(! cp)
printf("not found\n");
else
printf("Found at offset %u\n",cp - (char *)ary);
}
```

4. 参　阅

strchr( )。

5. 返回值

函数返回指针。该指针指向被寻内存区域中被寻字节首次出现的单元；否则返回 NULL。

3.31　MEMCMP 函数

1. 提　要

```
#include <string.h>
int memcmp (const void * s1, const void * s2, size_t n)
```

2. 描　述

memcmp()函数的功能是，比较 2 块长度为 n 的内存中变量的大小，类似 strncmp()函数，返回一个有符号数。与 strncmp()函数不同的是，memcmp()函数没有空格结束符。ASCII 码字符顺序被用来比较；但如果内存块中包含非 ASCII 码字符，则返回值不确定。测试是否相等总是可靠的。

3. 例　程

```
#include <stdio.h>
#include <string.h>
void
main (void)
{
int buf[10], cow[10], i;
```

```
buf[0] = 1;
buf[2] = 4;
cow[0] = 1;
cow[2] = 5;
buf[1] = 3;
cow[1] = 3;
i = memcmp(buf, cow, 3 * sizeof(int));
if(i < 0)
printf("less than\n");
else if(i > 0)
printf("Greater than\n");
else
printf("Equal\n");
}
```

4. 参 阅

strncpy(),strncmp(),strchr(),memset(),memchr()。

5. 返回值

当内存块变量 s1 分别小于、等于或大于内存块变量 s2 时,函数返回值分别为－1,0 或 1。

3.32 MEMCPY 函数

1. 提 要

```
#include <string.h>
/* 低级或中级系列单片机 */
void * memcpy (void * d, const void * s, size_t n)
/* 高级系列单片机 */
far void * memcpy (far void * d, const void * s, size_t n)
```

2. 描 述

memcpy()函数的功能是,将指针 s 指向的、内存开始的 n 个字节复制到指针 d 指向的、内存开始的单元。复制重叠区的结果不确定。与 strcpy()函数不同的是,memcpy()复制的是一定数量的字节,而不是复制所有结束符前的数据。

3. 例 程

```
#include <string.h>
#include <stdio.h>
void
```

```
main (void)
{
char buf[80];
memset(buf, 0, sizeof buf);
memcpy(buf, "a partial string", 10);
printf("buf = '%s'\n", buf);
}
```

4. 参　阅

strncpy(),strncmp(),strchr(),memset()。

5. 返回值

memcpy()函数返回值为函数的第 1 个参数。

3.33　MEMMOVE 函数

1. 提　要

```
#include <string.h>
/* 低级或中级系列单片机 */
void * memmove (void * s1, const void * s2, size_t n)
/* 高级系列单片机 */
far void * memmove (far void * s1, const void * s2, size_t n)
```

2. 描　述

memmove()函数与 memcpy()函数相似;但 memmove()函数能对重叠区进行准确的复制。也就是说,它可以适当向前或向后,正确地从一个块复制到另一个块,并将它覆盖。

3. 参　阅

strncpy(), strncmp(), strchr(), memcpy()。

4. 返回值

memmove()函数同样返回它的第 1 个参数。

3.34　MEMSET 函数

1. 提　要

```
#include <string.h>
/* 低级和中级系列的单片机 */
void * memset (void * s, int c, size_t n)
```

```
/* 高级系列单片机 */
far void * memset (far void * s, int c, size_t n)
```

2. 描　述

memset()函数将字节 c 存储到指针 s 指向的、内存开始的 n 个内存字节。

3. 例　程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
char abuf[20];
strcpy(abuf, "This is a string");
memset(abuf, 'x', 5);
printf("buf = '%s'\n", abuf);
}
```

4. 参　阅

strncpy(), strncmp(), strchr(), memcpy(), memchr()。

3.35　MODF 函数

1. 提　要

```
#include <math.h>
double modf (double value, double * iptr)
```

2. 描　述

modf()函数将参数 value 分为整数和小数 2 部分，每 1 部分都和 value 的符号相同。例如，−3.17 被分为整数部分(−3)和小数部分(−0.17)。其中整数部分以双精度数据类型存储在指针 iptr 指向的单元中。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
double i_val, f_val;
```

```
f_val = modf( -3.17, &i_val);
}
```

4. 返回值

函数返回值为 value 的带符号小数部分。

3.36　PERSIST_CHECK，PERSIST_VALIDATE 函数

1. 提　要

```
# include <sys. h>
int persist_check (int flag)
void persist_validate (void)
```

2. 描　述

persist_check()函数要用到非可变(non-volatile)的 RAM 变量，这些变量在定义时被加上限定词 persistent。在测试 NVRAM(非可变 RAM)区域时，先调用 persist_validate()函数，并用到一个存储在隐藏变量中的虚拟数据，且由 persist_validate()函数计算得到一个测试结果。如果虚拟数据和测试结果都正确，则返回值为真(非 0)；如果都不正确，则返回 0。在这种情况下，函数返回 0 并且重新检测 NVRAM 区域(通过调用 persist_validate()函数)。函数被执行的条件是，标志变量不为 0。persist_validate()函数应该在每次转换为永久变量之后调用。它将重新建立虚拟数据和计算测试结果。

3. 例　程

```
# include <sys. h>
# include <stdio. h>
persistent long reset_count;
void
main (void)
{
if(! persist_check(1))
printf("Reset count invalid - zeroed\n");
else
printf("Reset number %ld\n", reset_count);
reset_count++; /* update count */
persist_validate(); /* and checksum */
for(;;)
continue; /* sleep until next reset */
}
```

4. 返回值

如果 NVRAM 区域无效，则返回值为假(0)；如果 NVRAM 区域有效，则返回值为真(非 0)。

3.37 POW 函数

1. 提 要

```
# include <math. h>
double pow (double f, double p)
```

2. 描 述

pow()函数表示第 1 个参数 f 的 p 次幂。

3. 例 程

```
# include <math. h>
# include <stdio. h>
void
main (void)
{
double f;
for(f = 1.0 ; f <= 10.0 ; f += 1.0)
printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

4. 参 阅

log(), log10(), exp()。

5. 返回值

返回值为 f 的 p 次幂。

3.38 PRINTF 函数

1. 提 要

```
# include <stdio. h>
unsigned char printf (const char * fmt, ...)
```

2. 描 述

printf()函数是一个格式输出子程序，其运行的基础是标准输出(staout)。它有对应的程序形成字符缓冲区(sprintf()函数)。printf()函数以格式字符串、一系列 0 及其他作为参数。

格式字符串都转换为一定的格式，每一规格化都用来输出变量表。

转换格式的形式为%m.nc。其中%表示格式，m 表示选择的字符宽度，n 表示选择的精度，c 表示规格类型。字符宽度和精度只适于中级和高级系列单片机，并且精度只对格式%s 有效。

如果指针变量为十进制常数，例如格式为% * d 时，则一个整型数将从表中被取出，提供给指针变量。对低级系列单片机而言，有下列转换格式：

o x X u d　分别为整型格式——八进制、十六进制、十六进制、十进制及十进制。其中 d 为有符号十进制数，其他为无符号数。其精度值为被输出数的总的位数，也可以强制在前面加 0。例如%8.4x 将产生一 8 bit 的十六进制数，其中前 4 bit 为 0，后为 4 bit 十六进制数。X 输出的十六进制数中，字母为 A～F；x 输出的十六进制数中，字母为 a～f。当格式发生变化时，八进制格式前要加 0，十六进制格式的前面要加 0x 或 0X。

s　打印一个字符串——函数参数值被认为是字符型指针。最多从字符串中取 n 个字符打印，字符宽度为 m。

c　函数参数被认为一个单字节字符并可自由打印。任何其他有格式规定的字符将被输出，则%%将产生一个百分号。对中级和高级系列单片机而言，转换格式在低级系列单片机的基础上再加上：

l　长整型格式——在整型格式前加上关键字母 l，即表示长整型变量。

f　浮点格式——总的宽度为 m，小数点后的位数为 n。如果 n 没有写出，则默认为 6。如果精度为 0，则小数点被省略，除非精度已预先定义。

3. 例　程

printf("Total = %4d%%", 23)

输出为"Total = 23%"

printf("Size is %lx" , size)

这里 size 为长整型十六进制变量。当使用%s 时，精度只对中级和高级系列单片机有效。

printf("Name = %.8s", "a1234567890")

输出为 "Name = a1234567"

字符变量宽度只对中级和高级系列单片机有效。

printf("xx% * d", 3, 4)

输出为 "xx 4"

```
/* vprintf 例程 */
#include <stdio.h>
int
error (char * s, ...)
{
va_list ap;
```

```
va_start(ap, s);
printf("Error: ");
vprintf(s, ap);
putchar('\n');
va_end(ap);
}
void
main (void)
{
int i;
i = 3;
error("testing 1 2 %d", i);
}
```

4. 参　阅

sprintf()。

5. 返回值

printf()将返回的字符值写到标准输出口。注意返回值为字符型,而不是整型。

注意,printf()函数的部分特征只对中级和高级系列单片机有效。详见描述部分。输出浮点数要求浮点数不大于最大长整型变量。为了使用长整型变量或浮点数格式,必须将适当的函数库包含进来。参见有关 PICC - L 的描述及有关 HPDPIC 长整型格式在 printf 的菜单选项。

3.39 RAND 函数

1. 提　要

```
#include <stdlib.h>
int rand (void)
```

2. 描　述

rand()函数用来产生 1 个随机数数据。它返回 1 个 0～32 767 的整数,并且这个整数在每次被调用后,以随机数据形式出现。这一运算规则将产生一个从同一起点开始的确定顺序。起点通过调用 srand()函数获得。下面的例程说明了每次通过调用 time()函数获得不同的起点。

3. 例　程

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

```
void
main (void)
{
time_t toc;
int i;
time(&toc);
srand((int)toc);
for(i = 0 ; i ! = 10 ; i++)
printf("%d\t", rand());
putchar('\n');
}
```

4. 参　阅

srand()。

注意,例程中需要用户自己提供 time()函数,因为它不能由汇编器产生。更详细的情况参见 time()函数。

3.40　SIN 函数

1. 提　要

```
# include <math. h>
double sin (double f)
```

2. 描　述

这个函数返回参数的正弦值。

3. 例　程

```
# include <math. h>
# include <stdio. h>
# define C 3. 141592/180. 0
void
main (void)
{
double i;
for(i = 0 ; i <= 180. 0 ; i += 10)
printf("sin(%3. 0f) = %f, cos = %f\n", i, sin(i * C), cos(i * C));
}
```

4. 参　阅

cos(), tan(), asin(), acos(), atan(), atan2()。

5. 返回值

返回值为参数 f 的正弦值。

3.41 SPRINTF 函数

1. 提　要

```
#include <stdio.h>
/* 中级和低级系列单片机 */
unsigned char sprintf (char * buf, const char * fmt, ...)
/* 高级系列单片机 */
unsigned char sprintf (far char * buf, const char * fmt, ...)
```

2. 描　述

sprintf()函数和 printf()函数操作基本相同;只是输出在不同的输出终端,所有的字符被放到 buf 缓冲器。字符串带空格结束符,buf 缓冲器中的数据被返回。

3. 参　阅

printf()。

4. 返回值

sprintf()函数的返回值为被放入缓冲器中的数据。注意,返回值为字符型而非整型。

注意,对高级单片机而言,缓冲器是通过长指针访问的。

3.42 SQRT 函数

1. 提　要

```
#include <math.h>
double sqrt (double f)
```

2. 描　述

sqrt()函数利用牛顿法得到参数的近似平方根。

3. 例　程

```
#include <math.h>
#include <stdio.h>
void
main (void)
{
```

```
double i;
for(i = 0 ; i <= 20.0 ; i += 1.0)
printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

4. 参　阅

exp()。

5. 返回值

返回值为参数的平方根。

注意，如果参数为负，则出现错误。

3.43　SRAND 函数

1. 提　要

```
#include <stdlib.h>
void srand (unsigned int seed)
```

2. 描　述

srand()函数是在调用 rand()函数时，被用来初始化随机数据发生器的。它为 rand()函数产生不同起点虚拟数据顺序提供一个机制。在 Z80 上，随机数据最好从新的寄存器获得；否则控制台的响应时间或系统时间将充当这一数据。

3. 例　程

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
void
main (void)
{
time_t toc;
int i;
time(&toc);
srand((int)toc);
for(i = 0 ; i ! = 10 ; i++)
printf("%d\t", rand());
putchar('\n');
}
```

4. 参　阅

rand()。

3.44　STRCAT 函数

1. 提　要

```
#include <string.h>
/* 中级和低级系列单片机 */
char * strcat (char * s1, const char * s2)
/* 高级系列单片机 */
far char * strcat (far char * s1, const char * s2)
```

2. 描　述

这个函数将字符串 s2 连接到字符串 s1 的后面。新的字符串以空格作为结束符。指针型参数 s1 指向的字符数组必须保证大于结果字符串。

3. 例　程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
char buffer[256];
char * s1, * s2;
strcpy(buffer, "Start of line");
s1 = buffer;
s2 = "... end of line";
strcat(s1, s2);
printf("Length = %d\n", strlen(buffer));
printf("string = \"%s\"\n", buffer);
}
```

4. 参　阅

strcpy(), strcmp(), strncat(), strlen()。

5. 返回值

即为字符串 s1。

3.45　STRCHR, STRICHR 函数

1. 提　要

```
#include <string.h>
/* 中级和低级系列单片机 */
const char * strchr (const char * s, int c)
const char * strichr (const char * s, int c)
/* 高级系列单片机 */
char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

2. 描　述

strchr()函数查找字符串 s 中是否出现字符变量 c。

如果找到了,则字符指针被返回;否则返回 0。

strichr()函数与 strchr()函数作用相同。

3. 例　程

```
#include <strings.h>
#include <stdio.h>
void
main (void)
{
static char temp[] = "Here it is...";
char c = 's';
if(strchr(temp, c))
printf("Character %c was found in string\n", c);
else
printf("No character was found in string");
}
```

4. 参　阅

strrchr(), strlen(), strcmp()。

5. 返回值

如果找到,则返回第 1 个字符的指针;否则返回 0。

注意,函数对字符使用整型参数,只有低 8 位有效。

3.46 STRCMP, STRICMP 函数

1. 提 要

#include <string.h>

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

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

2. 描 述

strcmp()函数用来比较 2 个字符串的大小。

字符串带有空格结束符,根据字符串 s1 是否小于、等于或大于字符串 s2,返回一个有符号整数。

比较是根据 ASCII 字母的顺序表进行的。stricmp()函数和 strcmp()函数功能完全一样。

3. 例 程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
int i;
if((i = strcmp("ABC", "ABc")) < 0)
printf("ABC is less than ABc\n");
else if(i > 0)
printf("ABC is greater than ABc\n");
else
printf("ABC is equal to ABc\n");
}
```

4. 参 阅

strlen(), strncmp(), strcpy(), strcat()。

5. 返回值

返回一个有符号整数。

注意,其他的 C 应用程序也可以采用不同的字母顺序表。返回值为正、0 或负,即不一定是-1 或 1。

3.47　STRCPY 函数

1. 提　要

```
#include <string.h>
/* 低级和中级系列单片机 */
char * strcpy (char * s1, const char * s2)
/* 高级系列单片机 */
far char * strcpy (far char * s1, const char * s2)
```

2. 描　述

这个函数将以空格键结束的字符串 s2 拷贝到 s1 指向的字符数组。目的数组必须足够大，以容纳包括空格在内的字符串 s2。

3. 例　程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
char buffer[256];
char * s1, * s2;
strcpy(buffer, "Start of line");
s1 = buffer;
s2 = " ... end of line";
strcat(s1, s2);
strcat(s1, s2);
printf("Length = %d\n", strlen(buffer));
printf("string = \"%s\"\n", buffer);
}
```

4. 参　阅

strncpy(), strlen(), strcat(), strlen()。

5. 返回值

目的数组被返回。

3.48 STRCSPN 函数

1. 提 要

#include <string.h>

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

2. 描 述

strcspn()函数用于取得在字符串常数 s1 中有、而字符串常数 s2 中没有的字符的长度。

3. 例 程

```
#include <stdio.h>
#include <string.h>
void
main (void)
{
static char set[] = "xyz";
printf("%d\n", strcspn( "abcdevwxyz", set));
printf("%d\n", strcspn( "xxxbcadefs", set));
printf("%d\n", strcspn( "1234567890", set));
}
```

4. 参 阅

strspn()。

5. 返回值

剩余部分的长度。

3.49 STRLEN 函数

1. 提 要

#include <string.h>

size_t strlen (const char * s)

2. 描 述

strlen()函数用来测量字符串 s1 的长度,不包括空格结束符。

3. 例 程

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

```
void
main (void)
{
char buffer[256];
char * s1, * s2;
strcpy(buffer, "Start of line");
s1 = buffer;
s2 = " ... end of line";
strcat(s1, s2);
printf("Length = %d\n", strlen(buffer));
printf("string = \"%s\"\n", buffer);
}
```

4. 返回值

不包括结束符在内的字符长度。

3.50　STRNCAT 函数

1. 提　要

```
#include <string.h>
/* 低级和中级系列单片机 */
char * strncat (char * s1, const char * s2, size_t n)
/* 高级系列单片机 */
far char * strncat (far char * s1, const char * s2, size_t n)
```

2. 描　述

函数将字符串 s2 连接到字符串 s1 的尾端。最多只有 n 个字符被拷贝，结果包含空格结束符。指针 s1 指向的字符数组应足够大，以容纳结果字符串。

3. 例　程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
char buffer[256];
char * s1, * s2;
strcpy(buffer, "Start of line");
```

```
s1 = buffer;
s2 = "... end of line";
strncat(s1, s2, 5);
printf("Length = %d\n", strlen(buffer));
printf("string = \"%s\"\n", buffer);
}
```

4. 参 阅

strcpy(), strcmp(), strcat(), strlen()。

5. 返回值

字符串 s1。

3.51 STRNCMP, STRNICMP 函数

1. 提 要

```
#include <string.h>
int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

2. 描 述

strncmp()函数用来比较 2 个带有空格结束符的字符串的大小,最多比较 n 个字符。根据字符串 s1 是否小于、等于或大于字符串 s2,返回 1 个有符号数。比较是根据 ASCII 字母顺序表进行的。strnicmp()函数与之相同。

3. 例 程

```
#include <stdio.h>
#include <string.h>
void
main (void)
{
int i;
i = strcmp("abcxyz", "abcxyz");
if(i == 0)
printf("Both strings are equal\n");
else if(i > 0)
printf("String 2 less than string 1\n");
else
printf("String 2 is greater than string 1\n");
```

```
}
```

4. 参　阅

strlen(), strcmp(), strcpy(), strcat()。

5. 返回值

有符号整数。

注意,其他的 C 应用函数可以采用不同的字母顺序。返回值为负、0 或正,并不一定是－1 或 1。

3.52　STRNCPY 函数

1. 提　要

```
#include <string.h>
  * 低级和中级系列单片机 */
char * strncpy (char * s1, const char * s2, size_t n)
/* 高级系列单片机 */
far char * strncpy (far char * s1, const char * s2, size_t n)
```

2. 描　述

这个函数将带结束符的字符串 s2 拷贝到字符指针 s1 指向的字符数组。最多有 n 个字符被拷贝。如果 s2 的长度大于 n,则结果中不包含结束符。目的数组必须足够大,以容纳包括结束符在内的新字符串。

3. 例　程

```
#include <string.h>
#include <stdio.h>
void
main (void)
{
char buffer[256];
char * s1, * s2;
strncpy(buffer, "Start of line", 6);
s1 = buffer;
s2 = "... end of line";
strcat(s1, s2);
printf("Length = %d\n", strlen(buffer));
printf("string = \"%s\"\n", buffer);
```

4. 参　阅

strcpy()，strcat()，strlen()，strcmp()。

5. 返回值

指针 s1 指向的目的缓冲区。

3.53　STRPBRK 函数

1. 提　要

```
#include <string.h>
/* 低级和中级系列单片机 */
const char * strpbrk(const char * s1, const char * s2)
/* 高级系列单片机 */
char * strpbrk(const char * s1, const char * s2)
```

2. 描　述

strpbrk()函数查找字符串 s1 是否包含字符串 s2 的字符。如果包含，则返回被找到的第 1 个字符的指针；否则，不返回任何值。

3. 例　程

```
#include <stdio.h>
#include <string.h>
void
main(void)
{
char * str = "This is a string.";
while(str != NULL) {
printf("%s\n", str);
str = strpbrk(str+1, "aeiou");
}
}
```

4. 返回值

第 1 个匹配的字符；否则返回值为空。

3.54　STRRCHR，STRRICHR 函数

1. 提　要

```
#include <string.h>
/* 中级和低级系列单片机 */
const char * strrchr (char * s, int c)
const char * strrichr (char * s, int c)
/* 高级系列单片机 */
char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

2. 描　述

strrchr()函数和 strchr()函数相似；但它从字符串的尾端开始查找，即其返回值为字符 c 最后一次在字符串中出现时的指针。如果没出现，则返回值为空。strrichr()函数和 strrchr()函数完全一样。

3. 例　程

```
#include <stdio.h>
#include <string.h>
void
main (void)
{
char * str = "This is a string.";
while(str != NULL) {
printf( "%s\n", str );
str = strrchr( str+1, 's');
}
}
```

4. 参　阅

strchr()，strlen()，strcmp()，strcpy()，strcat()。

5. 返回值

字符指针，或者返回值为空。

3.55 STRSPN 函数

1. 提 要

```
#include <string.h>
size_t strspn (const char * s1, const char * s2)
```

2. 描 述

strspn()函数返回字符串 s1 中包含的、完全由字符串 s2 组成的字符的长度。

3. 例 程

```
#include <stdio.h>
#include <string.h>
void
main (void)
{
printf("%d\n", strspn("This is a string", "This"));
printf("%d\n", strspn("This is a string", "this"));
}
```

4. 参 阅

strcspn()。

5. 返回值

部分长度。

3.56 STRSTR, STRISTR 函数

1. 提 要

```
#include <string.h>
/* 中级和低级系列单片机 */
const char * strstr (const char * s1, const char * s2)
const char * stristr (const char * s1, const char * s2)
/* 高级系列单片机 */
char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

2. 描 述

strstr()函数返回字符数组 s1 中第 1 次出现字符数组 s2 的指针位置。stristr()函数与之