MPLAB® C18
C 编译器
函数库

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。 Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了《数字器件千年版权法案 （Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其他受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip 位于美国亚利桑那州 Chandler 和 Tempe 及位于加利福尼亚州 Mountain View 的全球总部、设计中心和晶圆生产厂均于 2003 年 10 月通过了 ISO/TS-16949:2002 质量体系认证。公司在 PICmicro® 8 位单片机、KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品方面的质量体系流程均符合 ISO/TS-16949:2002。此外，Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。

目录

前言

客户须知

所有文档均会过时，本文档也不例外。 Microchip 的工具和文档将不断演变以满足客户的需求，因此实际使用中有些对话框和 / 或工具说明可能与本文档所述之内容有所不同。请访问我们的网站（www.microchip.com）获取最新文档。

文档均标记有 “DS”编号。该编号出现在每页底部的页码之前。 DS 编号的命名约定为 “DSXXXXXA”，其中 “XXXXX”为文档编号，“A”为文档版本。

欲了解开发工具的最新信息，请参考 MPLAB® IDE 在线帮助。从 Help （帮助）菜单选择 Topics （主题），打开现有在线帮助文件列表。

简介

本文档旨在提供可供 Microchip MPLAB® C18 C 编译器使用的函数库和预编译目标文件的详细信息。

文档内容编排

文档内容编排如下：

- **第 1 章：概述 —** 描述提供的函数库和预编译目标文件。
- **第 2 章：硬件外设函数库** — 描述每个硬件外设库函数。
- **第 3 章：软件外设函数库** — 描述每个软件外设库函数。
- **第 4 章：通用软件函数库** — 描述每个通用软件库函数。
- **第 5 章：数学函数库** — 讲述数学库函数。
- **术语表** — 包括本指南使用的术语。
- **索引** — 本文档中的术语、特性以及各个章节的交叉引用列表。

本文档使用的约定

本文档使用如下文档约定：

文档约定

说明	涵义	示例
Arial 字体：		
斜体字	参考书目	*MPLAB® IDE User's Guide*
	需强调的文字	*... 仅有*的编译器 ...
Courier 字体：		
常规 Courier	源代码示例	`#define START`
	文件名	`autoexec.bat`
	文件路径	`c:\mcc18\h`
	关键字	`_asm, _endasm, static`
	命令行选项	`-Opa+, -Opa-`
	位值	`0, 1`
	常数	`0xFF, 'A'`
斜体 Courier	可变参数	*file*.o，其中 *file* 可以是任一有效文件名
0x*nnnn*	十六进制数， *n* 是一个十六进制数字	0xFFFF，0x007A
方括号 [ ]	可选参数	`mcc18 [options]` *file* `[options]`
花括号和竖线：{ | }	选择互斥参数； “或”选择	`errorlevel {0|1}`
省略号 ...	代替重复文字	`var_name [, var_name...]`
	表示由用户提供的代码	`void main (void) { ... }`

推荐读物

要了解更多关于编译器的函数库和预编译目标文件、 MPLAB IDE 及其他工具使用方面的信息，请阅读以下推荐读物。

readme.c18

关于使用 MPLAB C18 C 编译器的最新信息，请阅读本软件自带的 readme.c18 文件（ASCII 文本）。此 readme 文件包含了本文档可能未提供的更新信息。

readme.xxx

需要其他 Microchip 工具的最新信息 （MPLAB IDE 和 MPLINK™ 链接器等），请阅读软件自带的相关 readme 文件 （ASCII 文本文件）。

MPLAB® C18 C 编译器入门 （DS51295E_CN）

讲述如何安装 MPLAB C18 编译器，如何编写简单的程序以及如何在 MPLAB IDE 中使用编译器。

MPLAB® C18 C 编译器用户指南 （DS51288J_CN）

一个综合指南，讲述了针对 PIC18 器件设计的 Microchip MPLAB C18 C 编译器的使用及特征。

MPLAB® IDE V6.XX 快速入门指南 （DS51281C_CN）

介绍如何安装 MPLAB IDE 软件及如何使用 IDE 创建项目并烧写器件。

MPASM™ 汇编器、 MPLINK™ 目标链接器和 MPLIB™ 目标库管理器用户指南（DS33014J_CN）

这个用户指南描述了如何使用 Microchip 的 PICmicro® 单片机 （MCU）汇编器（MPASM ）、链接器 （MPLINK）和库管理器 （MPLIB）。

PICmicro® 18C 单片机系列参考手册 （DS39500A_CN）

重点介绍增强型单片机系列。说明了增强型单片机系列架构和外设模块的工作原理，但没有涉及到每个器件的具体细节。

PIC18 器件数据手册和应用笔记

讲述 PIC18 器件工作和电气特性的数据手册。应用笔记介绍了如何使用 PIC18 器件。

要获得上述任何文档，请访问 Microchip 的网站（www.microchip.com），获得 Adobe Acrobat （.pdf）格式的文档。

MICROCHIP 网站

Microchip 网站 （www.microchip.com）为客户提供在线支持。客户可通过该网站方便地获取文件和信息。只要使用常用的因特网浏览器即可访问。网站提供以下信息：

- **产品支持**——数据手册和勘误表、应用笔记和样本程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及存档软件
- **一般技术支持**——常见问题 （FAQ）、技术支持请求、在线讨论组以及 Microchip 顾问计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、 Microchip 销售办事处、代理商以及工厂代表列表

开发系统变更通知客户服务

Microchip 启动了客户通知服务，来帮助客户轻松获得关于 Microchip 产品的最新信息。订阅此项服务后，每当您指定的产品系列或感兴趣的开发工具有更改、更新、改进或有勘误时，您都会收到电子邮件通知。

登录 Microchip 网站 （http://www.microchip.com），点击 “客户变更通知”。按照指示注册。

开发系统产品组分类如下：

- **编译器** — 关于 Microchip C 编译器和其他语言工具的最新信息。这些工具包括 MPLAB® C17、 MPLAB C18 和 MPLAB C30 C 编译器； MPASM™ 和 MPLAB ASM30 汇编器； MPLINK™ 和 MPLAB LINK30 目标链接器； MPLIB™ 和 MPLAB LIB30 目标库管理器。
- **仿真器** — 关于Microchip在线仿真器的最新信息。包括MPLAB ICE 2000和MPLAB ICE 4000。
- **在线调试器** — 关于 Microchip 在线调试器的最新信息，包括 MPLAB ICD 2。
- **MPLAB IDE** — 关于 Microchip MPLAB® IDE 的最新信息，它是开发系统工具的 Windows® 集成开发环境。重点介绍 MPLAB IDE、 MPLAB SIM 软件仿真器、 MPLAB IDE 项目管理器以及一般的编辑和调试功能。
- **编程器** — 关于 Microchip 器件编程器的最新信息。编程器包括 MPLAB PM3 和 PRO MATE® II 器件编程器，以及 PICSTART® Plus 开发编程器。

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师 （FAE）
- 技术支持
- 开发系统信息热线

客户应联系其代理商、代表或应用工程师 （FAE）寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。

也可通过 http://support.microchip.com 获得网上技术支持。

第 1 章 概述

1.1 简介

本章概括了可在应用程序中包含的 MPLAB C18 库文件和预编译目标文件。

1.2 MPLAB C18 函数库概述

函数库是为便于引用和链接而分类形成的函数集合。可参阅 *《MPASM™ 汇编器、MPLINK™ 目标链接器和 MPLIB™ 目标库管理器用户指南》*（DS33014J_CN），获得更多关于创建和维护函数库的信息。

MPLAB C18 函数库在安装目录下的 `lib` 子目录中。这些函数库可通过 MPLINK 链接器直接链接到应用程序中。

这些文件在 `c:\mcc18\src` 目录下进行预编译。目录 `src\traditional` 包含非扩展模式的文件，目录 `src\extended` 包含扩展模式的文件。假如你选择不把编译器和相关文件安装到 `c:\mcc18` 目录下，那么链接器列表文件中不会显示函数库的源代码，使用 MPLAB IDE 时也不能单步执行函数库的源代码。

为了在 `.lst` 文件中包含库函数代码和能够单步执行库函数，可以按照**第 1.3.3 节**、**第 1.4.3 节**和**第 1.5.3 节**中的说明，使用 `src`、 `src\traditional` 和 `src\extended` 目录下的批处理文件 （`.bat`）重建函数库。

1.3 启动代码

1.3.1 概述

MPLAB C18 提供了三个版本的启动代码，其初始化级别不同。`c018*.o` 目标文件用于工作在非扩展模式的编译器。`c018*_e.o` 目标文件用于工作在扩展模式的编译器。按照复杂程度递增的顺序排列为：

`c018.o/c018_e.o` 初始化 C 软件堆栈，然后跳转到应用函数 `main()` 的开头。

`c018i.o/c018i_e.o` 执行所有与 `c018.o/c018_e.o` 相同的任务，且在调用用户的应用程序之前，为需要初始化的数据赋值。如果全局变量或静态变量在定义时已赋值，也需要进行初始化。这是包含在随 MPLAB C18 提供的链接描述文件中的启动代码。

`c018iz.o/c018iz_e.o` 执行所有与 `c018.o/c018_e.o` 相同的任务，并按照严格符合 ANSI 的要求，将所有未初始化的变量赋值为 0。

1.3.2 源代码

启动子程序的源代码保存在编译器安装目录的 src\traditional\ startup 和 src\extended\startup 子目录中。

1.3.3 重建

批处理文件 makestartup.bat 可用来创建启动代码，并把生成的目标文件复制到 lib 目录中。

使用 makestartup.bat 重建代码之前，检查 MPLAB C18（mcc18.exe）是否在正确的路径中。

1.4 处理器内核函数库

1.4.1 概述

标准 C 函数库（clib.lib 或 clib_e.lib）提供了 PIC18 内核架构支持的函数，本系列中所有处理器都支持这些函数。将在以下章节中描述这些函数：

- **第 4 章 “通用软件函数库”**
- **第 5 章 “数学函数库”**

1.4.2 源代码

可以在编译器安装目录的下列子目录中找到标准 C 函数库中函数的源代码：

- src\traditional\math
- src\extended\math
- src\traditional\delays
- src\extended\delays
- src\traditional\stdclib
- src\extended\stdclib

1.4.3 重建

可以使用批处理文件 makeclib.bat 重建处理器内核函数库。在调用这个批处理文件之前，确认下列工具在相应路径中：

- MPLAB C18 （mcc18.exe）
- MPASM 汇编器 （mpasm.exe）
- MPLIB 库管理器 （mplib.exe）

在重建标准 C 函数库之前，确保环境变量 MCC_INCLUDE 已经设置为 MPLAB C18 头文件的路径 （如 c:\mcc18\h）。

1.5 特定处理器的函数库

1.5.1 概述

特定处理器的库文件包含 PIC18 系列各成员的定义，对于不同的处理器，这些定义可能有所不同。其中包括所有外设子程序和特殊功能寄存器 （Special Function Register， SFR）定义。所提供的外设子程序包括为使用硬件外设设计的子程序以及使用通用 I/O 口实现外设接口的子程序。以下章节描述特定处理器函数库中的函数：

- **第 2 章 “硬件外设函数”**
- **第 3 章 “软件外设函数库”**

特定处理器的函数库命名为：

p *processor*.lib — 非扩展模式特定处理器函数库

p *processor*_e.lib — 扩展模式特定处理器函数库

例如，对于 PIC18F4620 库的非扩展版本，其库文件命名为 p18f4620.lib；对于库的扩展版本，其库文件命名为 p18f4620_e.lib。

1.5.2 源代码

特定处理器函数库的源代码可在编译器安装目录的以下子目录中找到：

- src\traditional\pmc
- src\extended\pmc
- src\traditional\proc
- src\extended\proc

1.5.3 重建

可以使用批处理文件 makeplib.bat 重建特定处理器的函数库。在调用此批处理文件之前，要确保下列工具在相应路径中：

- MPLAB C18 （mcc18.exe）
- MPASM 汇编器 （mpasm.exe）
- MPLIB 库管理器 （mplib.exe）

在调用 makeplib.bat 之前，确保环境变量 MCC_INCLUDE 已经设置为 MPLAB C18 头文件的路径 （如 c:\mcc18\h）。

注：

第 2 章 硬件外设函数

2.1 简介

本章描述特定处理器函数库中的硬件外设函数，所有这些函数的源代码都包含在 MPLAB C18 编译器安装目录的 `src\pmc` 和 `src\extended\pmc` 子目录下。

更多有关使用 MPLIB 函数库管理器管理函数库的信息，可参阅 *《MPASM™ 汇编器、MPLINK™ 目标链接器和 MPLIB™ 目标库管理器用户指南》*（DS33014J_CN）。

MPLAB C18 库函数支持下列外设：

- A/D 转换器 （**第 2.2 节 “A/D 转换器函数”**）
- 输入捕捉 （**第 2.3 节 “输入捕捉函数”**）
- I²C™ （**第 2.4 节 “I²C™ 函数”**）
- I/O 口 （**第 2.5 节 “I/O 口函数”**）
- Microwire （**第 2.6 节 “Microwire 函数”**）
- 脉宽调制 （Pulse-Width Modulation， PWM）（**第 2.7 节 “脉宽调制函数”**）
- SPI （**第 2.8 节 “SPI 函数”**）
- 定时器 （**第 2.9 节 “定时器函数”**）
- USART （**第 2.10 节 “USART 函数”**）

2.2 A/D 转换器函数

下列函数支持 A/D 外设：

表 2-1： A/D 转换器函数

函数	描述
`BusyADC`	A/D 转换器是否正在进行转换？
`CloseADC`	禁止 A/D 转换器。
`ConvertADC`	启动 A/D 转换。
`OpenADC`	配置 A/D 转换器。
`ReadADC`	读取 A/D 转换的结果。
`SetChanADC`	选择要使用的 A/D 通道。

2.2.1 函数描述

BusyADC

功能： A/D 转换器是否正在进行转换？

头文件： adc.h

函数原型：
```
char BusyADC( void );
```

说明： 该函数表明 A/D 外设是否正在进行转换。

返回值： 如果 A/D 外设正在进行转换，为 1；
如果 A/D 外设不在进行转换，为 0。

文件名： adcbusy.c

CloseADC

功能： 禁止 A/D 转换器。

头文件： adc.h

函数原型：
```
void CloseADC( void );
```

说明： 该函数禁止 A/D 转换器和 A/D 中断机制。

文件名： adcclose.c

ConvertADC

功能： 启动 A/D 转换过程。

头文件： adc.h

函数原型：
```
void ConvertADC( void );
```

说明： 该函数启动 A/D 转换。可用函数 BusyADC() 来检测转换是否完成。

文件名： adcconv.c

OpenADC
PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39

功能： 配置 A/D 转换器。

头文件： adc.h

函数原型：
```
void OpenADC( unsigned char config,
              unsigned char config2 );
```

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 adc.h 中定义。

A/D 时钟源：

ADC_FOSC_2	FOSC / 2
ADC_FOSC_4	FOSC / 4
ADC_FOSC_8	FOSC / 8
ADC_FOSC_16	FOSC / 16
ADC_FOSC_32	FOSC / 32
ADC_FOSC_64	FOSC / 64
ADC_FOSC_RC	内部 RC 振荡器

A/D 结果对齐：

ADC_RIGHT_JUST	结果向最低有效位对齐（右对齐）
ADC_LEFT_JUST	结果向最高有效位对齐（左对齐）

OpenADC
PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39 （续）

A/D 参考电压源：

ADC_8ANA_0REF	VREF+=VDD, VREF-=VSS, 所有通道都是模拟通道
ADC_7ANA_1REF	AN3=VREF+，除 AN3 外都是模拟通道
ADC_6ANA_2REF	AN3=VREF+, AN2=VREF
ADC_6ANA_0REF	VREF+=VDD, VREF-=VSS
ADC_5ANA_1REF	AN3=VREF+, VREF-=VSS
ADC_5ANA_0REF	VREF+=VDD, VREF-=VSS
ADC_4ANA_2REF	AN3=VREF+, AN2=VREF-
ADC_4ANA_1REF	AN3=VREF+
ADC_3ANA_2REF	AN3=VREF+, AN2=VREF-
ADC_3ANA_0REF	VREF+=VDD, VREF-=VSS
ADC_2ANA_2REF	AN3=VREF+, AN2=VREF-
ADC_2ANA_1REF	AN3=VREF+
ADC_1ANA_2REF	AN3=VREF+, AN2=VREF-, AN0=A
ADC_1ANA_0REF	AN0 为模拟输入
ADC_0ANA_0REF	所有通道都是数字 I/O

config2

从下面所列出各类型中分别取一个值并相与 （‘&’）所得的值。这些值在文件 adc.h 定义。

通道：

ADC_CH0	通道 0
ADC_CH1	通道 1
ADC_CH2	通道 2
ADC_CH3	通道 3
ADC_CH4	通道 4
ADC_CH5	通道 5
ADC_CH6	通道 6
ADC_CH7	通道 7

A/D 中断：

ADC_INT_ON	允许中断
ADC_INT_OFF	禁止中断

说明： 该函数把 A/D 外设复位到上电复位 （POR）状态，且根据指定的选择，配置与 A/D 相关的特殊功能寄存器 （SFR）。

文件名： adcopen.c

代码示例：

```
OpenADC( ADC_FOSC_32    &
         ADC_RIGHT_JUST &
         ADC_1ANA_0REF,
         ADC_CH0        &
         ADC_INT_OFF    );
```

OpenADC
PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20

功能： 配置 A/D 转换器。

头文件： adc.h

函数原型：
```
void OpenADC( unsigned char config,
              unsigned char config2 );
```

参数：

config

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 adc.h 定义。

A/D 时钟源：

ADC_FOSC_2	FOSC / 2
ADC_FOSC_4	FOSC / 4
ADC_FOSC_8	FOSC / 8
ADC_FOSC_16	FOSC / 16
ADC_FOSC_32	FOSC / 32
ADC_FOSC_64	FOSC / 64
ADC_FOSC_RC	内部 RC 振荡器

A/D 结果对齐：

ADC_RIGHT_JUST	结果向最低有效位对齐（右对齐）
ADC_LEFT_JUST	结果向最高有效位对齐（左对齐）

A/D 端口配置：

ADC_0ANA	所有端口都是数字端口	
ADC_1ANA	模拟端口：AN0	数字端口：AN1-AN15
ADC_2ANA	模拟端口：AN0-AN1	数字端口：AN2-AN15
ADC_3ANA	模拟端口：AN0-AN2	数字端口：AN3-AN15
ADC_4ANA	模拟端口：AN0-AN3	数字端口：AN4-AN15
ADC_5ANA	模拟端口：AN0-AN4	数字端口：AN5-AN15
ADC_6ANA	模拟端口：AN0-AN5	数字端口：AN6-AN15
ADC_7ANA	模拟端口：AN0-AN6	数字端口：AN7-AN15
ADC_8ANA	模拟端口：AN0-AN7	数字端口：AN8-AN15
ADC_9ANA	模拟端口：AN0-AN8	数字端口：AN9-AN15
ADC_10ANA	模拟端口：AN0-AN9	数字端口：AN10-AN15
ADC_11ANA	模拟端口：AN0-AN10	数字端口：AN11-AN15
ADC_12ANA	模拟端口：AN0-AN11	数字端口：AN12-AN15
ADC_13ANA	模拟端口：AN0-AN12	数字端口：AN13-AN15
ADC_14ANA	模拟端口：AN0-AN13	数字端口：AN14-AN15
ADC_15ANA	所有端口都是模拟端口	

config2

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 adc.h 定义。

OpenADC
PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20 （续）

通道：

ADC_CH0	通道 0
ADC_CH1	通道 1
ADC_CH2	通道 2
ADC_CH3	通道 3
ADC_CH4	通道 4
ADC_CH5	通道 5
ADC_CH6	通道 6
ADC_CH7	通道 7
ADC_CH8	通道 8
ADC_CH9	通道 9
ADC_CH10	通道 10
ADC_CH11	通道 11
ADC_CH12	通道 12
ADC_CH13	通道 13
ADC_CH14	通道 14
ADC_CH15	通道 15

A/D 中断：

ADC_INT_ON	允许中断
ADC_INT_OFF	禁止中断

A/D VREF+ 配置：

ADC_VREFPLUS_VDD	VREF+ = AVDD
ADC_VREFPLUS_EXT	VREF+ = 外接

A/D VREF- 配置：

ADC_VREFMINUS_VSS	VREF- = AVSS
ADC_VREFMINUS_EXT	VREF- = 外接

说明： 该函数把与 A/D 相关的寄存器复位到 POR 状态，然后配置时钟、结果格式、参考电压、端口和通道。

文件名： adcopen.c

代码示例：

```
OpenADC( ADC_FOSC_32    &
         ADC_RIGHT_JUST &
         ADC_14ANA,
         ADC_CH0        &
         ADC_INT_OFF    );
```

OpenADC
所有其他处理器

功能： 配置 A/D 转换器。

头文件： adc.h

函数原型：
```
void OpenADC(unsigned char config,
             unsigned char config2 ,
             unsigned char portconfig);
```

参数：

config

从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 adc.h 定义。

A/D 时钟源：

ADC_FOSC_2	Fosc / 2
ADC_FOSC_4	Fosc / 4
ADC_FOSC_8	Fosc / 8
ADC_FOSC_16	Fosc / 16
ADC_FOSC_32	Fosc / 32
ADC_FOSC_64	Fosc / 64
ADC_FOSC_RC	内部 RC 振荡器

A/D 结果对齐：

ADC_RIGHT_JUST	结果向最低有效位对齐 （右对齐）
ADC_LEFT_JUST	结果向最高有效位对齐 （左对齐）

A/D 采集时间选择：

ADC_0_TAD	0 Tad
ADC_2_TAD	2 Tad
ADC_4_TAD	4 Tad
ADC_6_TAD	6 Tad
ADC_8_TAD	8 Tad
ADC_12_TAD	12 Tad
ADC_16_TAD	16 Tad
ADC_20_TAD	20 Tad

config2

从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 adc.h 定义。

通道：

ADC_CH0	通道 0
ADC_CH1	通道 1
ADC_CH2	通道 2
ADC_CH3	通道 3
ADC_CH4	通道 4
ADC_CH5	通道 5
ADC_CH6	通道 6
ADC_CH7	通道 7
ADC_CH8	通道 8
ADC_CH9	通道 9
ADC_CH10	通道 10
ADC_CH11	通道 11
ADC_CH12	通道 12
ADC_CH13	通道 13
ADC_CH14	通道 14
ADC_CH15	通道 15

OpenADC
所有其他处理器 （续）

A/D 中断：

ADC_INT_ON	允许中断
ADC_INT_OFF	禁止中断

A/D 电压配置：

ADC_VREFPLUS_VDD	VREF+ = AVDD
ADC_VREFPLUS_EXT	VREF+ = 外接
ADC_VREFMINUS_VDD	VREF- = AVDD
ADC_VREFMINUS_EXT	VREF- = 外接

portconfig

对于 PIC18F1220/1320， portconfig 的取值范围是 0 到 127 之间（包括 0 和 127）；对于所有其他处理器， portconfig 的取值范围是 0 到 15 之间 （包括 0 和 15）。这是 ADCON1 寄存器的端口配置位 bit 0 至 bit 6 或 bit 0 至 bit 3 的值。

说明： 该函数把与 A/D 相关的寄存器复位到 POR 状态，然后配置时钟、结果格式、参考电压、端口和通道。

文件名： adcopen.c

代码示例：

```
OpenADC( ADC_FOSC_32     &
         ADC_RIGHT_JUST &
         ADC_12_TAD,
         ADC_CH0         &
         ADC_INT_OFF, 15  );
```

ReadADC

功能： 读取 A/D 转换的结果。

头文件： adc.h

函数原型：

```
int ReadADC( void );
```

说明： 该函数读取 A/D 转换的 16 位结果。

返回值： 该函数返回 A/D 转换的 16 位有符号结果。根据 A/D 转换器的配置（例如，使用函数 OpenADC()），结果会包含在 16 位结果的低有效位或高有效位中。

文件名： adcread.c

SetChanADC

功能： 选择用作 A/D 转换器输入的通道。

头文件： adc.h

函数原型： void SetChanADC(unsigned char ***channel***);

参数： ***channel***
下列值之一（在 adc.h 中定义）：

ADC_CH0	通道 0
ADC_CH1	通道 1
ADC_CH2	通道 2
ADC_CH3	通道 3
ADC_CH4	通道 4
ADC_CH5	通道 5
ADC_CH6	通道 6
ADC_CH7	通道 7
ADC_CH8	通道 8
ADC_CH9	通道 9
ADC_CH10	通道 10
ADC_CH11	通道 11

说明： 选择用作 A/D 转换器输入的引脚。

文件名： adcsetch.c

代码示例： SetChanADC(ADC_CH0);

2.2.2 使用 A/D 转换器函数的例子

```
#include <p18C452.h>
#include <adc.h>
#include <stdlib.h>
#include <delays.h>

int result;

void main( void )
{
  // configure A/D convertor
  OpenADC( ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_8ANA_0REF,
           ADC_CH0 & ADC_INT_OFF );

  Delay10TCYx( 5 );     // Delay for 50TCY
  ConvertADC();         // Start conversion
  while( BusyADC() );   // Wait for completion
  result = ReadADC();   // Read result
  CloseADC();           // Disable A/D converter
}
```

2.3 输入捕捉函数

下列函数支持捕捉外设。

表 2-2： 输入捕捉函数

函数	描述
CloseCapture***x***	禁止捕捉外设 ***x***。
OpenCapture***x***	配置捕捉外设 ***x***。
ReadCapture***x***	从捕捉外设 ***x*** 读取值。
CloseECapture***x***(1)	禁止增强型捕捉外设 ***x***。
OpenECapture***x***(1)	配置增强型捕捉外设 ***x***。
ReadECapture***x***(1)	从增强型捕捉外设 ***x*** 读取值。

注 1： 仅带有 ECCPxCONT 寄存器的器件具有增强捕捉功能。

2.3.1 函数描述

CloseCapture1
CloseCapture2
CloseCapture3
CloseCapture4
CloseCapture5
CloseECapture1

功能： 禁止输入捕捉 *x*。

头文件： capture.h

函数原型：

```
void CloseCapture1( void );
void CloseCapture2( void );
void CloseCapture3( void );
void CloseCapture4( void );
void CloseCapture5( void );
void CloseECapture1( void );
```

说明： 该函数禁止与指定输入捕捉相对应的中断。

文件名：

```
cp1close.c
cp2close.c
cp3close.c
cp4close.c
cp5close.c
ep1close.c
```

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4
OpenCapture5
OpenECapture1

功能： 配置并使能输入捕捉 *x*。

头文件： capture.h

函数原型：

```
void OpenCapture1( unsigned char config );
void OpenCapture2( unsigned char config );
void OpenCapture3( unsigned char config );
void OpenCapture4( unsigned char config );
void OpenCapture5( unsigned char config );
void OpenECapture1( unsigned char config );
```

参数： ***config***

从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在 capture.h 文件中定义。

使能 CCP 中断：

CAPTURE_INT_ON	使能中断
CAPTURE_INT_OFF	禁止中断

中断触发 （用 CCP 模块号代替 *x* ）：

Cx_EVERY_FALL_EDGE	每个下降沿产生中断
Cx_EVERY_RISE_EDGE	每个上升沿产生中断
Cx_EVERY_4_RISE_EDGE	每 4 个上升沿产生 1 个中断
Cx_EVERY_16_RISE_EDGE	每 16 个上升沿产生 1 个中断
EC1_EVERY_FALL_EDGE	每个下降沿产生中断 （增强型）
EC1_EVERY_RISE_EDGE	每个上升沿产生中断 （增强型）
EC1_EVERY_4_RISE_EDGE	每4个上升沿产生1个中断（增强型）
EC1_EVERY_16_RISE_EDGE	每 16 个上升沿产生 1 个中断（增强型）

说明： 该函数首先把捕捉模块复位到 POR 状态，然后将输入捕捉配置为指定的边沿检测。

捕捉函数使用在 capture.h 中定义的一个结构，来指示每个捕捉模块的溢出状态。此结构名为 CapStatus，包含如下位域：

```
Cap1OVF
Cap2OVF
Cap3OVF
Cap4OVF
Cap5OVF
ECap1OVF
```

进行任何捕捉操作之前，不仅要配置和使能捕捉模块，还要使能相应的定时器模块。关于 CCP 和定时器连接配置的信息，请参见相应的数据手册；关于函数 OpenTimer3 中使用的参数的信息，请参见**第 2.9 节 “定时器函数”**。

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4
OpenCapture5
OpenECapture1（续）

文件名： cp1open.c
cp2open.c
cp3open.c
cp4open.c
cp5open.c
ep1open.c

代码示例：
```
OpenCapture1( CAPTURE_INT_ON &
              C1_EVERY_4_RISE_EDGE );
```

ReadCapture1
ReadCapture2
ReadCapture3
ReadCapture4
ReadCapture5
ReadECapture1

功能： 从指定的输入捕捉中读取捕捉事件的结果。

头文件： capture.h

函数原型：
```
unsigned int ReadCapture1( void );
unsigned int ReadCapture2( void );
unsigned int ReadCapture3( void );
unsigned int ReadCapture4( void );
unsigned int ReadCapture5( void );
unsigned int ReadECapture1( void );
```

说明： 该函数读取各个输入捕捉特殊功能寄存器的值。

返回值： 该函数返回捕捉事件的结果。

文件名： cp1read.c
cp2read.c
cp3read.c
cp4read.c
cp5read.c
ep1read.c

2.3.2 使用输入捕捉函数的例子

该示例说明在 “查询”（非中断驱动）环境下如何使用捕捉库函数。

```
#include <p18C452.h>
#include <capture.h>
#include <timers.h>
#include <usart.h>
#include <stdlib.h>

void main(void)
{
  unsigned int result;
  char str[7];

  // Configure Capture1
  OpenCapture1( C1_EVERY_4_RISE_EDGE &
                CAPTURE_INT_OFF );

  // Configure Timer3
  OpenTimer3( TIMER_INT_OFF &
              T3_SOURCE_INT );

  // Configure USART
  OpenUSART( USART_TX_INT_OFF  &
             USART_RX_INT_OFF  &
             USART_ASYNCH_MODE &
             USART_EIGHT_BIT   &
             USART_CONT_RX,
             25 );

  while(!PIR1bits.CCP1IF);  // Wait for event
  result = ReadCapture1(); // read result
  ultoa(result,str);       // convert to string

  // Write the string out to the USART if
  // an overflow condition has not occurred.
  if(!CapStatus.Cap1OVF)
  {
    putsUSART(str);
  }

  // Clean up
  CloseCapture1();
  CloseTimer3();
  CloseUSART();
}
```

2.4 I²C™ 函数

为具有一个 I²C 外设的器件提供了下列函数：

表 2-3： 单个 I²C™ 外设函数

函数	描述
AckI2C	产生 I²C™ 总线*应答*条件。
CloseI2C	禁止 SSP 模块。
DataRdyI2C	I²C 缓冲区中是否有数据？
getcI2C	从 I²C 总线读取一个字节。
getsI2C	从工作在主 I²C 模式的 I²C 总线读取一个数据串。
IdleI2C	循环直到 I²C 总线空闲。
NotAckI2C	产生 I²C 总线*不应答*条件。
OpenI2C	配置 SSP 模块。
putcI2C	写一个字节到 I²C 总线。
putsI2C	写一个数据串到工作在主模式或从模式的 I²C 总线。
ReadI2C	从 I²C 总线上读取一个字节。
RestartI2C	产生 I²C 总线*重复启动*条件。
StartI2C	产生 I²C 总线*启动*条件。
StopI2C	产生 I²C 总线*停止*条件。
WriteI2C	写一个字节到 I²C 总线。

为具有多个 I²C 外设的器件提供了下列函数：

表 2-4： 多个 I²C™ 外设函数

函数	描述
AckI2C***x***	产生 I²C***x*** 总线*应答*条件。
CloseI2C***x***	禁止 SSP***x*** 模块。
DataRdyI2C***x***	I²C***x*** 缓冲区中是否有数据？
getcI2C***x***	从 I²C***x*** 总线读取一个字节。
getsI2C***x***	从工作在主 I²C 模式的 I²C***x*** 总线读取一个数据串。
IdleI2C***x***	循环直到 I²C***x*** 总线空闲。
NotAckI2C***x***	产生 I²C***x*** 总线*不应答*条件。
OpenI2C***x***	配置 SSP***x*** 模块。
putcI2C***x***	写一个字节到 I²C***x*** 总线。
putsI2C***x***	写一个数据串到工作在主模式或从模式的 I²C***x*** 总线。
ReadI2C***x***	从 I²C***x*** 总线上读取一个字节。
RestartI2C***x***	产生 I²C***x*** 总线*重复启动*条件。
StartI2C***x***	产生 I²C***x*** 总线*启动*条件。
StopI2C***x***	产生 I²C***x*** 总线*停止*条件。
WriteI2C***x***	写一个字节到 I²C***x*** 总线。

还提供了下列函数，用于与采用 I^2C 接口的电可擦除 （EE）存储器 （如 Microchip 的 24LC01B）接口：

表 2-5： 电可擦除存储器接口函数

函数	描述
EEAckPolling*x*	产生应答查询序列。
EEByteWrite*x*	写入一个字节。
EECurrentAddRead*x*	从下一个地址读取一个字节。
EEPageWrite*xx*	写入一个数据串。
EERandomRead*x*	从任意地址读取一个字节。
EESequentialRead*x*	读取一个数据串。

2.4.1 函数描述

AckI2C
AckI2C1
AckI2C2

功能： 产生 I^2C 总线*应答*条件。

头文件： i2c.h

函数原型：
```
void AckI2C( void );
void AckI2C1( void );
void AckI2C2( void );
```

说明： 该函数产生 I^2C*x* 总线*应答*条件。

文件名：
```
i2c_ack.c
i2c1ack.c
i2c2ack.c
```

CloseI2C
CloseI2C1
CloseI2C2

功能： 禁止 SSP*x* 模块。

头文件： i2c.h

函数原型：
```
void CloseI2C( void );
void CloseI2C1( void );
void CloseI2C2( void );
```

说明： 该函数禁止 SSP*x* 模块。

文件名：
```
i2c_close.c
i2c1close.c
i2c2close.c
```

DataRdyI2C
DataRdyI2C1
DataRdyI2C2

功能： I²C*x* 缓冲区中是否有数据？

头文件： i2c.h

函数原型：
```
unsigned char DataRdyI2C( void );
unsigned char DataRdyI2C1( void );
unsigned char DataRdyI2C2( void );
```

说明： 确定 SSP*x* 缓冲区中是否有数据可读。

返回值： 如果 SSP*x* 缓冲区中有数据，为 1；
如果 SSP*x* 缓冲区中没有数据，则为 0。

文件名：
```
i2c_dtrd.c
i2c1dtrd.c
i2c2dtrd.c
```

代码示例：
```
if (DataRdyI2C())
{
  var = getcI2C();
}
```

getcI2C
getcI2C1
getcI2C2

getcI2C*x* 定义为 ReadI2C*x*。*参见* **ReadI2C*x***。

getsI2C
getsI2C1
getsI2C2

功能： 从工作在主 I²C 模式的 I²C*x* 总线上读取一个固定长度的数据串。

头文件： i2c.h

函数原型：
```
unsigned char getsI2C(
            unsigned char  * rdptr,
            unsigned char    length );
unsigned char getsI2C1(
            unsigned char  * rdptr,
            unsigned char    length );
unsigned char getsI2C2(
            unsigned char  * rdptr,
            unsigned char    length );
```

参数：

rdptr
指向用于存储从 I²C*x* 器件所读取数据的 PICmicro RAM 的字符型指针。

length
从 I²C*x* 器件读取的字节数。

说明： 该函数从 I²C*x* 总线上读取一个预定义长度的数据串。

getsI2C
getsI2C1
getsI2C2 （续）

返回值： 如果所有字节都发送完毕，为 0；
如果发生总线冲突，则为 -1 。

文件名：
```
i2c_gets.c
i2c1gets.c
i2c2gets.c
```

代码示例：
```
unsigned char string[15];
getsI2C(string, 15);
```

IdleI2C
IdleI2C1
IdleI2C2

功能： 循环直到 I^2C***x*** 总线空闲。

头文件： `i2c.h`

函数原型：
```
void IdleI2C( void );
```

说明： 该函数检查 I^2C 外设的状态并且等待总线变为空闲。由于硬件 I^2C 外设不允许队列缓冲总线序列，所以需要函数 `IdleI2C`。在开始 I^2C 操作或者产生写冲突之前， I^2C 外设必须处于空闲状态。

文件名： `idlei2c.c`

NotAckI2C

功能： 产生 I^2C***x*** 总线*不应答*条件。

头文件： `i2c.h`

函数原型：
```
void NotAckI2C( void );
void NotAckI2C1( void );
void NotAckI2C2( void );
```

说明： 该函数产生 I^2C***x*** 总线*不应答*条件。

文件名：
```
i2c_nack.c
i2c1nack.c
i2c2nack.c
```

OpenI2C
OpenI2C1
OpenI2C2

功能： 配置 SSP***x*** 模块。

头文件： i2c.h

函数原型：
```
void OpenI2C( unsigned char sync_mode,
              unsigned char slew );
void OpenI2C1( unsigned char sync_mode,
              unsigned char slew );
void OpenI2C2( unsigned char sync_mode,
              unsigned char slew );
```

参数：

sync_mode
在 i2c.h 中定义的下列值之一：

SLAVE_7	I^2C 从模式， 7 位地址
SLAVE_10	I^2C 从模式， 10 位地址
MASTER	I^2C 主模式

slew
在 i2c.h 中定义的下列值之一：

SLEW_OFF	在 100 kHz 模式下禁止压摆率。
SLEW_ON	在 400 kHz 模式下使能压摆率。

说明： OpenI2C***x***函数把SSP***x***模块复位到POR状态，然后将模块配置为主/从模式及选择的压摆率。

文件名：
i2c_open.c
i2c1open.c
i2c2open.c

代码示例：
```
OpenI2C(MASTER, SLEW_ON);
```

putcI2C
putcI2C1
putcI2C2

putcI2Cx 定义为 WriteI2Cx。*参见* **WriteI2C*x***。

putsI2C
putsI2C1
putsI2C2

功能： 向工作在主模式或从模式的 I²C***x*** 总线写一个数据串。

头文件： i2c.h

函数原型：
```
unsigned char putsI2C(
                  unsigned char *wrptr );
unsigned char putsI2C1(
                  unsigned char *wrptr );
unsigned char putsI2C2(
                  unsigned char *wrptr );
```

参数：
wrptr
指向要写到 I²C***x*** 总线的数据的指针。

说明： 该函数向 I²C***x*** 总线写一个数据串，直到出现空字符为止。不传送空字符本身。该函数可以工作在主模式或从模式。

返回值：
主 I²C 模式：
如果在数据串中遇到空字符，为 0；
如果从 I²C***x*** 器件响应一个不应答 *Not Ack* 信号，为 -2；
如果发生写冲突，则为 -3。
从 I²C 模式：
如果在数据串中遇到空字符，为 0；
如果主 I²C***x*** 器件响应一个终止数据传送的不应答 *Not Ack* 信号，为 -2。

文件名：
i2c_puts.c
i2c1puts.c
i2c2puts.c

代码示例：
```
unsigned char string[] = "data to send";
putsI2C(string);
```

ReadI2C
ReadI2C1
ReadI2C2
getcI2C
getcI2C1
getcI2C2

功能： 从 I²C***x*** 总线读取一个字节。

头文件： i2c.h

函数原型：
```
unsigned char ReadI2C ( void );
unsigned char ReadI2C1 ( void );
unsigned char ReadI2C2 ( void );
unsigned char getcI2C ( void );
unsigned char getcI2C1 ( void );
unsigned char getcI2C2 ( void );
```

说明： 该函数从 I²C***x*** 总线上读入一个字节。 getcI2C*x* 在 i2c.h 中定义为 ReadI2C*x*。

返回值： 从 I²C***x*** 总线上读取的数据字节。

ReadI2C
ReadI2C1
ReadI2C2
getcI2C
getcI2C1
getcI2C2 （续）

文件名： i2c_read.c
i2c1read.c
i2c2read.c
define in i2c.h
define in i2c.h
define in i2c.h

代码示例：
```
unsigned char value;
value = ReadI2C();
```

RestartI2C
RestartI2C1
RestartI2C2

功能： 产生 I²C*x* 总线*重复启动*条件。

头文件： i2c.h

函数原型：
```
void RestartI2C( void );
void RestartI2C1( void );
void RestartI2C2( void );
```

说明： 该函数产生 I²C*x* 总线*重复启动*条件。

文件名： i2c_start.c
i2c1start.c
i2c2start.c

StartI2C
StartI2C1
StartI2C2

功能： 产生 I²C*x* 总线*启动*条件。

头文件： i2c.h

函数原型：
```
void StartI2C( void );
void StartI2C1( void );
void StartI2C2( void );
```

说明： 该函数产生 I²C*x* 总线*启动*条件。

文件名： i2c_start.c
i2c1start.c
i2c2start.c

StopI2C
StopI2C1
StopI2C2

功能： 产生 I²C*x* 总线*停止*条件。

头文件： i2c.h

函数原型：
```
void StopI2C( void );
void StopI2C1( void );
void StopI2C2( void );
```

说明： 该函数产生 I²C*x* 总线*停止*条件。

文件名：
```
i2c_stop.c
i2c1stop.c
i2c2stop.c
```

WriteI2C
putcI2C
WriteI2C1
WriteI2C2
putcI2C
putcI2C1
putcI2C2

功能： 向 I²C*x* 总线器件写一个字节。

头文件： i2c.h

函数原型：
```
unsigned char WriteI2C(
      unsigned char data_out );
unsigned char WriteI2C1(
      unsigned char data_out );
unsigned char WriteI2C2(
      unsigned char data_out );
unsigned char putcI2C(
      unsigned char data_out );
unsigned char putcI2C1(
      unsigned char data_out );
unsigned char putcI2C2(
      unsigned char data_out );
```

参数： ***data_out***
要写到 I²C*x* 总线器件的一字节数据。 putcI2C*x* 在 i2c.h 中定义为 iWriteI2C*x*。

说明： 该函数向 I²C*x* 总线器件写一字节数据。

返回值： 如果写入成功，为 0；
如果发生写冲突，则为 -1。

文件名：
```
i2c_write.c
i2c1write.c
i2c2write.c
#define in i2c.h
#define in i2c.h
#define in i2c.h
```

代码示例：
```
WriteI2C('a');
```

2.4.2 电可擦除存储器件接口函数描述

EEAckPolling
EEAckPolling1
EEAckPolling2

功能： 为 Microchip 的电可擦除 I^2C 存储器件产生应答查询序列。

头文件： i2c.h

函数原型：
```
unsigned char EEAckPolling(
     unsigned char control );
unsigned char EEAckPolling1(
     unsigned char control );
unsigned char EEAckPolling2(
     unsigned char control );
```

参数： ***control***
EEPROM 控制 / 总线器件的地址选择字节。

说明： 该函数为利用应答查询的电可擦除 I^2C 存储器件产生应答查询序列。

返回值： 如果没有发生错误，为 0；
如果发生总线冲突错误，为 -1；
如果发生写冲突错误，则为 -3。

文件名：
```
i2c_ecap.c
i2c1ecap.c
i2c2ecap.c
```

代码示例：
```
temp = EEAckPolling(0xA0);
```

EEByteWrite
EEByteWrite1
EEByteWrite2

功能： 向 I^2C***x*** 总线写一个字节。

头文件： i2c.h

函数原型：
```
unsigned char EEByteWrite(
     unsigned char control,
     unsigned char address,
     unsigned char data );
unsigned char EEByteWrite1(
     unsigned char control,
     unsigned char address,
     unsigned char data );
unsigned char EEByteWrite2(
     unsigned char control,
     unsigned char address,
     unsigned char data );
```

参数： ***control***
EEPROM 控制 / 总线器件的地址选择字节。
address
EEPROM 的内部地址单元。
data
要写到 EEPROM 中函数地址参数所指定地址的数据。

说明： 该函数把一字节数据写到 I^2C***x*** 总线，也适用于仅需单字节地址信息的任何 Microchip I^2C 电可擦除存储器件。

EEByteWrite
EEByteWrite1
EEByteWrite2 （续）

返回值： 如果没有发生任何错误，为 0；
如果发生总线冲突错误，为 -1；
如果发生不应答错误，为 -2；
如果发生写冲突错误，则为 -3。

文件名：
```
i2c_ecbw.c
i2c1ecbw.c
i2c2ecbw.c
```

代码示例：
```
temp = EEByteWrite(0xA0, 0x30, 0xA5);
```

EECurrentAddRead
EECurrentAddRead1
EECurrentAddRead2

功能： 从 I^2C***x*** 总线上读取一个字节。

头文件：
```
i2c.h
```

函数原型：
```
unsigned int EECurrentAddRead(
     unsigned char control );
unsigned int EECurrentAddRead1(
     unsigned char control );
unsigned int EECurrentAddRead2(
     unsigned char control );
```

参数： ***control***
EEPROM 控制 / 总线器件的地址选择字节。

说明： 该函数从 I^2C***x*** 总线上读取一个字节。要读取的数据位于 I^2C 电可擦除存储器件中当前指针所指向的地址。存储器件包含一个地址计数器，它保持最后访问的字的地址，并且以 1 为幅度递增。

返回值： 如果发生总线冲突错误，为 -1；
如果发生不应答错误，为 -2；
如果发生写冲突错误，则为 -3。
另外，该函数返回的结果为无符号的 16 位数据。因为缓冲区本身只有 8 位宽，这就意味着最高有效字节为 0，最低有效字节将包含读缓冲区的内容。

文件名：
```
i2c_eecr.c
i2c1eecr.c
i2c2eecr.c
```

代码示例：
```
temp = EECurrentAddRead(0xA1);
```

EEPageWrite
EEPageWrite1
EEPageWrite2

功能： 从 I^2C***x*** 总线写一个数据串到电可擦除存储器件中。

头文件： i2c.h

函数原型：

```
unsigned char EEPageWrite(
     unsigned char control,
     unsigned char address,
     unsigned char * wrptr );
unsigned char EEPageWrite1(
     unsigned char control,
     unsigned char address,
     unsigned char * wrptr );
unsigned char EEPageWrite2(
     unsigned char control,
     unsigned char address,
     unsigned char * wrptr );
```

参数：

control
EEPROM 控制 / 总线器件的地址选择字节。
address
EEPROM 的内部地址单元。
wrptr
PICmicro 单片机 RAM 的字符类型指针。***wrptr*** 指向的数据对象将会被写到电可擦除存储器件。

说明： 该函数把一个以空字符终止的数据串写到 I^2C 电可擦除存储器件，而空字符本身不会被传送。

返回值：
如果没有发生错误，为 0 ；
如果发生总线冲突错误，为 -1 ；
如果发生不应答错误，为 -2 ；
如果发生写冲突错误，则为 -3 。

文件名：
i2c_eepw.c
i2c1eepw.c
i2c2eepw.c

代码示例：

```
temp = EEPageWrite(0xA0, 0x70, wrptr);
```

EERandomRead
EERandomRead1
EERandomRead2

功能： 从 I²C***x*** 总线读取一个字节。

头文件： i2c.h

函数原型：

```
unsigned int EERandomRead(
     unsigned char control,
     unsigned char address );
unsigned int EERandomRead1(
     unsigned char control,
     unsigned char address );
unsigned int EERandomRead2(
     unsigned char control,
     unsigned char address );
```

参数：

control
EEPROM 控制 / 总线器件的地址选择字节。
address
EEPROM 的内部地址单元。

说明： 该函数从 I²C***x*** 总线上读取一个字节，也适用于仅需单字节地址信息的 Microchip I²C 电可擦除存储器件。

返回值： 返回值由两部分构成：一部分为在最低有效字节中读的值，另一部分为最高有效字节中的错误条件。错误条件为：
如果发生总线冲突错误，为 -1；
如果发生不应答错误，为 -2；
如果发生写冲突错误，则为 -3。

文件名：
i2c_eerr.c
i2c1eerr.c
i2c2eerr.c

代码示例：

```
unsigned int temp;
temp = EERandomRead(0xA0,0x30);
```

EESequentialRead
EESequentialRead1
EESequentialRead2

功能： 从 I²C***x*** 总线上读取一个数据串。

头文件： i2c.h

函数原型：

```
unsigned char EESequentialRead(
     unsigned char control,
     unsigned char address,
     unsigned char * rdptr,
     unsigned char length );
unsigned char EESequentialRead1(
     unsigned char control,
     unsigned char address,
     unsigned char * rdptr,
     unsigned char length );
unsigned char EESequentialRead2(
     unsigned char control,
     unsigned char address,
     unsigned char * rdptr,
     unsigned char length );
```

参数：

control
EEPROM 控制 / 总线器件的地址选择字节。

address
EEPROM 的内部地址单元。

rdptr
指向存放从 EEPROM 器件中所读出数据的 PICmicro 单片机 RAM 区的字符型指针。

length
从 EEPROM 器件中读出的字节数。

说明： 该函数从 I²C***x*** 总线上读取一个预定义长度的数据串，也适用于仅需单字节地址信息的 Microchip I²C 电可擦除存储器件。

返回值：
如果没有发生错误，为 0；
如果发生总线冲突错误，为 -1；
如果发生不应答错误，为 -2；
如果发生写冲突错误，则为 -3。

文件名：
i2c_eesr.c
i2c1eesr.c
i2c2eesr.c

代码示例：

```
unsigned char err;
err = EESequentialRead(0xA0,
                       0x70,
                       rdptr,
                       15);
```

2.4.3 使用示例

下面是一个简单的代码示例，该程序举例说明了配置为 I^2C 主通讯的 SSP 模块，及其和 Microchip 24LC01B I^2C 电可擦除存储器之间的 I^2C 通讯。

```
#include "p18cxx.h"
#include "i2c.h"

unsigned char arraywr[] = {1,2,3,4,5,6,7,8,0};
unsigned char arrayrd[20];

//*************************************************
void main(void)
{
  OpenI2C(MASTER, SLEW_ON);// Initialize I2C module
  SSPADD = 9;              //400kHz Baud clock(9) @16MHz
                           //100kHz Baud clock(39) @16MHz

  while(1)
  {
    EEByteWrite(0xA0, 0x30, 0xA5);
    EEAckPolling(0xA0);
    EECurrentAddRead(0xA0);
    EEPageWrite(0xA0, 0x70, arraywr);
    EEAckPolling(0xA0);
    EESequentialRead(0xA0, 0x70, arrayrd, 20);
    EERandomRead(0xA0,0x30);
  }
}
```

2.5 I/O 口函数

下列函数支持 PORTB。

表 2-6： I/O 口函数

函数	描述
ClosePORTB	禁止 PORTB 的中断和内部上拉电阻。
CloseRB***x***INT	禁止 PORTB 引脚 ***x*** 的中断。
DisablePullups	禁止 PORTB 的内部上拉电阻。
EnablePullups	使能 PORTB 的内部上拉电阻。
OpenPORTB	配置 PORTB 的中断和内部上拉电阻。
OpenRB***x***INT	使能 PORTB 引脚 ***x*** 的中断。

2.5.1 函数描述

ClosePORTB

函数： 禁止 PORTB 的中断和内部上拉电阻。

头文件： portb.h

函数原型：
```
void ClosePORTB( void );
```

说明： 该函数禁止 PORTB 的电平变化中断和内部上拉电阻。

文件名： pbclose.c

CloseRB0INT
CloseRB1INT
CloseRB2INT

功能： 禁止 PORTB 指定引脚的中断。

头文件： portb.h

函数原型：
```
void CloseRB0INT( void );
void CloseRB1INT( void );
void CloseRB2INT( void );
```

说明： 该函数禁止 PORTB 指定引脚的电平变化中断。

文件名：
```
rb0close.c
rb1close.c
rb2close.c
```

DisablePullups

功能： 禁止 PORTB 的内部上拉电阻。

头文件： portb.h

函数原型：
```
void DisablePullups( void );
```

说明： 该函数禁止 PORTB 的内部上拉电阻。

文件名： pulldis.c

EnablePullups

功能： 使能 PORTB 的内部上拉电阻。

头文件： portb.h

函数原型：
```
void EnablePullups( void );
```

说明： 该函数使能 PORTB 的内部上拉电阻。

文件名： pullen.c

OpenPORTB

功能： 配置 PORTB 的中断和内部上拉电阻。

头文件： portb.h

函数原型： void OpenPORTB(unsigned char ***config***);

参数： ***config***
从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 portb.h 中定义。

电平变化中断：

PORTB_CHANGE_INT_ON	允许中断
PORTB_CHANGE_INT_OFF	禁止中断

使能上拉电阻：

PORTB_PULLUPS_ON	使能上拉电阻
PORTB_PULLUPS_OFF	禁止上拉电阻

说明： 此函数配置 PORTB 的中断和内部上拉电阻。

文件名： pbopen.c

代码示例：
```
OpenPORTB( PORTB_CHANGE_INT_ON & PORTB_PULLUPS_ON);
```

OpenRB0INT
OpenRB1INT
OpenRB2INT

功能： 允许指定 PORTB 引脚的中断。

头文件： portb.h

函数原型：
```
void OpenRB0INT( unsigned char config );
void OpenRB1INT( unsigned char config );
void OpenRB2INT( unsigned char config );
```

参数： ***config***
从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在 portb.h 中定义。

电平变化中断：

PORTB_CHANGE_INT_ON	允许中断
PORTB_CHANGE_INT_OFF	禁止中断

边沿触发中断：

RISING_EDGE_INT	上升沿触发中断
FALLING_EDGE_INT	下降沿触发中断

使能上拉电阻：

PORTB_PULLUPS_ON	使能上拉电阻
PORTB_PULLUPS_OFF	禁止上拉电阻

说明： 此函数配置 PORTB 的中断和内部上拉电阻。

文件名： rb0open.c
rb1open.c
rb2open.c

代码示例：
```
OOpenRB0INT( PORTB_CHANGE_INT_ON & RISING_EDGE_INT &
PORTB_PULLUPS_ON);
```

2.6 MICROWIRE 函数

为具有一个 Microwire 外设的器件提供了下列函数：

表 2-7： 单个 MICROWIRE 外设函数

函数	描述
CloseMwire	禁止用于 Microwire 通讯的 SSP 模块。
DataRdyMwire	表明是否完成内部写循环。
getcMwire	从 Microwire 器件读取一个字节。
getsMwire	从 Microwire 器件读取一个数据串。
OpenMwire	配置 SSP 模块的 Microwire 通讯。
putcMwire	写一个字节到 Microwire 器件。
ReadMwire	从 Microwire 器件读取一个字节。
WriteMwire	写一个字节到 Microwire 器件。

为具有多个 Microwire 外设的器件提供了下列函数：

表 2-8： 多个 MICROWIRE 外设函数

函数	描述
CloseMwire***x***	禁止用于 Microwire 通讯的 SSP***x*** 模块。
DataRdyMwire***x***	表明是否完成内部写循环。
getcMwire***x***	从 Microwire 器件读取一个字节。
getsMwire***x***	从 Microwire 器件读取一个数据串。
OpenMwire***x***	配置 SSP***x*** 模块的 Microwire 通讯。
putcMwire***x***	写一个字节到 Microwire 器件。
ReadMwire***x***	从 Microwire 器件读取一个字节。
WriteMwire***x***	写一个字节到 Microwire 器件。

2.6.1 函数描述

CloseMwire
CloseMwire1
CloseMwire2

功能：	禁止 SSP***x*** 模块。
头文件：	mwire.h

函数原型：	void CloseMwire(void);
	void CloseMwire1(void);
	void CloseMwire2(void);

说明：	相关引脚恢复为普通 I/O 口功能。由 TRISC 和 LATC 负责实现 I/O 控制。
文件名：	mw_close.c mw1close.c mw2close.c

DataRdyMwire
DataRdyMwire1
DataRdyMwire2

功能： 表明 Microwire***x*** 器件是否已经完成内部写循环。

头文件： mwire.h

函数原型：
```
unsigned char DataRdyMwire( void );
unsigned char DataRdyMwire1( void );
unsigned char DataRdyMwire2( void );
```

说明： 确定 Microwire***x*** 器件是否已准备就绪。

返回值： 如果 Microwire***x*** 器件已经就绪，为 1；
如果内部写循环尚未完成或者发生总线错误，则为 0。

文件名：
```
mw_drdy.c
mw1drdy.c
mw2drdy.c
```

代码示例：
```
while (!DataRdyMwire());
```

getcMwire
getcMwire1
getcMwire2

getcMwire*x* 定义为 ReadMwire*x*。*参见* **ReadMwire*x***。

getsMwire
getsMwire1
getsMwire2

功能： 从 Microwire***x*** 器件读取一个数据串。

头文件： mwire.h

函数原型：
```
void getsMwire( unsigned char * rdptr,
                unsigned char length);
void getsMwire1( unsigned char * rdptr,
                unsigned char length);
void getsMwire2( unsigned char * rdptr,
                unsigned char length);
```

参数：
rdptr
指向存放从 Microwire***x*** 器件所读取数据的 PICmicro 单片机 RAM 的指针。
length
从 Microwire***x*** 器件读取的字节数。

说明： 该函数用于从 Microwire***x*** 器件读取一个预定义长度的数据串。在使用此函数前，必须向正确的地址发出一个 Read***x*** 命令。

文件名：
```
mw_gets.c
mw1gets.c
mw2gets.c
```

代码示例：
```
unsigned char arryrd[LENGTH];
putcMwire(READ);
putcMwire(address);
getsMwire(arrayrd, LENGTH);
```

OpenMwire

功能： 配置 SSP***x*** 模块。

头文件： mwire.h

函数原型：
```
void OpenMwire(
        unsigned char sync_mode );
```

参数： ***sync_mode***

在 mwire.h 中定义的下列值之一：

MWIRE_FOSC_4	clock = FOSC/4
MWIRE_FOSC_16	clock = FOSC/16
MWIRE_FOSC_64	clock = FOSC/64
MWIRE_FOSC_TMR2	clock = TMR2 output/2

说明： OpenMwire***x*** 函数把 SSP***x*** 模块复位到 POR 状态，然后配置该模块的 Microwire***x*** 通讯。

文件名：
```
mw_open.c
mw1open.c
mw2open.c
```

代码示例：
```
OpenMwire(MWIRE_FOSC_16);
```

putcMwire
putcMwire1
putcMwire2

putcMwire*x* 定义为 WriteMwire*x*。参见 **WriteMwire*x***。

ReadMwire
ReadMwire1
ReadMwire2
getcMwire
getcMwire1
getcMwire2

功能： 从 Microwire***x*** 器件读取一个字节。

头文件： mwire.h

函数原型：

```
unsigned char ReadMwire(
        unsigned char high_byte,
        unsigned char low_byte );
unsigned char ReadMwire1(
        unsigned char high_byte,
        unsigned char low_byte );
unsigned char ReadMwire2(
        unsigned char high_byte,
        unsigned char low_byte );
unsigned char getcMwire(
        unsigned char high_byte,
        unsigned char low_byte );
unsigned char getcMwire1(
        unsigned char high_byte,
        unsigned char low_byte );
unsigned char getcMwire2(
        unsigned char high_byte,
        unsigned char low_byte );
```

参数：

high_byte
16 位指令字的第一个字节。

low_byte
16 位指令字的第二个字节。

说明： 该函数从 Microwire***x*** 器件读取一个字节。启动位、操作码和地址组成传递给此函数的高字节和低字节。 getcMwire*x* 在 mwire.h 中定义为 ReadMwire*x*。

返回值： 返回值是从 Microwire***x*** 器件读取的单字节数据。

文件名：

```
mw_read.c
mw1read.c
mw2read.c
#define in mwire.h
#define in mwire.h
#define in mwire.h
```

代码示例：

```
ReadMwire(0x03, 0x00);
```

WriteMwire
WriteMwire1
WriteMwire2
putcMwire
putcMwire1
putcMwire2

功能： 该函数用于写一字节 （一个字符）数据到 Microwire***x*** 器件。

头文件： mwire.h

函数原型：

```
unsigned char WriteMwire(
        unsigned char data_out );
unsigned char WriteMwire1(
        unsigned char data_out );
unsigned char WriteMwire2(
        unsigned char data_out );
unsigned char putcMwire(
        unsigned char data_out );
unsigned char putcMwire1(
        unsigned char data_out );
unsigned char putcMwire2(
        unsigned char data_out );
```

参数： ***data_out***
要写到 Microwire***x*** 器件的单字节数据。

说明： 该函数利用 SSP***x*** 模块将一字节数据写到 Microwire***x*** 器件。
putcMwire*x* 在 mwire.h 中定义为 WriteMwire*x*。

返回值： 如果写入成功，为 0 ；
如果发生写冲突，则为 -1。

文件名：

```
mw_write.c
mw1write.c
mw2write.c
#define in mwire.h
#define in mwire.h
#define in mwire.h
```

代码示例：

```
WriteMwire(0x55);
```

2.6.2 使用示例

下面是一个简单的代码示例，举例说明了 SSP 模块与 Microchip 93LC66 Microwire 电可擦除存储器之间的通讯。

```
#include "p18cxxx.h"
#include "mwire.h"

// 93LC66 x 8
// FUNCTION Prototypes
void main(void);
void ew_enable(void);
void erase_all(void);
void busy_poll(void);
void write_all(unsigned char data);
void byte_read(unsigned char address);
void read_mult(unsigned char address,
               unsigned char *rdptr,
               unsigned char length);
void write_byte(unsigned char address,
                unsigned char data);

// VARIABLE Definitions
unsigned char arrayrd[20];
unsigned char var;

// DEFINE 93LC66 MACROS -- see datasheet for details
#define  READ   0x0C
#define  WRITE  0x0A
#define  ERASE  0x0E
#define  EWEN1  0x09
#define  EWEN2  0x80
#define  ERAL1  0x09
#define  ERAL2  0x00
#define  WRAL1  0x08
#define  WRAL2  0x80
#define  EWDS1  0x08
#define  EWDS2  0x00
#define  W_CS   LATCbits.LATC2

void main(void)
{
  TRISCbits.TRISC2 = 0;
  W_CS = 0;               //ensure CS is negated
  OpenMwire(MWIRE_FOSC_16);     //enable SSP peripheral
  ew_enable();            //send erase/write enable
  write_byte(0x13, 0x34); //write byte (address, data)
  busy_poll();
  Nop();
  byte_read(0x13);        //read single byte (address)
  read_mult(0x10, arrayrd, 10); //read multiple bytes
  erase_all();                  //erase entire array
  CloseMwire();                 //disable SSP peripheral
}
```

```
void ew_enable(void)
{
   W_CS = 1;         //assert chip select
   putcMwire(EWEN1); //enable write command byte 1
   putcMwire(EWEN2); //enable write command byte 2
   W_CS = 0;         //negate chip select
}
void busy_poll(void)
{
  W_CS = 1;
  while(! DataRdyMwire() );
  W_CS = 0;
}

void write_byte(unsigned char address,
                unsigned char data)
{
  W_CS = 1;
  putcMwire(WRITE);    //write command
  putcMwire(address);  //address
  putcMwire(data);     //write single byte
  W_CS = 0;
}

void byte_read(unsigned char address)
{
  W_CS = 1;
  getcMwire(READ,address);  //read one byte
  W_CS = 0;
}

void read_mult(unsigned char address,
               unsigned char *rdptr,
               unsigned char length)
{
  W_CS = 1;
  putcMwire(READ);          //read command
  putcMwire(address);       //address (A7 - A0)
  getsMwire(rdptr, length); //read multiple bytes
  W_CS = 0;
}

void erase_all(void)
{
  W_CS = 1;
  putcMwire(ERAL1); //erase all command byte 1
  putcMwire(ERAL2); //erase all command byte 2
  W_CS = 0;
}
```

2.7 脉宽调制函数

下列函数支持 PWM 外设：

表 2-9： PWM 函数

函数	描述
ClosePWM*x*	禁止 PWM 通道 *x*。
OpenPWM*x*	配置 PWM 通道 *x*。
SetDCPWM*x*	向 PWM 通道 *x* 写入一个新的占空比值。
SetOutputPWM*x*	设置 ECCP *x* 的 PWM 输出配置位。
CloseEPWM*x*(1)	禁止增强型 PWM 通道 *x*。
OpenEPWM*x*(1)	配置增强型 PWM 通道 *x*。
SetDCEPWM*x*(1)	写一个新的占空比值到增强型 PWM 通道 *x*。
SetOutputEPWM*x*(1)	设置 ECCP *x* 的增强型 PWM 输出配置位。

注 1： 增强型 PWM 函数仅可用于带有 ECCPxCON 寄存器的器件。

2.7.1 函数描述

ClosePWM1
ClosePWM2
ClosePWM3
ClosePWM4
ClosePWM5
CloseEPWM1

功能： 禁止 PWM 通道。

包含： pwm.h

函数原型：
```
void ClosePWM1( void );
void ClosePWM2( void );
void ClosePWM3( void );
void ClosePWM4( void );
void ClosePWM5( void );
void CloseEPWM1( void );
```

说明： 该函数禁止指定的 PWM 通道。

文件名：
```
pw1close.c
pw2close.c
pw3close.c
pw4close.c
pw5close.c
ew1close.c
```

OpenPWM1
OpenPWM2
OpenPWM3
OpenPWM4
OpenPWM5
OpenEPWM1

功能： 配置 PWM 通道。

包含： pwm.h

函数原型：

```
void OpenPWM1( char period );
void OpenPWM2( char period );
void OpenPWM3( char period );
void OpenPWM4( char period );
void OpenPWM5( char period );
void OpenEPWM1( char period );
```

参数： ***period***

可以是 0x00 到 0xff 之间的任何值，通过使用下面的公式，这个值可确定 PWM 频率：

PWM 周期 =[(***period***) + 1] x 4 x TOSC x TMR2 预分频比

说明： 该函数配置指定 PWM 通道的周期和时基。 PWM 只使用 Timer2。

在 PWM 工作之前，除了要配置 PWM 通道外，还要用 **OpenTimer2(...)** 语句配置 Timer2 。

文件名：

```
pw1open.c
pw2open.c
pw3open.c
pw4open.c
pw5open.c
ew1open.c
```

代码示例：

```
OpenPWM1(0xff);
```

SetDCPWM1
SetDCPWM2
SetDCPWM3
SetDCPWM4
SetDCPWM5
SetDCEPWM1

功能： 向指定 PWM 通道的占空比寄存器写入新的占空比值。

头文件： pwm.h

函数原型：
```
void SetDCPWM1( unsigned int dutycycle );
void SetDCPWM2( unsigned int dutycycle );
void SetDCPWM3( unsigned int dutycycle );
void SetDCPWM4( unsigned int dutycycle );
void SetDCPWM5( unsigned int dutycycle );
void SetDCEPWM1( unsigned int dutycycle );
```

参数： ***dutycycle***

dutycycle 的值可以是任何一个 10 位数。只有 *dutycycle* 的低 10 位写入到占空比寄存器。占空比，或者更具体地说是 PWM 波形的高电平时间，可以通过下面的公式计算出来：

PWM x 占空比 = (DCx<9:0>) x TOSC

其中， DCx<9:0> 是调用该函数时指定的 10 位值。

说明： 该函数向指定 PWM 通道的占空比寄存器写入新的占空比值。

PWM 波形的最大分辨率可以使用下面的公式、通过周期计算出来：

分辨率 （位）= log(FOSC/Fpwm) / log(2)

文件名：
```
pw1setdc.c
pw2setdc.c
pw3setdc.c
pw4setdc.c
pw5setdc.c
ew1setdc.c
```

代码示例：
```
SetDCPWM1(0);
```

SetOutputPWM1
SetOutputPWM2
SetOutputPWM3
SetOutputEPWM1

功能： 设置 ECCP 的 PWM 输出配置位。

头文件： pwm.h

函数原型：

```
void SetOutputPWM1 (
        unsigned char outputconfig,
        unsigned char outputmode);
void SetOutputPWM2 (
        unsigned char outputconfig,
        unsigned char outputmode);
void SetOutputPWM3 (
        unsigned char outputconfig,
        unsigned char outputmode);
void SetOutputEPWM1 (
        unsigned char outputconfig,
        unsigned char outputmode);
```

参数： ***outputconfig***

outputconfig 的值可以是下列值 （在 pwm.h 中定义）之一：

SINGLE_OUT	单端输出
FULL_OUT_FWD	全桥正向输出
HALF_OUT	半桥输出
FULL_OUT_REV	全桥反向输出

outputmode

outputmode 的值可以是下列值 （在 pwm.h 中定义）之一：

PWM_MODE_1	P1A 和 P1C 高电平有效 P1B 和 P1D 高电平有效
PWM_MODE_2	P1A 和 P1C 高电平有效 P1B 和 P1D 低电平有效
PWM_MODE_3	P1A 和 P1C 低电平有效 P1B 和 P1D 高电平有效
PWM_MODE_4	P1A 和 P1C 低电平有效 P1B 和 P1D 低电平有效

说明： 仅适用于带扩展型或增强型 CCP （ECCP）的器件。

文件名：

```
pw1setoc.c
pw2setoc.c
pw3setoc.c
ew1setoc.c
```

代码示例：

```
SetOutputPWM1 (SINGLE_OUT, PWM_MODE_1);
```

2.8 SPI 函数

为具有一个 SPI 外设的器件提供了下列函数：

表 2-10： 单个 SPI 外设函数

函数	描述
CloseSPI	禁止用于 SPI 通讯的 SSP 模块。
DataRdySPI	确定 SPI 缓冲区中是否有新值。
getcSPI	从 SPI 总线上读取一个字节。
getsSPI	从 SPI 总线上读取一个数据串。
OpenSPI	初始化用于 SPI 通讯的 SSP 模块。
putcSPI	向 SPI 总线写入一个字节。
putsSPI	向 SPI 总线写入一个数据串。
ReadSPI	从 SPI 总线上读取一个字节。
WriteSPI	向 SPI 总线写入一个字节。

另外为具有多个 SPI 外设的器件提供了下列函数：

表 2-11： 多个 SPI 外设函数

函数	描述
CloseSPI***x***	禁止用于 SPI 通讯的 SSP***x*** 模块。
DataRdySPI***x***	确定 SPI***x*** 缓冲区中是否有新值。
getcSPI***x***	从 SPI***x*** 总线上读取一个字节。
getsSPI***x***	从 SPI***x*** 总线上读取一个数据串。
OpenSPI***x***	初始化用于 SPI 通讯的 SSP***x*** 模块。
putcSPI***x***	向 SPI***x*** 总线写入一个字节。
putsSPI***x***	向 SPI***x*** 总线写入一个数据串。
ReadSPI***x***	从 SPI***x*** 总线上读取一个字节。
WriteSPI***x***	向 SPI***x*** 总线写入一个字节。

2.8.1 函数描述

CloseSPI
CloseSPI1
CloseSPI2

功能：	禁止 SSP***x*** 模块。
头文件：	spi.h

函数原型：	void CloseSPI(void);
	void CloseSPI1(void);
	void CloseSPI2(void);

说明：	该函数禁止 SSP***x*** 模块。相关引脚恢复为普通 I/O 口功能。由相应的 TRIS 和 LAT 寄存器控制 I/O 引脚。
文件名：	spi_clos.c spi1clos.c spi2clos.c

DataRdySPI
DataRdySPI1
DataRdySPI2

功能：	确定 SSPBUF***x*** 中是否有数据。
头文件：	spi.h

函数原型：	unsigned char DataRdySPI(void);
	unsigned char DataRdySPI1(void);
	unsigned char DataRdySPI2(void);

说明：	该函数确定 SSPBUF***x*** 寄存器中是否有数据字节可读。
返回值：	如果 SSPBUF***x*** 寄存器中没有数据，为 0； 如果 SSPBUF***x*** 寄存器中有数据，则为 1。
文件名：	spi_dtrd.c spi1dtrd.c spi2dtrd.c
代码示例：	while (!DataRdySPI());

getcSPI
getcSPI1
getcSPI2

getcSPI*x* 定义为 ReadSPI*x*。*参见* **ReadSPI*x***。

getsSPI
getsSPI1
getsSPI2

功能： 从 SPI*x* 总线读取一个数据串。

头文件： spi.h

函数原型：
```
void getsSPI( unsigned char *rdptr,
              unsigned char length );
void getsSPI1( unsigned char *rdptr,
              unsigned char length );
void getsSPI2( unsigned char *rdptr,
              unsigned char length );
```

参数：

rdptr
指向存放从 SPI*x* 器件中读取数据的地址的指针。

length
要从 SPI*x* 器件中读取数据的字节数。

说明： 该函数从 SPI*x* 总线读取一个预定义长度的数据串。

文件名：
spi_gets.c
spi1gets.c
spi2gets.c

代码示例：
```
unsigned char wrptr[10];
getsSPI(wrptr, 10);
```

OpenSPI
OpenSPI1
OpenSPI2

功能： 初始化 SSP*x* 模块。

头文件： spi.h

函数原型：
```
void OpenSPI( unsigned char sync_mode,
              unsigned char bus_mode,
              unsigned char smp_phase);
void OpenSPI1( unsigned char sync_mode,
              unsigned char bus_mode,
              unsigned char smp_phase);
void OpenSPI2( unsigned char sync_mode,
              unsigned char bus_mode,
              unsigned char smp_phase);
```

参数：

sync_mode
取下列值之一，在 spi.h 中定义：

SPI_FOSC_4	SPI 主模式，clock = FOSC/4
SPI_FOSC_16	SPI 主模式，clock = FOSC/16
SPI_FOSC_64	SPI 主模式， clock = FOSC/64
SPI_FOSC_TMR2	SPI 主模式，clock = TMR2 输出 /2
SLV_SSON	SPI 从模式，使能 /SS 引脚控制
SLV_SSOFF	SPI 从模式，禁止 /SS 引脚控制

bus_mode
取下列值之一，在 spi.h 中定义：

MODE_00	设置 SPI 总线为模式 0,0
MODE_01	设置 SPI 总线为模式 0,1
MODE_10	设置 SPI 总线为模式 1,0
MODE_11	设置 SPI 总线为模式 1,1

OpenSPI
OpenSPI1
OpenSPI2 （续）

smp_phase
取下列值之一，在 spi.h 中定义：

SMPEND	在输出数据的末端进行输入数据采样
SMPMID	在输出数据的中间进行输入数据采样

说明： 该函数设置供 SPI***x*** 总线器件使用的 SSP***x*** 模块。

文件名：
```
spi_open.c
spi1open.c
spi2open.c
```

代码示例：
```
OpenSPI(SPI_FOSC_16, MODE_00, SMPEND);
```

putcSPI
putcSPI1
putcSPI2

putcSPI*x* 定义为 WriteSPI*x*。*参见* **WriteSPI*x***。

putsSPI
putsSPI1
putsSPI2

功能： 向 SPI***x*** 总线写一个数据串。

头文件： spi.h

函数原型：
```
void putsSPI( unsigned char *wrptr );
void putsSPI1( unsigned char *wrptr );
void putsSPI2( unsigned char *wrptr );
```

参数： ***wrptr***
指向要写到 SPI***x*** 总线的值的指针。

说明： 该函数向 SPI***x*** 总线器件写一个数据串。当在数据串中读到一个空字符时函数终止（空字符不会写到总线）。

文件名：
```
spi_puts.c
spi1puts.c
spi2puts.c
```

代码示例：
```
unsigned char wrptr[] = "Hello!";
putsSPI(wrptr);
```

ReadSPI
ReadSPI1
ReadSPI2
getcSPI
getcSPI1
getcSPI2

功能：	从 SPI***x*** 总线上读取一个字节。
头文件：	spi.h

函数原型：	unsigned char ReadSPI(void);
	unsigned char ReadSPI1(void);
	unsigned char ReadSPI2(void);
	unsigned char getcSPI(void);
	unsigned char getcSPI1(void);
	unsigned char getcSPI2(void);

说明：	该函数启动一个 SPI***x*** 总线周期，来采集一字节数据。 getcSPI*x* 在 spi.h 中定义为 ReadSPI*x*。
返回值：	该函数返回在 SPI***x*** 读周期内读取的一字节数据。
文件名：	spi_read.c spi1read.c spi2read.c #define in spi.h #define in spi.h #define in spi.h
代码示例：	char x; x = ReadSPI();

WriteSPI
WriteSPI1
WriteSPI2
putcSPI
putcSPI1
putcSPI2

功能： 向 SPI***x*** 总线写一个字节。

头文件： spi.h

函数原型：

```
unsigned char WriteSPI(
            unsigned char data_out );
unsigned char WriteSPI1(
            unsigned char data_out );
unsigned char WriteSPI2(
            unsigned char data_out );
unsigned char putcSPI(
            unsigned char data_out );
unsigned char putcSPI1(
            unsigned char data_out );
unsigned char putcSPI2(
            unsigned char data_out );
```

参数： ***data_out***
要写到 SPI***x*** 总线的值。

说明： 该函数写一个字节的数据，然后检查是否有写冲突。 putcSPI*x* 在 spi.h 中定义为 WriteSPI*x*。

返回值： 如果没有发生写冲突，为 0；
如果发生写冲突，则为 -1。

文件名：

```
spi_writ.c
spi1writ.c
spi2writ.c
#define in spi.h
#define in spi.h
#define in spi.h
```

代码示例：

```
WriteSPI('a');
```

2.8.2 使用示例

下面的例子说明了如何使用 SSP 模块与 Microchip 的 25C080 SPI 电可擦除存储器进行通讯。

```
#include <p18cxxx.h>
#include <spi.h>

// FUNCTION Prototypes
void main(void);
void set_wren(void);
void busy_polling(void);
unsigned char status_read(void);
void status_write(unsigned char data);
void byte_write(unsigned char addhigh,
                unsigned char addlow,
                unsigned char data);
void page_write(unsigned char addhigh,
                unsigned char addlow,
                unsigned char *wrptr);
void array_read(unsigned char addhigh,
                unsigned char addlow,
                unsigned char *rdptr,
                unsigned char count);
unsigned char byte_read(unsigned char addhigh,
                        unsigned char addlow);

// VARIABLE Definitions
unsigned char arraywr[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,0};

//25C040/080/160 page write size
unsigned char arrayrd[16];
unsigned char var;

#define SPI_CS  LATCbits.LATC2

//************************************************
void main(void)
{
  TRISCbits.TRISC2 = 0;
  SPI_CS = 1;  // ensure SPI memory device
               // Chip Select is reset
  OpenSPI(SPI_FOSC_16, MODE_00, SMPEND);
  set_wren();
  status_write(0);

  busy_polling();
  set_wren();
  byte_write(0x00, 0x61, 'E');

  busy_polling();
  var = byte_read(0x00, 0x61);

  set_wren();
  page_write(0x00, 0x30, arraywr);
  busy_polling();

  array_read(0x00, 0x30, arrayrd, 16);
  var = status_read();
```

```
  CloseSPI();
  while(1);
}

void set_wren(void)
{
  SPI_CS = 0;               //assert chip select
  var = putcSPI(SPI_WREN);  //send write enable command
  SPI_CS = 1;               //negate chip select
}

void page_write (unsigned char addhigh,
                 unsigned char addlow,
                 unsigned char *wrptr)
{
  SPI_CS = 0;                 //assert chip select
  var = putcSPI(SPI_WRITE);   //send write command
  var = putcSPI(addhigh);     //send high byte of address
  var = putcSPI(addlow);      //send low byte of address
  putsSPI(wrptr);             //send data byte
  SPI_CS = 1;                 //negate chip select
}

void array_read (unsigned char addhigh,
                 unsigned char addlow,
                 unsigned char *rdptr,
                 unsigned char count)
{
  SPI_CS = 0;              //assert chip select
  var = putcSPI(SPI_READ); //send read command
  var = putcSPI(addhigh);  //send high byte of address
  var = putcSPI(addlow);   //send low byte of address
  getsSPI(rdptr, count);   //read multiple bytes
  SPI_CS = 1;
}

void byte_write (unsigned char addhigh,
                 unsigned char addlow,
                 unsigned char data)
{
  SPI_CS = 0;               //assert chip select
  var = putcSPI(SPI_WRITE); //send write command
  var = putcSPI(addhigh);   //send high byte of address
  var = putcSPI(addlow);    //send low byte of address
  var = putcSPI(data);      //send data byte
  SPI_CS = 1;               //negate chip select
}

unsigned char byte_read (unsigned char addhigh,
                         unsigned char addlow)
{
  SPI_CS = 0;               //assert chip select
  var = putcSPI(SPI_READ);  //send read command
  var = putcSPI(addhigh);   //send high byte of address
  var = putcSPI(addlow);    //send low byte of address
  var = getcSPI();          //read single byte
  SPI_CS = 1;
  return (var);
}
```

```
unsigned char status_read (void)
{
  SPI_CS = 0;               //assert chip select
  var = putcSPI(SPI_RDSR); //send read status command
  var = getcSPI();          //read data byte
  SPI_CS = 1;               //negate chip select
  return (var);
}

void status_write (unsigned char data)
{
  SPI_CS = 0;
  var = putcSPI(SPI_WRSR); //write status command
  var = putcSPI(data);     //status byte to write
  SPI_CS = 1;               //negate chip select
}

void busy_polling (void)
{
  do
  {
    SPI_CS = 0;               //assert chip select
    var = putcSPI(SPI_RDSR); //send read status command
    var = getcSPI();          //read data byte
    SPI_CS = 1;               //negate chip select
  } while (var & 0x01);      //stay in loop until !busy
}
```

2.9 定时器函数

下列函数支持定时器外设：

表 2-12： 定时器函数

函数	描述
CloseTimer***x***	禁止定时器 ***x***。
OpenTimer***x***	配置并使能定时器 ***x***。
ReadTimer***x***	读取定时器 ***x*** 的值。
WriteTimer***x***	向定时器 ***x*** 写入一个值。

2.9.1 函数描述

CloseTimer0
CloseTimer1
CloseTimer2
CloseTimer3
CloseTimer4

功能： 禁止指定的定时器。

头文件： timers.h

函数原型：

```
void CloseTimer0( void );
void CloseTimer1( void );
void CloseTimer2( void );
void CloseTimer3( void );
void CloseTimer4( void );
```

说明： 该函数禁止中断和指定的定时器。

文件名：

```
t0close.c
t1close.c
t2close.c
t3close.c
t4close.c
```

OpenTimer0

功能： 配置并使能 Timer0。

头文件： timers.h

函数原型： `void OpenTimer0( unsigned char config );`

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 timers.h 中定义。

允许 Timer0 中断：

TIMER_INT_ON	允许中断
TIMER_INT_OFF	禁止中断

定时器宽度：

T0_8BIT	8 位模式
T0_16BIT	16 位模式

时钟源：

T0_SOURCE_EXT	外部时钟源（I/O 引脚）
T0_SOURCE_INT	内部时钟源（TOSC）

外部时钟触发（T0_SOURCE_EXT）：

T0_EDGE_FALL	外部时钟下降沿
T0_EDGE_RISE	外部时钟上升沿

预分频值：

T0_PS_1_1	1:1 预分频
T0_PS_1_2	1:2 预分频
T0_PS_1_4	1:4 预分频
T0_PS_1_8	1:8 预分频
T0_PS_1_16	1:16 预分频
T0_PS_1_32	1:32 预分频
T0_PS_1_64	1:64 预分频
T0_PS_1_128	1:128 预分频
T0_PS_1_256	1:256 预分频

说明： 该函数按照指定的选项配置 Timer0，然后使能它。

文件名： t0open.c

代码示例：

```
OpenTimer0( TIMER_INT_OFF &
            T0_8BIT &
            T0_SOURCE_INT &
            T0_PS_1_32 );
```

OpenTimer1

功能： 配置并使能 Timer1。

头文件： timers.h

函数原型： void OpenTimer1(unsigned char ***config***);

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 timers.h 中定义。

允许 Timer1 中断：

TIMER_INT_ON	允许中断
TIMER_INT_OFF	禁止中断

定时器宽度：

T1_8BIT_RW	8 位模式
T1_16BIT_RW	16 位模式

时钟源：

T1_SOURCE_EXT	外部时钟源 （I/O 引脚）
T1_SOURCE_INT	内部时钟源 （Tosc）

预分频器：

T1_PS_1_1	1:1 预分频
T1_PS_1_2	1:2 预分频
T1_PS_1_4	1:4 预分频
T1_PS_1_8	1:8 预分频

振荡器使用：

T1_OSC1EN_ON	使能 Timer1 振荡器
T1_OSC1EN_OFF	禁止 Timer1 振荡器

同步时钟输入：

T1_SYNC_EXT_ON	同步外部时钟输入
T1_SYNC_EXT_OFF	不同步外部时钟输入

用于 CCP：

对于具有 1 个或 2 个 CCP 的器件

T3_SOURCE_CCP	Timer3 作为两个 CCP 的时钟源
T1_CCP1_T3_CCP2	Timer1 作为 CCP1 的时钟源，Timer3 作为 CCP2 的时钟源
T1_SOURCE_CCP	Timer1 作为两个 CCP 的时钟源

对于具有多于 2 个 CCP 的器件

T34_SOURCE_CCP	Timer3和Timer4作为所有CCP的时钟源
T12_CCP12_T34_CCP345	Timer1 和 Timer2 作为 CCP1 和 CCP2 的时钟源，Timer3 和 Timer4 作为 CCP3 到 CCP5 的时钟源
T12_CCP1_T34_CCP2345	Timer1 和 Timer2 作为 CCP1 的时钟源，Timer3 和 Timer4 作为 CCP2 到 CCP5 的时钟源
T12_SOURCE_CCP	Timer1和Timer2作为所有CCP的时钟源

说明： 该函数按照指定的选项配置 Timer1，然后使能它。

文件名： t1open.c

代码示例：

```
OpenTimer1( TIMER_INT_ON     &
            T1_8BIT_RW       &
            T1_SOURCE_EXT    &
            T1_PS_1_1        &
            T1_OSC1EN_OFF    &
            T1_SYNC_EXT_OFF );
```

OpenTimer2

功能： 配置并使能 Timer2。

头文件： timers.h

函数原型： `void OpenTimer2( unsigned char config );`

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 timers.h 中定义。

允许 Timer2 中断：

TIMER_INT_ON	允许中断
TIMER_INT_OFF	禁止中断

预分频值：

T2_PS_1_1	1:1 预分频
T2_PS_1_4	1:4 预分频
T2_PS_1_16	1:16 预分频

后分频值：

T2_POST_1_1	1:1 后分频
T2_POST_1_2	1:2 后分频
:	:
T2_POST_1_15	1:15 后分频
T2_POST_1_16	1:16 后分频

用于 CCP：

对于具有 1 个或 2 个 CCP 的器件

T3_SOURCE_CCP	Timer3 作为两个 CCP 的时钟源
T1_CCP1_T3_CCP2	Timer1 作为 CCP1 的时钟源，Timer3 作为 CCP2 的时钟源
T1_SOURCE_CCP	Timer1 作为两个 CCP 的时钟源

对于具有多于 2 个 CCP 的器件

T34_SOURCE_CCP	Timer3和Timer4作为所有CCP的时钟源
T12_CCP12_T34_CCP345	Timer1 和 Timer2 作为 CCP1 和 CCP2 的时钟源，Timer3 和 Timer4 作为 CCP3 到 CCP5 的时钟源
T12_CCP1_T34_CCP2345	Timer1 和 Timer2 作为 CCP1 的时钟源，Timer3 和 Timer4 作为 CCP2 到 CCP5 的时钟源
T12_SOURCE_CCP	Timer1和Timer2作为所有CCP的时钟源

说明： 该函数按照指定的选项配置 Timer2，然后使能它。

文件名： t2open.c

代码示例：

```
OpenTimer2( TIMER_INT_OFF &
            T2_PS_1_1     &
            T2_POST_1_8   );
```

OpenTimer3

功能： 配置并使能 Timer3。

头文件： timers.h

函数原型： void OpenTimer3(unsigned char ***config***);

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 timers.h 中定义。

允许 Timer3 中断：

TIMER_INT_ON	允许中断
TIMER_INT_OFF	禁止中断

定时器宽度：

T3_8BIT_RW	8 位模式
T3_16BIT_RW	16 位模式

时钟源：

T3_SOURCE_EXT	外部时钟源 （I/O 引脚）
T3_SOURCE_INT	内部时钟源 （Tosc）

预分频值：

T3_PS_1_1	1:1 预分频
T3_PS_1_2	1:2 预分频
T3_PS_1_4	1:4 预分频
T3_PS_1_8	1:8 预分频

同步时钟输入：

T3_SYNC_EXT_ON	同步外部时钟输入
T3_SYNC_EXT_OFF	不同步外部时钟输入

用于 CCP：

对于具有 1 个或 2 个 CCP 的器件

T3_SOURCE_CCP	Timer3 作为两个 CCP 的时钟源
T1_CCP1_T3_CCP2	Timer1 作为 CCP1 的时钟源，Timer3 作为 CCP2 的时钟源
T1_SOURCE_CCP	Timer1 作为两个 CCP 的时钟源

对于具有多于 2 个 CCP 的器件

T34_SOURCE_CCP	Timer3和Timer4作为所有CCP的时钟源
T12_CCP12_T34_CCP345	Timer1 和 Timer2 作为 CCP1 和 CCP2 的时钟源，Timer3 和 Timer4 作为 CCP3 到 CCP5 的时钟源
T12_CCP1_T34_CCP2345	Timer1 和Timer2 作为 CCP1 的时钟源，Timer3 和 Timer4 作为 CCP2 到 CCP5 的时钟源
T12_SOURCE_CCP	Timer1和Timer2作为所有CCP的时钟源

说明： 该函数按照指定的选项配置 Timer3，然后使能它。

文件名： t3open.c

代码示例：

```
OpenTimer3( TIMER_INT_ON     &
            T3_8BIT_RW       &
            T3_SOURCE_EXT    &
            T3_PS_1_1        &
            T3_OSC1EN_OFF    &
            T3_SYNC_EXT_OFF );
```

OpenTimer4

功能： 配置并使能 Timer4。

头文件： timers.h

函数原型： `void OpenTimer4( unsigned char config );`

参数： ***config***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 timers.h 中定义。

允许 Timer4 中断：

TIMER_INT_ON	允许中断
TIMER_INT_OFF	禁止中断

预分频值：

T4_PS_1_1	1:1 预分频
T4_PS_1_4	1:4 预分频
T4_PS_1_16	1:16 预分频

后分频值：

T4_POST_1_1	1:1 后分频
T4_POST_1_2	1:2 后分频
:	:
T4_POST_1_15	1:15 后分频
T4_POST_1_16	1:16 后分频

说明： 该函数按照指定的选项配置 Timer4，然后使能它。

文件名： t4open.c

代码示例：

```
OpenTimer4( TIMER_INT_OFF &
            T4_PS_1_1     &
            T4_POST_1_8   );
```

ReadTimer0
ReadTimer1
ReadTimer2
ReadTimer3
ReadTimer4

功能： 读取指定定时器的值。

头文件： timers.h

函数原型：

```
unsigned int  ReadTimer0( void );
unsigned int  ReadTimer1( void );
unsigned char ReadTimer2( void );
unsigned int  ReadTimer3( void );
unsigned char ReadTimer4( void );
```

说明： 这些函数读取各个定时器寄存器的值。

Timer0：	TMR0L,TMR0H
Timer1：	TMR1L,TMR1H
Timer2：	TMR2
Timer3：	TMR3L,TMR3H
Timer4：	TMR4

注： 当使用可配置为 16 位模式、但工作在 8 位模式的定时器 （如 Timer0）时，高字节并不保证为 0。用户可能希望将结果强制转换为字符型，以得到正确的结果。例如：

```
// Example of reading a 16-bit result
// from a 16-bit timer operating in
// 8-bit mode:
unsigned int result;
result = (unsigned char) ReadTimer0();
```

返回值： 定时器的当前值。

文件名：

```
t0read.c
t1read.c
t2read.c
t3read.c
t4read.c
```

WriteTimer0
WriteTimer1
WriteTimer2
WriteTimer3
WriteTimer4

功能： 向指定的定时器写入一个值。

头文件： timers.h

函数原型：

```
void WriteTimer0( unsigned int  timer );
void WriteTimer1( unsigned int  timer );
void WriteTimer2( unsigned char timer );
void WriteTimer3( unsigned int  timer );
void WriteTimer4( unsigned char timer );
```

参数： ***timer***

将装入指定定时器的值。

说明： 这些函数将值写入到相应的定时器寄存器：

Timer0：	TMR0L,TMR0H
Timer1：	TMR1L,TMR1H
Timer2：	TMR2
Timer3：	TMR3L,TMR3H
Timer4：	TMR4

文件名：

t0read.c
t1read.c
t2read.c
t3read.c
t4read.c

代码示例：

```
WriteTimer0( 10000 );
```

2.9.2 使用示例

```
#include <p18C452.h>
#include <timers.h>
#include <usart.h>
#include <stdlib.h>

void main( void )
{
  int result;
  char str[7];

  // configure timer0
  OpenTimer0( TIMER_INT_OFF &
              T0_SOURCE_INT  &
              T0_PS_1_32 );

  // configure USART
  OpenUSART( USART_TX_INT_OFF  &
             USART_RX_INT_OFF  &
             USART_ASYNCH_MODE &
             USART_EIGHT_BIT   &
             USART_CONT_RX,
             25                );

  while( 1 )
  {
    while( ! PORTBbits.RB3 ); // wait for RB3 high
    result = ReadTimer0();    // read timer

    if( result > 0xc000 )     // exit loop if value
      break;                  //   is out of range

    WriteTimer0( 0 );         // restart timer

    ultoa( result, str );     // convert timer to string
    putsUSART( str );         // print string
  }

  CloseTimer0();              // close modules
  CloseUSART();
}
```

2.10 USART 函数

下列函数支持带有单个 USART 外设的器件：

表 2-13： 单个 USART 外设函数

函数	描述
BusyUSART	USART 是否正在发送？
CloseUSART	禁止 USART。
DataRdyUSART	USART 读缓冲区中是否有数据？
getcUSART	从 USART 读取一个字节。
getsUSART	从 USART 上读取一个数据串。
OpenUSART	配置 USART。
putcUSART	向 USART 写入一个字节。
putsUSART	把数据存储器中的字符串写到 USART。
putrsUSART	把程序存储器中的字符串写到 USART。
ReadUSART	从 USART 上读取一个字节。
WriteUSART	写一个字节到 USART。
baudUSART	设置增强型 USART 的波特率配置位。

下列函数支持带有多个 USART 外设的器件：

表 2-14： 多个 USART 外设函数

函数	描述
Busy***x***USART	USART ***x*** 是否正在发送？
Close***x***USART	禁止 USART ***x***。
DataRdy***x***USART	USART ***x*** 读缓冲区中是否有数据？
getc***x***USART	从 USART ***x*** 读取一个字节。
gets***x***USART	从 USART ***x*** 读取一个字符串。
Open***x***USART	配置 USART ***x***。
putc***x***USART	向 USART ***x*** 写入一个字节。
puts***x***USART	把数据存储器中的字符串写到 USART ***x***。
putrs***x***USART	把程序存储器中的字符串写到 USART ***x***。
Read***x***USART	从 USART ***x*** 读取一个字节。
Write***x***USART	写一个字节到 USART ***x***。
baud***x***USART	设置增强型 USART ***x*** 的波特率配置位。

2.10.1 函数描述

BusyUSART
Busy1USART
Busy2USART

功能： USART 是否正在发送？

头文件： usart.h

函数原型：
```
char BusyUSART(  void );
char Busy1USART( void );
char Busy2USART( void );
```

说明： 返回值表明 USART 发送器现在是否正忙。应该在开始新的发送之前使用该函数。
当器件仅有一个 USART 外设时使用 BusyUSART，而当器件有多个 USART 外设时，使用 Busy1USART 和 Busy2USART。

返回值： 如果 USART 发送器空闲，为 0；
如果 USART 发送器正在使用，则为 1。

文件名：
```
ubusy.c
u1busy.c
u2busy.c
```

代码示例：
```
while (BusyUSART());
```

CloseUSART
Close1USART
Close2USART

功能： 禁止指定的 USART。

头文件： usart.h

函数原型：
```
void CloseUSART(  void );
void Close1USART( void );
void Close2USART( void );
```

说明： 该函数禁止指定 USART 的中断、发送器和接收器。
当器件仅有一个 USART 外设时使用 CloseUSART，而当器件有多个 USART 外设时，使用 Close1USART 和 Close2USART。

文件名：
```
uclose.c
u1close.c
u2close.c
```

DataRdyUSART
DataRdy1USART
DataRdy2USART

功能： 读缓冲区中是否有数据？

头文件： usart.h

函数原型：
```
char DataRdyUSART(  void );
char DataRdy1USART( void );
char DataRdy2USART( void );
```

说明： 该函数返回 PIR 寄存器中 RCI 标志位的状态。
当器件仅有一个 USART 外设时使用 DataRdyUSART，而当器件有多个 USART 外设时，使用 DataRdy1USART 和 DataRdy2USART。

返回值： 如果有数据，为 1；
如果没有数据，则为 0。

文件名：
```
udrdy.c
u1drdy.c
u2drdy.c
```

代码示例：
```
while (!DataRdyUSART());
```

getcUSART
getc1USART
getc2USART

getc*x*USART 定义为 Read*x*USART。*参见* **ReadUSART**。

getsUSART
gets1USART
gets2USART

功能： 从指定的 USART 中读取一个固定长度的字符串。

头文件： usart.h

函数原型：
```
void getsUSART ( char * buffer,
                 unsigned char len );
void gets1USART ( char * buffer,
                  unsigned char len );
void gets2USART ( char * buffer,
                  unsigned char len );
```

参数：
buffer
指向存储读取字符的地址的指针。
len
要从 USART 读取的字符数。

说明： 该函数仅适用于 8 位发送 / 接收模式。该函数将等待到从指定 USART 中读出 *len* 个字符为止。当等待字符到达时，不会出现超时。
当器件仅有一个 USART 外设时使用 getsUSART，而当器件有多个 USART 外设时，使用 gets1USART 和 gets2USART。

文件名：
```
ugets.c
u1gets.c
u2gets.c
```

代码示例：
```
char inputstr[10];
getsUSART( inputstr, 5 );
```

OpenUSART
Open1USART
Open2USART

功能： 配置指定的 USART 模块。

头文件： usart.h

函数原型：

```
void OpenUSART( unsigned char config,
                unsigned int spbrg);
void Open1USART( unsigned char config,
                 unsigned int spbrg);
void Open2USART( unsigned char config,
                 unsigned int spbrg);
```

参数：

config

从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 usart.h 中定义。

发送中断：

USART_TX_INT_ON	允许发送中断
USART_TX_INT_OFF	禁止发送中断

接收中断：

USART_RX_INT_ON	允许接收中断
USART_RX_INT_OFF	禁止接收中断

USART 模式：

USART_ASYNCH_MODE	异步模式
USART_SYNCH_MODE	同步模式

发送宽度：

USART_EIGHT_BIT	8 位发送 / 接收
USART_NINE_BIT	9 位发送 / 接收

主 / 从模式选择 *：

USART_SYNC_SLAVE	同步从模式
USART_SYNC_MASTER	同步主模式

接收模式：

USART_SINGLE_RX	单字节接收
USART_CONT_RX	连续接收

波特率：

USART_BRGH_HIGH	高波特率
USART_BRGH_LOW	低波特率

* 仅适用于同步模式

spbrg

这是写到波特率发生器寄存器中的值，它决定 USART 工作的波特率。计算波特率的公式为：

异步模式，高速：

FOSC / (16 * (***spbrg*** + 1))

异步模式，低速：

FOSC / (64 * (***spbrg*** + 1))

同步模式：

FOSC / (4 * (***spbrg*** + 1))

其中， FOSC 为振荡器频率。

说明： 该函数按照指定的配置选项配置 USART 模块。

当器件仅有一个 USART 外设时使用 OpenUSART ，而当器件有多个 USART 外设时，使用 Open1USART 和 Open2USART。

文件名：

uopen.c
u1open.c
u2open.c

OpenUSART
Open1USART
Open2USART （续）

代码示例：

```
OpenUSART1( USART_TX_INT_OFF  &
            USART_RX_INT_OFF  &
            USART_ASYNCH_MODE &
            USART_EIGHT_BIT   &
            USART_CONT_RX     &
            USART_BRGH_HIGH,
            25                );
```

putcUSART
putc1USART
putc2USART

putc*x*USART 定义为 Write*x*USART。*参见* **WriteUSART**。

putsUSART
puts1USART
puts2USART
putrsUSART
putrs1USART
putrs2USART

功能： 向 USART 写入一个包含空字符的字符串。

头文件： usart.h

函数原型：

```
void putsUSART(  char *data );
void puts1USART( char *data );
void puts2USART( char *data );
void putrsUSART(  const rom char *data );
void putrs1USART( const rom char *data );
void putrs2USART( const rom char *data );
```

参数：

data
指向以空字符结尾的数据串的指针。

说明： 该函数仅适用于 8 位发送 / 接收模式。该函数向 USART 写入一个包含空字符的字符串。
对于位于数据存储器中的字符串，应该使用这些函数的 “puts”形式。
对于位于程序存储器中的字符串，其中包括字符串常量，应该使用这些函数的 “putrs”形式。
当器件仅有一个 USART 外设时使用 putsUSART 和 putrsUSART，而当器件有多个 USART 外设时，使用其他函数。

文件名：

uputs.c
u1puts.c
u2puts.c
uputrs.c
u1putrs.c
u2putrs.c

代码示例：

```
putrsUSART( “Hello World!” );
```

ReadUSART
Read1USART
Read2USART
getcUSART
getc1USART
getc2USART

函数： 从 USART 接收缓冲区读取一个字节 （一个字符），包括第 9 位 （如果使能了 9 位模式的话）。

头文件： usart.h

函数原型：
```
char ReadUSART(  void );
char Read1USART( void );
char Read2USART( void );
char getcUSART(  void );
char getc1USART( void );
char getc2USART( void );
```

说明： 此函数从 USART 接收缓冲区读取一个字节。状态位和第 9 个数据位保存在定义如下的联合中：

```
union USART
{
  unsigned char val;
  struct
  {
    unsigned RX_NINE:1;
    unsigned TX_NINE:1;
    unsigned FRAME_ERROR:1;
    unsigned OVERRUN_ERROR:1;
    unsigned fill:4;
  };
};
```

如果使能了 9 位模式，则第 9 位是只读的。状态位将始终被读取。
对于具有一个 USART 外设的器件，应该使用 getcUSART 和 ReadUSART 函数，且状态信息读入名为 USART_Status 的变量，此变量类型为上述的 USART 联合。
对于具有多个 USART 外设的器件，应该使用 getc*x*USART 和 Read*x*USART 函数，状态信息读入名为 USART*x*_Status 的变量，此变量类型为上述的 USART 联合。

返回值： 此函数返回 USART 接收缓冲区中的下一个字符。

文件名：
```
uread.c
u1read.c
u2read.c
#define in usart.h
#define in usart.h
#define in usart.h
```

代码示例：
```
int result;
result = ReadUSART();
result |= (unsigned int)
          USART_Status.RX_NINE << 8;
```

WriteUSART
Write1USART
Write2USART
putcUSART
putc1USART
putc2USART

函数： 写一个字节 （一个字符）到 USART 发送缓冲区，包括第 9 位 （如果使能了 9 位模式的话）。

头文件： usart.h

函数原型：

```
void WriteUSART(  char data );
void Write1USART( char data );
void Write2USART( char data );
void putcUSART(  char data );
void putc1USART( char data );
void putc2USART( char data );
```

参数：

data
要写到 USART 的值。

说明： 此函数写一个字节到 USART 发送缓冲区。如果使能了 9 位模式，则第 9 位从字段 TX_NINE 写入，此字段包含在类型为 USART 的一个变量中。

```
union USART
{
  unsigned char val;
  struct
  {
    unsigned RX_NINE:1;
    unsigned TX_NINE:1;
    unsigned FRAME_ERROR:1;
    unsigned OVERRUN_ERROR:1;
    unsigned fill:4;
  };
};
```

对于带有一个 USART 外设的器件，应该使用 putcUSART 和 WriteUSART 函数， Status 寄存器名为 USART_Status，其类型为上述的 USART 联合。
对于带有多个 USART 外设的器件，应该使用 putc*x*USART 和 Write*x*USART 函数，状态寄存器名为 USART*x*_Status，其类型为上述的 USART 联合。

文件名：

```
uwrite.c
u1write.c
u2write.c
#define in usart.h
#define in usart.h
#define in usart.h
```

代码示例：

```
unsigned int outval;
USART1_Status.TX_NINE = (outval & 0x0100)
                        >> 8;
Write1USART( (char) outval );
```

baudUSART
baud1USART
baud2USART

函数： 为增强型 USART 的运行设置波特率配置位。

头文件： usart.h

函数原型：
```
void baudUSART(  unsigned char baudconfig );
void baud1USART( unsigned char baudconfig  );
void baud2USART( unsigned char baudconfig  );
```

参数： ***baudconfig***

从下面所列出各类型中分别取一个值并相与（'&'）所得的值。这些值在文件 usart.h 中定义：

时钟空闲状态：

BAUD_IDLE_CLK_HIGH	时钟空闲状态为高电平
BAUD_IDLE_CLK_LOW	时钟空闲状态为低电平

波特率发生：

BAUD_16_BIT_RATE	16 位波特率发生
BAUD_8_BIT_RATE	8 位波特率发生

RX 引脚监视：

BAUD_WAKEUP_ON	监测 RX 引脚
BAUD_WAKEUP_OFF	不监测 RX 引脚

波特率测量：

BAUD_AUTO_ON	使能自动波特率测量
BAUD_AUTO_OFF	禁止自动波特率测量

说明： 这些函数仅适用于带有增强型 USART 功能的处理器。

文件名：
ubaud.c
u1baud.c
u2baud.c

代码示例：
```
baudUSART (BAUD_IDLE_CLK_HIGH &
           BAUD_16_BIT_RATE &
           BAUD_WAKEUP_ON &
           BAUD_AUTO_ON);
```

2.10.2 使用示例

```
#include <p18C452.h>
#include <usart.h>

void main(void)
{
  // configure USART
  OpenUSART( USART_TX_INT_OFF  &
             USART_RX_INT_OFF  &
             USART_ASYNCH_MODE &
             USART_EIGHT_BIT   &
             USART_CONT_RX     &
             USART_BRGH_HIGH,
             25 );

  while(1)
  {
    while( ! PORTAbits.RA0 );  //wait for RA0 high

    WriteUSART( PORTD );       //write value of PORTD

    if(PORTD == 0x80)          // check for termination
      break;                   //   value
  }

  CloseUSART();
}
```

第 3 章 软件外设函数库

3.1 简介

本章讲述软件外设库函数。所有这些函数的源代码可以在 MPLAB C18 编译器安装目录的 src\traditional\pmc 和 src\extended\pmc 子目录下找到。

请参阅 *《MPASM™ 汇编器、MPLINK™ 目标链接器和 MPLIB™ 目标库管理器用户指南》*（DS33014J_CN），获得更多有关创建函数库的信息。

MPLAB C18 库函数支持下列外设：

- 外部 LCD 函数 （**第 3.2 节 “外部 LCD 函数”**）
- 外部 CAN2510 函数 （**第 3.3 节 “外部 CAN2510 函数”**）
- 软件 I²C™ 函数 （**第 3.4 节 “软件 I²C 函数”**）
- 软件 SPI 函数 （**第 3.5 节 “软件 SPI 函数”**）
- 软件 UART 函数 （**第 3.6 节 “软件 UART 函数”**）

3.2 外部 LCD 函数

设计这些函数，旨在使用 PIC18 单片机的 I/O 引脚控制 Hitachi 的 HD44780 LCD 控制器。所提供函数见下表：

表 3-1： 外部 LCD 函数

函数	描述
BusyXLCD	LCD 控制器是否正忙？
OpenXLCD	配置用于控制 LCD 的 I/O 线，并初始化 LCD 。
putcXLCD	向 LCD 控制器写一个字节。
putsXLCD	从数据存储器写一个字符串到 LCD 。
putrsXLCD	从程序存储器写一个字符串到 LCD 。
ReadAddrXLCD	从 LCD 控制器中读出地址字节。
ReadDataXLCD	从 LCD 控制器中读取一字节数据。
SetCGRamAddr	设置字符发生器的地址。
SetDDRamAddr	设置显示数据地址。
WriteCmdXLCD	写一个命令到 LCD 控制器。
WriteDataXLCD	写一字节数据到 LCD 控制器。

这些函数的预编译形式使用默认的引脚分配。通过在文件 xlcd.h 中重新定义下列宏，可以改变引脚的分配， xlcd.h 文件在编译器安装目录的 h 子目录下。

表 3-2： 选择 LCD 引脚分配的宏

LCD 控制器线	宏	默认值	用途
E 引脚	E_PIN TRIS_E	PORTBbits.RB4 DDRBbits.RB4	用于 E 线的引脚。 控制与 E 线有关引脚的方向的位。
RS 引脚	RS_PIN TRIS_RS	PORTBbits.RB5 DDRBbits.RB5	用于 RS 线的引脚。 控制与 RS 线有关引脚的方向的位。
RW 引脚	RW_PIN TRIS_RW	PORTBbits.RB6 DDRBbits.RB6	用于 RW 线的引脚。 控制与 RW 线有关引脚的方向的位。
数据线	DATA_PORT TRIS_DATA_PORT	PORTB DDRB	用于数据线的引脚。这些函数假设所有引脚都在一个端口上。 和数据线有关的数据方向寄存器。

所提供的函数库可工作在 4 位模式或 8 位模式。工作在 8 位模式时，使用一个端口的所有引脚。当工作在 4 位模式时，只使用一个端口的低 4 位或者高 4 位。下表列出了用于选择 4 位或 8 位模式的宏，以及用于选择工作在 4 位模式时使用端口哪些位的宏。

表 3-3： 选择 4 位或 8 位模式的宏

宏	默认值	用途
BIT8	未定义	如果创建库函数时定义了此值，库函数将工作在 8 位传输模式；否则，将工作在 4 位传输模式。
UPPER	未定义	当未定义 BIT8 时，该值将决定使用 DATA_PORT 的哪一个半字节来传输数据。 如果定义了 UPPER，使用 DATA_PORT 的高 4 位（4:7）。 如果没有定义 UPPER，则使用 DATA_PORT 的低 4 位（0:3）。

完成上述定义后，用户必须重新编译 XLCD 子程序，然后在项目中包含更新过的文件。这可通过把 XLCD 源文件添加到项目中，或者使用提供的批处理文件重新编译库文件来完成。

XLCD 函数库还需要用户定义下列函数，以提供适当的延时：

表 3-4： XLCD 延时函数

函数	功能
DelayFor18TCY	延时 18 个周期。
DelayPORXLCD	延时 15 ms。
DelayXLCD	延时 5 ms。

3.2.1 函数描述

BusyXLCD

功能： LCD 控制器是否正忙？

头文件： xlcd.h

函数原型：
```
unsigned char BusyXLCD( void );
```

说明： 该函数返回 Hitachi HD44780 LCD 控制器忙 （busy）标志的状态。

返回值： 如果控制器忙，返回 1；
否则，返回 0。

文件名： busyxlcd.c

代码示例：
```
while( BusyXLCD() );
```

OpenXLCD

功能： 配置 PIC® 单片机的 I/O 引脚，并初始化 LCD 控制器。

头文件： xlcd.h

函数原型：
```
void OpenXLCD( unsigned char lcdtype );
```

参数： ***lcdtype***
从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 xlcd.h 中定义。

数据接口：

FOUR_BIT	4 位数据接口模式
EIGHT_BIT	8 位数据接口模式

LCD 配置：

LINE_5X7	5x7 个字符，单行显示
LINE_5X10	5x10 个字符显示
LINES_5X7	5x7 个字符，多行显示

说明： 该函数配置用于控制 Hitachi HD44780 LCD 控制器的 PIC18 I/O 引脚，并初始化 LCD 控制器。

文件名： openxlcd.c

代码示例：
```
OpenXLCD( EIGHT_BIT & LINES_5X7 );
```

putcXLCD

参见 **WriteDataXLCD**。

putsXLCD
putrsXLCD

功能： 向 Hitachi HD44780 LCD 控制器写入一个字符串。

头文件： xlcd.h

函数原型：
```
void putsXLCD( char *buffer );
void putrsXLCD( const rom char *buffer );
```

参数：
buffer
指向要写入 LCD 控制器的字符的指针。

说明： 该函数把 *buffer* 中的字符串写到 Hitachi HD44780 LCD 控制器。当遇到空字符时会停止传输，不传输空字符。
对于位于数据存储器中的字符串，应该使用这些函数的 “puts”形式。
对于位于程序存储器中的字符串，包括字符串常量，应该使用这些函数的 “putrs”形式 。

文件名：
putsxlcd.c
putrxlcd.c

代码示例：
```
char mybuff [20];
putrsXLCD( "Hello World" );
putsXLCD( mybuff );
```

ReadAddrXLCD

功能： 从 Hitachi HD44780 LCD 控制器中读出地址字节。

头文件： xlcd.h

函数原型：
```
unsigned char ReadAddrXLCD( void );
```

说明： 该函数从 Hitachi HD44780 LCD 控制器中读出地址字节。当执行此操作时， LCD 控制器不能处于忙状态 — 这可通过 BusyXLCD 函数来检查。
从控制器中读出的地址是字符发生器 RAM 的地址还是显示数据 RAM 的地址，取决于前面调用 的 Set??RamAddr 函数。

返回值： 该函数返回一个 8 位的值。地址保存在低 7 位中， BUSY 状态标志保存在最高有效位中。

文件名： readaddr.c

代码示例：
```
char addr;
while ( BusyXLCD() );
addr = ReadAddrXLCD();
```

ReadDataXLCD

功能： 从 Hitachi HD44780 LCD 控制器中读出一字节数据。

头文件： xlcd.h

函数原型：
```
char ReadDataXLCD( void );
```

说明： 该函数从 Hitachi HD44780 LCD 控制器中读出一字节数据。当执行此操作时， LCD 控制器不能处于忙状态 — 这可以通过使用 BusyXLCD 函数来检查。
从控制器读出的数据是字符发生器 RAM 的数据还是显示数据 RAM 的数据，取决于前面调用 的 Set??RamAddr 函数。

返回值： 该函数返回一个 8 位的数据值。

文件名： readdata.c

代码示例：
```
char data;
while ( BusyXLCD() );
data = ReadAddrXLCD();
```

SetCGRamAddr

功能： 设置字符发生器的地址。

头文件： xlcd.h

函数原型：
```
void SetCGRamAddr( unsigned char addr );
```

参数： ***addr***
字符发生器的地址。

说明： 该函数设置 Hitachi HD44780 LCD 控制器字符发生器的地址。当执行此操作时， LCD 控制器不能处于忙状态 -- 这可以通过使用 BusyXLCD 函数来检查。

文件名： setcgram.c

代码示例：
```
char cgaddr = 0x1F;
while( BusyXLCD() );
SetCGRamAddr( cgaddr );
```

SetDDRamAddr

功能： 设置显示数据的地址。

头文件： xlcd.h

函数原型：
```
void SetDDRamAddr( unsigned char addr );
```

参数： ***addr***
显示数据的地址。

说明： 该函数设置 Hitachi HD44780 LCD 控制器中显示数据的地址。 LCD 控制器不能处于忙状态 — 这可以通过使用 BusyXLCD 函数来检查。

文件名： setddram.c

代码示例：
```
char ddaddr = 0x10;
while( BusyXLCD() );
SetDDRamAddr( ddaddr );
```

WriteCmdXLCD

功能： 写一个命令到 Hitachi HD44780 LCD 控制器。

头文件： xlcd.h

函数原型： `void WriteCmdXLCD( unsigned char cmd );`

参数： ***cmd***

指定要执行的命令。命令可以是 xlcd.h 中定义的下列值之一：

DOFF	关闭显示
CURSOR_OFF	使能无光标显示
BLINK_ON	使能闪烁光标显示
BLINK_OFF	使能不闪烁光标显示
SHIFT_CUR_LEFT	光标左移
SHIFT_CUR_RIGHT	光标右移
SHIFT_DISP_LEFT	显示左移
SHIFT_DISP_RIGHT	显示右移

或者，命令也可以是从下面所列出各类型中分别取一个值并相与 （'&'）所得的值。这些值在文件 xlcd.h 中定义。

数据传输模式：

FOUR_BIT	4 位数据接口模式
EIGHT_BIT	8 位数据接口模式

显示类型：

LINE_5X7	5x7 个字符，单行显示
LINE_5X10	5x10 个字符显示
LINES_5X7	5x7 个字符，多行显示

说明： 该函数把命令字节写入到 Hitachi HD44780 LCD 控制器。执行此操作时 LCD 控制器不能处于忙状态 — 这可以通过使用 BusyXLCD 函数来检查。

文件名： wcmdxlcd.c

代码示例：

```
while( BusyXLCD() );
WriteCmdXLCD( EIGHT_BIT & LINES_5X7 );
WriteCmdXLCD( BLINK_ON );
WriteCmdXLCD( SHIFT_DISP_LEFT );
```

putcXLCD
WriteDataXLCD

功能 把一字节数据写入到 Hitachi HD44780 LCD 控制器。

头文件： xlcd.h

函数原型： `void WriteDataXLCD( char data );`

参数： ***data***

data 的值可以是任意 8 位值，但是应该和 HD44780 LCD 控制器的字符 RAM 表相对应。

说明： 该函数把一字节数据写入到 Hitachi HD44780 LCD 控制器。当执行此操作时， LCD 控制器不能处于忙状态 — 这可通过 BusyXLCD 函数来检查。

从控制器中读出的数据是字符发生器 RAM 的数据还是显示数据 RAM 的数据，取决于前面调用 的 Set??RamAddr 函数。

文件名： writdata.c

3.2.2 使用示例

```
#include <p18C452.h>
#include <xlcd.h>
#include <delays.h>
#include <usart.h>

void DelayFor18TCY( void )
{
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
  Nop();
}

void DelayPORXLCD (void)
{
  Delay1KTCYx(60); // Delay of 15ms
                   // Cycles = (TimeDelay * Fosc) / 4
                   // Cycles = (15ms * 16MHz) / 4
                   // Cycles = 60,000
  return;
}

void DelayXLCD (void)
{
  Delay1KTCYx(20); // Delay of 5ms
                   // Cycles = (TimeDelay * Fosc) / 4
                   // Cycles = (5ms * 16MHz) / 4
                   // Cycles = 20,000
  return;
}
void main( void )
{
  char data;

  // configure external LCD
  OpenXLCD( EIGHT_BIT & LINES_5X7 );

  // configure USART
  OpenUSART( USART_TX_INT_OFF  & USART_RX_INT_OFF &
             USART_ASYNCH_MODE & USART_EIGHT_BIT  &
             USART_CONT_RX,
             25);

  while(1)
  {
    while(!DataRdyUSART());  //wait for data
    data = ReadUSART();      //read data
    WriteDataXLCD(data);     //write to LCD
    if(data=='Q')
      break;
  }

  CloseUSART();
}
```

3.3 外部 CAN2510 函数

本节讲述 MCP2510 外部外设库函数。所提供函数见下表：

表 3-5： 外部 CAN2510 函数

函数	描述
CAN2510BitModify	将一个寄存器中的指定位修改为新值。
CAN2510ByteRead	读取由地址指定的 MCP2510 寄存器。
CAN2510ByteWrite	写一个值到由地址指定的 MCP2510 寄存器。
CAN2510DataRead	从指定的接收缓冲区中读取报文。
CAN2510DataReady	确定指定的接收缓冲区中是否有数据可读。
CAN2510Disable	将所选择的 PIC18CXXX I/O 引脚设置为高电平，禁止 MCP2510 的片选。(1)
CAN2510Enable	将所选择的 PIC18CXXX I/O 引脚驱动为低电平，片选 MCP2510。(1)
CAN2510ErrorState	读 CAN 总线的当前错误状态。
CAN2510Init	初始化 PIC18CXXX 的 SPI 口以与 MCP2510 进行通讯，然后配置 MCP2510 寄存器以与 CAN 总线接口。
CAN2510InterruptEnable	将 CAN2510 的中断允许位 （CANINTE 寄存器）修改为新值。
CAN2510InterruptStatus	指明 CAN2510 的中断源。
CAN2510LoadBufferStd	把标准数据帧装入指定的发送缓冲区。
CAN2510LoadBufferXtd	把扩展数据帧装入指定的发送缓冲区。
CAN2510LoadRTRStd	把标准远程帧装入指定的发送缓冲区。
CAN2510LoadRTRXtd	把扩展远程帧装入指定的发送缓冲区。
CAN2510ReadMode	读取 MCP2510 的当前工作模式。
CAN2510ReadStatus	读取 MCP2510 发送缓冲区和接收缓冲区的状态。
CAN2510Reset	复位 MCP2510。
CAN2510SendBuffer	请求发送指定发送缓冲区中的报文。
CAN2510SequentialRead	从 MCP2510 中，自指定地址开始连续读取指定的字节数。这些值存储在 DataArray 中。
CAN2510SequentialWrite	自指定地址开始，向 MCP2510 连续写入指定的字节数。这些值来自于 DataArray。
CAN2510SetBufferPriority	为指定发送缓冲区装入指定的优先级。
CAN2510SetMode	配置 MCP2510 的工作模式。
CAN2510SetMsgFilterStd	为标准报文配置某个接收缓冲区的所有过滤器和屏蔽器。

表 3-5： 外部 CAN2510 函数 （续）

函数	描述
CAN2510SetMsgFilterXtd	为扩展报文配置某个接收缓冲区的所有过滤器和屏蔽器。
CAN2510SetSingleFilterStd	为标准 （Std）报文配置指定的接收过滤器。
CAN2510SetSingleFilterXtd	为扩展 （Xtd）报文配置指定的接收过滤器。
CAN2510SetSingleMaskStd	为标准 （Std）格式报文配置指定接收缓冲区的屏蔽器。
CAN2510SetSingleMaskXtd	为扩展 （Xtd）报文配置指定接收缓冲区的屏蔽器。
CAN2510WriteBuffer	启动所选发送缓冲区的 CAN 报文发送。
CAN2510WriteStd	使用第一个可用的发送缓冲区，将标准格式报文写到 CAN 总线。
CAN2510WriteXtd	使用第一个可用的发送缓冲区，将扩展格式报文写到 CAN 总线。

注 1： 在下列情况下，函数 CAN2510Enable 和 CAN2510Disable 需要重新编译：
- PICmicro® 单片机 $\overline{CS}$ 引脚的分配做了修改，不和 RC2 相连
- 需要修改器件的头文件

3.3.1 函数描述

CAN2510BitModify

功能： 将一个寄存器中的指定位修改为新值。

所要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
void CAN2510BitModify(
    unsigned char addr
    unsigned char mask
    unsigned char data );
```

参数：

addr

addr 的值指定要修改的 MCP2510 寄存器的地址。

mask

mask 的值指定将被修改的位。

data

data 的值指定位的新状态。

说明： 该函数修改由地址指定的寄存器的内容， mask 指定要修改哪些位，data 指定要装入这些位的新值。只能对某些特定寄存器使用位修改命令。

文件名： canbmod.c

CAN2510ByteRead

功能： 读取由地址指定的 MCP2510 寄存器。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
unsigned char CAN2510ByteRead(
    unsigned char address );
```

参数：

address
要从中读取数据的 MCP2510 寄存器的地址。

说明： 该函数按照从指定地址的 MCP2510 中读取一字节数据。

返回值： 指定地址的内容。

文件名： readbyte.c

CAN2510ByteWrite

功能： 写一个值到由地址指定的 MCP2510 寄存器。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
void CAN2510ByteWrite(
    unsigned char address,
    unsigned char value );
```

参数：

address
要写入数据的 MCP2510 的地址。

value
要写入的值。

说明： 该函数向由地址指定的 MCP2510 寄存器写入一字节数据。

文件名： wrtbyte.c

CAN2510DataRead

功能： 从指定的接收缓冲区中读取报文。

要求的 CAN 模式： 除配置模式外的所有模式

头文件： can2510.h

函数原型：
```
unsigned char CAN2510DataRead(
    unsigned char bufferNum,
    unsigned long *msgId,
    unsigned char *numBytes,
    unsigned char *data );
```

参数：

bufferNum
要从中读取报文的接收缓冲区，取下列值之一：

CAN2510_RXB0	读取接收缓冲区 0
CAN2510_RXB1	读取接收缓冲区 1

msgId
指向将被函数修改、保存 CAN 标准报文标识符的地址。

CAN2510DataRead （续）

numBytes
指向将被函数修改、保存此报文中字节数的地址。

data
指向将被该函数修改、保存报文数据的数组。此数组应该至少 8 个字节长，因为这是报文数据的最大长度。

说明： 该函数确定报文是标准报文还是扩展报文，将 ID 和报文长度解码，并用适当的信息来填充用户提供的地址。应该使用 CAN2510DataReady 函数来确定指定的缓冲区是否有数据可读。

返回值： 函数返回下列值之一：

CAN2510_XTDMSG	扩展格式报文
CAN2510_STDMSG	标准格式报文
CAN2510_XTDRTR	远程发送请求（XTD 报文）
CAN2510_STDRTR	远程发送请求（STD 报文）

文件名： canread.c

CAN2510DataReady

功能： 确定指定的接收缓冲区中是否有数据可读。

要求的 CAN 模式： 除配置模式外的所有模式

头文件： can2510.h

函数原型：
```
unsigned char CAN2510DataReady(
   unsigned char bufferNum );
```

参数： ***bufferNum***
要检查以确定其中是否有等待报文的接收缓冲区，其值为：

CAN2510_RXB0	检查接收缓冲区 0
CAN2510_RXB1	检查接收缓冲区 1
CAN2510_RXBX	检查接收缓冲区 0 和接收缓冲区 1

说明： 该函数检测 CANINTF 寄存器中相应的 RXnIF 位。

返回值： 如果没有检测到任何报文，返回 0；若检测到报文，则返回一个非 0 的值。
1 = 缓冲区 0
2 = 缓冲区 1
3 = 缓冲区 0 和缓冲区 1

文件名： canready.c

CAN2510Disable

功能： 将所选择的 PIC18CXXX I/O 引脚设置为高电平，禁止 MCP2510 的片选。

要求的 CAN 模式： 所有模式均可

头文件： canenabl.h

注： 如果片选信号没有和 PICmicro 单片机的 RC2 引脚相连，则应修改此头文件。

函数原型：
```
void CAN2510Disable( void );
```

参数： 无

说明： 该函数要求用户修改文件，指定用于和 MCP2510 $\overline{CS}$ 引脚相连的 PIC18CXXX I/O 引脚 （及端口）。默认引脚是 RC2。

注： 包含此函数 （和 CAN2510Enable 函数）的源文件必须修改定义，正确指定用于控制 MCP2510 $\overline{CS}$ 引脚的端口 （A, B, C, ...）和引脚号 （1, 2, 3, ...）。修改后，必须重建特定处理器的函数库。重建函数库的信息可参见**第 1.5.3 节 “重建”**。

文件名： canenabl.c

CAN2510Enable

功能： 将所选择的 PIC18CXXX I/O 引脚驱动为低电平，片选 MCP2510。

要求的 CAN 模式： 所有模式均可

头文件： canenabl.h

注： 如果片选信号没有和 PICmicro 单片机的 RC2 引脚相连，则应修改此头文件。

函数原型：
```
void CAN2510Enable( void );
```

说明： 该函数要求用户修改文件，指定用于和 MCP2510 $\overline{CS}$ 引脚相连的 PIC18CXXX I/O 引脚 （及端口）。默认引脚是 RC2。

注： 包含此函数 （和 CAN2510Enable 函数）的源文件必须修改定义，正确指定用于控制 MCP2510 $\overline{CS}$ 引脚的端口 （A, B, C, ...）和引脚号 （1, 2, 3, ...）。修改后，必须重建特定处理器的函数库。重建函数库的信息可参见**第 1.5.3 节 “重建”**。

文件名： canenabl.c

CAN2510ErrorState

功能： 读 CAN 总线的当前错误状态。

要求的 CAN 模式： 正常模式、环回测试模式和监听模式
（在配置模式中复位错误计数器）

头文件： can2510.h

函数原型：
```
unsigned char CAN2510ErrorState( void );
```

说明： 该函数返回 CAN 总线的错误状态。错误状态取决于 TEC 寄存器和 REC 寄存器中的值。

返回值： 函数返回下列值之一：

CAN2510_BUS_OFF	TEC > 255
CAN2510_ERROR_PASSIVE_TX	TEC > 127
CAN2510_ERROR_PASSIVE_RX	REC > 127
CAN2510_ERROR_ACTIVE_WITH_TXWARN	TEC > 95
CAN2510_ERROR_ACTIVE_WITH_RXWARN	REC > 95
CAN2510_ERROR_ACTIVE	TEC ≤ 95 且 REC ≤ 95

文件名： canerrst.c

CAN2510Init

功能： 初始化 PIC18CXXX 的 SPI 口以与 MCP2510 进行通讯，然后配置 MCP2510 寄存器以与 CAN 总线接口。

要求的 CAN 模式： 配置模式

头文件： can2510.h

函数原型：
```
unsigned char CAN2510Init(
    unsigned short long BufferConfig,
    unsigned short long BitTimeConfig,
    unsigned char interruptEnables,
    unsigned char SPI_syncMode,
    unsigned char SPI_busMode,
    unsigned char SPI_smpPhase );
```

参数： 下列参数的值在头文件 can2510.h 中定义。

BufferConfig

BufferConfig 的值是下列选项的位与（'&'）操作得到的。每组功能中只能选择一项，用**粗体字**标出的选项是默认值。

复位 MCP2510

指定是否要发送 MCP2510 复位命令。这并不和 MCP2510 寄存器中的一位相对应。

CAN2510_NORESET	**不复位 MCP2510**
CAN2510_RESET	复位 MCP2510

缓冲区 0 过滤

由 RXB0M1:RXB0M0 位（RXB0CTRL 寄存器）控制

CAN2510_RXB0_USEFILT	**接收所有报文，使用过滤器**
CAN2510_RXB0_STDMSG	只接收标准报文
CAN2510_RXB0_XTDMSG	只接收扩展报文
CAN2510_RXB0_NOFILT	接收所有报文，不使用过滤器

缓冲区 1 过滤

由 RXB1M1:RXB1M0 位（RXB1CTRL 寄存器）控制

CAN2510_RXB1_USEFILT	**接收所有报文，使用过滤器**
CAN2510_RXB1_STDMSG	只接收标准报文
CAN2510_RXB1_XTDMSG	只接收扩展报文
CAN2510_RXB1_NOFILT	接收所有报文，不使用过滤器

CAN2510Init （续）

接收缓冲区 0 向接收缓冲区 1 转存
由 BUKT 位 （RXB0CTRL 寄存器）控制

CAN2510_RXB0_ROLL	**如果接收缓冲区 0 已满，则报文会转存到接收缓冲区 1**
CAN2510_RXB0_NOROLL	禁止转存

RX1BF 引脚设置
由 B1BFS:B1BFE:B1BFM 位 （BFPCTRL 寄存器）控制

CAN2510_RX1BF_OFF	**RX1BF 引脚处于高阻态**
CAN2510_RX1BF_INT	RX1BF 引脚为输出，表明接收缓冲区 1 装入数据，也可用作中断信号。
CAN2510_RX1BF_GPOUTH	RX1BF 引脚为通用的数字输出，输出为高电平。
CAN2510_RX1BF_GPOUTL	RX1BF 引脚为通用的数字输出，输出为低电平。

RX0BF 引脚设置
由 B0BFS:B0BFE:B0BFM 位 （BFPCTRL 寄存器）控制

CAN2510_RX0BF_OFF	**RX0BF 引脚处于高阻态**
CAN2510_RX0BF_INT	RX0BF 引脚为输出，表明接收缓冲区 0 装入数据，也可用作中断信号。
CAN2510_RX0BF_GPOUTH	RX0BF 引脚为通用的数字输出，输出为高电平。
CAN2510_RX0BF_GPOUTL	RX0BF 引脚为通用的数字输出，输出为低电平。

TX2 引脚设置
由 B2RTSM 位 （TXRTSCTRL 寄存器）控制

CAN2510_TX2_GPIN	**TX2RTS 引脚为数字输入**
CAN2510_TX2_RTS	TX2RTS 引脚为输入，用于初始化来自 TXBUF2 的发送请求帧。

TX1 引脚设置
由 B1RTSM 位 （TXRTSCTRL 寄存器）控制

CAN2510_TX1_GPIN	**TX1RTS 引脚为数字输入**
CAN2510_TX1_RTS	TX1RTS 引脚为输入，用于初始化来自 TXBUF1 的发送请求帧。

TX0 引脚设置
由 B0RTSM 位 （TXRTSCTRL 寄存器）控制

CAN2510_TX0_GPIN	**TX0RTS 引脚为数字输入。**
CAN2510_TX0_RTS	TX0RTS 引脚为输入，用于初始化来自 TXBUF0 的发送请求帧。

请求工作模式
由 REQOP2:REQOP0 位 （CANCTRL 寄存器）控制

CAN2510_REQ_CONFIG	**配置模式**
CAN2510_REQ_NORMAL	正常工作模式
CAN2510_REQ_SLEEP	休眠模式
CAN2510_REQ_LOOPBACK	环回测试模式
CAN2510_REQ_LISTEN	监听模式

CLKOUT 引脚设置
由 CLKEN:CLKPRE1:CLKPRE0 位 （CANCTRL 寄存器）控制

CAN2510_CLKOUT_8	**CLKOUT = Fosc / 8**
CAN2510_CLKOUT_4	CLKOUT = Fosc / 4
CAN2510_CLKOUT_2	CLKOUT = Fosc / 2
CAN2510_CLKOUT_1	CLKOUT = Fosc
CAN2510_CLKOUT_OFF	禁止 CLKOUT

CAN2510Init （续）

BitTimeConfig

BitTimeConfig 的值是下列值的位与 （'&'）。每组功能中只可选择一个选项，用**粗体字**标出的选项是默认值。

波特率预分频器 （BRP）

通过 BRP5:BRP0 位 （CNF1 寄存器）控制

CAN2510_BRG_1X	**TQ = 1 x (2TOSC)**
:	:
CAN2510_BRG_64X	TQ = 64 x (2TOSC)

同步跳转宽度 （SJW）

通过 SJW1:SJW0 位 （CNF1 寄存器）控制

CAN2510_SJW_1TQ	**SJW 长度 = 1 TQ**
CAN2510_SJW_2TQ	SJW 长度 = 2 TQ
CAN2510_SJW_3TQ	SJW 长度 = 3 TQ
CAN2510_SJW_4TQ	SJW 长度 = 4 TQ

相位缓冲段 2 宽度

通过 PH2SEG2:PH2SEG0 位 （CNF3 寄存器）控制

CAN2510_PH2SEG_2TQ	**长度 = 2 TQ**
CAN2510_PH2SEG_3TQ	长度 = 3 TQ
CAN2510_PH2SEG_4TQ	长度 = 4 TQ
CAN2510_PH2SEG_5TQ	长度 = 5 TQ
CAN2510_PH2SEG_6TQ	长度 = 6 TQ
CAN2510_PH2SEG_7TQ	长度 = 7 TQ
CAN2510_PH2SEG_8TQ	长度 = 8 TQ

相位缓冲段 1 宽度

通过 PH1SEG2:PH1SEG0 位 （CNF2 寄存器）控制

CAN2510_PH1SEG_1TQ	**长度 = 1 TQ**
CAN2510_PH1SEG_2TQ	长度 = 2 TQ
CAN2510_PH1SEG_3TQ	长度 = 3 TQ
CAN2510_PH1SEG_4TQ	长度 = 4 TQ
CAN2510_PH1SEG_5TQ	长度 = 5 TQ
CAN2510_PH1SEG_6TQ	长度 = 6 TQ
CAN2510_PH1SEG_7TQ	长度 = 7 TQ
CAN2510_PH1SEG_8TQ	长度 = 8 TQ

传播段宽度

通过 PRSEG2:PRSEG0 位 （CNF2 寄存器）控制

CAN2510_PROPSEG_1TQ	**长度 = 1 TQ**
CAN2510_PROPSEG_2TQ	长度 = 2 TQ
CAN2510_PROPSEG_3TQ	长度 = 3 TQ
CAN2510_PROPSEG_4TQ	长度 = 4 TQ
CAN2510_PROPSEG_5TQ	长度 = 5 TQ
CAN2510_PROPSEG_6TQ	长度 = 6 TQ
CAN2510_PROPSEG_7TQ	长度 = 7 TQ
CAN2510_PROPSEG_8TQ	长度 = 8 TQ

相位段 2 的源

通过 BTLMODE 位 （CNF2 寄存器）控制。该值将确定相位段 2 的长度是由 PH2SEG2:PH2SEG0 位决定，还是由 PH1SEG2:PH1SEG0 位和 (2TQ) 中的较大者来决定。

CAN2510_PH2SOURCE_PH2	**长度 = PH2SEG2:PH2SEG0**
CAN2510_PH2SOURCE_PH1	长度 = PH1SEG2:PH1SEG0 和 2TQ 中的较大者

位采样点频度

通过 SAM 位 （CNF2 寄存器）来控制。它将确定在采样点上此位被采样 1 次还是被采样 3 次 。

CAN2510_SAMPLE_1x	**只采样 1 次**
CAN2510_SAMPLE_3x	采样 3 次

CAN2510Init （续）

休眠模式下RX 引脚噪声滤波

由 WAKFIL 位 （CNF3 寄存器）控制。这将确定当器件处于休眠模式时， RX 引脚是否使用滤波器来抑制噪声。

CAN2510_RX_FILTER	**休眠模式时在 RX 引脚上滤波**
CAN2510_RX_NOFILTER	休眠模式时不在 RX 引脚上滤波

interruptEnables

interruptEnables 的值是下列值的位与（'&'）。其中用**粗体字**标出的选项是默认值。通过 CANINTE 寄存器的所有位来控制。

CAN2510_NONE_EN	**不允许任何中断**
CAN2510_MSGERR_EN	在报文接收或发送过程中出现错误时产生中断
CAN2510_WAKEUP_EN	在 CAN 总线工作时产生中断
CAN2510_ERROR_EN	在 EFLG 错误条件改变时产生中断
CAN2510_TXB2_EN	在发送缓冲区 2 为空时产生中断
CAN2510_TXB1_EN	在发送缓冲区 1 为空时产生中断
CAN2510_TXB0_EN	在发送缓冲区 0 为空时产生中断
CAN2510_RXB1_EN	当接收缓冲区 1 收到报文时产生中断
CAN2510_RXB0_EN	当接收缓冲区 0 收到报文时产生中断

SPI_syncMode

指定 PIC18CXXX SPI 同步频率：

CAN2510_SPI_FOSC4	**以 FOSC/4 频率通讯**
CAN2510_SPI_FOSC16	以 FOSC/16 频率通讯
CAN2510_SPI_FOSC64	以 FOSC/64 频率通讯
CAN2510_SPI_FOSCTMR2	以 TMR2/2 频率通讯

SPI_busMode

指定 PIC18CXXX SPI 总线模式：

CAN2510_SPI_MODE00	**使用 SPI 00 模式进行通讯**
CAN2510_SPI_MODE01	使用 SPI 01 模式进行通讯

SPI_smpPhase

指定 PIC18CXXX SPI 采样点位置：

CAN2510_SPI_SMPMID	**在 SPI 位的中间采样**
CAN2510_SPI_SMPEND	在 SPI 位的末端采样

说明： 该函数初始化 PIC18CXXX SPI 模块，复位 MCP2510 器件 （如果需要），并配置 MCP2510 寄存器。

注： 该函数执行完后，MCP2510 处于配置模式。

返回值： 表明 MCP2510 是否完成初始化。
如果初始化完成，返回 0 ；
如果没有完成，则返回 -1。

文件名： caninit.c

CAN2510InterruptEnable

功能： 将 CAN2510 的中断允许位 （CANINTE 寄存器）修改为新值。

所要求的 CAN 模式： 所有模式均可

头文件：
```
can2510.h,
spi_can.h
```

函数原型：
```
void CAN2510InterruptEnable(
    unsigned char interruptEnables );
```

参数： ***interruptEnables***

interruptEnable 的值是下列值的位与 （'&'）。由**粗体**标出的选项是默认值。通过 CANINTE 寄存器的所有位来控制。

CAN2510_NONE_EN	**不允许任何中断 （00000000）**
CAN2510_MSGERR_EN	在报文接收或发送过程中出现错误时产生中断 （10000000）
CAN2510_WAKEUP_EN	在 CAN 总线工作时产生中断 （01000000）
CAN2510_ERROR_EN	EFLG 错误条件变化时产生中断 （00100000）
CAN2510_TXB2_EN	在发送缓冲区 2 为空时产生中断 （00010000）
CAN2510_TXB1_EN	在发送缓冲区 1 为空时产生中断 （00001000）
CAN2510_TXB0_EN	在发送缓冲区 0 为空时产生中断 （00000100）
CAN2510_RXB1_EN	当接收缓冲区 1 收到报文时产生中断 （00000010）
CAN2510_RXB0_EN	当接收缓冲区 0 收到报文时产生中断 （00000001）

说明： 该函数用对所期望中断源进行位与操作所得的值来更新 CANINTE 寄存器。

文件名： caninte.c

CAN2510InterruptStatus

功能： 指明 CAN2510 的中断源。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h,
spi_can.h

函数原型：
```
unsigned char CAN2510InterruptStatus(
    void );
```

说明： 该函数读取 CANSTAT 寄存器，并且根据 ICODE2:ICODE0 位的状态指定编码。

返回值： 函数返回下列值之一：

CAN2510_NO_INTS	没有中断产生
CAN2510_WAKEUP_INT	CAN 总线工作时产生中断
CAN2510_ERROR_INT	EFLG 错误条件改变时产生中断
CAN2510_TXB2_INT	发送缓冲区 2 为空时产生中断
CAN2510_TXB1_INT	发送缓冲区 1 为空时产生中断
CAN2510_TXB0_INT	发送缓冲区 0 为空时产生中断
CAN2510_RXB1_INT	接收缓冲区 1 接收到报文时产生中断
CAN2510_RXB0_INT	接收缓冲区 0 接收到报文时产生中断

文件名： canints.c

CAN2510LoadBufferStd

功能： 把标准数据帧装入指定的发送缓冲区。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
void CAN2510LoadBufferStd(
    unsigned char bufferNum,
    unsigned int  msgId,
    unsigned char numBytes,
    unsigned char *data );
```

参数：

bufferNum

指定要装入报文的缓冲区，取下列值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

msgId

CAN 报文标识符。对于标准报文，标识符可达 11 位。

numBytes

要发送的数据的字节数，取值为 0 到 8。如果值大于 8，则只存储前 8 个字节。

data

要装入的数组。此数组长度必须大于等于 ***numBytes*** 中指定的值。

CAN2510LoadBufferStd （续）

说明： 该函数仅装入报文，而不发送报文。
可使用函数 CAN2510WriteBuffer() 将报文写到 CAN 总线。
该函数不设置缓冲区的优先级。可使用函数
CAN2510SetBufferPriority() 来设置缓冲区优先级。

文件名： canloads.c

CAN2510LoadBufferXtd

功能： 把扩展数据帧装入指定的发送缓冲区。

所要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：

```
void CAN2510LoadBufferXtd(
    unsigned char bufferNum,
    unsigned long  msgId,
    unsigned char numBytes,
    unsigned char *data );
```

参数：

bufferNum
指定要装入报文的缓冲区，取值如下：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

msgId
CAN 报文标识符，对于扩展报文，可达 29 位。

numBytes
要发送的数据的字节数，取值为从 0 到 8。如果值大于 8，则只存储数据的前 8 个字节。

data
要装入的数组。此数组长度必须大于等于 ***numBytes*** 中指定的值。

说明： 该函数仅装入报文，而不发送报文。
可使用函数 CAN2510WriteBuffer() 将报文写到 CAN 总线。
该函数不设置缓冲区的优先级。可使用函数
CAN2510SetBufferPriority() 来设置缓冲区的优先级。

文件名： canloadx.c

CAN2510LoadRTRStd

功能： 把标准远程帧装入指定的发送缓冲区。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：

```
void CAN2510LoadBufferStd(
    unsigned char bufferNum,
    unsigned int  msgId,
    unsigned char numBytes);
```

参数：

bufferNum

指定要装入报文的缓冲区，取下列值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

msgId

CAN 报文标识符，对于标准报文可达 11 位。

numBytes

要发送的数据的字节数，取值为从 0 到 8。如果此值大于 8，则只存储数据的前 8 个字节。

说明： 该函数仅装入报文，而不发送报文。
可使用函数 CAN2510WriteBuffer() 将报文写到 CAN 总线。
该函数不设置缓冲区的优先级。可使用函数 CAN2510SetBufferPriority() 来设置缓冲区的优先级。

文件名： canlrtrs.c

CAN2510LoadRTRXtd

功能： 把扩展远程帧装入指定的发送缓冲区。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：

```
void CAN2510LoadBufferXtd(
    unsigned char bufferNum,
    unsigned long msgId,
    unsigned char numBytes);
```

参数：

bufferNum

指定要装入报文的缓冲区，取下列值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

msgId

CAN 报文标识符，对于扩展报文可达 29 位。

numBytes

要发送的数据的字节数，取值为从 0 到 8。如果此值大于 8，则只存储数据的前 8 个字节。

说明： 该函数仅装入报文，而不发送报文。
可使用函数 CAN2510WriteBuffer() 将报文写到 CAN 总线。
该函数不设置缓冲区的优先级。可使用函数 CAN2510SetBufferPriority() 来设置缓冲区的优先级。

文件名： canlrtrx.c

CAN2510ReadMode

功能： 读取 MCP2510 的当前工作模式。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
unsigned char CAN2510ReadMode( void );
```

说明： 该函数读取当前的工作模式。对于新模式，可能有等待请求。

返回值： ***mode***

mode 的值可以为下列值之一（在文件 can2510.h 中定义）。由 OPMODE2:OPMODE0 位 （CANSTAT 寄存器）指定。取值如下：

CAN2510_MODE_CONFIG	可修改配置寄存器
CAN2510_MODE_NORMAL	正常 （发送和接收报文）
CAN2510_MODE_SLEEP	等待中断
CAN2510_MODE_LISTEN	仅监听，不发送
CAN2510_MODE_LOOPBACK	用于测试，自发自收

文件名： canmoder.c

CAN2510ReadStatus

功能： 读取 MCP2510 发送缓冲区和接收缓冲区的状态。

所要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
unsigned char CAN2510ReadStatus( void );
```

说明： 该函数读取发送缓冲区和接收缓冲区的当前状态。

返回值： ***status***

status（一个无符号字节）的值有下列格式：

bit 7	TXB2IF
bit 6	TXB2REQ
bit 5	TXB1IF
bit 4	TXB1REQ
bit 3	TXB0IF
bit 2	TXB0REQ
bit 1	RXB1IF
bit 0	RXB0IF

文件名： canstats.c

CAN2510Reset

功能： 复位 MCP2510。

要求的 CAN 模式： 所有模式均可

头文件：
```
can2510.h
spi_can.h
spi.h
```

函数原型：
```
void CAN2510Reset( void );
```

说明： 该函数复位 MCP2510。

文件名： canreset.c

CAN2510SendBuffer

功能： 请求发送指定发送缓冲区中的报文。

要求的 CAN 模式： 正常模式

头文件： can2510.h

函数原型：
```
void CAN2510WriteBuffer
    ( unsigned char bufferNum );
```

参数： ***bufferNum***

指定请求发送哪些（个）缓冲区中的报文，取下列值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2
CAN2510_TXB0_B1	发送缓冲区 0 和发送缓冲区 1
CAN2510_TXB0_B2	发送缓冲区 0 和发送缓冲区 2
CAN2510_TXB1_B2	发送缓冲区 1 和发送缓冲区 2
CAN2510_TXB0_B1_B2	发送缓冲区 0、发送缓冲区 1 和发送缓冲区 2

说明： 该函数请求发送先前装入、存储在指定缓冲区中的报文。要装入报文，使用函数 CAN2510LoadBufferStd() 或 CAN2510LoadBufferXtd()。

文件名： cansend.c

CAN2510SequentialRead

功能： 从 MCP2510 中，自指定地址开始连续读取指定的字节数。这些值存储在 ***DataArray*** 中。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
void CAN2510SequentialRead(
    unsigned char *DataArray
    unsigned char CAN2510addr
    unsigned char numbytes );
```

参数： ***DataArray***

存储连续读取数据的数据数组的首地址。

CAN2510addr

MCP2510 中开始连续读取处的地址。

numbytes

连续读取的字节数。

CAN2510SequentialRead （续）

说明： 该函数从 MCP2510 中，自指定的地址开始读取连续的字节。这些值存储到自指定数组首地址开始的地址。

文件名： readseq.c

CAN2510SequentialWrite

功能： 自指定地址开始，向 MCP2510 连续写入指定的字节数。这些值来自于 ***DataArray***。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：

```
void CAN2510SequentialWrite(
    unsigned char *DataArray
    unsigned char CAN2510addr
    unsigned char numbytes );
```

参数：

DataArray
包含连续写数据的数组的首地址。

CAN2510addr
MCP2510 中连续写数据开始处的地址。

numbytes
要连续写的字节数。

说明： 该函数自指定的地址开始，将连续的字节写入 MCP2510。这些值来自于自指定数组首地址开始的地址。

文件名： wrtseq.c

CAN2510SetBufferPriority

功能： 为指定发送缓冲区装入指定的优先级。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：

```
void CAN2510SetBufferPriority(
    unsigned char bufferNum,
    unsigned char bufferPriority );
```

参数：

bufferNum
指定要配置优先级的缓冲区，取下列值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

bufferPriority
缓冲区的优先级，取下列值之一：

CAN2510_PRI_HIGHEST	最高的报文优先级
CAN2510_PRI_HIGH	高报文优先级
CAN2510_PRI_LOW	低报文优先级
CAN2510_PRI_LOWEST	最低的报文优先级

说明： 该函数为指定缓冲区装入指定的优先级。

文件名： cansetpr.c

CAN2510SetMode

功能： 配置 MCP2510 的工作模式。

要求的 CAN 模式： 所有模式均可

头文件： can2510.h

函数原型：
```
void CAN2510SetMode( unsigned char mode );
```

参数： ***mode***

mode 的值可以是下列值之一（在 can2510.h 中定义）。由 REQOP2:REQOP0 位（CANCTRL 寄存器）控制。

CAN2510_MODE_CONFIG	可修改配置寄存器
CAN2510_MODE_NORMAL	正常（发送和接收报文）
CAN2510_MODE_SLEEP	等待中断
CAN2510_MODE_LISTEN	仅监听，不发送
CAN2510_MODE_LOOPBACK	用于测试，自发自收

说明： 该函数配置指定的模式。直到所有等待发送的报文都发送完，模式才会改变。

文件名： canmodes.c

CAN2510SetMsgFilterStd

功能： 为标准报文配置某个接收缓冲区的所有过滤器和屏蔽器。

要求的 CAN 模式： 所有模式

头文件： can2510.h

函数原型：
```
unsigned char CAN2510SetMsgFilterStd(
    unsigned char bufferNum,
    unsigned int  mask,
    unsigned int  *filters );
```

参数： ***bufferNum***

指定要配置过滤器和屏蔽器的接收缓冲区，取下列值之一：

CAN2510_RXB0	配置 RXM0、RXF0 和 RXF1
CAN2510_RXB1	配置 RXM1、RXF2、RXF3、RXF4 和 RXF5

mask

屏蔽器设定值

filters

过滤器设定值。

对于缓冲区 0：

标准长度的报文：2 个无符号整数组成的数组

对于缓冲区 1：

标准长度的报文：4 个无符号整数组成的数组

说明： 该函数将 MCP2510 配置为配置模式，然后将屏蔽器设定值和过滤器设定值写到相应的寄存器。在返回前，将 MCP2510 配置为原来的模式。

返回值： 表明能否正确修改 MCP2510 的模式。
如果工作模式的初始化和恢复完成，返回 0；
如果工作模式的初始化和恢复未完成，则返回 -1。

文件名： canfms.c

CAN2510SetMsgFilterXtd

功能： 为扩展报文配置某个接收缓冲区的所有过滤器和屏蔽器。

要求的 CAN 模式： 所有模式

头文件： can2510.h

函数原型：

```
unsigned char CAN2510SetMsgFilterXtd(
    unsigned char bufferNum,
    unsigned long mask,
    unsigned long *filters );
```

参数：

bufferNum

指定要配置过滤器和屏蔽器的接收缓冲区，取下列值之一：

CAN2510_RXB0	配置 RXM0、 RXF0 和 RXF1
CAN2510_RXB1	配置 RXM1、 RXF2、 RXF3、 RXF4 和 RXF5

mask

屏蔽器设定值

filters

过滤器设定值：

对于缓冲区 0：

扩展长度的报文：2 个无符号长整型组成的数组

对于缓冲区 1：

扩展长度的报文：4 个无符号长整型组成的数组

说明： 该函数将 MCP2510 设置为配置模式，然后将过滤器设定值和屏蔽器设定值写到相应的寄存器。在返回前，将 MCP2510 配置为原来的模式。

返回值： 表明能否正确修改 MCP2510 的模式。
如果工作模式的初始化和恢复完成，返回 0；
如果工作模式的初始化和恢复未完成，则返回 -1。

文件名： canfmx.c

CAN2510SetSingleFilterStd

功能： 为标准 （Std）报文配置指定的接收过滤器。

要求的 CAN 模式： 配置模式

头文件： can2510.h

函数原型：

```
void CAN2510SetSingleFilterStd(
   unsigned char filterNum,
   unsigned int filter );
```

参数：

filterNum

指定要配置的接收过滤器，取下列值之一：

CAN2510_RXF0	配置 RXF0	（用于 RXB0）
CAN2510_RXF1	配置 RXF1	（用于 RXB0）
CAN2510_RXF2	配置 RXF2	（用于 RXB1）
CAN2510_RXF3	配置 RXF3	（用于 RXB1）
CAN2510_RXF4	配置 RXF4	（用于 RXB1）
CAN2510_RXF5	配置 RXF5	（用于 RXB1）

filter

过滤器设定值。

说明： 该函数将过滤器设定值写入到相应的寄存器。在执行该函数前 MCP2510 必须处于配置模式。

文件名： canfilts.c

CAN2510SetSingleFilterXtd

功能： 为扩展 （Xtd）报文配置指定的接收过滤器。

要求的 CAN 模式： 配置模式

头文件： can2510.h

函数原型：

```
void CAN2510SetSingleFilterXtd(
   unsigned char filterNum,
   unsigned long  filter );
```

参数：

filterNum

指定要配置的接收过滤器，取下列值之一：

CAN2510_RXF0	配置 RXF0	（用于 RXB0）
CAN2510_RXF1	配置 RXF1	（用于 RXB0）
CAN2510_RXF2	配置 RXF2	（用于 RXB1）
CAN2510_RXF3	配置 RXF3	（用于 RXB1）
CAN2510_RXF4	配置 RXF4	（用于 RXB1）
CAN2510_RXF5	配置 RXF5	（用于 RXB1）

filter

过滤器设定值。

说明： 该函数将过滤器设定值写入到相应的寄存器。在执行该函数前 MCP2510 必须处于配置模式。

文件名： canfiltx.c

CAN2510SetSingleMaskStd

功能： 为标准 （Std）格式报文配置指定接收缓冲区的屏蔽器。

要求的 CAN 模式： 配置模式

头文件： can2510.h

函数原型：

```
unsigned char CAN2510SetSingleMaskStd(
    unsigned char maskNum,
    unsigned int  mask );
```

参数：

maskNum

指定要配置的接收屏蔽器，取下列值之一：

CAN2510_RXM0	配置 RXM0	（用于 RXB0）
CAN2510_RXM1	配置 RXM1	（用于 RXB1）

mask

屏蔽器设定值。

说明： 该函数将屏蔽器设定值写到相应的寄存器。在执行该函数前 MCP2510 必须处于配置模式。

文件名： canmasks.c

CAN2510SetSingleMaskXtd

功能： 为扩展 （Xtd）报文配置指定接收缓冲区的屏蔽器。

要求的 CAN 模式： 配置模式

头文件： can2510.h

函数原型：

```
unsigned char CAN2510SetSingleMaskXtd(
    unsigned char maskNum,
    unsigned long mask );
```

参数：

maskNum

指定要配置的接收屏蔽器，取下列值之一：

CAN2510_RXM0	配置 RXM0	（用于 RXB0）
CAN2510_RXM1	配置 RXM1	（用于 RXB1）

mask

屏蔽器设定值。

说明： 该函数将屏蔽器设定值写到相应的寄存器。在执行该函数前 MCP2510 必须处于配置模式。

文件名： canmaskx.c

CAN2510WriteBuffer

功能： 启动所选发送缓冲区的 CAN 报文发送。

要求的 CAN 模式： 所有模式

头文件： can2510.h

函数原型：
```
unsigned char CAN2510WriteBuffer(
   unsigned char bufferNum )
```

参数： ***bufferNum***

指定要装入报文的缓冲区，取如下值之一：

CAN2510_TXB0	发送缓冲区 0
CAN2510_TXB1	发送缓冲区 1
CAN2510_TXB2	发送缓冲区 2

说明： 该函数启动所选发送缓冲区的报文发送。

文件名： canwrbuf.c

CAN2510WriteStd

功能： 使用第一个可用的发送缓冲区，将标准格式报文写到 CAN 总线。

要求的 CAN 模式： 正常模式

头文件： can2510.h

函数原型：

```
unsigned char CAN2510WriteStd(
    unsigned int  msgId,
    unsigned char msgPriority,
    unsigned char numBytes,
    unsigned char *data );
```

参数：

msgId
CAN 报文标识符，对于标准报文有 11 位，这个 11 位的标识符存储在 msgId（无符号整型）的低 11 位中。

msgPriority
缓冲区的优先级，取下列值之一：

CAN2510_PRI_HIGHEST	最高的报文优先级
CAN2510_PRI_HIGH	高报文优先级
CAN2510_PRI_LOW	低报文优先级
CAN2510_PRI_LOWEST	最低的报文优先级

numBytes
要发送数据的字节数，取值为从 0 到 8。如果值大于 8，则仅发送数据的前 8 个字节。

data
要写的数据值的数组。此数组长度必须大于等于 ***numBytes*** 中指定的值。

说明： 该函数将查询每个发送缓冲区，以确定是否有等待发送的报文，并将指定的报文传送到第一个可用的缓冲区。

返回值： 此值表明使用了哪一个缓冲区发送报文 （0、1 或 2）。
-1 表明没有发送报文。

文件名： canwrits.c

CAN2510WriteXtd

功能： 使用第一个可用的发送缓冲区，将扩展格式报文写到 CAN 总线。

要求的 CAN 模式： 正常模式

头文件： can2510.h

函数原型：

```
unsigned char CAN2510WriteXtd(
    unsigned long msgId,
    unsigned char msgPriority,
    unsigned char numBytes,
    unsigned char *data );
```

参数：

msgId
CAN 报文标识符，对于扩展报文有 29 位，这个 29 位的标识符存储在 msgId （无符号长整型）的低 29 位中。

msgPriority
缓冲区的优先级，取下列值之一：

CAN2510_PRI_HIGHEST	最高的报文优先级
CAN2510_PRI_HIGH	高报文优先级
CAN2510_PRI_LOW	低报文优先级
CAN2510_PRI_LOWEST	最低的报文优先级

numBytes
要发送数据的字节数，取值为从 0 到 8。如果值大于 8，则仅发送数据的前 8 个字节。

data
要写的数据值的数组。此数组长度必须大于等于 ***numBytes*** 中指定的值。

说明： 该函数将查询每个发送缓冲区，以确定是否有等待发送的报文，并将指定的报文传送到第一个可用的缓冲区。

返回值： 此值表明使用了哪一个缓冲区发送报文 （0、1 或 2）。
-1 表明没有发送报文。

文件名： canwritx.c

3.4 软件 I²C 函数

设计这些函数的目的是使用 PIC18 单片机的 I/O 引脚来实现 I²C 总线，具体函数见下表：

表 3-6： I²C™ 软件函数

函数	描述
Clock_test	为延长从时钟低电平时间产生延时。
SWAckI2C	产生 I²C™ 总线*应答*条件。
SWGetcI2C	从 I²C 总线读取一个字节。
SWGetsI2C	读取一个数据串。
SWNotAckI2C	产生 I²C 总线*不应答*条件。
SWPutcI2C	将一个字节写到 I²C 总线。
SWPutsI2C	将一个数据串写到 I²C 总线。
SWReadI2C	从 I²C 总线读取一个字节。
SWRestartI2C	产生 I²C 总线*重复启动*条件。
SWStartI2C	产生 ²C 总线*启动*条件。
SWStopI2C	产生 I²C 总线*停止*条件。
SWWriteI2C	将一个字节写到 I²C 总线。

这些函数的预编译形式使用默认的引脚分配。通过在文件 sw_i2c.h（在编译器安装目录的 h 子目录下）中重新定义下列宏，可以改变引脚的分配：

表 3-7： 选择 I²C™ 引脚分配的宏

I²C 线	宏	默认值	用途
DATA 引脚	DATA_PIN	PORTBbits.RB4	用于数据 （DATA）线的引脚。
	DATA_LAT	LATBbits.RB4	与 DATA 引脚有关的锁存器。
	DATA_LOW	TRISBbits.TRISB4 = 0;	将 DATA 引脚配置为输出的语句。
	DATA_HI	TRISBbits.TRISB4 = 1;	将 DATA 引脚配置为输入的语句。
CLOCK 引脚	SCLK_PIN	PORTBbits.RB3	用于时钟 （CLOCK）线的引脚。
	SCLK_LAT	LATBbits.LATB3	与 CLOCK 引脚有关的锁存器。
	CLOCK_LOW	TRISBbits.TRISB3 = 0;	将 CLOCK 引脚配置为输出的语句。
	CLOCK_HI	TRISBbits.TRISB3 = 1;	将 CLOCK 引脚配置为输入的语句。

完成这些定义后，用户必须重新编译 I²C 函数，然后在项目中使用更新过的文件。这可通过把库源文件添加到项目中，或者使用提供的批处理文件重新编译库文件来完成。

3.4.1 函数描述

Clock_test

功能： 为延长从时钟低电平时间产生延时。

头文件： sw_i2c.h

函数原型：
```
char Clock_test( void );
```

说明： 调用该函数可延长从时钟低电平时间。可能需要根据应用的要求调节延时时间。如果在延时周期结束时，时钟线为低电平，则返回一个表明时钟错误的值。

返回值： 如果没有发生时钟错误，返回 0；
如果发生时钟错误，则返回 -2。

文件名： swckti2c.c

SWAckI2C
SWNotAckI2C

功能： 产生 I^2C 总线*应答／不应答*条件。

头文件： sw_i2c.h

函数原型：
```
char SWAckI2C( void );
char SWNotAckI2C( void );
```

说明： 调用该函数将产生 I^2C 总线*应答*序列。

返回值： 如果从器件应答，返回 0；
如果从器件不应答，则返回 -1。

文件名： swacki2c.c

SWGetcI2C

参见 **SWReadI2C**。

SWGetsI2C

功能： 从 I^2C 总线读取一个数据串。

头文件： sw_i2c.h

函数原型：
```
char SWGetsI2C(
      unsigned char *rdptr,
      unsigned char length );
```

参数：
rdptr
存储从 I^2C 总线上读取的数据的地址。
length
要读取的字节数。

说明： 该函数读取一个预定义长度的数据串。

返回值： 如果主器件在所有字节接收完前产生一个*不应答（NOT ACK）*总线条件，返回 -1；
否则，返回 0。

文件名： swgtsi2c.c

代码示例：
```
char x[10];
SWGetsI2C( x,5 );
```

SWNotAckI2C

参见 **SWAckI2C**。

SWPutcI2C

参见 **SWWriteI2C**。

SWPutsI2C

功能： 写一个数据串到 I²C 总线。

头文件： sw_i2c.h

函数原型：
```
char SWPutsI2C(
      unsigned char *wrdptr );
```

参数： ***wrdptr***
指向要写到 I²C 总线的数据的指针。

说明： 该函数将写一个数据串到 I²C 总线，直到空字符为止（但不包括空字符）。

返回值： 如果写到 I²C 总线时有错误，返回 -1；
否则，返回 0。

文件名： swptsi2c.c

代码示例：
```
char mybuff [] = "Hello";
SWPutsI2C(mybuff);
```

SWReadI2C
SWGetcI2C

功能： 从 I²C 总线上读取一个字节。

头文件： sw_i2c.h

函数原型：
```
char SWReadI2C( void );
```

说明： 该函数通过在预定义的 I²C 时钟线上产生适当的信号，来读取一个数据字节。

返回值： 该函数返回已读取的 I²C 数据字节。
如果该函数出现错误，则返回 -1。

文件名： swgtci2c.c

SWRestartI2C

功能： 产生 I²C 总线*重复启动*条件。

头文件： sw_i2c.h

函数原型：
```
void SWRestartI2C( void );
```

说明： 调用该函数，可产生 I²C 总线重复启动条件。

文件名： swrsti2c.c

SWStartI2C

功能： 产生 I²C 总线*启动*条件。

头文件： sw_i2c.h

函数原型：
```
void SWStartI2C( void );
```

说明： 调用该函数来产生 I²C 总线*启动*条件。

文件名： swstri2c.c

SWStopI2C

功能： 产生 I²C 总线*停止*条件。

头文件： sw_i2c.h

函数原型：
```
void SWStopI2C( void );
```

说明： 调用该函数来产生 I²C 总线*停止*条件。

文件名： swstpi2c.c

SWWriteI2C
SWPutcI2C

功能： 写一个字节到 I²C 总线。

头文件： sw_i2c.h

函数原型：
```
char SWWriteI2C(
      unsigned char data_out );
```

参数：

data_out
要写到 I²C 总线的一个数据字节。

说明： 该函数将一个数据字节写到预定义的数据引脚。

返回值： 如果写入成功，返回 0；
如果出现错误，返回 -1。

文件名： swptci2c.c

代码示例
```
if(SWWriteI2C(0x80))
  {
    errorHandler();
  }
```

3.4.2 使用示例

下面是一个简单的代码示例，举例说明了与 Microchip 24LC01B I^2C 电可擦除存储器件进行 I^2C 通讯的软件实现。

```
#include <p18cxxx.h>
#include <sw_i2c.h>
#include <delays.h>

// FUNCTION Prototype
void main(void);
void byte_write(void);
void page_write(void);
void current_address(void);
void random_read(void);
void sequential_read(void);
void ack_poll(void);
unsigned char warr[] = {8,7,6,5,4,3,2,1,0};
unsigned char rarr[15];
unsigned char far *rdptr = rarr;
unsigned char far *wrptr = warr;
unsigned char var;

#define W_CS  PORTA.2

//*************************************************
void main( void )
{
  byte_write();
  ack_poll();
  page_write();
  ack_poll();
  Nop();
  sequential_read();
  Nop();
  while (1);  // Loop indefinitely
}

void byte_write( void )
{
  SWStartI2C();
  var = SWPutcI2C(0xA0); // control byte
  SWAckI2C();
  var = SWPutcI2C(0x10); // word address
  SWAckI2C();
  var = SWPutcI2C(0x66); // data
  SWAckI2C();
  SWStopI2C();
}

void page_write( void )
{
  SWStartI2C();
  var = SWPutcI2C(0xA0); // control byte
  SWAckI2C();
  var = SWPutcI2C(0x20); // word address
  SWAckI2C();
  var = SWPutsI2C(wrptr); // data
  SWStopI2C();
}
```

```
void sequential_read( void )
{
  SWStartI2C();
  var = SWPutcI2C( 0xA0 ); // control byte
  SWAckI2C();
  var = SWPutcI2C( 0x00 ); // address to read from
  SWAckI2C();
  SWRestartI2C();
  var = SWPutcI2C( 0xA1 );
  SWAckI2C();
  var = SWGetsI2C( rdptr, 9 );
  SWStopI2C();
}

void current_address( void )
{
  SWStartI2C();
  SWPutcI2C( 0xA1 ); // control byte
  SWAckI2C();
  SWGetcI2C();     // word address
  SWNotAckI2C();
  SWStopI2C();
}

void ack_poll( void )
{
  SWStartI2C();
  var = SWPutcI2C( 0xA0 );  // control byte
  while( SWAckI2C() )
  {
    SWRestartI2C();
    var = SWPutcI2C(0xA0); // data
  }
  SWStopI2C();
}
```

3.5 软件 SPI 函数

设计这些函数的目的是使用 PIC18 单片机的 I/O 引脚来实现 SPI 。具体函数见下表：

表 3-8： 软件 SPI 函数

函数	描述
ClearCSSWSPI	将片选（$\overline{CS}$）引脚清零。
OpenSWSPI	配置用于 SPI 的 I/O 引脚。
putcSWSPI	向软件 SPI 写一字节数据。
SetCSSWSPI	置位片选（$\overline{CS}$）引脚。
WriteSWSPI	向软件 SPI 总线写一字节数据。

这些函数的预编译形式使用默认的引脚分配。通过在文件 sw_spi.h （在编译器安装目录的 h 子目录下）中重新定义下列宏，可以改变引脚分配：

表 3-9： 选择 SPI 引脚分配的宏

LCD 控制器线	宏	默认值	用途
$\overline{CS}$ 引脚	SW_CS_PIN TRIS_SW_CS_PIN	PORTBbits.RB2 TRISBbits.TRISB2	用于片选（$\overline{CS}$）线的引脚。 控制与 $\overline{CS}$ 线相连引脚的方向的位。
DIN 引脚	SW_DIN_PIN TRIS_SW_DIN_PIN	PORTBbits.RB3 TRISBbits.TRISB3	用于 DIN 线的引脚。 控制与 DIN 线相连引脚的方向的位。
DOUT 引脚	SW_DOUT_PIN TRIS_SW_DOUT_PIN	PORTBbits.RB7 TRISBbits.TRISB7	用于 DOUT 线的引脚。 控制与 DOUT 线相连引脚的方向的位。
SCK 引脚	SW_SCK_PIN TRIS_SW_SCK_PIN	PORTBbits.RB6 TRISBbits.TRISB6	用于 SCK 线的引脚。 控制与 SCK 线相连引脚的方向的位。

所提供的函数库能工作在四种模式之一。下表列出了用于在这些模式之间进行选择的宏。在重建软件 SPI 函数库时，必须要定义其中一种宏。

表 3-10： 用于选择模式的宏

宏	默认值	含义
MODE0	已定义	CKP = 0 CKE = 0
MODE1	未定义	CKP = 1 CKE = 0
MODE2	未定义	CKP = 0 CKE = 1
MODE3	未定义	CKP = 1 CKE = 1

完成这些定义后，用户必须重新编译软件 SPI 函数，然后在项目中包含更新过的文件。这可通过将软件 SPI 源文件添加到项目中，或者使用所提供的批处理文件重新编译库文件来完成。

3.5.1 函数描述

ClearCSSWSPI

功能： 将头文件 sw_spi.h 中指定的片选 （$\overline{CS}$）引脚清零。

头文件： sw_spi.h

函数原型：
```
void ClearCSSWSPI( void );
```

说明： 该函数将头文件 sw_spi.h 中指定用于软件 SPI 片选 （$\overline{CS}$）引脚的 I/O 引脚清零。

文件名： clrcsspi.c

OpenSWSPI

功能： 配置用于软件 SPI 的 I/O 引脚。

头文件： sw_spi.h

函数原型：
```
void OpenSWSPI( void );
```

说明： 该函数将用于软件 SPI 的 I/O 引脚配置为正确的输入或输出状态和逻辑电平。

文件名： opensspi.c

putcSWSPI

参见 **WriteSWSPI**。

SetCSSWSPI

功能： 将头文件 sw_spi.h 中指定的片选 （$\overline{CS}$）引脚置位。

头文件： sw_spi.h

函数原型：
```
void SetCSSWSPI( void );
```

说明： 该函数将头文件 sw_spi.h 中指定用于软件 SPI 片选 （$\overline{CS}$）引脚的 I/O 引脚置位。

文件名： setcsspi.c

WriteSWSPI
putcSWSPI

功能： 向软件 SPI 写一个字节。

头文件： sw_spi.h

函数原型：
```
char WriteSWSPI( char data );
```

参数：
data
要写到软件 SPI 的数据。

说明： 该函数将指定的数据字节写到软件 SPI，并且返回读取的数据字节。该函数不提供对片选引脚 （$\overline{CS}$）的控制。

返回值： 该函数返回从软件 SPI 的 （DIN）引脚的数据中读取的数据字节。

文件名： wrtsspi.c

代码示例；
```
char addr = 0x10;
char result;
result = WriteSWSPI( addr );
```

3.5.2 使用示例

```
#include <p18C452.h>
#include <sw_spi.h>
#include <delays.h>

void main( void )
{
  char address;

  // configure software SPI
  OpenSWSPI();

  for( address=0; address<0x10; address++ )
  {
    ClearCSSWSPI();         //clear CS pin
    WriteSWSPI( 0x02 );     //send write cmd
    WriteSWSPI( address ); //send address hi
    WriteSWSPI( address ); //send address low
    SetCSSWSPI();           //set CS pin
    Delay10KTCYx( 50 );     //wait 5000,000TCY
  }
}
```

3.6 软件 UART 函数

设计这些函数的目的是使用 PIC18 单片机的 I/O 引脚来实现 UART 。具体函数见下表：

表 3-11： 软件 UART 函数

函数	描述
getcUART	从软件 UART 中读取一个字节。
getsUART	从软件 UART 中读取一个字符串。
OpenUART	配置用于 UART 的 I/O 引脚。
putcUART	写一个字节到软件 UART。
putsUART	写一个字符串到软件 UART。
ReadUART	从软件 UART 中读取一个字节。
WriteUART	写一个字节到软件 UART。

这些函数的预编译形式使用默认的引脚分配。通过重新定义文件 writuart.asm、readuart.asm 和 openuart.asm 中的 equate（equ）语句，可以改变引脚分配。这些文件包含在编译器安装目录的 src/traditional/pmc/sw_uart 或 scr/extended/pmc/sw_uart 子目录中。

表 3-12： 用于选择 UART 引脚分配的宏

LCD 控制器线	定义	默认值	用途
TX 引脚	SWTXD	PORTB	用于发送线的端口。
	SWTXDpin	4	SWTXD 端口中用于 TX 线的位。
	TRIS_SWTXD	TRISB	与用于 TX 线的端口相关的数据方向寄存器。
RX 引脚	SWRXD	PORTB	用于接收线的端口。
	SWRXDpin	5	SWTXD 端口中用于 RX 线的位。
	TRIS_SWRXD	TRISB	与用于 RX 线的端口相关的数据方向寄存器。

更改这些定义后，用户必须重新编译软件 UART 函数，然后在项目中包含更新过的文件。这可通过把软件 UART 源文件添加到项目中，或使用 MPLAB C18 编译器安装目录中提供的批处理文件重新编译库文件来完成。

UART 函数库还要求用户定义下列函数，以提供适当的延时：

表 3-13： 软件 UART 延时函数

函数	功能
DelayTXBitUART	延时： ((((2*FOSC) / (4*baud)) + 1) / 2) - 12 周期
DelayRXHalfBitUART	延时： ((((2*FOSC) / (8*baud)) + 1) / 2) - 9 周期
DelayRXBitUART	延时： ((((2*FOSC) / (4*baud)) + 1) / 2) - 14 周期

3.6.1 函数描述

getcUART

参见 **ReadUART**。

getsUART

功能： 从软件 UART 中读取一个字符串。

头文件： sw_uart.h

函数原型：
```
void getsUART( char * buffer,
               unsigned char len);
```

参数：

buffer
指向从软件 UART 中读取字符串的指针。

len
要从软件 UART 中读取的字符数。

说明： 该函数从软件 UART 中读取 ***len*** 个字符，并放入 ***buffer***。

文件名： getsuart.c

代码示例：
```
char x[10];
getsUART( x, 5 );
```

OpenUART

功能： 配置用于软件 UART 的 I/O 引脚。

头文件： sw_uart.h

函数原型：
```
void OpenUART( void );
```

说明： 该函数将用于软件 UART 的 I/O 引脚配置为正确的输入或输出状态和逻辑电平。

文件名： openuart.asm

代码示例：
```
OpenUART();
```

putcUART

参见 **WriteUART。**

putsUART

功能： 写一个字符串到软件 UART。

头文件： sw_uart.h

函数原型：
```
void putsUART( char * buffer );
```

参数：

buffer
要写到软件 UART 的字符串。

说明： 该函数将一个字符串写到软件 UART。包括空字符在内的全部字符串将被写到 UART。

文件名： putsuart.c

代码示例：
```
char mybuff [] = "Hello";
putsUART( mybuff );
```

ReadUART
getcUART

功能： 从软件 UART 读取一个字节。

头文件： sw_uart.h

函数原型： char ReadUART(void);

说明： 该函数从软件 UART 读取一个数据字节。

返回值： 返回从软件 UART 的接收数据 （RXD）引脚读取的数据字节。

文件名： readuart.asm

代码示例：
```
char x;
x = ReadUART();
```

WriteUART
putcUART

功能： 写一个字节到软件 UART。

头文件： sw_uart.h

函数原型： void WriteUART(char ***data***);

参数 ×¸½¨： ***data***
要写到软件 UART 的数据字节。

说明： 该函数将指定的数据字节写到软件 UART。

文件名： writuart.asm

代码示例：
```
char x = 'H';
WriteUART( x );
```

3.6.2 使用示例

```
#include <p18C452.h>
#include <sw_uart.h>

void main( void )
{
  char data

  // configure software UART
  OpenUART();

  while( 1 )
  {
    data = ReadUART();   //read a byte
    WriteUART( data );   //bounce it back
  }
}
```

第 4 章 通用软件函数库

4.1 简介

本章讲述预编译标准 C 库文件中的通用软件库函数。所有这些函数的源代码都包含在 MPLAB C18 编译器安装目录的如下子目录中：

- `src\traditional\stdlib`
- `src\extended\stdlib`
- `src\traditional\delays`
- `src\extended\delays`

MPLAB C18 函数库支持如下函数类别：

- 字符分类函数
- 数据转换函数
- 存储器和字符串操作函数
- 延时函数
- 复位函数
- 字符输出函数

4.2 字符分类函数

这些函数与 ANSI 1989 标准 C 库函数中的同名函数是一致的，见下表：

表 4-1： 字符分类函数

函数	描述
`isalnum`	确定字符是否为字母数字字符。
`isalpha`	确定字符是否为字母。
`iscntrl`	确定字符是否为控制字符。
`isdigit`	确定字符是否为十进制数字。
`isgraph`	确定字符是否为图形字符。
`islower`	确定字符是否为小写字母。
`isprint`	确定字符是否为可打印字符。
`ispunct`	确定字符是否为标点字符。
`isspace`	确定字符是否为空白字符。
`isupper`	确定字符是否为大写字母。
`isxdigit`	确定字符是否为十六进制数字。

4.2.1 函数描述

isalnum

功能： 确定字符是否为字母数字字符。

头文件： ctype.h

函数原型：
```
unsigned char isalnum( unsigned char ch );
```

参数： ***ch***
要检查的字符。

说明： 如果字符在“A”到“Z”，“a”到“z”或者“0”到“9”的范围内，就认为它是字母数字字符。

返回值： 如果是字母数字字符，返回非 0；
否则，返回 0。

文件名： isalnum.c

isalpha

功能： 确定字符是否为字母。

头文件： ctype.h

函数原型：
```
unsigned char isalpha( unsigned char ch );
```

参数： ***ch***
要检查的字符。

说明： 如果字符在“A”到“Z”或者“a”到“z”的范围内，就认为它是字母。

返回值： 如果字符是字母，返回非 0；
否则，返回 0。

文件名： isalpha.c

iscntrl

功能： 确定字符是否为控制字符。

头文件： ctype.h

函数原型：
```
unsigned char iscntrl( unsigned char ch );
```

参数： ***ch***
要检查的字符。

说明： 如果字符不是由 isprint() 定义的可打印字符，则认为它是控制字符。

返回值： 如果字符为控制符，返回非 0；
否则，返回 0。

文件名： iscntrl.c

isdigit

功能： 确定字符是否为十进制数字。

头文件： ctype.h

函数原型： unsigned char isdigit(unsigned char ***ch***);

参数： ***ch***
要检查的字符。

说明： 如果字符在 “0” 到 “9” 范围内，就认为它是十进制数字。

返回值： 如果字符是十进制数字，返回非 0；
否则，返回 0。

文件名： isdigit.c

isgraph

功能： 确定字符是否为图形字符。

头文件： ctype.h

函数原型： unsigned char isgraph(unsigned char ***ch***);

参数： ***ch***
要检查的字符。

说明： 如果字符是除空格外的任何可打印字符，就认为它是图形字符。

返回值： 如果字符是图形字符，返回非 0；
否则，返回 0。

文件名： isgraph.c

islower

功能： 确定字符是否为小写字母。

头文件： ctype.h

函数原型： unsigned char islower(unsigned char ***ch***);

参数： ***ch***
要检查的字符。

说明： 如果字符在 “a” 到 “z” 范围内，就认为它是小写字母。

返回值： 如果字符是小写字母，返回非 0；
否则，返回 0。

文件名： islower.c

isprint

功能： 确定字符是否为可打印字符。

头文件： ctype.h

函数原型： `unsigned char isprint( unsigned char ch );`

参数： ***ch***
要检查的字符。

说明： 如果字符是在 0x20 至 0x7e （包括 0x20 和 0x7e）范围内，就认为它是可打印字符。

返回值： 如果字符是可打印字符，返回非 0；
否则，返回 0。

文件名： isprint.c

ispunct

功能： 确定字符是否为标点字符。

头文件： ctype.h

函数原型： `unsigned char ispunct( unsigned char ch );`

参数： ***ch***
要检查的字符。

说明： 如果字符是可打印字符，且既不是空格，也不是字母数字字符，则认为它是标点字符。

返回值： 如果字符是标点字符，返回非 0；
否则，返回 0。

文件名： ispunct.c

isspace

功能： 确定字符是否为空白字符。

头文件： ctype.h

函数原型： `unsigned char isspace (unsigned char ch);`

参数： ***ch***
要检查的字符。

说明： 如果字符属于下列一种：空格 （“ ”）、制表符 （“\t”）、回车符（“\r”）、换行符 （“\n”）、换页符 （“\f”）或者垂直制表符（“\v”），则认为它是空白字符。

返回值： 如果字符是空白字符，返回非 0；
否则，返回 0。

文件名： isspace.c

isupper

功能： 确定字符是否为大写字母。

头文件： ctype.h

函数原型： unsigned char isupper (unsigned char ***ch***);

参数： ***ch***
要检查的字符。

说明： 如果字符在 “A” 到 “Z” 范围内，就认为它是大写字母字符。

返回值： 如果字符是大写字母字符，返回非 0；
否则，返回 0。

文件名： isupper.c

isxdigit

功能： 确定字符是否为十六进制数字。

头文件： ctype.h

函数原型： unsigned char isxdigit(unsigned char ***ch***);

参数： ***ch***
要检查的字符。

说明： 如果字符在 “0” 到 “9”， “a” 到 “f” 或” “A” 到 “F” 范围内，就认为它是十六进制数字字符。

返回值： 如果字符是十六进制数字字符，返回非 0；
否则，返回 0。

文件名： isxdig.c

4.3 数据转换函数

除非在函数描述中另有注明，这些函数和 ANSI 1989 标准 C 库函数中的同名函数是一致的。具体函数见下表：

表 4-2： 数据转换函数

函数	描述
atob	将一个字符串转换为 8 位有符号数。
atof	将一个字符串转换为浮点数。
atoi	将一个字符串转换为 16 位有符号整型。
atol	将一个字符串转换为长整型。
btoa	将一个 8 位有符号数转换为字符串。
itoa	将一个 16 位有符号整型转换为字符串。
ltoa	将有符号长整型转换为字符串。
rand	生成一个伪随机整数。
srand	设置伪随机数发生器的起始种子值。
tolower	将字符转换为小写字母的 ASCII 字符。
toupper	将字符转换为大写字母的 ASCII 字符。
ultoa	将无符号长整型转换为字符串。

4.3.1 函数描述

atob

功能： 将一个字符串转换为一个 8 位有符号数。

头文件： stdlib.h

函数原型：
```
signed char atob( const char * s );
```

参数： ***s***
指向要转换的 ASCII 字符串的指针。

说明： 该函数将 ASCII 字符串 ***s*** 转换为 8 位有符号数 （-128 到 127）。输入的字符串必须以 10 为基数 （十进制），且可以字符指示符号 （“+”或“-”）开始。溢出结果未定义。该函数是 MPLAB C18 对 ANSI 标准函数库的扩展。

返回值： -128 到 127 范围内所有字符串的 8 位有符号数。

文件名： atob.asm

atof

功能： 将一个字符串转换为浮点数。

头文件： stdlib.h

函数原型：
```
double atof ( const char * s );
```

参数： ***s***
指向要转换的 ASCII 字符串的指针。

说明： 该函数将 ASCII 字符串 ***s*** 转换为浮点数。以下是可识别的浮点字符串示例：
```
-3.1415
1.0E2
1.0E+2
1.0E-2
```

返回值： 函数返回转换的结果。

文件名： atof.c

atoi

功能： 将一个字符串转换为 16 位有符号整数。

头文件： stdlib.h

函数原型：
```
int atoi( const char * s );
```

参数： ***s***
指向要转换的 ASCII 字符串的指针。

说明： 该函数将 ASCII 字符串 ***s*** 转换为 16 位有符号整型 （-32768 到 32767）。输入的字符串必须以 10 为基数 （十进制），且可以指示符号（“+”或 “-”）开始。溢出结果未定义。该函数是 MPLAB C18 对 ANSI 标准函数库的扩展。

返回值： 在 -32768 到 32767 范围内所有字符串的 16 位有符号整型

文件名： atoi.asm

atol

功能： 将一个字符串转换为长整型表示。

头文件： stdlib.h

函数原型：
```
long atol( const char * s );
```

参数： ***s***
指向要转换的 ASCII 字符串的指针。

说明： 该函数将 ASCII 字符串 ***s*** 转换为长整型。输入的字符串必须以 10 为基数 （十进制），且可以指示符号 （“+”或 “-”）开始。溢出结果未定义。该函数是 MPLAB C18 对 ANSI 标准函数库的扩展。

返回值： 返回转换的结果。

文件名： atol.asm

btoa

功能： 将一个 8 位有符号数转换为字符串。

头文件： stdlib.h

函数原型：
```
char * btoa( signed char value,
             char * string );
```

参数： ***value***
8 位有符号数。
string
指向保存结果的 ASCII 字符串的指针。 ***string*** 必须足够长才能保存 ASCII 表示，包括负值的符号字符和结尾的空字符。

说明： 该函数将参数 ***value*** 中的 8 位有符号数转换为 ASCII 字符串。
该函数是 MPLAB C18 对 ANSI 所需函数库的扩展。

返回值： 指向结果 ***string*** 的指针。

文件名： btoa.asm

itoa

功能： 将 16 位有符号整型转换为字符串。

头文件： stdlib.h

函数原型：
```
char * itoa( int value,
             char * string );
```

参数：

value
16 位有符号整型。

string
指向保存结果的 ASCII 字符串的指针。 ***string*** 必须足够长才能保存 ASCII 表示，包括负值的符号字符和结尾的空字符。

说明： 该函数将参数 ***value*** 中的 16 位有符号整型转换为 ASCII 字符串表示。该函数是 MPLAB C18 对 ANSI 所需函数库的扩展。

返回值： 指向结果 ***string*** 的指针。

文件名： itoa.asm

ltoa

功能： 将有符号长整型转换为字符串。

头文件： stdlib.h

函数原型：
```
char * ltoa( long value,
             char * string );
```

参数：

value
要转换的有符号长整型。

string
指向保存结果的 ASCII 字符串的指针。

说明： 该函数将参数 ***value*** 中的有符号长整型转换为 ASCII 字符串。***string*** 必须足够长才能保存 ASCII 表示，包括负值的符号字符和结尾的空字符。该函数是 MPLAB C18 对 ANSI 所需函数库的扩展。

返回值： 指向结果 ***string*** 指针。

文件名： ltoa.asm

rand

功能： 生成一个伪随机整数。

头文件： stdlib.h

函数原型：
```
int rand( void );
```

说明： 调用该函数会返回 [0,32767] 范围内的一个伪随机整数。为了有效使用该函数，必须使用 srand() 函数来设置随机数发生器的种子值。当使用相同的种子值时，该函数总会返回相同的整数序列。

返回值： 一个伪随机整数。

文件名： rand.asm

srand

功能： 为伪随机数字序列设置起始种子值。

头文件： stdlib.h

函数原型： `void srand( unsigned int seed );`

参数： ***seed***
伪随机数字序列的起始值。

说明： 该函数为 rand() 函数生成的伪随机数字序列设置起始种子值。当使用相同的种子值时， rand() 函数总会返回相同的整数序列。如果在调用 rand() 以前，未调用 srand()，则生成的数字序列与种子值为 1 调用 srand() 函数时相同。

文件名： rand.asm

tolower

功能： 将字符转换为小写字母 ASCII 字符。

头文件： ctype.h

函数原型： `char tolower( char ch );`

参数： ***ch***
要转换的字符。

说明： 假如参数是有效大写字母字符，则该函数会将 ***ch*** 转换为小写字母 ASCII 字符。

返回值： 如果参数是大写字母字符，则返回小写字符；否则返回原字符。

文件名： tolower.c

toupper

功能： 将字符转换为大写字母 ASCII 字符。

头文件： ctype.h

函数原型： `char toupper( char ch );`

参数： ***ch***
要转换的字符。

说明： 假如参数是有效的小写字母字符，则该函数会将 ***ch*** 转换为大写字母 ASCII 字符。

返回值： 如果参数是大写字母字符，则返回小写字符；否则返回原字符。

文件名： toupper.c

ultoa

功能： 将无符号长整型转换为字符串。

头文件： stdlib.h

函数原型：
```
char * ultoa( unsigned long value,
              char * string);
```

参数：
value
要转换的无符号长整型。
string
指向保存结果的 ASCII 字符串的指针。

说明： 该函数将参数 ***value*** 中的无符号长整型转换为 ASCII 字符串表示。***string***必须足够长才能保存ASCII表示，包括负值的符号字符和结尾的空字符。该函数是 MPLAB C18 对 ANSI 所需函数库的扩展。

返回值： 指向结果 ***string*** 的指针。

文件名： ultoa.asm

4.4 存储器和字符串操作函数

除非在函数描述中另有注明，这些函数和 ANSI （1989）标准 C 库函数中的同名函数是一致的。具体函数见下表：

表 4-3： 存储器和字符串操作函数

函数	描述
memchr memchrpgm	在指定的存储区中查找某个值。
memcmp memcmppgm memcmppgm2ram memcmpram2pgm	比较两个数组的内容。
memcpy memcpypgm memcpypgm2ram memcpyram2pgm	复制缓冲区。
memmove memmovepgm memmovepgm2ram memmoveram2pgm	复制缓冲区，源缓冲区和目标缓冲区可以重叠。
memset memsetpgm	用某个重复的值初始化数组。
strcat strcatpgm strcatpgm2ram strcatram2pgm	将源字符串的拷贝添加到目标字符串的末尾。
strchr strchrpgm	查找某个值在字符串中首次出现的位置。
strcmp strcmppgm strcmppgm2ram strcmpram2pgm	比较两个字符串。
strcpy strcpypgm strcpypgm2ram strcpyram2pgm	将一个字符串从数据存储器或程序存储器复制到数据存储器。

表 4-3： 存储器和字符串操作函数 （续）

strcspn strcspnpgm strcspnpgmram strcspnrampgm	计算从字符串开头不包含在另一组字符中的连续字符数。
strlen strlenpgm	确定字符串的长度。
strlwr strlwrpgm	将字符串中所有大写字符转换为小写。
strncat strncatpgm strncatpgm2ram strncatram2pgm	将源字符串中指定数目的字符添加到目标字符串的末尾。
strncmp strncmppgm strncmppgm2ram strncmpram2pgm	比较两个字符串，直到指定的字符数。
strncpy strncpypgm strncpypgm2ram strncpyram2pgm	将源字符串中的字符复制到目标字符串，直到指定的字符数。
strpbrk strpbrkpgm strpbrkpgmram strpbrkrampgm	查找一组字符中的某个字符在另一个字符串中首次出现的位置。
strrchr strrchrpgm	在字符串中查找指定字符最后一次出现的位置。
strspn strspnpgm strspnpgmram strspnrampgm	计算从字符串开头包含在另一组字符中的连续字符数。
strstr strstrpgm strstrpgmram strstrrampgm	查找某字符串在另一字符串中首次出现的位置。
strtok strtokpgm strtokpgmram strtokrampgm	将空字符插入到指定的分隔符处，把某个字符串分隔为子字符串或标记 （token）。
strupr struprpgm	将某个字符串中的所有小写字符转换为大写。

4.4.1 函数描述

memchr
memchrpgm

函数： 在指定存储区中查找某个单字节值首次出现的位置。

头文件： string.h

函数原型：
```
void * memchr( const void *mem,
               unsigned char c,
               size_t n );
rom char * memchrpgm( const rom char *mem,
                      const unsigned char c,
                      sizerom_t n );
```

参数：

mem
指向存储区的指针。

c
要查找的单字节值。

n
查找的最大字节数。

说明： 该函数在存储区 ***mem*** 中查找 ***n*** 个字节，查找 ***c*** 首次出现的位置。该函数和 ANSI 中指定函数的不同之处在于， ***c*** 定义为 unsigned char 参数，而不是 int 参数。

返回值： 如果 ***c*** 在 ***mem*** 的前 ***n*** 个字节中出现，则函数返回指向 ***mem*** 内该字符的指针；否则，返回一个空指针。

文件名：
memchr.asm
mchrpgm.asm

memcmp
memcmppgm
memcmppgm2ram
memcmpram2pgm

功能： 比较两个数组的内容。

头文件： string.h

函数原型：
```
signed char memcmp(
       const void * buf1,
       const void * buf2,
       size_t memsize );
signed char memcmppgm(
       const rom void * buf1,
       const rom void * buf2,
       sizerom_t memsize );
signed char memcmppgm2ram(
       const void * buf1,
       const rom void * buf2,
       sizeram_t memsize );
signed char memcmpram2pgm(
       const rom void * buf1,
       const void * buf2,
       sizeram_t memsize );
```

memcmp
memcmppgm
memcmppgm2ram
memcmpram2pgm （续）

参数：

buf1
指向第一个数组的指针。
buf2
指向第二个数组的指针。
memsize
数组中要比较的元素个数。

说明： 该函数将 ***buf1*** 中前 ***memsize*** 个字节与 ***buf2*** 中前 ***memsize*** 个字节进行比较，然后返回一个值，表明其中一个缓冲区是小于、等于还是大于另一个缓冲区。

返回值： 返回值为：

<0 ***buf1*** 小于 ***buf2***
==0 ***buf1*** 等于 ***buf2***
>0 ***buf1*** 大于 ***buf2***

文件名：

memcmp.asm
memcmpp2p.asm
memcmpp2r.asm
memcmpr2p.asm

memcpy
memcpypgm
memcpypgm2ram
memcpyram2pgm

功能： 将源缓冲区的内容复制到目标缓冲区。

头文件： string.h

函数原型：

```
void * memcpy(
             void * dest,
             const void * src,
             size_t memsize );
rom void * memcpypgm(
             rom void * dest,
             const rom void * src,
             sizerom_t memsize );
void * memcpypgm2ram(
             void * dest,
             const rom void * src,
             sizeram_t memsize );
rom void * memcpyram2pgm(
             rom void * dest,
             const void * src,
             sizeram_t memsize );
```

参数：

dest
指向目标数组的指针。
src
指向源数组的指针。
memsize
从 ***src*** 数组复制到 ***dest*** 数组的字节数。

说明： 该函数将 ***src*** 中前 ***memsize*** 个字节复制到数组 ***dest***。如果 ***src*** 与 ***dest*** 地址有重叠，则无法执行此操作 （未定义）。

memcpy
memcpypgm
memcpypgm2ram
memcpyram2pgm （续）

返回值： 返回 ***dest*** 的值。

文件名：
```
memcpy.asm
memcpyp2p.asm
memcpyp2r.asm
memcpyr2p.asm
```

memmove
memmovepgm
memmovepgm2ram
memmoveram2pgm

功能： 将源缓冲区中内容复制到目标缓冲区，即使两者的地址重叠。

头文件： string.h

函数原型：
```
void * memmove( void * dest,
                const void * src,
                size_t memsize );
rom void * memmovepgm(
                rom void * dest,
                const rom void * src,
                sizerom_t memsize );
void * memmovepgm2ram(
                void * dest,
                const rom void * src,
                sizeram_t memsize );
rom void * memmoveram2pgm(
                rom void * dest,
                const void * src,
                sizeram_t memsize );
```

参数：

dest
指向目标数组的指针。

src
指向源数组的指针。

memsize
从 ***src*** 复制到 ***dest*** 的字节数。

说明： 该函数将 ***src*** 中前 ***memsize*** 字节复制到 ***dest*** 数组。即使 ***src*** 与 ***dest*** 地址重叠，该函数也能正确执行。

返回值： 返回 ***dest*** 的值。

文件名：
```
memmove.asm
memmovp2p.asm
memmovp2r.asm
memmovr2p.asm
```

memset
memsetpgm

功能： 将指定字符复制到目标数组。

头文件： string.h

函数原型：
```
void * memset( void * dest,
               unsigned char value,
               size_t memsize );
rom void * memsetpgm(
               rom void * dest,
               unsigned char value,
               sizerom_t memsize );
```

参数：

dest
指向目标数组的指针。

value
要复制的字符值。

memsize
dest 中将 ***value*** 复制到的字节数。

说明： 该函数将字符 ***value*** 复制到数组 ***dest*** 的前 ***memsize*** 个字节。该函数与 ANSI 中指定函数的不同之处在于， ***value*** 定义为 unsigned char 型，而不是 int 参数。

返回值： 返回 ***dest*** 的值。

文件名：
memset.asm
memsetpgm.asm

strcat
strcatpgm
strcatpgm2ram
strcatram2pgm

功能： 将源字符串的一个拷贝添加到目标字符串的末尾。

头文件： string.h

函数原型：
```
char * strcat( char * dest,
               const char * src );
rom char * strcatpgm(
               rom char * dest,
               const rom char * src );
char * strcatpgm2ram(
               char * dest,
               const rom char * src );
rom char * strcatram2pgm(
               rom char * dest,
               const char * src );
```

参数：

dest
指向目标数组的指针。

src
指向源数组的指针。

说明： 该函数将 ***src*** 中的字符串复制到 ***dest*** 中字符串的末尾，且从 ***dest*** 中的空字符处开始添加 ***src*** 字符串。添加后，在 ***dest*** 的末尾处，将加上一个空字符。如果 ***src*** 与 ***dest*** 地址有重叠，则无法执行此操作 （未定义）。

返回值： 返回 ***dest*** 的值。

strcat
strcatpgm
strcatpgm2ram
strcatram2pgm （续）

文件名： strcat.asm
scatp2p.asm
scatp2r.asm
scatp2p.asm

strchr
strchrpgm

功能： 查找指定字符在字符串中首次出现的位置。

头文件： string.h

函数原型：
```
char * strchr( const char * str,
               unsigned char c );
rom char * strchrpgm(
               const rom char * str,
               unsigned char c );
```

参数：

str
指向要查找的字符串的指针。

c
要查找的字符。

说明： 该函数查找字符串 ***str***，找出字符 ***c*** 首次出现的位置。
此函数和 ANSI 中指定函数的不同之处在于， ***c*** 定义为 unsigned char 参数，而不是 int 型参数。

返回值： 如果 ***c*** 出现在 ***str*** 中，则返回指向 ***str*** 中此字符的指针。否则，返回一个空指针。

文件名： strchr.asm
schrpgm.asm

strcmp
strcmppgm
strcmppgm2ram
strcmpram2pgm

功能： 比较两个字符串。

头文件： string.h

函数原型：
```
signed char strcmp(
      const char * str1,
      const char * str2 );
signed char strcmppgm(
      const rom char * str1,
      const rom char * str2 );
signed char strcmppgm2ram(
      const char * str1,
      const rom char * str2 );
signed char strcmpram2pgm(
      const rom char * str1,
      const char * str2 );
```

strcmp
strcmppgm
strcmppgm2ram
strcmpram2pgm

参数： ***str1***
指向第一个字符串的指针。
str2
指向第二个字符串的指针。

说明： 该函数将 ***str1*** 中的字符串与 ***str2*** 中的字符串进行比较，且返回一个值，表明 ***str1*** 是小于、等于还是大于 ***str2***。

返回值： 返回值为：
<0 ***str1*** 小于 ***str2***
==0 ***str1*** 等于 ***str2***
>0 ***str1*** 大于 ***str2***

文件名：
strcmp.asm
scmpp2p.asm
scmpp2r.asm
scmpr2p.asm

strcpy
strcpypgm
strcpypgm2ram
strcpyram2pgm

功能： 将源字符串复制到目标字符串。

头文件： string.h

函数原型：
```
char * strcpy( char * dest,
               const char * src );
rom char * strcpypgm(
               rom char * dest,
               const rom char * src );char *
char * strcpypgm2ram(
               char * dest,
               const rom char *src );
rom char * strcpyram2pgm(
               rom char * dest,
               const char * src );
```

参数： ***dest***
指向目标字符串的指针。
src
指向源字符串的指针。

说明： 该函数将 ***src*** 中的字符串复制到 ***dest*** 中，直到终止空字符，所复制的字符包括 ***src*** 中的终止空字符。如果 ***src*** 与 ***dest*** 地址有重叠，则无法执行此操作 （操作未定义）。

返回值： 该函数返回 ***dest*** 的值。

文件名：
strcpy.asm
scpyp2p.asm
scpyp2r.asm
scpyr2p.asm

strcspn
strcspnpgm
strcspnpgmram
strcspnrampgm

功能： 计算从字符串开头不包含在另一组字符中的连续字符数。

头文件： string.h

函数原型：
```
size_t strcspn( const char * str1,
                const char * str2 );
sizerom_t strcspnpgm(
                  const rom char * str1,
                  const rom char * str2 );
sizerom_t strcspnpgmram(
                  const rom char * str1,
                  const char * str2 );
sizeram_t strcspnrampgm(
                  const char * str1,
                  const rom char * str2 );
```

参数：

str1
指向要查找字符串的指针。

str2
指向视为另一组字符的字符串的指针。

说明： 该函数确定从 ***str1*** 中首字符开始不包含在 ***str2*** 中的连续字符数。例如：

str1	***str2***	**结果**
"hello"	"aeiou"	1
"antelope"	"aeiou"	0
"antelope"	"xyz"	8

返回值： 如上例所示，该函数返回 ***str1*** 中首字符开始不包含在 ***str2*** 中的连续字符数。

文件名：
strcspn.asm
scspnpp.asm
scspnpr.asm
scspnrp.asm

strlen
strlenpgm

功能： 返回字符串的长度。

头文件： string.h

函数原型：
```
size_t strlen( const char * str );
sizerom_t strlenpgm( const rom char * str );
```

参数：

str
指向字符串的指针。

说明： 该函数确定字符串的长度，但不包括结尾的空字符。

返回值： 返回字符串的长度。

文件名：
strlen.asm
slenpgm.asm

strlwr
strlwrpgm

功能： 将字符串中所有大写字符转换为小写。

头文件： string.h

函数原型：
```
char * strlwr( char * str );
rom char * strlwrpgm( rom char * str );
```

参数： ***str***
指向字符串的指针。

说明： 该函数将 ***str*** 中所有大写字符转换为小写字符。不会影响不在大写字符范围 （A 到 Z）内的字符。

返回值： 返回 ***str*** 的值。

文件名：
strlwr.asm
slwrpgm.asm

strncat
strncatpgm
strncatpgm2ram
strncatram2pgm

功能： 将源字符串中指定数目的字符添加到目标字符串。

头文件： string.h

函数原型：
```
char * strncat( char * dest,
                const char * src,
                size_t n );
rom char * strncatpgm(
                rom char * dest,
                const rom char * src,
                sizerom_t n );
char * strncatpgm2ram(
                char * dest,
                const rom char * src,
                sizeram_t n );
rom char * strncatram2pgm(
                rom char * dest,
                const char * src,
                sizeram_t n );
```

参数： ***dest***
指向目标数组的指针。
src
指向源数组的指针。
n
要添加的字符数。

说明： 该函数将 ***src*** 字符串中的 ***n*** 个字符添加到 ***dest*** 字符串的末尾。如果在 ***n*** 个字符复制完前也复制了空字符，则这些空字符将添加到 ***dest***，直到正好添加了 ***n*** 个字符。
如果 ***src*** 和 ***dest*** 地址有重叠，则无法执行此操作 （操作未定义）。
如果没有遇到空字符，则不会添加空字符。

返回值： 返回 ***dest*** 的值。

strncat
strncatpgm
strncatpgm2ram
strncatram2pgm （续）

文件名： strncat.asm
sncatp2p.asm
sncatp2r.asm
sncatr2p.asm

strncmp
strncmppgm
strncmppgm2ram
strncmpram2pgm

功能： 比较两个字符串，直到指定的字符数。

头文件： string.h

函数原型：

```
signed char strncmp( const char * str1,
                     const char * str2,
                     size_t n );
signed char strncmppgm(
                     const rom char * str1,
                     const rom char * str2,
                     sizerom_t n );
signed char strncmppgm2ram(
                     const char * str1,
                     const rom char * str2,
                     sizeram_t n );
signed char strncmpram2pgm(
                     const rom char * str1,
                     const char * str2,
                     sizeram_t n );
```

参数：

str1
指向第一个数组的指针。
str2
指向第二个数组的指针。
n
要比较的最大字符数。

说明： 该函数将 ***str1*** 中的字符串与 ***str2*** 中的字符串进行比较，并返回一个值，表明 ***str1*** 是小于、等于还是大于 ***str2***。如果比较了 ***n*** 个字符后，没有发现差别，则该函数会返回一个值，表明两个字符串是相等的。

返回值： 根据 ***str1*** 和 ***str2*** 之间的第一个不同字符返回下列值：

<0 ***str1*** 小于 ***str2***
==0 ***str1*** 等于 ***str2***
>0 ***str1*** 大于 ***str2***

文件名： strncmp.asm
sncmpp2p.asm
sncmpp2r.asm
sncmpr2p.asm

strncpy
strncpypgm
strncpypgm2ram
strncpyram2pgm

功能： 将源字符串中的字符复制到目标字符串，直到指定的字符数。

头文件： string.h

函数原型：

```
char * strncpy( char * dest,
                const char * src,
                size_t n );
rom char * strncpypgm(
                rom char * dest,
                const rom char * src,
                sizerom_t n );
char *strncpypgm2ram(
                char * dest,
                const rom char * src,
                sizeram_t n );
rom char * strncpyram2pgm(
                rom char * dest,
                const char * src,
                sizeram_t n );
```

参数：

dest
指向目标字符串的指针。

src
指向源字符串的指针。

n
要复制的最大字符数。

说明： 该函数将 ***src*** 中的字符串复制到 ***dest***。当遇到终止空字符或者已复制 ***n*** 个字符时，复制结束。如果复制了 ***n*** 个字符而没有空字符，则 ***dest*** 将不会以空字符结束。
如果在地址重叠的对象之间复制，则无法执行此操作 （操作未定义）。

返回值： 返回 ***dest*** 的值。

文件名：

strncpy.asm
sncpyp2p.asm
sncpyp2r.asm
sncpyr2p.asm

strpbrk
strpbrkpgm
strpbrkpgmram
strpbrkrampgm

功能： 查找指定的一组字符中包含的某个字符在另一个字符串中首次出现的位置。

头文件： string.h

函数原型：

```
char * strpbrk( const char * str1,
                const char * str2 );
rom char * strpbrkpgm(
                const rom char * str1,
                const rom char * str2 );
rom char * strpbrkpgmram(
                const rom char * str1,
                const char * str2 );
char * strpbrkrampgm(
                const char * str1,
                const rom char * str2 );
```

参数：

str1
指向要搜索的字符串。
str2
指向视为一组字符的字符串的指针。

说明： 该函数将搜索 ***str1***，查找 ***str2*** 中包含的某个字符在 ***str1*** 中首次出现的位置。

返回值： 如果在 ***str1*** 中找到 ***str2*** 中的指定字符，则返回指向 ***str1*** 中那个字符的指针。如果没有在 ***str1*** 中找到 ***str2*** 中的指定字符，则返回空指针。

文件名：
strpbrk.asm
spbrkpp.asm
spbrkpr.asm
spbrkrp.asm

strrchr

功能： 查找字符串中指定字符最后一次出现的位置。

头文件： string.h

函数原型：

```
char * strrchr( const char * str,
                const char c );
```

参数：

str
指向要查找字符串的指针。
c
要查找的字符。

说明： 该函数对包括终止空字符在内的字符串 ***str*** 进行搜索，以找出字符 ***c*** 最后一次出现的位置。该函数与 ANSI 中指定函数的不同之处在于， ***c*** 定义为 unsigned char 型参数，而不是 int 型参数。

返回值： 如果 ***c*** 在 ***str*** 中出现，则返回指向 ***str*** 中该字符的指针；否则返回一个空指针。

文件名： strrchr.asm

strspn
strspnpgm
strspnpgmram
strspnrampgm

功能： 计算从字符串首字符开始、包含在另一组字符中的连续字符数。

头文件： string.h

函数原型：

```
size_t strspn( const char * str1,
               const char * str2 );
sizerom_t strspnpgm(
               const rom char * str1,
               const rom char * str2 );
sizerom_t strspnpgmram(
               const rom char * str1,
               const char * str2 );
sizeram_t strspnrampgm(
               const char * str1,
               const rom char * str2 );
```

参数：

str1
指向要查找的字符串的指针。

str2
指向作为另一组字符的字符串的指针。

说明： 该函数将确定 ***str1*** 中首字符开始、包含在 ***str2*** 中的连续字符数。例如：

str1	***str2***	**结果**
"banana"	"ab"	2
"banana"	"abn"	6
"banana"	"an"	0

返回值： 如上例所示，返回 ***str1*** 中首字符开始、包含在 ***str2*** 中的连续字符数。

文件名：

strspn.asm
sspnpp.asm
sspnpr.asm
sspnrp.asm

strstr
strstrpgm
strstrpgmram
strstrrampgm

功能： 查找某个字符串在另一字符串中首次出现的位置。

头文件： string.h

函数原型：

```
char * strstr( const char * str,
               const char * substr );
rom char * strstrpgm(
               const rom char * str,
               const rom char * substr );
rom char * strstrpgmram(
               const rom char * str,
               const char * substr );
char * strstrrampgm(
               const char * str,
               const rom char * substr );
```

参数：

str
指向要搜索的字符串。
substr
指向要查找的字符串模式。

说明： 该函数将在字符串 ***str*** 中查找字符串 ***substr***（空字符除外）首次出现的位置。

返回值： 如果找到了字符串，返回指向 ***str*** 中该字符串的指针。否则，返回一个空指针。

文件名：

```
strstr.asm
sstrpp.asm
sstrpr.asm
sstrrp.asm
```

strtok
strtokpgm
strtokpgmram
strtokrampgm

函数： 将空字符插入到指定的分隔符处，把某个字符串分隔为子字符串或者标记（token）。

头文件： string.h

函数原型：

```
char * strtok( char * str,
               const char * delim );
rom char * strtokpgm(
               rom char * str,
               const rom char * delim );
char * strtokpgmram(
               char * str,
               const rom char * delim );
rom char * strtokrampgm(
               rom char * str,
               const char * delim );
```

参数：

str
指向要搜索的字符串。
delim
指向表明标记结尾的一组字符的指针。

strtok
strtokpgm
strtokpgmram
strtokrampgm （续）

说明： 该函数通过将空字符插入到指定的字符处，把字符串分隔为子字符串。在第一次对某个字符串调用此函数时，该字符串要传递到 ***str***。此后，通过向 ***str*** 传递空值调用该函数，从上一个分隔符继续解析该字符串。
当调用 strtok 时，如果 ***str*** 是一个非空参数，则它从 ***str*** 字符串的首字符开始搜索。它略过字符串 ***delim*** 中出现的所有前导字符，然后略过所有 ***delim*** 中没有出现的字符，最后将下一字符设为空字符。
当调用 strtok 时，如果 ***str*** 是一个空参数，则它从上次调用时设置为空的字符后面一个字符开始，查找最新检测过的字符串。它略过所有 ***delim*** 中没有出现的字符，然后将下一字符设为空字符。
如果 strtok 函数在发现分隔符前遇到了字符串的结尾，则它不会修改该字符串。
在每次调用 strtok 函数时，传递给 ***delim*** 的字符组不必相同。

返回值： 如果找到分隔符，则函数返回指向 ***str*** 中找到不在字符组 ***delim*** 中的第一个字符的指针。该字符代表本次调用所创建标记的第一个字符。
如果在遇到终止空字符前没有找到分隔符，则返回一个空指针。

文件名：
strtok.asm
stokpgm.asm
stokpr.asm
stokrp.asm

strupr
struprpgm

功能： 将字符串中的所有小写字符转换为大写。

头文件： string.h

函数原型：
```
char * strupr( char * str );
rom char * struprpgm( rom char * str );
```

参数： ***str***
指向字符串的指针。

说明： 该函数将 ***str*** 中所有小写字符转换为大写字符，而所有不在小写字符范围 （a 到 z）内的字符不受影响。

返回值： 返回 ***str*** 的值。

文件名：
strupr.asm
suprpgm.asm

4.5 延时函数

延时函数执行需要若干个处理器指令周期的代码。对于基于时间的延时，必须考虑处理器的工作频率。具体函数见下表：

表 4-4： 延时函数

函数	描述
Delay1TCY	延时一个指令周期。
Delay10TCYx	延时 10 的整数倍个指令周期。
Delay100TCYx	延时 100 的整数倍个指令周期。
Delay1KTCYx	延时 1,000 的整数倍个指令周期。
Delay10KTCYx	延时 10,000 的整数倍个指令周期。

4.5.1 函数描述

Delay1TCY

功能： 延时 1 个指令周期 （Tcy）。

头文件： delays.h

函数原型：
```
void Delay1TCY( void );
```

说明： 该函数实际上是用 #define 定义的 NOP 指令。当在源代码中遇到该函数时，编译器会仅仅插入一条 NOP 指令。

文件名： delays.h

Delay10TCYx

功能： 延时 10 的整数倍个指令周期 （Tcy）。

头文件： delays.h

函数原型：
```
void Delay10TCYx( unsigned char unit );
```

参数： ***unit***

unit 的值可以为任何 8 位值。对 [1,255] 范围内的值，将会延时（***unit*** * 10）个周期。值为 0 时则延时 2,560 个周期。

说明： 该函数延时 10 的整数倍个指令周期。

文件名： d10tcyx.asm

Delay100TCYx

功能： 延时 100 的整数倍个指令周期 （Tcy）。

头文件： delays.h

函数原型：
```
void Delay100TCYx( unsigned char unit );
```

参数： ***unit***

unit 的值可以为任何 8 位值。对于 [1,255] 范围内的值，将会延时（***unit*** * 100）个周期。值为 0 时延时 25,600 个周期。

Delay100TCYx（续）

说明： 延时 100 的整数倍个指令周期。该函数使用全局分配的变量 DelayCounter1。如果在中断服务程序和一般代码中都使用了该函数，则应该在中断服务程序中保存和恢复变量。请参见 #pragma interrupt 或 #pragma interruptlow 伪指令的 save= 子句获得更多信息。要注意，其他延时函数也使用全局分配的变量 DelayCounter1。

文件名： d100tcyx.asm

Delay1KTCYx

功能： 延时 1,000 的整数倍个指令周期 （Tcy）。

头文件： delays.h

函数原型： `void Delay1KTCYx( unsigned char unit );`

参数： ***unit***

unit 的值可以为任何 8 位值。对 [1,255] 范围内的值，将会延时（***unit*** * 1000）个周期。值为 0 时延时 256,000 个周期。

说明： 延时 1,000 的整数倍个指令周期。该函数使用全局分配的变量 DelayCounter1 和 DelayCounter2。如果在中断服务程序和一般代码中都使用了该函数，则应该在中断服务程序中保存和恢复变量 DelayCounter1 和 DelayCounter2。请参见 #pragma interrupt 或 #pragma interruptlow 伪指令的 save= 子句获得更多信息。要注意，其他延时函数也使用全局分配的变量 DelayCounter1。

文件名： d1ktcyx.asm

Delay10KTCYx

功能： 延时 10,000 的整数倍个指令周期 （Tcy）。

头文件： delays.h

函数原型： `void Delay10KTCYx( unsigned char unit );`

参数： ***unit***

unit 的值可以为任何 8 位值。对 [1,255] 范围内的值，将会延时（***unit*** * 10000）个周期。值为 0 时延时 2,560,000 个周期。

说明： 延时 10,000 的整数倍个指令周期。该函数使用全局分配的变量 DelayCounter1。如果在中断服务程序和一般代码中都使用了该函数，则应该在中断服务程序中保存和恢复变量。请参见 #pragma interrupt 或 #pragma interruptlow 伪指令的 save= 子句获得更多信息。要注意，其他延时函数也使用全局分配的变量 DelayCounter1。

文件名： d10ktcyx.asm

4.6 复位函数

复位函数可用来帮助确定复位或者唤醒事件的源，并在复位之后重新配置处理器的状态。具体函数见下表：

表 4-5： 复位函数

函数	描述
isBOR	确定复位是否是由欠压复位电路引起的。
isLVD	确定复位是否是由检测到低电压条件引起的。
isMCLR	确定复位是否是由 $\overline{\text{MCLR}}$ 引脚引起的。
isPOR	检测到上电复位条件。
isWDTTO	确定复位是否是由看门狗定时器超时引起的。
isWDTWU	确定唤醒是否是由看门狗定时器引起的。
isWU	检测单片机从休眠状态唤醒是由 $\overline{\text{MCLR}}$ 引脚还是中断引起的。
StatusReset	置位 $\overline{\text{POR}}$ 位和 $\overline{\text{BOR}}$ 位。

注： 如果正在使用欠压复位 （Brown-out Reset， BOR）或看门狗定时器（Watchdog Timer， WDT），则必须在头文件 reset.h 中定义使能宏（分别为 #define BOR_ENABLED 和 #define WDT_ENABLED）并重新编译源代码。
如果器件配置为在堆栈上溢 / 下溢时复位，则必须在头文件 reset.h 中定义使能宏 （#define STVR_ENABLED）并重新编译源代码。

4.6.1 函数描述

isBOR

功能： 确定复位是否是由欠压复位电路引起的。

头文件： reset.h

函数原型：
```
char isBOR( void );
```

说明： 该函数检测单片机复位是否因欠压复位电路引起。下面的状态位表明了此条件：
$\overline{\text{POR}}$ = 1
$\overline{\text{BOR}}$ = 0

返回值： 如果复位是由欠压复位电路引起，返回 1；
否则，返回 0。

文件名： isbor.c

isLVD

功能： 确定复位是否是由检测到低电压条件引起的。

头文件： reset.h

函数原型：
```
char isLVD( void );
```

说明： 该函数检测器件电压是否要比 LVDCON 寄存器 （LVDL3:LVDL0 位）中指定的值低。

返回值： 如果复位是由正常工作时 LVD （低电压检测）引起，则返回 1；
否则，返回 0。

文件名： islvd.c

isMCLR

功能： 确定复位是否是由 $\overline{\text{MCLR}}$ 引脚引起的。

头文件： reset.h

函数原型：
```
char isMCLR( void );
```

说明： 该函数检测单片机在正常工作时是否因 $\overline{\text{MCLR}}$ 引脚引起复位。下列状态位表明了这种情况：
$\overline{\text{POR}}$ = 1
如果使能了欠压复位， $\overline{\text{BOR}}$ = 1
如果使能了看门狗定时器， $\overline{\text{TO}}$ = 1
$\overline{\text{PD}}$ = 1
如果使能了堆栈上溢 / 下溢复位，则 STKPTR 寄存器中的堆栈上溢和下溢标志位将被清零。

返回值： 如果在正常工作时因 $\overline{\text{MCLR}}$ 而复位，则返回 1；
否则，返回 0。

文件名： ismclr.c

isPOR

功能： 检测上电复位条件。

头文件： reset.h

函数原型：
```
char isPOR( void );
```

说明： 该函数检测单片机是否刚刚发生上电复位。下列状态位表明了此条件：
$\overline{\text{POR}}$ = 0
$\overline{\text{BOR}}$ = 0
$\overline{\text{TO}}$ = 1
$\overline{\text{PD}}$ = 1
正常工作时 $\overline{\text{MCLR}}$ 也会引起上电复位，执行 CLRWDT 指令时也会引起上电复位。
在调用 isPOR 后，应该调用 StatusReset 来置位 $\overline{\text{POR}}$ 位和 $\overline{\text{BOR}}$ 位。

返回值： 如果器件刚刚发生上电复位，则返回 1；
否则，返回 0。

文件名： ispor.c

isWDTTO

功能： 确定复位是否是由看门狗定时器超时引起的。

头文件： reset.h

函数原型：
```
char isWDTTO( void );
```

说明： 该函数检测单片机在正常工作时是否因 WDT 而复位。下面的状态位表明了此条件：
$\overline{\text{POR}}$ = 1
$\overline{\text{BOR}}$ = 1
$\overline{\text{TO}}$ = 0
$\overline{\text{PD}}$ = 1

返回值： 如果在正常工作期间因看门狗而复位，则返回 1；
否则，返回 0。

文件名： iswdtto.c

isWDTWU

功能： 确定唤醒是否是由看门狗定时器 （WDT）引起的。

头文件： reset.h

函数原型： char isWDTWU(void);

说明： 该函数检测单片机是否被 WDT 从休眠状态中唤醒。下面的状态位表明了此条件：
$\overline{POR}$ = 1
$\overline{BOR}$ = 1
$\overline{TO}$ = 0
$\overline{PD}$ = 0

返回值： 如果单片机被看门狗定时器唤醒，则返回 1；
否则，返回 0。

文件名： iswdtwu.c

isWU

功能： 检测是 $\overline{MCLR}$ 引脚还是中断将单片机从休眠状态唤醒。

头文件： reset.h

函数原型： char isWU(void);

说明： 该函数检测是因 $\overline{MCLR}$ 引脚还是中断将单片机从休眠状态唤醒。下面的状态位表明了此条件：
$\overline{POR}$ = 1
$\overline{BOR}$ = 1
$\overline{TO}$ = 1
$\overline{PD}$ = 0

返回值： 如果是因 $\overline{MCLR}$ 引脚或者中断唤醒单片机，则返回 1；
否则，返回 0。

文件名： iswu.c

StatusReset

功能： 置位 CPUSTA 寄存器中的 $\overline{POR}$ 位和 $\overline{BOR}$ 位。

头文件： reset.h

函数原型： void StatusReset(void);

说明： 该函数置位 CPUSTA 寄存器中的 $\overline{POR}$ 和 $\overline{BOR}$ 位。在发生上电复位后，必须在软件中置位这些位。

文件名： statrst.c

4.7 字符输出函数

字符输出函数提供了一组用于处理到外设、存储缓冲区及其他使用字符数据设备输出的主要函数。

当处理对函数 fprintf、 printf、 sprintf、 vfprintf、 vprintf 或 vsprintf 的调用时， MPLAB C18 将始终使能整型的提升来处理参数列表中的可变长度参数 （参见*《MPLAB® C18 C 编译器用户指南》*（DS51288J_CN）中的“整型的提升”小节）。这使得标准函数库可与编译器平滑接口，且输出格式一致，这正是这些函数所要求的。

4.7.1 输出流

输出基于目标流的使用。一个流可以是外设，也可以是存储缓冲区，或任何其他使用数据的设备，流表示为一个指向 FILE 类型对象的指针。 MPLAB C18 在标准函数库中定义了两种流：

_H_USER：通过用户定义的输出函数 _user_putc 输出。

_H_USART：通过函数库中的输出函数 _usart_putc 输出。

当前版本的函数库仅支持这两种输出流。这两种流始终被视为是打开的，因此不需要使用 fopen 和 fclose 等函数。

函数库定义了全局变量 stdout 和 stderr，且具有默认值 _H_USART。为将目标流更改为 _H_USER，要将这个值赋给变量。例如，将标准输出更改为使用用户定义的输出函数：

```
stdout = _H_USER;
```

表 4-6： 字符输出函数

函数	描述
fprintf	格式化字符串并输出到流。
fputs	将一个字符串输出到流。
printf	格式化字符串并输出到 stdout。
putc	将一个字符输出到流。
puts	将一个字符串输出到 stdout。
sprintf	格式化字符串并输出到数据存储缓冲区。
vfprintf	格式化字符串并输出到流，通过 stdarg 头文件提供处理格式字符串的参数。
vprintf	格式化字符串并输出到 stdout，通过 stdarg 头文件提处理格式字符串的参数。
vsprintf	格式化字符串并输出到数据存储缓冲区，通过 stdarg 头文件提供处理格式字符串的参数。
_usart_putc	输出一个字符到 USART （对于具有多个 USART 的器件，输出到 USART1）。
_user_putc	以应用定义的方式输出一个字符。

4.7.2 函数描述

fprintf

功能： 格式化字符串并输出到流。

头文件： stdio.h

函数原型： `int fprintf (FILE *f, const rom char *fmt, ...);`

说明： fprintf 函数格式化输出，并通过 putc 函数将字符传递到指定的流。一次处理格式字符串的一个字符，并按照字符在格式字符串中的顺序输出字符，格式说明符除外。格式说明符在格式字符串中以百分号指示，后跟格式规范的格式说明符，格式说明符包含如下组成部分。[1] 除了转换操作外，所有格式说明符都是可选的：

1. 标志字符 （顺序没有关系），其中标志字符为 #、 -、 +、 0 或空格之一。
2. *域宽度*，如果为星号 *，则为一个十进制常量。
3. *域精度*，是一个句点 （.），后跟可选的十进制整数或星号 *。
4. *长度说明*，为说明符 h、 H、 hh、 j、 z、 Z、 t、 T 或 l 之一。
5. *转换操作*，为 c、 b、 B、 d、 i、 n、 o、 p、 P、 s、 S、 u、 x、 X 或 % 之一。

[1] 并非所有的组成部分都对所有的转换操作有效。详细信息请参见转换操作符中的描述。

fprintf （续）

标志字符

#	将给出结果的备用形式。对于 o 转换，备用形式就像精度提高了一样，结果的第一个数字被强制为 0。对于 x 转换，将向非零的结果添加一个 0x 作为前缀。对于 b 转换，将向非零的结果添加一个 0b 作为前缀。对于 B 转换，将向非零的结果添加一个 0B 作为前缀。对于其他转换，忽略这个标志。
-	结果左对齐。如果不指定这个标志，结果将右对齐。
+	对于有符号转换，结果将始终以 + 或 - 符号开头。默认情况下，仅当结果为负时，才向结果添加符号字符。对于其他转换，忽略这个标志。
空格	对于有符号转换，如果结果非负或没有字符，将向结果添加一个空格作为前缀。如果同时指定了空格和 + 标志，将忽略空格标志。对于其他转换，忽略这个标志。
0	对于整型转换（d、i、o、u、b、B、x 和 X），向结果添加 0 作为前缀（在任何符号和 / 或基数指示符之后），以使结果满足域宽度。不进行空格填充。如果还指定了 - 标志，将忽略 0 标志。如果指定了精度，也将忽略 0 标志。对于其他转换，忽略这个标志。

域宽度

域宽度指定转换后值的最少字符数。如果转换后的值长度比域宽度短，那么将对值进行填充，以使字符数与域宽度相等。默认情况下，使用前导空格进行填充；标志字符用于修改填充字符和值的对齐。

如果域宽度是一个星号字符 *，将读取一个 int 参数来指定域宽度。如果值为负，就像指定了 - 标志一样，后跟一个正的域宽度。

域精度

对于 d、i、o、u、b、B、x 或 X 转换，域精度指定转换后的值中的最小位数；对于 s 转换，指定转换后的值中的最大字符数。

如果域宽度是一个星号字符 *，将读取一个 int 参数来指定域宽度。如果值为负，则就像没有指定精度一样。

对于 d、i、o、u、b、B、x 或 X 转换操作符，默认的精度为 1。对于所有其他转换操作符，当不指定精度时的操作描述如下。

fprintf （续）

长度说明

长度说明字符适用于整型转换说明符 d、 i、 o、 u、 b、 B、 x 或 X，以及指针转换说明符 p 和 P。对于其他转换操作符，忽略长度说明字符。

hh 对于整型转换说明符，要转换的参数为signed char或unsigned char 型参数。[2] 对于 n 转换说明符，这个说明符表示一个指向 signed char 型参数的指针。

h 对于整型转换说明符，要转换的参数为 short int 或 unsigned short int 型。对于 n 转换说明符，这个说明符表示一个指向 short int 型参数的指针。对于 MPLAB C18，单独的、没有符号说明的 int 与 short int 长度相同，这个选项没有实际的作用，提供这个选项只是为了兼容目的。对于指针转换说明符，要转换的参数为 16 位的指针。

H 对于整型转换说明符，要转换的参数为 short long int 或 unsigned short long int 型。对于 n 转换说明符，这个说明符表示一个指向 short long int 型参数的指针。对于指针转换说明符，要转换的参数为 24 位的指针。[3] 例如，当输出一个 far rom char * 时，应该使用长度说明符 H （%HS）。

j 对于整型转换说明符，要转换的参数为 intmax_t 或 uintmax_t 型参数。对于 n 转换说明符，这个说明符表示一个指向 intmax_t 型参数的指针。对于 MPLAB C18，这个说明符与 l 长度说明符作用相同。

l 对于整型转换说明符，要转换的参数为 long int 或 unsigned long int 型。对于 n 转换说明符，这个说明符表示一个指向 long int 型参数的指针。对于指针转换说明符，忽略长度说明符。

t 对于整型转换说明符，要转换的参数为 ptrdiff_t 型参数。对于 n 转换说明符，这个说明符表示一个指向与 ptrdiff_t 型参数相对应的有符号整型的指针。对于 MPLAB C18，这个说明符与 h 长度说明符作用相同。

T 对于整型转换说明符，要转换的参数为 ptrdiffrom_t 型参数。对于 n 转换说明符，这个说明符表示一个指向与 ptrdiffrom_t 型参数相对应的有符号整型的指针。对于 MPLAB C18，这个说明符与 H 长度说明符作用相同。[4]

z 对于整型转换说明符，要转换的参数为 size_t 型参数。对于 n 转换说明符，这个说明符表示一个指向与 size_t 型参数相对应的有符号整型的指针。对于 MPLAB C18，这个说明符与 h 长度说明符作用相同。

Z 对于整型转换说明符，要转换的参数为 sizerom_t 型参数。对于 n 转换说明符，这个说明符表示一个指向与 sizerom_t 型参数相对应的有符号整型的指针。对于 MPLAB C18，这个说明符与 H 长度说明符作用相同。[5]

---

[2] 注意，当传递参数时整型提升仍适用。这个说明符使得在使用参数的值之前将参数强制转换为 8 位长度。

[3] H 长度说明符是 MPLAB C18 对 ANSI C 的特定扩展。

[4] T 长度说明符是 MPLAB C18 对 ANSI C 的特定扩展。

[5] Z 长度说明符是 MPLAB C18 对 ANSI C 的特定扩展。

fprintf （续）

转换操作符

c 将 int 型参数转换为 unsigned char 值，并写该值表示的字符。

d, i 将 int 型参数格式化为有符号十进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。

o 将 unsigned int 型参数转换为无符号八进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。

u 将 unsigned int 型参数格式化为无符号十进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。

b 将 unsigned int 型参数格式化为无符号二进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。[6]

B 将 unsigned int 型参数格式化为无符号二进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。[7]

x 将 unsigned int 型参数格式化为无符号十六进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。如果参数值在十进制数字 10 至 15 之间，则用 abcdef 来表示。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。

X 将 unsigned int 型参数格式化为无符号十六进制数，精度表示要写的最小位数。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。如果参数值在十进制数字 10 至 15 之间，则用 ABCDEF 来表示。如果转换后的值位数不够，则在其前面添加零。如果转换后的值为零，且精度为零，则不写任何字符。

s 写数据存储区中字符数组中的字符，直到遇到终止 “\0” 字符（不写 “\0” 字符），或者已写的字符数目与指定精度相等。如果精度指定为大于数组的长度或未指定精度，则数组必须包含一个终止 “\0” 字符。

S 写程序存储区中字符数组中的字符，直到遇到终止 “\0” 字符（不写 “\0” 字符），或者已写的字符数目与指定精度相等。如果精度指定为大于数组的长度或未指定精度，则数组必须包含一个终止 “\0” 字符。[8] 当输出一个 far rom char * 型参数时，一定要使用 H 长度说明符 （即 %HS）。

---

[6] b 转换操作符是 MPLAB C18 对 ANSI C 的特定扩展。

[7] B 转换操作符是 MPLAB C18 对 ANSI C 的特定扩展。

[8] S 转换操作符是 MPLAB C18 对 ANSI C 的特定扩展。

fprintf （续）

p 将指向 void 型参数 （在数据或程序存储区中）的指针转换为长度相同的无符号整型，并且对这个值的处理与使用 x 转换操作符时相同。如果提供了 H 长度说明符，则指针为 一个 24 位的指针，否则为 16 位的指针。

P 将指向 void 型参数 （在数据或程序存储区中）的指针转换为长度相同的无符号整型，并且对这个值的处理与使用 x 转换操作符时相同。如果提供了 H 长度说明符，则指针为 一个 24 位的指针，否则为 16 位的指针。[9]

n 目前已写的字符数应该存储到由参数指向的地址，该参数是一个指向数据存储区中整型的指针。整型的长度由为转换指定的长度说明符决定，或者，如果不指定长度说明符，将为一个无符号说明的 16 位整型。

% 写一个字面字符 %。转换说明应该仅有 %%，不应该提供其他标志或其他说明符。

如果转换操作符非法，（如，将标志字符提供给 %% 转换说明符）。则操作未定义。

返回值： 如果发生错误， fprintf 返回 EOF ； 否则返回输出的字符数。

文件名： fprintf.c

代码示例：

```
#include <stdio.h>
void main (void)
{
  far rom char * S = ”Hello, World!”;
  int n = 0x1234;
  fprintf (_H_USART, "test output to USART\n");
  fprintf (_H_USER, "test output to application”
           “defined function\n" );
  fprintf (stdout, "hex output: %#x", n);
  fprintf (stderr, “%HS\n”, S);
}
```

[9] P 转换操作符是 MPLAB C18 对 ANSI C 的特定扩展。

fputs

功能： 将字符串输出到流。

头文件： stdio.h

函数原型：

```
int fputs (const rom char *s, FILE *f);
```

说明： fputs 通过 putc 将一个以空字符结尾的字符串输出到指定的输出流，一次输出一个字符。向输出添加换行符。不输出终止空字符。

返回值： 如果发生错误， fputs 返回 EOF ； 否则返回一个非负值。

文件名： fputs.c

printf

功能： 格式化字符串并输出到 stdout。

头文件： stdio.h

函数原型： int printf (const rom char *fmt, ...);

说明： printf 函数对输出进行格式化，并通过 putc 函数将字符传递到 stdout。对格式字符串的处理，与函数 fprintf 所述相同。

返回值： 如果发生错误， printf 返回 EOF；否则返回输出的字符数。

文件名： printf.c

代码示例：

```
#include <stdio.h>
void main (void)
{
  /* will output via stdout (_H_USART by default) */
  printf ("Hello, World!\n");
}
```

putc

功能： 将一个字符输出到流。

头文件： stdio.h

函数原型： int putc (char c, FILE *f);

说明： putc 将一个字符输出到指定的输出流。

返回值： 如果发生错误， putc 返回 EOF；否则返回输出的字符。

文件名： putc.c

puts

功能： 将一个字符串输出到 stdout。

头文件： stdio.h

函数原型： int puts (const rom char *s);

说明： puts 通过 putc 将一个以空字符结尾的字符串输出到 stdout，一次输出一个字符。向输出添加换行符。不输出终止空字符。

返回值： 如果发生错误， puts 返回 EOF；否则返回一个非负值。

文件名： puts.c

代码示例：

```
#include <stdio.h>
void main (void)
{
  puts ("test message");
}
```

sprintf

功能： 格式化字符串并输出到数据存储缓冲区。

头文件： stdio.h

函数原型：
```
int sprintf (char *buf, const rom char *fmt, ...);
```

说明： sprintf 函数对输出进行格式化，并将字符存储到目标数据存储缓冲区 buf。对格式字符串 fmt 的处理，与函数 fprintf 中的描述相同。

返回值： 如果发生错误， sprintf 返回 EOF ；否则返回输出的字符数。

文件名： sprintf.c

代码示例：
```
#include <stdio.h>
void main (void)
{
  int i = 0xA12;
  char buf[20];
  sprintf (buf, "%#010x", i);
  /* buf will contain the string "0x00000a12"
}
```

vfprintf

功能： 格式化字符串并输出到流，通过 stdarg 头文件提供处理格式字符串的参数。

头文件： stdio.h

函数原型：
```
int vfprintf (FILE *f, const rom char *fmt,
              va_list ap);
```

说明： vfprintf 函数对输出进行格式化，并通过函数 putc 将字符传递到指定的输出流 f。对格式字符串 fmt 的处理，与函数 fprintf 中的描述基本相同，但处理格式字符串时使用的参数是通过 stdarg 头文件中定义的可变长度参数宏获得的。

返回值： 如果发生错误， vfprintf 返回 EOF ；否则返回输出的字符数。

文件名： vfprintf.c

vprintf

功能： 格式化字符串并输出到 stdout，通过 stdarg 头文件提供用于处理格式字符串的参数。

头文件： stdio.h

函数原型：
```
int vprintf (const rom char *fmt, va_list ap);
```

说明： vprintf 函数对输出进行格式化，并通过 putc 函数将字符传递到 stdout。对格式字符串 fmt 的处理，与函数 fprintf 中的描述基本相同，但处理格式字符串时使用的参数是通过 stdarg 头文件中定义的可变长度参数宏获得的。

返回值： 如果发生错误， vprintf 返回 EOF ；否则返回输出的字符数。

文件名： vprintf.c

vsprintf

功能： 格式化字符串并输出到数据存储缓冲区，通过 stdarg 头文件提供用于处理格式字符串的参数。

头文件： stdio.h

函数原型：
```
int vsprintf (char *buf, const rom char *fmt,
              va_list ap);
```

说明： vsprintf 函数对输出进行格式化，并将字符存储到目标数据存储缓冲区 buf。对格式字符串 fmt 的处理，与函数 fprintf 中的描述基本相同，但处理格式字符串时使用的参数是通过 stdarg 头文件中定义的可变长度参数宏获得的。

返回值： 如果发生错误， vsprintf 返回 EOF ；否则返回输出的字符数。

文件名： vsprintf.c

_usart_putc

功能： 将一个字符输出到 USART （对于有多个 USART 的器件，输出到 USART1）。

头文件： stdio.h

函数原型：
```
int _usart_putc (char c);
```

说明： 当 _H_USART 为目标流时， _usart_putc 为由 putc 调用的输出库函数。当 USART 准备好进行输出 （TRMT 置 1）时，要输出的字符将被传送到发送寄存器 （TXREG）。
当调用 _usart_putc 时，如果 USART 未使能 （TXSTA 中的位 TXEN 清零），将使能 USART （TXEN 和 SPEN 将被置 1），并设置为最大波特率输出 （SPBRG 将赋值为零）。这种配置允许在没有显式外设配置的情况下，将字符输出库函数用在 MPLAB IDE 中支持 USART 调试输出。

返回值： _usart_putc 返回输出字符的值。

文件名： _usart_putc.c

_user_putc

功能： 以应用定义的方式输出一个字符。

头文件： stdio.h

函数原型：
```
int _user_putc (char c);
```

说明： _user_putc 是应用定义的函数。当目标流为 _H_USER 时，对于每个要输出的字符，字符输出函数都要调用这个函数。

返回值： _user_putc 返回输出字符的值。

注：

第 5 章 数学函数库

5.1 简介

本章讲述数学库函数。包括两个部分：

- 32 位浮点数数学函数库
- C 标准数学库函数

5.2 32 位浮点数数学函数库

除两点例外情况外，基本浮点运算——如加法、减法、乘法、除法以及浮点数和整数之间的转换——都符合单精度浮点数的 IEEE 754 标准。这两种例外情况将在次归一化（**第 5.2.1.2 节 “次归一化 （Subnormal）”**）和舍入 （**第 5.2.2 节 “舍入”**）部分中讲述。扩展模式和传统模式使用相同的浮点表示，且浮点运算的结果也是相同的。

1985年公布了二进制浮点运算的IEEE标准ANSI/IEEE Std 754-1985 [IEEE85]。该标准有三个重要的要求：

- 采用这个标准的所有机器对浮点数的表示要一致；
- 采用不同的舍入模式正确地对浮点运算进行舍入；
- 对例外情形 （如被零除）的处理要一致。

5.2.1 浮点数表示

C18 的浮点数表示遵循单精度浮点 IEEE 754 标准。一个浮点数由以下四个部分组成：

1. 符号
2. 尾数
3. 底数
4. 指数

这些部分组成浮点数的形式为：

$$x = \pm d_0.d_1.d_2.d_3 \cdots d_{23} \times 2^E$$

其中， ± 为符号， $d_0.d_1.d_2.d_3 \cdots d_{23}$ 为尾数， E 为以 2 为底的指数。每个 d_i 为一位（0 或 1）。指数 E 为整数，范围为 *Emin* 到 *Emax*，其中 *Emin* = -126， *Emax* = 127。

单精度浮点数使用 32 位， 1 个符号位， 8 位带偏移量的指数 e = E + 127，以及 23 位小数，这是尾数的小数部分。

不存储尾数 （d_0）的最高位。这样做是因为其值可从指数值中推导出来：如果带偏移量的指数值为 0，则 $d_0 = 0$，否则 $d_0 = 1$。采用这个约定，可允许在 23 个物理位中存储 24 位精度。

符号	8 位带偏移量的指数 *E*	23 位无符号小数 *f*
±	$e_7e_6e_5e_4e_3e_2e_1e_0$	$d_0d_1d_2d_3 \cdots d_{23}$

在 C18 实现中，不使用 $d_0 = 0$ 的数字 （见**第 5.2.1.2 节 “次归一化（Subnormal）”**）。

5.2.1.1 归一化 （Normal）

在表 5-1 中，除第一行和最后一行外，其他所有行都称为归一化数字。指数位 $e_7e_6e_5...e_0$采用带偏移量的表示；指数位存储为E+127的二进制表示，其中E为不带偏移量的指数。加到指数 E 的数字 127，称为*指数偏移量*。例如，数字 1=$(1.000...0)_2\ 2^0$ 存储为：

0	01111111	00000000000000000000000

这里，指数位为 0+127 的二进制表示，小数位为 0 的二进制表示 （1.0 的小数部分）。

归一化数字的指数位范围为 00000001 至 11111110 （十进制数字的 1 至 254），表示 E_{min} =-126 至 E_{max} = 127 范围内的实际指数。

表 5-1： IEEE-754 单精度格式

带偏移量的指数	表示的数字
$(00000000)_2 = (00)_{16} = (0)_{10}$	$\pm (0.d_1d_2d_3...d_{23})_2 \times 2^{-126}$
$(00000001)_2 = (01)_{16} = (1)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{-126}$
$(00000010)_2 = (02)_{16} = (2)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{-125}$
$(00000011)_2 = (03)_{16} = (3)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{-124}$
↓	↓
$(01111110)_2 = (7E)_{16} = (126)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{-1}$
$(01111111)_2 = (7F)_{16} = (127)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{0}$
$(10000000)_2 = (80)_{16} = (128)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{1}$
↓	↓
$(11111100)_2 = (FC)_{16} = (252)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{125}$
$(11111101)_2 = (FD)_{16} = (253)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{126}$
$(11111110)_2 = (FE)_{16} = (254)_{10}$	$\pm (1.d_1d_2d_3...d_{23})_2 \times 2^{127}$
$(11111111)_2 = (FF)_{16} = (255)_{10}$	$\pm\infty$ if $d_1...d_{23} = 0$ NaN if $d_1...d_{23} \neq 0$

可存储的最小非零正归一化数字表示为：

0	00000001	00000000000000000000000

这可以用下面的公式来表示：

$N_{min} = (1.000\ldots 0)_2 \times 2^{-126} = 2^{-126} \sim 1.2 \times 10^{38}$

C 编程人员可通过 <float.h> 中定义的明示常量（manifest constant ）FLT_MIN，求出常量 N_{min}。

最大归一化数字 （等同于最大有限数字）表示为：

0	11111110	11111111111111111111111

这可以用下面的公式来表示：

$N_{max} = (1.111...1)_2 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} \sim 2^{128} \sim 3.4 \times 10^{38}$

C 编程人员可通过 <float.h> 中定义的明示常量（manifest constant ）FLT_MAX，求出常量 N_{max}。

5.2.1.2 次归一化 （Subnormal）

可表示的最小归一化数字为 2^{-126}。 IEEE 754 标准使用一个零偏移量的指数 *e* 和一个非零小数 *f* 的组合来表示更小的数字，称为次归一化数字 （subnormal number）。次归一化数字的构成如表 5-1 的第 1 行所示。在 C18 浮点数实现中，始终将次归一化数字转换为有符号的零。

IEEE 754 使用两种不同的零表示：+ 0 和 -0。+0 由全零位表示。-0 除符号位外，其他位为全零位。

如果浮点数运算的结果比最小的归一化数字小，那么在返回结果前将结果设置为有符号零。因为在 C18 实现中，浮点运算不会产生次归一化结果，次归一化数字仅当其由立即数显式构成或由非标准浮点运算产生时才会出现。如果在浮点运算中使用次归一化值，那么在运算中使用次归一化值之前，将其自动转换为有符号零。

5.2.1.3 NaN

除了支持有符号无穷数、有符号零和有符号非零有限数外， IEEE 浮点格式还指定了错误模式的编码。这些模式不是数字，是为了记录进行的非法运算。任何这样的模式都是一个错误指示符，而不是一个浮点数，因此称为非数字 （Not a Number， NaN）。IEEE 标准定义的非法运算包括：

- 无穷数相减，如 (+∞) + (-∞)
- 零与无穷数相乘，如 (0) x (+∞)
- 零除零或无穷数除无穷数，如 (+∞)/(-∞) 或 (+∞)/(+∞)

NaN 具有一个带偏移量的指数 255，这也是用于编码无穷数的指数。当带偏移量的指数为 255 时可这样解释：如果小数为零，则编码表示一个无穷数；如果小数非零，则编码表示NaN（不是一个数字）。忽略符号位，标准不解释NaN的符号位，因此可能有 $2^{23} – 1$ 个 NaN。为响应非法运算，C18 实现返回 NaN 模式 $7FFF\ FFFF_{16}$。即，符号位为 0，指数为 255，小数位为全 1。

5.2.2 舍入

IEEE-754 标准要求对运算进行正确舍入。标准将 x 的正确舍入值（表示为 round(x)）定义如下：如果 x 是一个浮点数，那么 round(x) = x。否则，正确的舍入值取决于四种舍入模式中哪个模式有效。 C18 浮点实现使用舍入到最接近的值模式，针对 IEEE-754 标准略微做了修改。向上舍入的临界点大约为 0.502，而不是恰好 0.5。这相对于舍入到零有微小的偏离。这一修改具有显著节省代码空间和缩短执行时间的优点，而对于实际的计算几乎没有什么影响。

5.3 C 标准数学库函数

对于标准 C 函数库中的所有数学函数，如果其一个或多个参数满足下述条件，则函数将返回 NaN：

- 为 NaN。
- 超出了函数有定义实数值的值范围，例如负数的平方根。

表 5-2 列出了数学库函数。

表 5-2： 数学库函数

函数	说明
acos	计算反余弦值 （arccosine）。
asin	计算反正弦值 （arcsine）。
atan	计算反正切值 （arctangent）。
atan2	计算两个参数之比的反正切值 （arctangent）。
ceil	计算上限整数值 （最小整数）。
cos	计算余弦值。
cosh	计算双曲余弦值。
exp	计算指数 e^x。
fabs	计算绝对值。
floor	计算下限整数值 （最大整数）
fmod	计算余数。
frexp	分成小数和指数两个部分。
ieeetomchp	将 IEEE-754 格式的 32 位浮点值转换为 Microchip 32 位浮点格式。
ldexp	与指数相乘——计算 $x * 2^n$。
log	计算自然对数。
log10	计算普通对数 （以 10 为底）。
mchptoieee	将 Microchip 格式的 32 位浮点值转换为 IEEE-754 32 位浮点格式。
modf	计算模。
pow	计算指数 x^y。
sin	计算正弦值。
sinh	计算双曲正弦值。
sqrt	计算平方根。
tan	计算正切值。
tanh	计算双曲正切值。

5.3.1 函数描述

acos

功能：	计算反余弦值 （arccosine）。
头文件：	`math.h`
函数原型：	`float acos( float x );`
说明：	该函数计算参数 x 的反余弦值 （arccosine），参数的值必须在 –1 和 +1 之间。如果参数超出允许范围，将发生定义域错误，且结果为 NaN。
返回值：	返回值为反余弦值，单位为弧度，在 0 和 π 之间。
文件名：	`acos.c`

asin

功能：	计算反正弦值 （arcsine）。
头文件：	`math.h`
函数原型：	`float asin( float x );`
说明：	该函数计算参数 x 的反正弦值 （arcsine），参数的值必须在 –1 和 +1 之间。如果参数超出允许范围，将发生定义域错误，且结果为 NaN。
返回值：	返回值为反余弦值，单位为弧度，在 –π/2 和 π/2 之间。
文件名：	`asin.c`

atan

功能：	计算反正切值 （arctangent）。
头文件：	`math.h`
函数原型：	`float atan( float x );`
说明：	该函数计算参数 x 的反正切值 （arctangent）。如果 x 为 NaN，将发生定义域错误，且返回值为 NaN。
返回值：	返回值的单位为弧度，在 –π/2 和 π/2 之间。
文件名：	`atan.c`

atan2

功能：	计算两个参数之比的反正切值 （arctangent）。
头文件：	`math.h`
函数原型：	`float atan2( float y, float x );`
说明：	该函数计算 y/x 的反正切值 （arctangent）。如果 x 或 y 为 NaN，将发生定义域错误，且返回值为 NaN。如果 x 为 NaN，或 x = y = 0，或 x = y = ∞，将发生定义域错误，且返回值为 NaN。
返回值：	返回值的单位为弧度，在 –π 和 π 之间。
文件名：	`atan2.c`

ceil

功能： 计算上限整数值 （最小整数）。
头文件： `math.h`
函数原型： `float ceil ( float x );`
说明： 无。
返回值： 大于等于 x 的最小整数。
文件名： `ceil.c`

cos

功能： 计算余弦值。
头文件： `math.h`
函数原型： `float cos ( float x );`
说明： 计算 x 的余弦值 （单位为弧度）。如果参数为无穷或 NaN，将发生定义域错误。这两种情况下都返回 NaN。
返回值： 参数 x 的余弦值。
文件名： `cos.c`

cosh

功能： 计算双曲余弦值。
头文件： `math.h`
函数原型： `float cosh ( float x );`
说明： 无。
返回值： 参数 x 的双曲余弦值。
文件名： `cosh.c`

exp

功能： 计算指数值 e^x。
头文件： `math.h`
函数原型： `float exp ( float x );`
说明： 如果 x 的绝对值太大，将发生值域错误。该函数的值域限制为指数值大约在-103.2789 和 88.722283 之间。结果的最小值为 2^{-149}，最大值为 2^{127}。
返回值： 指数值 e^x。
文件名： `exp.c`

fabs

功能： 计算绝对值。
头文件： `math.h`
函数原型： `float fabs( float x );`
说明： 对于为零或无穷的浮点型参数，返回值为将参数的符号位清零后的值。
返回值： x 的绝对值。
文件名： `fabs.c`

floor

功能： 计算下限整数值 （最大整数）。

头文件： math.h

函数原型： `float floor( float x );`

说明： 无。

返回值： 小于等于 x 的最大整数。

文件名： floor.c

fmod

功能： 计算余数。

头文件： math.h

函数原型： `float fmod( float x, float y );`

说明： 无。

返回值： x/y 的余数。

文件名： fmod.c

frexp

功能： 分成小数和指数两个部分。

头文件： math.h

函数原型： `float frexp( float x, int *pexp );`

说明： 将参数 x 分成满足如下公式的两部分：
$x = frexp(x, *pexp) \times 2^{*pexp}$
对保存到 pexp 中的整数值的选取，要使得结果的小数部分在 ½ 和 1 之间。

返回值： 满足上述条件的小数结果。

文件名： frexp.c

ieeetomchp

功能： 将 IEEE-754 格式的 32 位浮点值转换为 Microchip 32 位浮点格式。

头文件： math.h

函数原型： `unsigned long ieeetomchp( float v );`

说明： 该函数按照 Microchip 格式的要求，调整浮点表示的符号位位置：

	eb	f0	f1	f2
IEEE-754 32 位	seee eeee	exxx xxxx	xxxx xxxx	xxxx xxxx
Microchip 32 位	eeee eeee	sxxx xxxx	xxxx xxxx	xxxx xxxx

s= 符号位　　e= 指数　　x= 尾数

返回值： 转换后的 32 位值。

文件名： ieeetomchp.c

ldexp

功能： 与指数相乘——计算 x * 2^n。

头文件： math.h

函数原型： `float ldexp( float x, int n );`

说明： 无。

返回值： 返回 x * 2^n 的值。

文件名： ldexp.c

log

功能： 计算自然对数。

头文件： math.h

函数原型： `float log( float x );`

说明： 如果参数不在 [0, +∞] 范围内，将发生定义域错误。

返回值： x 的自然对数。

文件名： log.c

log10

功能： 计算普通对数 （以 10 为底）。

头文件： math.h

函数原型： `float log10( float x );`

说明： 如果参数不在 [0, +∞] 范围内，将发生定义域错误。

返回值： $\log_{10}x$。

文件名： log10.c

mchptoieee

功能： 将 Microchip 格式的 32 位浮点值转换为 IEEE-754 32 位浮点格式。

头文件： math.h

函数原型： `float ieeetomchp( unsigned long v );`

说明： 该函数按照 IEEE 格式的要求，调整浮点表示的符号位位置：

	eb	f0	f1	f2
IEEE-754 32 位	seee eeee	exxx xxxx	xxxx xxxx	xxxx xxxx
Microchip 32 位	eeee eeee	sxxx xxxx	xxxx xxxx	xxxx xxxx

s= 符号位　　e= 指数　　x= 尾数

返回值： 转换后的浮点值。

文件名： mchptoieee.c

modf

功能： 计算模。

头文件： math.h

函数原型： float modf(float x, float *ipart);

说明： 该函数将参数 x 分成整数和小数两个部分。返回小数部分，并将整数部分保存到 ipart。如果参数为 NaN，那么小数部分和整数部分的结果也都将为 NaN。

返回值： x 的小数部分。

文件名： modf.c

pow

功能： 计算指数 x^y。

头文件： math.h

函数原型： float pow(float x, float y);

说明： 如果 x 为有限的负数，且 y 为有限的非整数；或者， x 为零，且 y 小于等于零，将发生定义域错误。如果 x^y 的值太大或太小而无法表示，将发生值域错误。在这种情况下，将返回符号正确的无穷数或零，并发出值域错误消息。

返回值： xy.

文件名： pow.c

sin

功能： 计算正弦值。

头文件： math.h

函数原型： float sin(float x);

说明： 计算 x 的正弦值（单位为弧度）。如果参数为无穷或 NaN，将发生定义域错误。这两种情况下都将返回 NaN。

返回值： x 的正弦值。

文件名： sin.c

sinh

功能： 计算双曲正弦值。

头文件： math.h

函数原型： float sinh(float x);

说明： 无。

返回值： x 的双曲正弦值。

文件名： sinh.c

sqrt

功能： 计算平方根。

头文件： math.h

函数原型：
```
float sqrt( float x );
```

说明： 如果参数 x 严格为负，将发生定义域错误。对于每个非负的浮点数 x，主平方根存在且可计算。

返回值： x 的平方根。

文件名： sqrt.c

tan

功能： 计算正切值。

头文件： math.h

函数原型：
```
float tan( float x );
```

说明： 计算 x 的正切值 （单位为弧度）。如果参数为无穷或 NaN，将发生定义域错误。这两种情况下都返回 NaN。

返回值： x 的正切值。

文件名： tan.c

tanh

功能： 计算双曲正切值。

头文件： math.h

函数原型：
```
float tanh( float x );
```

说明： 如果参数为 NaN，则返回值为 NaN。

返回值： x 的双曲正切值。

文件名： tanh.c

术语表

A

ANSI

美国国家标准学会

B

八进制 （Octal）

使用数字 0-7，以 8 为基数的计数体制。最右边的位表示 1 的倍数，右侧第二位表示 8 的倍数，右侧第三位表示 $8^2 = 64$ 的倍数，以此类推。

编译器 （Compiler）

将用高级语言编写的源文件翻译成机器代码的程序。

C

CPU

中央处理单元

存储类别 （Storage Class）

确定与指定对象相关联的存储区的生存时间。

存储模型 （Memory Model）

一种描述，它指定指向程序存储器的指针的位数。

存储限定符 （Storage Qualifier）

表明所定义的对象的特性 （例如 `const`）。

错误文件 （Error File）

包含 MPLAB C18 所生成的诊断信息的文件。

D

单片机 （Microcontroller）

高度集成的芯片，它包括 CPU、 RAM、某种类型的 ROM、 I/O 口和定时器。

递归函数 （Recursive）

自调用的函数 （即调用自己的函数）。

地址 （Address）

确定信息在存储器中位置的代码。

段 （Section）

位于特定存储器地址的一段应用程序。

段属性 （Section Attribute）

段的特性 （如 `access` 段）。

E

二进制 （Binary）

使用数字 0 和 1，以 2 为基数的计数体制。最右边的位表示 1 的倍数，右边第二位表示 2 的倍数，右边第三位表示 $2^2 = 4$ 的倍数，以此类推。

F

Free-standing

一种实现，它接受任何不使用复杂数据类型的严格符合程序，而且在这种实现中，对库条款中规定的属性的使用，仅限于标准头文件：<float.h>、<iso646.h>、<limits.h>、<stdarg.h>、<stdbool.h>、<stddef.h> 和 <stdint.h>。

非扩展模式 （Non-extended Mode）

在非扩展模式下，编译器不会使用扩展指令和立即数变址寻址。

G

高级语言 （High-level Language）

编写程序的语言，与汇编语言相比，它不依赖于具体的处理器。

H

汇编器 （Assembler ）

把汇编源代码翻译成机器代码的语言工具。

汇编语言 （Assembly）

以可读形式描述二进制机器代码的符号语言。

I

ICD

在线调试器

ICE

在线仿真器

IDE

集成开发环境

IEEE

电子和电气工程师协会

ISO

国际标准化组织

ISR

中断服务程序

J

绝对段 （Absolute Section）

具有链接器不能改变的固定地址的段。

K

可重定位 （Relocatable）

没有被指定到固定的存储器地址的对象。

可重入函数 （Reentrant）

可以有多个同时运行的实例的函数。在下面两种情况下可能发生函数重入：直接或间接递归调用函数；或者在由函数转入的中断处理程序中又执行此函数。

库 （Library）

可重定位目标模块的集合。

库管理器 （Librarian）

创建并管理库的程序。

快速存取存储区 （Access Memory）

PIC18 PICmicro 单片机的一些特殊通用寄存器 （General Purpose Registers，GPR），对这些寄存器的访问与存储区选择寄存器 （BSR）的设置无关。

扩展模式 （Extended Mode）

在扩展模式下，编译器将使用扩展指令 （即 ADDFSR、 ADDULNK、 CALLW、 MOVSF、MOVSS、 PUSHL、 SUBFSR 和 SUBULNK）以及立即数变址寻址。

L

链接器 （Linker）

把目标文件和库文件结合起来生成可执行代码的程序。

M

MPASM 汇编器 （MPASM Assembler）

Microchip PICmicro 系列单片机的可重定位宏汇编器。

MPLIB 目标库管理器 （MPLIB Object Librarian ）

Microchip PICmicro 系列单片机的库管理器。

MPLINK 目标链接器 （MPLINK Object Linker）

Microchip PICmicro 系列单片机的链接器。

目标代码 （Object Code）

由汇编器或编译器生成的机器代码。

目标文件 （Object File）

包含目标代码的文件。它可以直接执行或需要与其他目标代码文件 （比如库文件）链接，以生成完全可执行的程序。

N

匿名结构 （Anonymous Structure）

未命名的对象。

P

Pragma

一种伪指令，它对于特定的编译器有意义。

R

RAM

随机访问存储器

ROM

只读存储器

S

十六进制 （Hexadecimal）

使用数字 0-9 以及字母 A-F （或 A-F），以 16 为基数的计数体制。字母 A-F 表示十进制数 10 到 15。最右边的位表示 1 的倍数，右边第二位表示 16 的倍数，第三位表示 $16^2 = 256$ 的倍数，以此类推。

随机访问存储器 （Random Access Memory）

一种存储器，可以在这种存储器中以任意顺序读写信息。

T

特殊功能寄存器 （Special Function Register）

控制 I/O 处理函数、I/O 状态、定时器、其他模式或外围模块的寄存器。

条件编译 （Conditional Compilation）

只有当预处理伪指令指定的某个常数表达式为真时才编译程序段的操作。

X

向量 （Vector）

发生复位或中断时，应用程序跳转到的存储器地址。

小尾数法 （Little Endian）

将给定对象的最低有效字节存储在较低的地址。

Y

已分配段 （Assigned Section）

在链接器命令文件中已分配到目标存储区的段。

异步 （Asynchronously）

不同时发生的多个事件。通常用来指可能在处理器执行过程中的任意时刻发生的中断。

运行时模型 （Runtime Model）

编译器运行所遵循的各项前提。

Z

帧指针 （Frame Pointer）

指向堆栈中地址的指针，它用于区分堆栈中的函数参数和局部变量。

只读存储器 （Read Only Memory）

存储器硬件，它允许快速访问其中永久存储的数据，但不允许添加或更改数据。

致命错误 （Fatal Error）

引起编译立即停止的错误。不产生其他消息。

中断 （Interrupt）

发送到 CPU 的信号，它使 CPU 暂停正在运行的应用程序，把控制权转交给中断服务程序，以处理事件。执行完中断服务程序后，继续正常执行应用程序。

中断服务程序 （Interrupt Service Routine）

处理中断的函数。

中断响应时间 （Latency）

从事件发生到得到响应的时间。

中央处理单元 （Central Processing Unit）

芯片的一部分，其功能是取出要执行的指令，再对指令进行译码，然后执行指令。如果有必要，它和算术逻辑单元 （ Arithmetic Logic Unit， ALU）一起工作，来完成指令的执行。它控制程序存储器的地址总线、数据存储器的地址总线和对堆栈的访问。

字节存储顺序 （Endianness）

多字节对象的字节存储顺序

注：

索引

I

J

K

L

M

T

U

V

W

X

Y

Z

注：

全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://support.microchip.com
网址：www.microchip.com

亚特兰大 Atlanta
Alpharetta, GA
Tel: 1-770-640-0034
Fax: 1-770-640-0307

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

科科莫 Kokomo
Kokomo, IN
Tel: 1-765-864-8360
Fax: 1-765-864-8387

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

圣何塞 San Jose
Mountain View, CA
Tel: 1-650-215-1444
Fax: 1-650-961-0286

加拿大多伦多 Toronto
Mississauga, Ontario,
Canada
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

中国 - 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 - 成都
Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

中国 - 福州
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

中国 - 香港特别行政区
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 - 青岛
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 - 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 - 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 - 深圳
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 - 顺德
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

中国 - 武汉
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

中国 - 西安
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

台湾地区 - 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾地区 - 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

台湾地区 - 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

印度 India - Bangalore
Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

印度 India - New Delhi
Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

印度 India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

日本 Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

韩国 Korea - Gumi
Tel: 82-54-473-4301
Fax: 82-54-473-4302

韩国 Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 或
82-2-558-5934

马来西亚 Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

菲律宾 Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

新加坡 Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

泰国 Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

欧洲

奥地利 Austria - Wels
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

丹麦 Denmark-Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

法国 France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

德国 Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

意大利 Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

荷兰 Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

西班牙 Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

英国 UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

10/31/05