dsPIC® 语言工具库

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

所有文档均会更新，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

本文档的目的是对使用 Microchip 的 dsPIC 语言工具时用到的库进行定义和说明，这些语言工具是基于 GNU 编译器集 （GNU Compiler Collection， GCC）技术的。相关的语言工具有：

- MPLAB® ASM30 汇编器
- MPLAB C30 C 编译器
- MPLAB LINK30 链接器
- MPLAB LIB30 归档程序 / 库管理器
- 其他使用工具

本章讨论的内容包括：

- 关于本指南
- 推荐读物
- 疑难解答
- Microchip 网站
- 开发系统变更通知客户服务
- 客户支持

关于本指南

文档内容编排

本文档介绍了如何使用 GNU 语言工具为 dsPIC® 单片机应用编写代码。文档内容编排如下：

- **库的概述**—提供对库的概述。
- **DSP 函数库—**列出了 DSP 操作的库函数。
- **dsPIC 外设函数库**—列出了 dsPIC 芯片的软件和硬件外设操作的库函数与宏。
- **标准 C 数学函数库**—列出了标准 C 操作的库函数与宏。
- **MPLAB C30 内建函数—**列出了 C 编译器 MPLAB C30 的内建函数。

本指南中使用的约定

本手册采用以下文档约定：

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 ...
首字母大写	窗口	Output 窗口
	对话框	Settings 对话框
	菜单选项	选择 Enable Programmer
引用	窗口或对话框中的字段名	“Save project before build”
带右尖括号且有下划线的斜体文字	菜单路径	*File>Save*
粗体字	对话框按钮	单击 **OK**
	选项卡	单击 **Power** 选项卡
‘b*nnnn*	二进制数， *n* 是其中一位	‘b00100, ‘b10
尖括号 < > 括起的文字	键盘上的按键	按 <Enter>，<F1>
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	变量参数	*file*.o，其中 *file* 可以是任一有效文件名
0x*nnnn*	十六进制数， *n* 是其中一位	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

本文档介绍了 dsPIC 的库函数和宏。更多关于 dsPIC 语言工具和其他工具的使用信息，可以从下面的推荐读物中获得：

README 文件

关于 Microchip 工具的最新信息，请阅读软件附带的 README 文件 （ASCII 文本文件）。

Getting Started with dsPIC Language Tools （DS51316）

关于安装和使用 Microchip dsPIC 数字信号控制器 （digital signal controller， DSC）语言工具 （MPLAB ASM30、 MPLAB LINK30 和 MPLAB C30）的指南。同时提供了使用 dsPIC 仿真器和 MPLAB SIM30 的示例。

MPLAB® ASM30, MPLAB LINK30 and Utilities User's Guide （DS51317）

指导如何使用 dsPIC DSC 汇编器 MPLAB ASM30、 dsPIC DSC 链接器 MPLAB LINK30 和各种 dsPIC DSC 实用程序，包括 MPLAB LIB30 归档程序 / 库管理器。

MPLAB® C30 C 编译器用户指南 （DS51284C_CN）

使用 dsPIC DSC C 编译器的指南。 MPLAB LINK30 与这个工具配合使用。

GNU HTML 文档

在语言工具的光盘中提供了这个文档。它介绍了标准 GNU 开发工具， dsPIC DSC 的语言工具就是以此为基础的。

dsPIC30F Family Overview （DS70043）

关于 dsPIC30F 系列芯片和架构的概述。

dsPIC30F Programmer’s Reference Manual （DS70030）

dsPIC30F 编程人员的指南。包括编程模型和指令集。

Microchip 网站

Microchip 的网站 （http://www.microchip.com）提供了丰富的文档资料，可以方便地下载各个数据手册、应用笔记、教程和用户指南。所有文档都采用 Adobe Acrobat （pdf）格式。

疑难解答

本文档中没有提到的常见问题信息可查阅 README 文件。

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归档软件
- **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 的客户通知服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

欲注册，请登录 Microchip 网站 www.microchip.com，点击 “变更通知客户（Customer Change Notification）”服务并按照注册说明完成注册。

开发系统产品的分类如下：

- **编译器**——Microchip C 编译器及其他语言工具的最新信息，包括 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，以及 MPLIB™ 和 MPLAB LIB30 目标库管理器。
- **仿真器**——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**——Microchip 在线调试器 MPLAB ICD 2 的最新信息。
- **MPLAB® IDE**——用于开发系统工具的Windows®集成开发环境Microchip MPLAB IDE 的最新信息，主要针对 MPLAB IDE、 MPLAB SIM 模拟器、 MPLAB IDE 项目管理器以及一般编辑和调试功能。
- **编程器**——Microchip 编程器的最新信息，包括 MPLAB PM3 和 PRO MATE® II 器件编程器以及 PICSTART® Plus 和 PICkit® 1 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 库的概述

1.1 引言

所谓库就是将一些常用的函数集合在一起，这样用户可以方便地引用和链接。请查阅 *MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide* 获得有关创建和使用库的更多信息。

1.1.1 汇编代码的应用

可以从 Microchip 网站下载 dsPIC 语言工具库的免费版本。提供了 DSP 函数库和 dsPIC 外设函数库的目标文件和源代码。仅提供了数学库的目标文件，数学库包含标准 C 语言头文件 <math.h> 中的函数。其完整的标准 C 语言库随 MPLAB C30 C 编译器提供。

1.1.2 C 代码的应用

dsPIC 语言工具库存放在 c:\pic30_tools\lib 目录下，其中， c:\pic30_tools 是 MPLAB C30 C 编译器的安装目录。可以通过 MPLAB LINK30 将这些库直接链接到应用程序中。

1.1.3 本章内容

本章包括以下内容：

- 特定于 OMF 的库 / 启动模块
- 启动代码
- DSP 函数库
- dsPIC 外设函数库
- 标准 C 语言库 （包含数学函数）
- MPLAB C30 内建函数

1.2 特定于 OMF 的库 / 启动模块

库文件和启动模块是特定于目标模块格式（Object Module Format， OMF）的。 OMF 可以为以下格式之一：

- COFF — 默认格式。
- ELF — 用于 ELF 目标文件的调试格式是 DWARF 2.0。

选择 OMF 的方法有两种：

1. 为所有工具设定一个名为 PIC30_OMF 的环境变量。
2. 当调用工具时在命令行中选择 OMF，如：-omf=*omf* 或 -momf=*omf*。

当我们创建应用程序时， dsPIC 工具先搜索一般的库文件 （非 OMF 格式）。如果没有找到， dsPIC 工具搜索 OMF 规范的库文件，并确定使用哪个库文件。

例如，如果没有找到 libdsp.a，并且没有设定任何环境变量或命令行选项，默认情况下将会使用文件 libdsp-coff.a。

1.3 启动代码

为初始化数据存储器中的变量，链接器创建一个数据初始化模板。这个模板必须在应用程序获得控制权之前，在启动时处理。对于 C 程序，这个函数是由 libpic30-coff.a 中的启动模块（crt0.o 或 crt1.o），或 libpic30-elf.a 中的启动模块 （crt0.eo 或 crt1.eo）来完成的。汇编语言程序可以通过与需要的启动模块文件链接直接来利用这些模块。启动模块的源代码在相应的 .s 文件中。

主启动模块 （crt0）对所有变量 （持久数据段中的变量除外）进行初始化 （根据 ANSI 标准的要求，未初始化的变量设置为 0）。备用启动模块 （crt1）不会进行数据初始化。

关于启动代码的更多信息，请参阅 *MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide*。对于 C 语言应用程序，可参阅 *《MPLAB C30 C 编译器用户指南》*。

1.4 DSP 函数库

DSP 函数库 （libdsp-*omf*.a）为将在 dsPIC30F 数字信号控制器 （DSC）中执行的程序提供了一套数字信号处理函数。 DSP 函数库共支持 49 个函数。

1.5 dsPIC 外设函数库

dsPIC （软件和硬件）外设函数库为设置和控制 dsPIC30F DSC 外设提供了函数和宏。本书的相关章节中有相应的使用示例。

这些库是特定于处理器的，形式为 libp*Device*-*omf*.a，其中， *Device* 为 dsPIC 器件的编号 （如：libp30F6014-coff.a 是特定于 dsPIC30F6014 器件的）。

1.6 标准 C 语言库 （包含数学函数）

提供了一整套符合 ANSI-89 的库。标准的 C 语言库文件是 libc-*omf*.a （由业界领先的 Dinkumware 公司编写）和 libm-*omf*.a （由 Microchip 公司编写的数学函数）。

另外，一些在使用 dsPIC 器件时必须进行修改的 dsPIC 标准 C 语言库辅助函数和标准函数包含在 libpic30-*omf*.a。

一个典型的 C 应用程序必须包含全部三个库。

1.7 MPLAB C30 内建函数

对于开发人员来说， MPLAB C30 C 编译器包含的内建函数与库函数工作方式相同。

第 2 章 DSP 函数库

2.1 简介

DSP 函数库为将在 dsPIC30F 数字信号控制器 （DSC）中执行的程序提供了一系列数字信号处理函数。 DSP 函数库旨在为 C 软件开发人员提供大部分通用信号处理函数的高效实现。 DSP 函数库共支持 49 个函数。

函数库的主要目的是最大限度地缩短每个函数的执行时间。为了达到这个目的， DSP 函数库主要由优化的汇编语言编写。与 ANSI C 相比，在代码大小相同的情况下，应用 DSP 函数库可以大大地提高执行速度。另外，由于 DSP 函数库已经过严格的测试，应用 DSP 函数库将缩短应用程序的开发时间。

2.1.1 汇编代码应用程序

可以从 Microchip 网站获得该库以及相关头文件的免费版本，其中包括源代码。

2.1.2 C 代码应用程序

MPLAB C30 C 编译器安装目录 （c:\pic30_tools）包含以下与库有关的子目录：

- lib—DSP 函数库 / 归档文件。
- src\dsp—库函数的源代码以及用于重建库的批处理文件。
- support\h—DSP 函数库的头文件。

2.1.3 本章内容

本章的内容包括：

- DSP 函数库的使用
- 矢量函数
- 窗函数
- 矩阵函数
- 滤波函数
- 变换函数

2.2 DSP 函数库的使用

2.2.1 利用 DSP 函数库构建应用程序

利用 DSP 函数库构建应用程序只需要两个文件：dsp.h 和 libdsp-*omf*.a。 dsp.h 是提供函数库中所有函数的原型以及 #define 和 typedef 的头文件。libdsp-omf.a 是一个归档库文件，它包含所有库函数的目标文件。（更多特定于 OMF 的库的信息，请参阅**第 1.2 节 “特定于 OMF 的库 / 启动模块”**。）

当编译应用程序时，调用 DSP 库中函数或者使用其符号或 typedef 的所有源文件必须都 （使用 #include）引用 dsp.h。当链接应用程序时， libdsp-*omf*.a 必须作为链接器的一个输入 （使用 --library 或 -l 链接器开关），以便将应用程序使用的函数链接到应用程序。

链接器将 DSP 库的函数存放到一个名为 .libdsp 的特殊文本段中。这可以通过查看链接器生成的映射文件看到。

2.2.2 存储模型

DSP 函数库是使用 “小代码”和 “小数据”存储模型创建的，以尽可能生成最小的库。因为一些 DSP 库函数是用 C 语言编写的，并且使用编译器的浮点库，因此 MPLAB C30 链接描述文件将 .libm 和 .libdsp 文本段相邻存放。这保证了 DSP 函数库可以安全地使用 RCALL 指令来调用浮点库中所需的浮点函数。

2.2.3 DSP 库函数的调用约定

DSP 函数库中的所有目标模块都遵循 dsPIC30F DSC 的 C 语言兼容性原则，以及 *《MPLAB C30 C 编译器用户指南》* 中陈述的函数调用约定。特别地，函数可以使用前 8 个工作寄存器 （W0 ～ W7）作为函数参数。任何其他函数参数都通过堆栈传递。

W0 ～ W7 工作寄存器作为暂存存储器，函数调用后，其中的值不能保存。另一方面，如果函数使用 W8 ～ W13 中地任何一个工作寄存器，工作寄存器的内容先被保存起来，然后该寄存器，函数返回时恢复其原来的值。函数 （非 void 函数）的返回值可以保存在 W0 （也称为 WREG）中。当需要时，可以按照 *《MPLAB C30 C 编译器用户指南》* 中描述的 C 语言系统堆栈规则，使用运行时软件堆栈。根据这些规则， DSP 函数库的目标模块可以链接到 C 程序、汇编程序或者混合使用这两种语言编写的程序。

2.2.4 数据类型

DSP 函数库提供的运算利用了 DSP 指令系统和 dsPIC30F DSC 的架构特点。从这种意义上来讲，大部分运算都是小数计算。

DSP 函数库利用整型定义了小数类型：

```
#ifndef fractional
typedef int fractional;
#endif
```

fractional 数据类型用来表示带有 1 位符号位和 15 位小数位的数据。使用这种格式定义的数据通常被称为 “1.15”数据。

对于使用乘法器的函数，结果的计算要使用 40 位的累加器，并利用 “9.31” 算法。这种数据格式带有 9 位符号 / 指数位， 31 位小数位，它提供了更大的计算空间，比 “1.15” 类型提供的范围 （-1.00 ～ +1.00）大得多。当这些函数计算出结果时，它们自然会还原为 “1.15” 格式的数据类型。

使用小数运算时，要对特定函数的允许输入值集合施加某些约束。如果保证了这些约束， DSP 函数库提供的运算通常会产生精确到 14 位的数字结果。然而，一些函数会对输入数据和 / 或输出结果进行隐含的定标，这样会使得输出结果的精度下降 （与浮点运算相比）。

DSP 函数库中的某些运算需要更高级别的数据精度，要采用浮点算法运算。然而，这些运算的结果会被转换为小数值以与整个应用程序统一。唯一例外的是 `MatrixInvert` 函数，它以浮点型算法计算浮点型矩阵的逆矩阵，而提供浮点型格式的结果。

2.2.5 数据存储器的使用

DSP 函数库不对 RAM 进行分配，这个任务留给用户自己完成。如果您没有分配适当的存储空间并正确地对齐数据，当函数运行时会出现非期望结果。另外，为了最大限度地缩短执行时间， DSP 函数库并不会检查提供的函数参数 （包括指向数据存储器的指针），来判断这些参数是否合法。

大部分函数都接受数据指针作为函数的参数，函数参数包含要操作的数据，通常情况下还包含存放结果的存储单元。为方便起见， DSP 函数库中的大部分函数期望将其输入参数分配到默认的 RAM 存储空间 （X 数据空间或 Y 数据空间），而将输出存放回默认的 RAM 存储空间。然而，计算密集型的函数要求将一些操作数存放到 X 数据空间和 Y 数据空间 （或是程序存储器和 Y 数据空间）中，这样运算能够利用 dsPIC30F 架构的取双数据功能。

2.2.6 CORCON 寄存器的使用

DSP 函数库中的很多函数通过修改 CORCON 寄存器来使 dsPIC30F 器件工作在特定的运算模式。在这些函数的入口， CORCON 寄存器被压入堆栈。接着，它被修改以正确地执行期望的运算，最后 CORCON 寄存器从堆栈中弹出以恢复它原来的值。这种机制允许库的执行尽可能正确，而不破坏 CORCON 的设定值。

当修改 CORCON 寄存器时，一般情况下它会被设置为 0x00F0。这使得 dsPIC30F 器件工作在以下运算模式：

- DSP 乘法被设定为有符号的小数运算
- 使能累加器 A 和累加器 B 的饱和模式
- 饱和模式设定为 “9.31” 饱和 （超饱和）模式
- 使能数据空间写饱和 （Data Space Write Saturation）
- 禁止程序空间可视性 （Program Space Visibility， PSV）
- 使能收敛 （无偏的）舍入法

关于 CORCON 寄存器及其影响的详细信息，可参阅 *《dsPIC30F 系列参考手册》*。

2.2.7 溢出和饱和处理

DSP 函数库使用 “9.31”饱和进行大多数计算，但是必须以 “1.15”格式存储输出结果。如果在运算过程中，使用的累加器发生饱和（超过 0x7F FFFF FFFF 或低于 0x80 0000 0000），状态寄存器中的相应饱和位 （SA 或 SB）就会被置位。这个位将一直保持置位直到被清零。这样允许您在执行函数后检查 SA 或 SB 位，以确定是否应该对函数的输入数据进行定标。

同样地，如果一个计算导致累加器溢出（累加器值超过 0x00 7FFF FFFF 或低于 0xFF 8000 0000），状态寄存器中相应的溢出位（OA 或 OB）就会被置位。与 SA 和 SB 位不同， OA 和 OB 位不会一直保持置位到被清零。每次执行使用累加器的运算时都会更新这些位。如果超出这个指定的范围，这是很重要的事件，建议要通过 INTCON1 寄存器中的 OVATE 位和 OVBTE 位来使能累加器溢出陷阱。这使得一旦发生溢出就会产生算术错误陷阱，这样您就可以进行需要的操作。

2.2.8 集成到利用中断和 RTOS 的应用程序中

DSP 函数库可以很容易地集成到利用中断和 RTOS 的应用程序中，但要遵循某些规则。为了最大限度地缩短执行时间， DSP 函数库利用了 DO 循环、 REPEAT 循环，模寻址和位反转寻址等。这些都是 dsPIC30F DSC 中的有限硬件资源，当中断 DSP 库函数的执行时，后台代码必须考虑每个资源的使用。

当与 DSP 函数库集成时，必须检查下文中每个函数描述中的 “资源使用情况”，以确定使用哪些资源。如果库函数将被中断，要自己保存和恢复函数用到的所有寄存器的内容，包括 DO、 REPEAT 和特殊寻址硬件的状态。这当然也包括存储和恢复 CORCON 寄存器和状态寄存器的内容。

2.2.9 重建 DSP 函数库

提供了名为 makedsplib.bat 的批处理文件来重建 DSP 函数库。 MPLAB C30 编译器需要重建 DSP 函数库，批处理文件假定编译器安装在默认的目录 c:\pic30_tools 下。如果您的语言工具安装在不同的目录下，您必须修改批处理文件中的目录以符合语言工具的实际位置。

2.3 矢量函数

本节将讨论 DSP 函数库的小数矢量的概念，并描述执行矢量操作的各个函数。

2.3.1 小数矢量运算

小数矢量是矢量元素和相应的元素编号组成的集合，在存储器中连续分配，其中第一个元素位于最低的存储地址。存储单元的一个字 （两个字节）用于存储一个元素的值，且这个量必须解释为用 “1.15”数据格式表示的小数。

指向矢量第一个元素的指针用作访问每个矢量值的句柄。第一个元素的地址称为矢量的基地址。因为矢量的每个元素都是 16 位的，基地址必须对齐到偶数地址。

一维矢量的配置与器件的存储器存储模型相适应，所以对于一个具有 N 个元素的矢量，其第 n 个元素可以用如下方法通过矢量的基地址 BA 访问：

BA + 2(n-1)，其中 $1 \leqslant n \leqslant N$

因 dsPIC30F 器件的字节寻址能力，故系数选择为 2。

在 DSP 函数库中，实现了一元和二元的小数矢量运算。一元运算中的操作数矢量称为源矢量。在二元运算中，第一个操作数称为第一源矢量，第二个操作数称为第二源矢量。每个运算都对一个或几个源矢量元素进行计算。某些运算的结果是标量值 （同样会被表示为 “1.15”类型的小数），而另外一些运算的结果是矢量。当结果也是矢量时，也称为为目标矢量。

某些计算结果是矢量的运算允许 “原址”计算。即运算的结果被重新存放到源矢量中（或者是二元运算中的第一源矢量）。在这种情况下，目标矢量被认为会 （在物理上）替代 （第一）源矢量。如果一个运算可以 “原址”计算，那么下文中函数描述的 “说明”中会指出这一点。

对于某些二元运算，（在物理上）两个操作数可以有相同的源矢量，这表示对源矢量和其本身进行运算。如果一个给定的运算可以进行这种类型的计算，那么下文中函数描述的 “说明”中会指出这一点。

某些运算既可以对源矢量和其本身进行计算，又可以 “原址”计算。

DSP 函数库中的所有小数矢量运算都将操作数矢量的基数 （元素数目）作为参数。基于这个参数的值，作了以下假设：

a) 一个特定运算所涉及到的所有矢量总共占用的存储空间在目标器件的可用数据存储空间范围内。
b) 在二元运算情况下，两个操作数矢量的基数都*必须*符合矢量代数规则 （具体请参阅 `VectorConvolve` 和 `VectorCorrelate` 函数的 “说明”）。
c) 目标矢量*必须*足够大以接受运算的结果。

2.3.2 用户需要注意的事项

a) 不对这些函数执行边界检查。基数超出范围（包括零长度的矢量）以及二元运算中源矢量大小不一致都可能产生预想不到的结果。

b) 如果源矢量中相应元素的运算结果大于 $1-2^{-15}$ 或者小于 -1.0，矢量的加减运算会导致饱和。同样地，如果运算结果大于 $1-2^{-15}$ 或小于 -1.0，矢量的点积和幂运算也会导致饱和。

c) 建议在每个函数调用结束后检查状态寄存器（Status register， SR）。尤其是，用户可以在函数返回后检查 SA、 SB 和 SAB 标志，以判断是否发生了饱和。

d) 所有函数都设计为对分配到默认 RAM 存储空间（X 数据空间或 Y 数据空间）中的小数矢量进行运算。

e) 返回一个目标矢量的运算可以是嵌套的，例如，如果：

   a = Op1 (b, c)，且 b = Op2 (d)， c = Op3 (e, f)，则

   a = Op1 (Op2 (d), Op3 (e, f))。

2.3.3 附加说明

函数的描述将其范围限制在了这些运算的正常使用范围内。然而，由于这些函数的计算过程中不进行边界检查，因此可根据特定需要自由地解释运算及其结果。

例如，当计算函数 VectorMax 时，源矢量的长度可能会超过 numElems。这种情况下，可以使用这个函数*仅仅*在源矢量的前 numElems 个元素中查找最大值。

再如，您可能对用位于 M 与 M+numElems-1 之间的源矢量的 numElems 个元素来替换位于 N 与 N+numElems-1 之间的目标矢量的 numElems 个元素感兴趣。可以用以下方法使用 VectorCopy 函数：

```
fractional* dstV[DST_ELEMS] = {...};
fractional* srcV[SRC_ELEMS] = {...};
int n = NUM_ELEMS;
int N = N_PLACE;    /* NUM_ELEMS+N ≤ DST_ELEMS */
int M = M_PLACE;    /* NUM_ELEMS+M ≤ SRC_ELEMS */
fractional* dstVector = dstV+N;
fractional* srcVector = srcV+M;

dstVector = VectorCopy (n, dstVector, srcVector);

dstVector = VectorCopy (n, dstVector, srcVector);
```

在这个例子中， VectorZeroPad 函数可以“原址”运算，现在 dstV = srcV，其中， numElems 为要保存的源矢量开头的元素数目， numZeros 为要设置为零的矢量末尾的元素数目。

另外，可以利用不执行边界检查开发出更多的功能。

2.3.4 各个函数

下面将说明实现矢量运算的各函数。

VectorAdd

描述： VectorAdd 函数将第一源矢量中每个元素的值与第二源矢量中对应元素的值相加，并将结果存放在目标矢量中。

头文件： dsp.h

函数原型：

```
extern fractional* VectorAdd (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

参数：

numElems	源矢量中的元素数目
dstV	指向目标矢量的指针
srcV1	指向第一源矢量的指针
srcV2	指向第二源矢量的指针

返回值： 指向目标矢量基地址的指针

说明： 如果 *rcV1*[n] + *srcV2*[n] 的绝对值大于 $1-2^{-15}$，运算会导致第 n 个元素饱和。
该函数可“原址”计算。
该函数可以对源矢量和其本身进行计算。

源文件： vadd.asm

资源使用情况： 系统资源的使用：

W0..W4	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
13

周期数 （包括 C 函数调用和返回开销）：
17 + 3(*numElems*)

VectorConvolve

描述： VectorConvolve 函数计算两个源矢量的卷积，并且将结果存储在目标矢量中。结果用以下公式计算：

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k)，\text{其中 } 0 \le n < M$$

$$y(n) = \sum_{k=n-M+1}^{n} x(k)h(n-k)，\text{其中 } M \le n < N$$

$$y(n) = \sum_{k=n-M+1}^{N-1} x(k)h(n-k)，\text{其中 } N \le n < N+M-1$$

式中， x(k) 为大小为 N 的第一源矢量， h(k) 为大小为 M 的第二源矢量（M ≤ N）。

头文件： dsp.h

源文件：

```
extern fractional* VectorConvolve (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

参数：

numElems1	第一源矢量中的元素数目
numElems2	第二源矢量中的元素数目
dstV	指向目标矢量的指针
srcV1	指向第一源矢量的指针
srcV2	指向第二源矢量的指针

返回值： 指向目标矢量基地址的指针

说明： 第二源矢量中的元素数目*必须*小于或等于第一源矢量中的元素数目。目标矢量*必须*已经存在，且元素数目应为 *numElems1*+*numElems2*-1。
该函数可以对源矢量和其本身进行计算。

源文件： vcon.asm

VectorConvolve （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W10	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
58

周期数 （包括 C 函数调用和返回开销）
对于 N = *numElems1*， M = *numElems2*，

$$28 + 13M + 6\sum_{m=1}^{M} m + (N - M)(7 + 3M)$$ ，其中 M < N

$$28 + 13M + 6\sum_{m=1}^{M} m$$ ，其中 M = N

VectorCopy

描述： VectorCopy 函数将源矢量的元素复制到目标矢量 （已经存在）的开头，使得：
dstV[n] = srcV[n]， 0 ≤ n < *numElems*

头文件： dsp.h

函数原型：

```
extern fractional* VectorCopy (
    int numElems,
    fractional* dstV,
    fractional* srcV
);
```

参数：

numElems	源矢量中的元素数目
dstV	指向目标矢量的指针
srcV	指向源矢量的指针

返回值： 指向目标矢量基地址的指针。

说明： 目标矢量*必须*已经存在，并且*必须*至少有 *numElems* 个元素 （可以更多）。
函数可以 “原址” 计算。关于这种运算模式的更多信息，可参阅本章的 “附加说明” 部分。

源文件： vcopy.asm

资源使用情况： 系统资源的使用：

W0..W3	使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
6

周期数 （包括 C 函数的调用与返回开销）：
12 + *numElems*

VectorCorrelate

描述： VectorCorrelate 函数计算两个源矢量的相关，并且将结果存储在目标矢量中。其结果可以通过下式计算：

$$r(n) = \sum_{k=0}^{N-1} x(k)y(k+n)$$，其中 $0 \le n < N+M-1$

式中，x(k) 为大小为 N 的第一源矢量
y(k) 为大小为 M 的第二源矢量，其中 （M ≤ N）。

头文件： dsp.h

函数原型：

```
extern fractional* VectorCorrelate (
    int numElems1,
    int numElems2,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

参数：

numElems1	第一源矢量中的元素数目
numElems2	第二源矢量中的元素数目
dstV	指向目标矢量的指针
srcV1	指向第一源矢量的指针
srcV2	指向第二源矢量的指针

返回值： 指向目标矢量基地址的指针。

说明： 第二源矢量中的元素数目*必须*小于或等于第一源矢量中的元素数目。目标矢量*必须*已经存在，其元素数目应为 *numElems1*+*numElems2*-1。
该函数可以对源矢量和其本身进行计算。
该函数使用 VectorConvolve 函数。

源文件： vcor.asm

资源使用情况：

系统资源的使用：
W0..W7 使用，不恢复，
以及 VectorConvolve 使用的资源

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令
以及 VectorConvolve 的 DO/REPEAT 指令。

程序字 （24 位指令）：
14，
加上 VectorConvolve 的程序字。

周期数 （包括 C 函数的调用与返回开销）：
19 + floor(M/2)*3，其中 M = *numElems2*，
加上 VectorConvolve 的周期。

注： 在 VectorConvolve 的描述中，周期数包括 4 个周期的 C 函数调用开销。因此， VectorCorrelate 中因 VectorConvolve 而增加的实际周期数比单独的 VectorConvolve 所用的周期数少 4 个周期。

VectorDotProduct

描述： VectorDotProduct 函数计算第一源矢量和第一源矢量中相应元素乘积的和 （矢量点积）。

头文件： dsp.h

函数原型：
```
extern fractional VectorDotProduct (
    int numElems,
    fractional* srcV1,
    fractional* srcV2
);
```

参数：

numElems	源矢量中的元素数目
srcV1	指向第一源矢量的指针
srcV2	指向第二源矢量的指针

返回值： 乘积的和。

说明： 如果乘积之和的绝对值大于 $1-2^{-15}$，该运算将导致饱和。
该函数可以对源矢量和其本身进行计算。

函数原型： vdot.asm

资源使用情况： 系统资源的使用：

W0..W2	使用，不恢复
W4..W5	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
13

周期数 （包括 C 函数的调用与返回开销）：
17 + 3(*numElems*)

.

VectorMax

描述： VectorMax 函数在源矢量中查找值大于或等于前面任何一个矢量元素的最后一个元素。然后，它输出最大值以及最大值元素的下标。

头文件： dsp.h

函数原型：
```
extern fractional VectorMax (
    int numElems,
    fractional* srcV,
    int* maxIndex
);
```

参数：

numElems	源矢量中的元素数目
srcV	指向源矢量的指针
maxIndex	指向保存 （最后一个）最大元素的下标的存储单元的指针

返回值： 矢量中的最大值。

说明： 如果 *srcV*[i] = *srcV*[j] = maxVal，且 i < j，则 **maxIndex* = j。

源文件： vmax.asm

VectorMax （续）

资源使用情况：

系统资源的使用：
W0..W5 使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
13

周期数 （包括 C 函数的调用与返回开销）：
14
如果 *numElems* = 1
20 + 8(*numElems*-2)
如果 *srcV*[n] ≤ *srcV*[n+1]， 0 ≤ n < *numElems*-1
19 + 7(*numElems*-2)
如果 *srcV*[n] > *srcV*[n+1]， 0 ≤ n < *numElems*-1

VectorMin

描述： VectorMin 函数在源矢量中查找值小于或等于前面任何一个矢量元素的最后一个元素。然后，它输出最小值以及最小值元素的下标。

头文件： dsp.h

函数原型：

```
extern fractional VectorMin (
    int numElems,
    fractional* srcV,
    int* minIndex
);
```

参数：

numElems 源矢量中的元素数目
srcV 指向源矢量的指针
minIndex 指向保存 （最后一个）最小元素下标的存储单元的指针

返回值： 矢量中的最小值。

说明： 如果 *srcV*[i] = *srcV*[j] = minVal，且 i < j，则 **minIndex* = j。

源文件： vmin.asm

资源使用情况：

系统资源的使用：
W0..W5 使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
13

周期数 （包括 C 函数的调用与返回开销）：
14
如果 *numElems* = 1
20 + 8(*numElems*-2)
如果 *srcV*[n] ≥ *srcV*[n+1]， 0 ≤ n < *numElems*-1
19 + 7(*numElems*-2)
如果 *srcV*[n] < *srcV*[n+1]， 0 ≤ n < *numElems*-1

VectorMultiply

描述： VectorMultiply 函数将第一源矢量中每个元素的值与第二源矢量中对应元素的值相乘，并将结果存放在目标矢量相应的元素中。

头文件： dsp.h

函数原型：

```
extern fractional* VectorMultiply (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

参数：

numElems	源矢量中元素的数目
dstV	指向目标矢量的指针
srcV1	指向第一源矢量的指针
srcV2	指向第二源矢量的指针

返回值： 指向目标矢量基地址的指针。

说明： 该运算也可看作是矢量中逐个元素相乘。
该函数可以 “原址”计算。
该函数可以对源矢量和其本身进行计算。

源文件： vmul.asm

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
14

周期数 （包括 C 函数的调用和返回开销）：
17 + 4(*numElems*)

VectorNegate

描述： VectorNegate 函数对源矢量中元素的值取反 （改变其符号），并将结果存放在目标矢量中。

头文件： dsp.h

函数原型：

```
extern fractional* VectorNeg (
    int numElems,
    fractional* dstV,
    fractional* srcV
);
```

参数：

numElems	源矢量中的元素的数目
dstV	指向目标矢量的指针
srcV	指向源矢量的指针

返回值： 指向目标矢量基地址的指针

说明： 0x8000 取反的值设置为 0x7FFF。
该函数可以 “原址”计算。

源文件： vneg.asm

VectorNegate （续）

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
16

周期数 （包括 C 函数的调用和返回开销）：
19 + 4(*numElems*)

VectorPower

描述： VectorPower 函数计算源矢量元素的平方和。

头文件： dsp.h

函数原型：

```
extern fractional VectorPower (
    int numElems,
    fractional* srcV
);
```

参数：

numElems	源矢量中元素的数目
srcV	指向源矢量的指针

返回值： 矢量幂 （元素平方和）的值。

说明： 如果元素平方和的绝对值大于 $1-2^{-15}$，该运算将导致饱和。
该函数可以对源矢量和其本身进行计算。

源文件： vpow.asm

资源使用情况： 系统资源的使用：

W0..W2	使用，不恢复
W4	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
12

周期数 （包括 C 函数的调用和返回开销）：
16 + 2(*numElems*)

VectorScale

描述： VectorScale 函数用一个定标值对源矢量中所有元素的值进行换算（乘以定标值），并将结果存放在目标矢量中。

头文件： dsp.h

函数原型：

```
extern fractional* VectorScale (
    int numElems,
    fractional* dstV,
    fractional* srcV,
    fractional sclVal
);
```

参数：

numElems	源矢量中元素的数目
dstV	指向目标矢量的指针
srcV	指向源矢量的指针
sclVal	换算矢量元素的定标值

返回值： 指向目标矢量基地址的指针

说明： sclVal 必须是 “1.15 “格式的小数。
该函数可以 “原址”计算。

源文件： vscl.asm

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
14

周期数 （包括 C 函数的调用和返回开销）：
18 + 3(*numElems*)

VectorSubtract

描述： VectorSubtract 函数将第一源矢量中每个元素的值减去第二源矢量中对应元素的值，并将结果存放在目标矢量中。

头文件： dsp.h

函数原型：

```
extern fractional* VectorSubtract (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

参数：

numElems	源矢量中元素的数目
dstV	指向目标矢量的指针
srcV1	指向第一源矢量 （被减数）的指针
srcV2	指向第二源矢量 （减数）的指针

返回值： 指向目标矢量基地址的指针。

说明： 如果 *rcV1*[n] - *srcV2*[n] 结果的绝对值大于 $1\text{-}2^{-15}$，该运算会导致第 n 个元素饱和。
该函数可以 “原址”计算。
该函数可以对源矢量和其本身进行计算。

VectorSubtract （续）

源文件： vsub.asm

资源使用情况： 系统资源的使用：

W0..W4	使用，不恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
14

周期数 （包括 C 函数的调用和返回开销）
17 + 4(*numElems*)

VectorZeroPad

描述： VectorZeroPad 函数将源矢量复制到已经存在的目标矢量的开头，并对目标矢量中剩余的 numZeros 个元素填充零：
dstV[n] = *srcV*[n]， 0 ≤ n < *numElems*
dstV[n] = 0， *numElems*≤n<*numElems*+*numZeros*

头文件： dsp.h

函数原型：

```
extern fractional* VectorZeroPad (
    int numElems,
    int numZeros,
    fractional* dstV,
    fractional* srcV
);
```

参数：

numElems	源矢量中元素的数目
numZeros	目标矢量末端被填充零的元素数目
dstV	指向目标矢量的指针
srcV	指向源矢量的指针

返回值： 指向目标矢量基地址的指针。

说明： 目标矢量*必须*已经存在，并且元素数目为 *numElems*+*numZeros* 。
该函数可以“原址”计算。关于这种运算模式的更多信息，可参阅本章的“附加说明”部分。
该函数使用了 VectorCopy 函数。

源文件： vzpad.asm

VectorZeroPad （续）

资源使用情况：

系统资源的使用：
W0..W6 使用，不恢复
以及 VectorCopy 使用的资源。

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令
以及 VectorCopy 使用的 DO/REPEAT 指令

程序字：
13，
加上 VectorCopy 的程序字

周期数 （包括 C 函数的调用和返回开销）：
18 + *numZeros*
加上 VectorCopy 函数的周期数。

注：在对 VectorCopy 的描述中，周期数包括 C 函数调用开销的 3 个周期。因此， VectorZeroPad 中因 VectorCopy 而增加的实际周期数比单独的 VectorCopy 函数所用的周期数少 3 个周期。

2.4 窗函数

窗是在其定义域 （0 ≤ n < numElems）内具有特定值分布的矢量。特定的值分布取决于窗生成时的特性。

对矢量加窗可改变其值分布。在这种情况下，窗*必须*与修改后的矢量具有相同数目的元素。

对矢量加窗之前，必须先创建窗。窗初始化操作将生成窗元素的值。为获得更高的精度，这些值是以浮点型算法计算的，其结果以 “1.15” 小数格式存储。

当应用窗操作时，为了避免过多的开销，在程序的执行过程中只生成一次特定的窗，但多次使用生成的窗。因此，建议将任何初始化操作返回的窗存储在持久 （静态）矢量中。

2.4.1 用户需要注意的事项

a) 所有窗初始化函数都设计为将生成的窗矢量存放到默认的 RAM 存储空间 （X 数据空间或 Y 数据空间）中。
b) 窗函数设计为对存放在默认 RAM 存储空间 （X 数据空间或 Y 数据空间）中的矢量进行操作。
c) 建议在每个函数调用结束后，对状态寄存器 （SR）进行检查。
d) 由于窗初始化函数是用 C 语言实现的，关于周期数的最新信息，可参考最新发布的有关电子文档。

2.4.2 各个函数

下面将描述实现窗操作的各函数。

BartlettInit

描述：	BartlettInit 函数初始化一个长度为 *numElems* 的三角形（Barlett）窗。
头文件：	dsp.h

函数原型：	extern fractional* BartlettInit (
	  int *numElems*,
	  fractional* *window*
	);

参数：	*numElems* 窗中元素的数目 *window* 指向要被初始化的窗的指针
返回值：	指向被初始化窗的基地址的指针。
说明：	window 矢量*必须*已经存在，且其元素数目应为 *numElems*。
源文件：	initbart.c

BartlettInit （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W14	保存，使用，不恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

BlackmanInit

描述： BlackmanInit 函数初始化一个长度为 *numElems* 的布莱克曼（Blackman）（3 term）窗。

头文件： dsp.h

函数原型：

```
extern fractional* BlackmanInit (
    int numElems,
    fractional* window
);
```

参数：

numElems	窗中元素的数目
window	指向要被初始化的窗的指针

返回值： 指向被初始化窗的基地址的指针。

说明： window 矢量*必须*已经存在，且其元素的数目应为 *numElems*。

源文件： initblck.c

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W14	保存，使用，不恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

HammingInit

描述： HammingInit 函数初始化一个长度为 *numElems* 的海明（Hamming）窗。

头文件： dsp.h

函数原型：
```
extern fractional* HammingInit (
    int numElems,
    fractional* window
);
```

参数：
numElems 窗中元素的数目
window 指向要被初始化的窗的指针

返回值： 指向被初始化窗的基地址的指针。

说明： window 矢量*必须*已经存在，且其元素的数目应为 *numElems*。

源文件： inithamm.c

资源使用情况：
系统资源的使用：
W0..W7 使用，不恢复
W8..W14 保存，使用，不恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

HanningInit

描述： HanningInit 函数初始化一个长度为 *numElems* 的汉宁 （Hanning）窗。

头文件： dsp.h

函数原型：
```
extern fractional* HanningInit (
    int numElems,
    fractional* window
);
```

参数：
numElems 窗中元素的数目
window 指向要被初始化的窗的指针

返回值： 指向被初始化窗的基地址的指针。

说明： window 矢量*必须*已经存在，且其元素数目应为 *numElems*。

源文件： inithann.c

资源使用情况：
系统资源的使用：
W0..W7 使用，不恢复
W8..W14 保存，使用，不恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

KaiserInit

描述： KaiserInit 函数初始化一个形状由参数 betaVal 定义，长度为 *numElems* 的凯撒 （Kaiser）窗。

头文件： dsp.h

函数原型：
```
extern fractional* KaiserInit (
    int numElems,
    fractional* window,
    float betaVal
);
```

参数：

numElems	窗中元素的数目
window	指向要被初始化的窗的指针
betaVal	窗形状参数

返回值： 指向被初始化窗的基地址的指针。

说明： window 矢量*必须*已经存在，且其元素的数目应为 *numElems*。

源文件： initkais.c

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W14	保存，使用，不恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

VectorWindow

描述： VectorWindow 函数对给定的源矢量进行加窗操作，并将加窗后的矢量存放到目标矢量中。

头文件： dsp.h

函数原型：
```
extern fractional* VectorWindow (
    int numElems,
    fractional* dstV,
    fractional* srcV,
    fractional* window
);
```

参数：

numElems	源矢量元素的数目
dstV	指向目标矢量的指针
srcV	指向源矢量的指针
window	指向被初始化窗的指针

返回值： 目标矢量基地址的指针。

说明： window 矢量*必须*已经存在，且其元素的数目应为 *numElems*。
该函数可以 “原址” 计算。
该函数可以对源矢量和其本身进行计算。
该函数用到 VectorMultiply 函数。

源文件： dowindow.asm

VectorWindow （续）

资源使用情况：

系统资源的使用：
VectorMultiply 函数使用的资源

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令，
VectorMultiply 函数使用的 DO/REPEAT 指令。

程序字 （24 位指令）：
3
加上 VectorMultiply 函数的程序字。

周期数 （包括 C 函数的调用和返回开销）：
9
加上 VectorMultiply 函数的周期数。

注：在 VectorMultiply 的描述中，周期数包括 C 函数调用开销的 3 个周期。因此， VectorWindow 函数中因 VectorMultiply 函数增加的实际周期数比单独的 VectorMultiply 函数所用的周期数少 3 个周期。

2.5 矩阵函数

这一节将讲述 DSP 函数库中的小数矩阵的概念，并描述进行矩阵运算的各个函数。

2.5.1 小数矩阵运算

小数矩阵是由矩阵元素及相应的元素编号组成的集合，按照地址连续地分配在存储器中，第一个元素存放到最低的存储器地址。用存储器的一个字 （2 个字节）来存储每个元素的值，这个量必须解释为用 “1.15” 数据格式表示的小数。

指向矩阵第一个元素的指针可用作访问每个矩阵值的句柄。矩阵第一个元素的地址称为矩阵的基地址。因为每个矩阵元素都是 16 位的，基地址*必须*对齐到偶数地址。

二阶矩阵的元素是按照行主序存储在存储器中的。因此，存储到存储器中的第一个值是第一行的第一个元素，后面是第一行的其他元素。接着是第二行的元素，依此类推，直至分配好存储器的所有行。采用这种方式，对于一个基地址为 BA，共 r 行 c 列的矩阵，其第 r 行、 c 列的元素地址为：

$$BA + 2(C(r-1) + c-1)，其中\ 1 \leqslant r \leqslant R，\ 1 \leqslant c \leqslant C$$

注：使用系数 2 是由于 dsPIC30F 的字节寻址能力。

DSP 函数库实现了一元和二元的小数矩阵运算。一元运算中的操作数矩阵称为源矩阵，而二元运算中的第一个操作数称为第一源矩阵，第二个操作数称为第二源矩阵。每个运算对源矩阵的一个或几个元素进行计算。运算的结果为矩阵，称为目标矩阵。

某些小数矩阵运算允许 “原址” 计算。即运算的结果被存放回源矩阵中 （或者是二元运算中的第一源矩阵）。在这种情况下，目标矩阵被认为 （在物理上）替代了 （第一）源矩阵。如果一个运算可以 “原址” 计算，那么下文中函数描述的 “说明” 中会指出这一点。

对于某些二元运算，两个操作数 （在物理上）可以是同一个源矩阵，这表明是对源矩阵和其本身进行运算。如果一个给定的运算可以进行这种类型的计算，那么下文中函数描述的 “说明” 中会指出这一点。

某些运算可以对源矢量和其本身进行计算，且可以 “原址” 计算。

本函数库中的所有小数矩阵运算都将操作数矩阵的行数和列数作为参数。基于这些参数的值，作了以下的假设：

a) 特定操作中涉及到的所有矩阵的总的大小必须在目标器件的可用数据存储空间范围内。

b) 对于二元运算，操作数矩阵的行数和列数*必须*遵从矢量代数规则；即，对于矩阵的加减运算，两个矩阵必须具有相同的行数和列数；而对于矩阵的乘法运算，第一个操作数的列数必须与第二个操作数的行数相同。矩阵求逆运算的源矩阵必须是方阵 （行数和列数相等），并且是非奇的 （其行列式不等于零）。

c) 目标矩阵*必须*足够大以存放运算的结果。

2.5.2 用户需要注意的事项

a) 不对这些函数执行边界检查。阶数超出范围 （包括零行和 / 或零列的矩阵）以及二元运算中矩阵大小不一致都可能产生预想不到的结果。

b) 在矩阵加减运算中，如果源矩阵中相应元素的运算结果大于 $1-2^{-15}$ 或者小于 -1.0，矩阵的加减运算会导致饱和。

c) 在矩阵乘法运算中，如果相应行和列的运算结果大于 $1-2^{-15}$ 或小于 -1.0，矩阵乘法运算也会导致饱和。

d) 建议在每个函数调用结束后检查状态寄存器 （SR）。尤其是，用户可以在函数返回后检查 SA、 SB 和 SAB 标志，以判断是否发生了饱和。

e) 所有函数都设计为对分配到默认 RAM 存储空间 （X 数据空间或 Y 数据空间）中的小数矩阵进行运算。

f) 返回一个目标矩阵的运算可以是嵌套的，例如，如果：

   a = Op1 (b, c)，且 b = Op2 (d)， c = Op3 (e, f)，则

   a = Op1 (Op2 (d), Op3 (e, f))。

2.5.3 附加说明

函数的描述将其范围限制在了这些运算的正常使用范围内。然而，由于这些函数的计算过程中不进行边界检查，可以根据特定的需要自由地解释运算及其结果。

例如，当计算 `MatrixMultiply` 函数时，矩阵的阶数不需要满足第一源矩阵为 { *numRows1*, *numCos1Rows2* }，第二源矩阵为 { *numCols1Rows2*, *numCols2* }，以及目标矩阵为 { *numRows1*, *numCols2* }。事实上，只要求在计算过程中它们的大小足够大，以使指针不会超出存储空间范围。

再如，当对阶数为 { *numRows*, *numCols* } 的源矩阵进行转置运算后，目标矩阵的阶数为 { *numCols*, *numRows* }。因此，恰当地说，*仅*当源矩阵是方阵时，矩阵转置运算才能“原址”计算。然而，矩阵转置运算可以对非方阵进行 “原址”计算；需要注意的是矩阵阶数的隐含变化。

另外，可以利用不执行边界检查开发出更多的功能。

2.5.4 各个函数

下面将描述实现矩阵运算的各函数。

MatrixAdd

描述： MatrixAdd 函数将第一源矩阵中每个元素的值与第二源矩阵中相应元素的值相加，并将结果存放在目标矩阵中。

头文件： dsp.h

函数原型：

```
extern fractional* MatrixAdd (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

参数：

numRows	源矩阵的行数
numCols	源矩阵的列数
dstM	指向目标矩阵的指针
srcM1	指向第一源矩阵的指针
srcM2	指向第二源矩阵的指针

返回值： 指向目标矩阵基地址的指针。

说明： 如果 *srcM1*[r][c]+*srcM2*[r][c] 的绝对值大于 $1\text{-}2^{-15}$，该运算将导致第 (r,c) 个元素饱和。
该函数可以 “原址” 计算。
该函数可以对源矢量和其本身进行计算。

源文件： madd.asm

资源使用情况：

系统资源的使用：

W0..W4	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）
14

周期数 （包括 C 函数的调用和返回开销）：
20 + 3(*numRows***numCols*)

MatrixMultiply

描述： MatrixMultiply 函数对第一源矩阵和第二源矩阵执行矩阵乘法，并将结果存放在目标矩阵中。如下式：

$$dstM[i][j] = \sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

式中：
$0 \le i < numRows1$
$0 \le j < numCols2$
$0 \le k < numCols1Rows2$

头文件： dsp.h

函数原型

```
extern fractional* MatrixMultiply (
    int numRows1,
    int numCols1Rows2,
    int numCols2,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

参数：

numRows1	第一源矩阵的行数
numCols1Rows2	第一源矩阵的列数，其值*必须*与第二源矩阵的行数相等
numCols2	第二源矩阵的列数
dstM	指向目标矩阵的指针
srcM1	指向第一源矩阵的指针
srcM2	指向第二源矩阵的指针

返回值： 指向目标矩阵基地址的指针。

说明： 如果

$$\sum_{k}(srcM1[i][k])(srcM2i[k][j])$$

的绝对值大于 $1\text{-}2^{-15}$，该运算会导致第 (i,j) 个元素饱和。
如果第一源矩阵为方阵，则该函数可以“原址”计算，且可以对源矢量和其本身进行计算。关于这种运算模式的更多信息，可参阅本节开始部分的“附加说明”。

源文件： mmul.asm

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
35

周期数 （包括 C 函数的调用和返回开销）：
36+*numRows1**(8+*numCols2**)7+4**numCols1Rows2*))

MatrixScale

描述： MatrixScale 函数用一个定标值对源矩阵中所有元素的值进行换算（乘以定标值），并将结果存放在目标矩阵中。

头文件： dsp.h

函数原型：

```
extern fractional* MatrixScale (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM,
    fractional sclVal
);
```

参数：

numRows	源矩阵的行数
numCols	源矩阵的列数
dstM	指向目标矩阵的指针
srcM	指向源矩阵的指针
sclVal	定标值

返回值： 指向目标矩阵基地址的指针。

说明： 该函数可以“原址”计算。

源文件： mscl.asm

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
14

周期数 （包括 C 函数的调用和返回开销）：
20 + 3(*numRows***numCols*)

MatrixSubtract

描述： MatrixSubtract 函数将第一源矩阵中每个元素的值与第二源矩阵中对应的元素值相减，并将结果存放在目标矩阵中。

头文件： dsp.h

函数原型：

```
extern fractional* MatrixSubtract (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

参数：

numRows	源矩阵的行数
numCols	源矩阵的列数
dstM	指向目标矩阵的指针
srcM1	指向第一源矩阵 （被减数）的指针
srcM2	指向第二源矩阵 （减数）的指针

返回值： 指向目标矩阵基地址的指针。

MatrixSubtract （续）

说明： 如果 *srcM1*[r][c]-*srcM2*[r][c] 结果的绝对值大于 $1-2^{-15}$，该运算会导致 (r,c) 个元素饱和。
该函数可以 “原址” 计算。
该函数可以对源矢量和其本身进行计算。

源文件： msub.asm

资源使用情况： 系统资源的使用：

W0..W4	使用，不恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
15

周期数 （包括 C 函数的调用和返回开销）：
20 + 4(*numRows***numCols*)

MatrixTranspose

描述： MatrixTranspose 函数对源矩阵的行和列进行转置，并将结果存放在目标矩阵中。运算结果：
dstM[i][j] = *srcM*[j][i],
$0 \le i < numRows$， $0 \le j < numCols$。

头文件： dsp.h

函数原型：

```
extern fractional* MatrixTranspose (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM
);
```

参数：

numRows	源矩阵的行数
numCols	源矩阵的列数
dstM	指向目标矩阵的指针
srcM	指向源矩阵的指针

返回值： 指向目标矩阵基地址的指针

说明： 如果源矩阵是方阵，该函数可以 “原址” 计算。关于这种运算模式的更多信息，可参阅本节开始部分的 “附加说明”。

源文件： mtrp.asm

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
14

周期数 （包括 C 函数的调用和返回开销）：
16 + *numCols**(6 + (*numRows*-1)*3)

2.5.5 矩阵求逆

对一个非奇的小数方阵求逆的结果也是一个方阵 （阶数相同），其元素值没必要限制在离散的小数集合 { -1，……， $1-2^{-15}$ } 内。因此，对于小数矩阵，没有提供矩阵求逆运算。

然而，因为矩阵求逆是非常有用的运算， DSP 函数库中提供了基于浮点型数字表示与算法的实现。如下所述。

MatrixInvert

描述： MatrixInvert 函数对源矩阵进行求逆计算，并将结果存放在目标矩阵中。

头文件： dsp.h

函数原型：

```
extern float* MatrixInvert (
    int numRowsCols,
    float* dstM,
    float* srcM,
    float* pivotFlag,
    int* swappedRows,
    int* swappedCols
);
```

参数：

numRowCols	源 （方）矩阵的行数和列数
dstM	指向目标矩阵的指针
srcM	指向源矩阵的指针

内部使用必需的参数：

pivotFlag	指向长度为 numRowsCols 的矢量的指针
swappedRows	指向长度为 numRowsCols 的矢量的指针
swappedCols	指向长度为 numRowsCols 的矢量的指针

返回值： 指向目标矩阵基地址的指针，当源矩阵为奇矩阵时为 NULL。

说明： 尽管矢量 *pivotFlag*、 *swappedRows* 和 *swappedCols* 只在内部使用，它们需要在调用该函数之前进行分配。
如果源矩阵为奇矩阵 （行列式等于零），矩阵不能进行求逆。这种情况下，函数返回 NULL。
该函数可以 “原址”计算。

源文件： minv.asm （由 C 代码汇编的）

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8, W14	保存，使用，恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

2.6 滤波函数

本节将讲述 DSP 函数库中小数滤波器的概念，并介绍执行滤波运算的各个函数。

2.6.1 小数滤波器运算

对小数矢量 x[n] （0 ≤ n < N）表示的数据序列进行滤波运算等价于求解下面的差分方程：

$$y[n] + \sum_{p=1}^{P-1} (-a[p])(y[n-p]) = \sum_{m=0}^{M-1} (b[m])(x[n-m])$$

对于每第 n 个样本，其结果都是经滤波的数据序列 y[n]。这样，小数滤波器就可以用称为滤波器系数集的小数矢量 a[p] （0 ≤ p < P）和 b[m] （0 ≤ m < M）来表示，它们设计为在输入数据序列表示的信号中引起一些预先指定的变化。

当进行滤波运算时，了解和管理输入和输出数据序列 （x[n]，-M+1 ≤ n < 0，和 y[n]，-P+1 ≤ n < 0）的历史记录非常重要，这些历史记录提供了滤波运算的初始条件。而且，当对输入数据序列中连续的数据段重复应用滤波器时，必须记住最后一次滤波运算 （x[n]，N-M+1 ≤ n < N-1，和 y[n]，N-P+1 ≤ n < N-1）的最终状态。在下一次级滤波的计算中将要考虑到这个最终状态。要进行正确的滤波运算，必须考虑到历史记录和当前状态。

对滤波运算的历史记录和当前状态的管理通常是通过附加序列 （同样是小数矢量）来实现的，它们称为滤波器的延迟线。在应用滤波器运算之前，延迟记述滤波器的历史记录。当完成滤波运算后，延迟包含最新的滤波数据样本集合以及最新的输出数据。（注：为确保特定滤波器实现的正确运算，建议通过调用相应的初始化函数将延迟值初始化为零。）

在 DSP 函数库提供的滤波器实现中，输入数据序列称为源样本序列，而滤波运算所得的序列称为目标序列。在建立一个滤波器结构时，通常要考虑系数 (a， b) 和延迟。在所有的滤波器实现中，输入和输出数据样本可分配到默认的 RAM 存储空间 （X 数据空间或 Y 数据空间）中。滤波器系数可以存放在 X 数据存储空间或程序存储器中，而滤波器延迟的值必须 *仅*通过 Y 数据进行访问。

2.6.2 FIR 和 IIR 滤波器实现

滤波器的特性取决于它的系数值分布。特别要注意滤波器有两种重要的类型：有限冲激响应（Finite Impulse Response， FIR）滤波器，当 1 ≤ m < M 时，其 a[m] = 0；和无限冲激响应（Infinite Impulse Response， IIR）滤波器，其 a[0] ≠ 0，且当 m 在 { 1, ·····, M } 范围内， a[m] ≠ 0。考虑到滤波器运算对输入数据序列的影响， FIR 和 IIR 滤波器系列内部还有其他分类。

此外，尽管滤波运算是对上述差分方程进行求解，还提供了一些比直接求解差分方程更有效的一些实现。另外，另外一些实现设计为在小数算法的限制下执行滤波运算。

所有这些考虑使得滤波运算函数范围很广， DSP 函数库也提供了一部分这样的函数。

2.6.3 单样本滤波

DSP 函数库提供的滤波函数设计为进行块处理。每个滤波函数都接受一个名为 *numSamps* 的参数，它表示要运算的输入数据的字数 （块的大小）。如果是单样本滤波器，应将 *numSamps* 置为 1。这样将对一个输入样本进行滤波运算，而函数将计算滤波器的单个输出样本。

2.6.4 用户需要注意的事项

本库中的所有小数滤波运算都依赖于输入参数或数据结构成员的值，这些值指定要处理的样本数以及系数和延迟矢量的大小。基于这些值，作了以下假设：

a) 某个运算中涉及的所有矢量 （样本序列）的总的大小是有限的，该限制取决于当前所选用芯片的数据存储器 （RAM）的大小。

b) 目标矢量*必须*足够大以保存运算的结果。

c) 不对这些函数执行边界检查。超出范围 （包括零长度矢量）以及源矢量和系数集使用的不一致都会产生预想不到的结果。

d) 建议在每个函数调用结束后检查状态寄存器 （SR）。尤其是，用户可以在函数返回后检查 SA、 SB 和 SAB 标志，以判断是否发生了饱和。

e) 返回一个目标矢量的运算可以是嵌套的，例如，如果：

   a = Op1 (b, c)，且 b = Op2 (d)， c = Op3 (e, f)，则

   a = Op1 (Op2 (d), Op3 (e, f))。

2.6.5 各个函数

下面将讲述实现滤波运算的各函数。关于数字滤波器的更多论述，请参阅 Alan Oppenheim 和 Ronald Schafer 的 *Discrete-Time Signal Processing*， Prentice Hall，1989。关于最小均方 FIR 滤波器的实现细节，请参阅 T. Hsia 的 *Convergence Analysis of LMS and NLMS Adaptive Algorithms*，Proc. ICASSP，pp. 667-670，1983，以及 Sangil Park 和 Garth Hillman 的 *On Acoustic-Echo Cancellation Implementation with Multiple Cascadable Adaptive FIR Filter Chips*， Proc. ICASSP， 1989。

FIRStruct

结构： FIRStruct 函数描述任何 FIR 滤波器的结构。

头文件： dsp.h

描述：

```
typedef struct {
  int numCoeffs;
  fractional* coeffsBase;
  fractional* coeffsEnd;
  int coeffsPage;
  fractional* delayBase;
  fractional* delayEnd;
  fractional* delay;
} FIRStruct;
```

参数：

numCoeffs	滤波器中系数的数目 （也可为 M）
coeffsBase	滤波器系数的基地址 （也可为 h）
coeffsEnd	滤波器系数的末地址
coeffsPage	系数缓冲区的页号
delayBase	延迟缓冲区的基地址
delayEnd	延迟缓冲区的末地址
delay	延迟指针的当前值 （也可为 d）

说明：

缓冲区中的系数数目为 M。
系数 h[m]， 0 ≤ m < M 中，可存储在 X 数据空间或程序存储器中。

延迟缓冲区 d[m]， 0 ≤ m < M，*仅*可存储在 Y 数据空间中。

如果系数存储在 X 数据空间中， *coeffsBase* 指向系数分配的实际地址。如果系数存储在程序存储器中， *coeffsBase* 为包含系数的程序页边界与系数在该页中分配的地址之间的偏移量。这个偏移量值可以通过使用行内汇编操作符 psvoffset() 计算出来。

coeffsEnd 为滤波器系数缓冲区最后一个字节在 X 数据空间中的地址（如果是程序存储器，则为偏移量）。

如果系数是存放在 X 数据空间数据存储器中， *coeffsPage* 必须被置为 0xFF00 （定义值 COEFFS_IN_DATA）。如果系数是存放在程序存储器中，则为包含系数的程序页号。后面这个值可通过使用行内汇编操作符 psvpage() 计算出来。

delayBase 指向延迟缓冲区分配的实际地址。

delayEnd 为滤波器延迟缓冲区的最后一个字节的地址。

FIRStruct （续）

当系数和延迟缓冲区作为循环递增模缓冲区时， *coeffsBase* 和 *delayBase* *必须*对齐到两个地址的“0”幂（*oeffsEnd* 和 *delayEnd* 都是奇数地址）。这些缓冲区是否作为循环增递增模缓冲区，会在每个 FIR 滤波器函数描述中的 “说明”部分中指出。

当系数和延迟缓冲区不作为循环递增模缓冲区时， *coeffsBase* 和 *delayBase* 不需要对齐到两个地址的“0”幂，而在特定的 FIR 滤波器函数实现中， *coeffsEnd* 和 *delayEnd* 的值将被忽略。

FIR

描述： FIR 函数对源样本序列应用 FIR 滤波器，将结果存放在目标样本序列中，并更新延迟值。

头文件： dsp.h

函数原型：

```
extern fractional* FIR (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

参数：

numSamps	滤波器的输入样本数 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	FIRStruct 滤波器结构的指针

返回值： 指向目标样本基地址的指针。

说明： 滤波器中系数的数目为 M。
系数 h[m] （定义在 0 ≤ m < M 范围内），作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M 范围内），作为循环递增模缓冲区。
源样本 x[n] 定义在 0 ≤ n < N 范围内。
目标样本 y[n] 定义在 0 ≤ n < N 范围内。
（参见 FIRStruct、 FIRStructInit 和 FIRDelayIni。）

源文件： fir.asm

FIR （续）

资源使用情况： 系统资源的使用：

W0..W6	使用，不恢复
W8, W10	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复
MODCON	保存，使用，恢复
XMODSTRT	保存，使用，恢复
XMODEND	保存，使用，恢复
YMODSTRT	保存，使用，恢复
PSVPAG	保存，使用，恢复（仅当系数在程序存储器中时）

DO 和 REPEAT 指令的使用：
一级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
55

周期数 （包括 C 函数的调用和返回开销）：
53 + N(4+M)，
或当系数在程序存储器中，为 56 + N(8+M)。

FIRDecimate

描述： FIRDecimate 函数以 1/R 的比率从源样本序列中抽取样本，或者以系数 R 对信号降采样。实际上，
y[n] = x[Rn]。
为了减小混叠的影响，要先对源样本进行滤波，然后降采样。抽取结果存放在目标样本序列中，并更新延迟的值。

头文件： dsp.h

函数原型：

```
extern fractional* FIRDecimate (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    int rate
);
```

参数：

numSamps	*输出*样本的数目 （也可为 N， N = Rp， p 为整数）
dstSamp	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 FIRStruct 滤波器结构的指针
rate	抽取比率 （降采样系数，也可为 R）

返回值： 指向目标样本基地址的指针。

说明： 滤波器中系数的数目为 M，而 M 是 R 的整数倍。
系数 h[m] （定义在 0 ≤ m < M 范围内），不作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M 范围内），不作为循环递增模缓冲区。
源样本 x[n]，定义在 0 ≤ n < NR 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
（参见 FIRStruct、 FIRStructInit 和 FIRDelayInit。）

源文件： firdecim.asm

FIRDecimate （续）

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W12	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复 （仅当系数在程序存储器中时）

DO 和 REPEAT 指令的使用：
一级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
48

周期数 （包括 C 函数的调用和返回开销）：
45 + N(10 + 2M)，
或者，当系数在程序存储器中，为 48 + N(13 + 2M)。

FIRDelayInit

描述： FIRDelayInit 函数将 FIRStruct 滤波器结构中的延迟值初始化为零。

头文件： dsp.h

函数原型：

```
extern void FIRDelayInit (
    FIRStruct* filter
);
```

参数： *filter* 指向 FIRStruct 滤波器结构的指针。

说明： 参阅上面关于 FIRStruct 结构的描述。
注： FIR 插值器的延迟由函数 FIRInterpDelayInit 初始化。

源文件： firdelay.asm

资源使用情况：

系统资源的使用：

W0..W2	使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
7

周期数 （包括 C 函数的调用和返回开销）：
11 + M

FIRInterpolate

描述： FIRInterpolate 函数以 1:R 比率在源样本序列中插入样本，或者以系数 R 对信号进行过采样。实际上，
y[n] = x[n/R]。
为了减小信号混叠的影响，要先对源样本进行过采样，然后滤波。结果存放在目标样本序列中，并更新延迟的值。

头文件： dsp.h

函数原型：

```
extern fractional* FIRInterpolate (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    int rate
);
```

参数：

numSamps	输入样本的数目 （也可为 N， N = Rp， p 为整数）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	FIRStruct 指向滤波器结构的指针
rate	插值的比率 （过采样系数，也可为 R）

返回值： 指向目标样本基地址的指针。

说明： 滤波器中系数的数目为 M，而 M 是 R 的整数倍。
系数 h[m] （定义在 0 ≤ m < M 范围内），不作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M/R 范围内），不作为循环递增模缓冲区。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < NR 范围内。
（参见 FIRStruct、 FIRStructInit 和 FIRInterpDelayInit。）

源文件： firinter.asm

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复 （仅当系数在程序存储器中时）

DO 和 REPEAT 指令的使用：
两级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
63

周期数 （包括 C 函数的调用和返回开销）：
45 + 6(M/R) + N(14 + M/R + 3M + 5R)，
或者，当系数在程序存储器中，为
48 + 6(M/R) + N(14 + M/R + 4M + 5R)

FIRInterpDelayInit

描述： FIRInterpDelayInit函数将FIRStruct滤波器结构中的延迟值初始化为零，优化供 FIR 插值滤波器使用。

头文件： dsp.h

函数原型：

```
extern void FIRDelayInit (
   FIRStruct* filter,
   int rate
);
```

参数：

filter	指向 FIRStruct 滤波器结构的指针
rate	插值的比率 （过采样系数，也可为 R）

说明： 延迟 d[m]，定义在 0 ≤ m < M/R 范围内，其中 M 为插值器中滤波器系数的数目。
参阅上面关于 FIRStruct 结构的描述。

源文件： firintdl.asm

资源使用情况： 系统资源的使用：
W0..W4 使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
13

周期数 （包括 C 函数的调用和返回开销）：
10 + 7M/R

FIRLattice

描述： FIRLattice 函数使用格型 （Lattice）结构实现对源样本序列应用 FIR 滤波器，接着将结果存放在目标样本序列中，并更新延迟值。

头文件： dsp.h

函数原型：

```
extern fractional* FIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

参数：

numSamps	滤波器输入样本的数目 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 FIRStruct 滤波器结构的指针

返回值： 指向目标样本基地址的指针。

说明： 滤波器中的系数数目为 M。
格型滤波器系数 k[m] （定义在 0 ≤ m < M 范围内），不作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M 范围内），不作为循环递增模缓冲区。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
（参见 FIRStruct、 FIRStructInit 和 FIRDelayInit。）

源文件： firlatt.asm

FIRLattice （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W12	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复 （仅当系数在程序存储器中时）

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
50

周期数 （包括 C 函数的调用和返回开销）：
41 + N(4 + 7M)，
当系数在程序存储器中，为 44 + N(4 + 8M）。

FIRLMS

描述： FIRLMS 函数对源样本序列应用自适应 FIR 滤波器，将结果存放在目标样本序列中，并更新延迟值。
针对每个样本，使用最小均方算法，对参考样本进行处理，同时滤波器的系数也被更新。

头文件： dsp.h

函数原型：

```
extern fractional* FIRLMS (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    fractional* refSamps,
    fractional muVal
);
```

参数：

numSamps	输入样本的数目 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 FIRStruct 滤波器结构的指针
refSamps	指向参考样本的指针 （也可为 r）
muVal	自适应系数 （也可为 mu）

返回值： 指向目标样本基地址的指针。

FIRLMS （续）

说明：

滤波器中的系数数目为 M。
系数 h[m] （定义在 0 ≤ m < M 范围内），作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M-1 范围内），作为循环递增模缓冲区。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
参考样本 r[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
修改：

h_m[n] = h_m[n-1] + mu*（r[n] - y[n]） *x[n-m]，

式中， 0 ≤ n < N, 0 ≤ m < M。

当（r[n] - y[n]）的绝对值大于或等于 1，该运算会导致饱和。
滤波器系数*不能*存放在程序存储器中，因为在这种情况下，其值不能被修改。如果检测到滤波器系数存放在程序存储器中，函数会返回 NULL。
（参见 FIRStruct、FIRStructInit 和 FIRDelayInit。）

源文件：

firlms.asm

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W12	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复
MODCON	保存，使用，恢复
XMODSTRT	保存，使用，恢复
XMODEND	保存，使用，恢复
YMODSTRT	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
76

周期数 （包括 C 函数的调用和返回开销）：
61 + N(13 + 5M)

FIRLMSNorm

描述：

FIRLMSNorm 函数对源样本序列应用自适应 FIR 滤波器，将结果存放在目标样本序列中，并更新延迟值。
针对每个样本，使用最小均方算法，对参考样本进行处理，同时滤波器的系数也被更新。

头文件：

dsp.h

函数原型：

```
extern fractional* FIRLMSNorm (
    int numSamps,
    fractional* dstSamps,
    fractional* srcSamps,
    FIRStruct* filter,
    fractional* refSamps,
    fractional muVal,
    fractional* energyEstimate
);
```

FIRLMSNorm （续）

参数：

numSamps	输入样本的数目 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 FIRStruct 滤波器结构的指针
refSamps	参考样本的指针 （也可为 r）
muVal	自适应参数 （也可为 mu）
energyEstimate	最后M个输入信号样本的估计能量值，其中M为滤波器系数的数目。

返回值：

指向目标样本基地址的指针。

说明：

滤波器中系数的数目为 M。
系数 h[m] （定义在 0 ≤ m < M 范围内），作为循环递增模缓冲区。
延迟 d[m] （定义在 0 ≤ m < M 范围内），作为循环递增模缓冲区。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
参考样本 r[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
修改：

h_m[n] = h_m[n-1] + nu[n]*（r[n] - y[n]） *x[n-m]，
0 ≤ n < N， 0 ≤ m < M，
其中 nu[n] = mu/（mu+E[n]），
E[n]=E[n-1]+（x[n]） ^{2}-（x[n-M+1]） ^{2} 为输入信号能量的估计值。

再该函数的开头，energyEstimate 应被初始化为 E[-1] 的值 （当第一次调用该滤波器时为零）。返回时， energyEstimate 被更新为 E[N-1] 的值 （如果对扩展的输入信号进行滤波，该值可作为下一次函数调用的初始值）。
当 (r[n] - y[n]) 的绝对值大于或等于 1 时，该运算会导致饱和。
注：能量估计值另外的另外一个表达式为：
E[n] = （x[n]） ^{2} + （x[n-1） ^{2} + ... + （x[n-M+2]） ^{2}。
因此，为了避免在估计值计算中出现饱和，输入样本值应该受到限制，依下式：

$$\sum_{m=0}^{-M+2} (x[n+m])^2 < 1, \quad 0 \le n < N。$$

滤波器系数*不能*存放到程序存储器中，因为在这种情况下，它们的值不能被修改。如果检测到滤波器系数存放在程序存储器中，函数会返回 NULL。
（参见 FIRStruct、 FIRStructInit 和 FIRDelayInit。）

源文件：

firlmsn.asm

FIRLMSNorm （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复
MODCON	保存，使用，恢复
XMODSTRT	保存，使用，恢复
XMODEND	保存，使用，恢复
YMODSTRT	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
91

周期数 （包括 C 函数的调用和返回开销）：
66 + N(49 + 5M)

FIRStructInit

描述： FIRStructInit 函数的功能是：对 FIR 滤波器结构 FIRStruct 中参数的值进行初始化 。

头文件： dsp.h

函数原型：

```
extern void FIRStructInit (
  FIRStruct* filter,
  int numCoeffs,
  fractional* coeffsBase,
  int coeffsPage,
  fractional* delayBase
);
```

参数：

filter	指向 FIRStruct 滤波器结构的指针
numCoeffs	滤波器中系数的数目 （也可为 M）
coeffsBase	滤波器系数的基地址 （也可为 h）
coeffsPage	系数缓冲区的页号
delayBase	延迟缓冲区的基地址

说明： 参见上面关于 FIRStruct 结构的描述。
函数结束时， FIRStructInit 相应地初始化 coeffsEnd 和 delayEnd 指针。而且， delay 设置为等于 delayBase。

源文件： firinit.asm

资源使用情况： 系统资源的使用：

W0..W5	使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
10

周期数 （包括 C 函数的调用和返回开销）：
19

IIRCanonic

描述： IIRCanonic 函数对源样本序列应用正准型 （直接型 II）的二阶节级联 IIR 滤波器，将结果存放在目标样本序列中，并更新延迟值。

头文件： dsp.h

函数原型：

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase;
  int initialGain;
  int finalShift;
} IIRCanonicStruct;

extern fractional* IIRCanonic (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRCanonicStruct* filter
);
```

参数： 滤波器结构：

numSectionsLess1	级联的二阶节数减去 1 （可以为 S － 1）
coeffsBase	指向X数据空间或程序存储器中滤波器系数的指针 （也可为 { a, b }）
coeffsPage	系数缓冲区的页号，或者当系数在数据存储空间时，为 0xFF00 （定义的值 COEFFS_IN_DATA）
delayBase	指向滤波器延迟的指针 （也可为 d），*只能*在 Y 数据空间
initialGain	初始增益值
finalShift	输出定标 （左移）

滤波器的描述：

numSamps	滤波器输入样本的数目 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 IIRCanonicStruct 滤波器结构的指针

返回值： 指向目标样本基地址的指针。

说明： 每二阶节有 5 个系数，排列为有序集合 {a2[s], a1[s], b2[s], b1[s], b0[s]}，0 ≤ s < S。利用 Momentum Data Systems 公司的 dsPICFD 滤波器设计包或相似工具算出系数值。
延迟由每节的滤波器状态的两个字 （{d1[s], d2[s]}， 0 ≤ s < S）组成。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
初始增益值在*进入*滤波器结构前应用于每个输入样本。
在将结果存放到输出序列中之前，对滤波器结构的输出进行移位进行输出定标。这用于将滤波器的增益恢复为 0 dB。移位计数可以为 0 ；如果不为 0，它表示移位的位数：负数表示左移，正数表示右移。

源文件： iircan.asm

IIRCanonic （续）

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W11	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
42

周期数 （包括 C 函数的调用和返回开销）：
36 + N(8 + 7S)，
或者，当系数在程序存储器中，为 39 + N(9 + 12S)。

IIRCanonicInit

描述： IIRCanonicInit 函数将 IIRCanonicStruct 滤波器结构中的延迟值初始化为零。

头文件： dsp.h

函数原型：

```
extern void IIRCanonicInit (
    IIRCanonicStruct* filter
);
```

参数：

滤波器结构：
（参见 IIRCanonic 函数的描述。）

初始化描述：
filter 指向 IIRCanonicStruct 滤波器结构的指针。

说明： 每二阶节的滤波器状态的两个字 （{d1[s], d2[s]}， $0 \le s < S$）。

源文件： iircan.asm

资源使用情况：

系统资源的使用：

W0， W1	使用，不恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
7

周期数 （包括 C 函数的调用和返回开销）：
10 + S2

IIRLattice

描述： IIRLattice 函数的功能是：使用格型结构 （Lattice) 对源样本序列进行 IIR 滤波，将结果存放在目标样本序列中，并更新延迟值。

头文件： dsp.h

函数原型：

```
typedef struct {
  int order;
  fractional* kappaVals;
  fractional* gammaVals;
  int coeffsPage;
  fractional* delay;
} IIRLatticeStruct;

extern fractional* IIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRLatticeStruct* filter
);
```

参数： 滤波器结构：

order	滤波器阶次 （也可为 M， M ≤ N ； N 参见 FIRLattice）
kappaVals	X 数据空间或程序存储器中的格型系数的基地址 （也可为 k）
gammaVals	X 数据空间或程序存储器中梯度系数的基地址 （也可为 g）。如果为 NULL，函数将实现一个全极点滤波器。
coeffsPage	系数缓冲区的页号，或者当系数在数据存储空间中时，为 0xFF00 （定义的值 COEFFS_IN_DATA）
delay	延迟的基地址 （也可为 d），*仅*存放在 Y 数据空间中

滤波器描述：

numSamps	滤波器输入样本的数目（也可为 N，N ≥ M； M 可参见 IIRLatticeStruct ）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 IIRLatticeStruct 滤波器结构的指针

返回值： 指向目标样本基地址的指针。

说明： 格型系数 k[m]，定义在 0 ≤ m ≤ M 范围内。
梯度系数 g[m]，定义在 0 ≤ m ≤ M 范围内 （除非是实现一个全极点滤波器）。
延迟 d[m]，定义在 0 ≤ m ≤ M 范围内。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
注：本函数亚库提供的小数实现容易造成饱和。可利用如 OCTAVE 模型 （见本节最后部分）这样的小数实现 “离线”设计和测试滤波器。然后，应该在执行浮点型运算的过程中监控前向和后向中间值，查找超出 [-1, 1）范围的值。如果任意一个中间值超出该范围，则应在实时应用小数滤波器之前，将最大绝对值用于定标输入信号；也就是说，用该最大值的倒数与信号相乘。这样可以避免小数实现出现饱和。

源文件： iirlatt.asm

IIRLattice （续）

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
76

周期数 （包括 C 函数的调用和返回开销）：
46 + N(16 + 7M)，
或者，当系数在程序存储器中，为 49 + N(20 + 8M)。

如果实现一个全极点滤波器：
46 + N(16 + 6M)，
或者，当系数在程序存储器中，为 49 + N(16 + 7M)。

IIRLatticeInit

描述： IIRLatticeInit函数将IIRLatticeStruct滤波器结构中的延迟值初始化为零。

头文件： dsp.h

函数原型：

```
extern void IIRLatticeInit (
    IIRLatticeStruct* filter
);
```

参数：

滤波器结构：
（参见 IIRLattice 函数的描述）。

初始化描述：
filter 指向 IIRLatticeStruct 滤波器结构的指针。

源文件： iirlattd.asm

资源使用情况：

系统资源的使用：

W0..W2	使用，不恢复

DO 和 REPEAT 指令的使用：
无 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
6

周期数 （包括 C 函数的调用和返回开销）：
10 + M

IIRTransposed

描述： IIRTransposed 函数对源样本序列应用转置型 （直接型 II）的二阶节级联 IIR 滤波器，将结果存放在目标样本序列中，并更新延迟值。

头文件： dsp.h

函数原型：

```
typedef struct {
  int numSectionsLess1;
  fractional* coeffsBase;
  int coeffsPage;
  fractional* delayBase1;
  fractional* delayBase2;
  int finalShift;
} IIRTransposedStruct;

extern fractional* IIRTransposed (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   IIRTransposedStruct* filter
);
```

参数： 滤波器结构：

numSectionsLess1	级联的二阶节数减去 1 （可以为 S-1）
coeffsBase	指向X数据空间或程序存储器中滤波器系数的指针 （也可为 { a, b }）
coeffsPage	系数缓冲区的页号，或者当系数在数据存储空间时，为 0xFF00 （定义的值 COEFFS_IN_DATA）
delayBase1	指向滤波器状态 1 的指针，每二阶节延迟一个字，*仅*存放在 Y 数据空间中 （也可为 d1）。
delayBase2	指向滤波器状态 2 的指针，每二阶节延迟一个字，*仅*存放在 Y 数据空间中 （也可为 d2）。
finalShift	输出定标 （左移）

滤波器描述：

numSamps	滤波器输入样本的数目 （也可为 N）
dstSamps	指向目标样本的指针 （也可为 y）
srcSamps	指向源样本的指针 （也可为 x）
filter	指向 IIRTransposedStruct 滤波器结构的指针

返回值： 指向目标样本基地址的指针。

说明： 每二阶节有 5 个系数，排列为有序集合 {b0[s], b1[s], a1[s], b2[s],a2[s]}， 0 ≤ s < S。利用 Momentum Data Systems 公司的 dsPICFD 滤波器设计包或相似工具算出系数值。
延迟由每节的滤波器状态的两个字 （{d1[s], d2[s]}， 0 ≤ s < S）组成。
源样本 x[n]，定义在 0 ≤ n < N 范围内。
目标样本 y[n]，定义在 0 ≤ n < N 范围内。
初始增益值在*进入*滤波器结构前应用于每个输入样本。
在将结果存放到输出序列中之前，对滤波器结构的输出进行移位进行输出定标。这用于将滤波器的增益恢复为 0 dB。移位计数可以为 0 ；如果不为 0，它表示移位的位数：负数表示左移，正数表示右移。

源文件： iirtrans.asm

IIRTransposed （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W11	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复

DO 和 REPEAT 指令的使用：
两级 DO 指令
一级 REPEAT 指令

程序字 （24 位指令）：
48

周期数 （包括 C 函数的调用和返回开销）：
35 + N(11 + 11S)，
或者，当系数存放在程序存储器中，为 38 + N(9 + 17S)。
其中， S 为二阶节的数目。

IIRTransposedInit

描述： IIRTransposedInit 函数将 IIRTransposedStruct 滤波器结构中的延迟值初始化为零。

头文件： dsp.h

函数原型：

```
extern void IIRTransposedInit (
    IIRTransposedStruct* filter
);
```

参数： 滤波器结构：
（参见 IIRTransposed 函数的描述）。

初始化描述：
filter 指向 IIRTransposedStruct 滤波器结构的指针。

说明： 延迟由两个独立的缓冲区组成，每个缓冲区包含每节的滤波器状态的一个字 （{d2[s], d1[s]}， 0 ≤ s < S）。

源文件： iirtrans.asm

资源使用情况： 系统资源的使用：

W0..W2	使用，不恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
8

周期数 （包括 C 函数的调用和返回开销）：
11 + 2S，
S 为 二阶节的数目。

2.6.6 可用于分析格型 IIR 滤波器的 OCTAVE 模型

下面的 OCTAVE 模型可以用来在使用 IIRLattice 函数提供的小数实现之前检验格型 IIR 滤波器的性能。

IIRLattice OCTAVE model

```
function [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
## FUNCTION.-
## IIRLATT: IIR Fileter Lattice implementation.
##
##     [out, del, forward, backward] = iirlatt (in, kappas, gammas, delay)
##
##     forward: records intermediate forward values.
##     backward: records intermediate backward values.

#.....................................................................

## Get implicit parameters.
numSamps = length(in); numKapps = length(kappas);
if (gammas != 0)
   numGamms = length(gammas);
else
   numGamms = 0;
endif
numDels = length(delay); filtOrder = numDels-1;

## Error check.
if (numGamms != 0)
   if (numGamms != numKapps)
      fprintf ("ERROR! %d should be equal to %d.\n", numGamms, numKapps);
      return;
   endif
endif
if (numDels != numKapps)
   fprintf ("ERROR! %d should equal to %d.\n", numDels, numKapps);
   return;
endif

## Initialize.
M = filtOrder; out = zeros(numSamps,1); del = delay;
forward = zeros(numSamps*M,1); backward = forward; i = 0;

## Filter samples.
for n = 1:numSamps
   ## Get new sample.
   current = in(n);
```

```
   ## Lattice structure.
   for m = 1:M
      after     = current  - kappas(M+1-m) * del(m+1);
      del(m)    = del(m+1) + kappas(M+1-m) * after;
      i = i+1;
      forward(i) = current;
      backward(i) = after;
      current   = after;
   end
   del(M+1) = after;

   ## Ladder structure (computes output).
   if (gammas == 0)
      out(n) = del(M+1);
   else
      for m = 1:M+1
         out(n) = out(n) + gammas(M+2-m)*del(m);
      endfor
   endif
endfor

## Return.
return;

#.....................................................................

endfunction
```

2.7 变换函数

本节将讲述 DSP 函数库中小数变换的概念，并描述执行变换运算的各个函数。

2.7.1 小数变换运算

小数变换是线性、时不变和离散的运算，当应用于一个小数的时域样本序列时，会在频域内生成小数频率。相反地，当将小数变换运算的反变换应用于频域数据时，会生成其时域表示。

DSP 函数库提供了一套变换 （以及一部分反变换）。第一组变换对复数数据集 （参见下面关于复杂小数复数值的描述）应用离散傅里叶变换 （Discrete Fourier transform，DFT）（或反变换）。第二组变换对实数值序列应用类型 II 离散余弦变换 （Discrete Cosine Transform， DCT）。这些变换设计为可以 “原址”运算，或非 “原址”运算。前一种类型将变换结果保存到输出序列中。而对于后一种类型，输入序列 （在物理上）被变换后的序列替代。对于非 “原址”运算，需要有足够的存储空间来存放计算的结果。

变换使用了在调用变换函数时必须提供的变换因子 （或常数）。这些因子都是复数的数据集合，以浮点型算法进行计算，然后变换成小数以供运算使用。当应用变换时，为了避免过多的计算开销，可以一次生成一组特定的变换因子，在程序执行过程中多次使用这组因子。因此，建议将任何初始化运算返回的因子存放在持久 （静态）复数矢量中。“离线”生成因子，并将它们存放在程序存储器中，以备程序以后指向时使用，这样做也是非常有用的。这样，当一个应用程序有变换运算时，可以节省运行时间 （周期数）和 RAM 空间。

2.7.2 小数型复数矢量

复数矢量通过数据集合来表示，在该数据集合中，矢量中每个元素由两个值组成。其中第一个值是元素的实部，第二个值是元素的虚部。用存储器的一个字 （2 个字节）来存储实部和虚部，且必须都表示为 “1.15”小数格式。与小数矢量一样，小数型复数矢量将其元素连续地存放在存储器中。

小数型复数矢量的结构可以通过下面的数据结构访问来说明：

```
#ifdef fractional
#ifndef fractcomplex
typedef struct {
  fractional real;
  fractional imag;
} fractcomplex;
#endif
#endif
```

2.7.3 用户需要注意的事项

a) 不对这些函数执行边界检查。超出范围（包括零长度的矢量）以及源复数矢量和因子集合使用的不一致都可能产生预想不到的结果。

b) 建议在每个函数调用结束后检查状态寄存器（SR）。尤其是，用户可以在函数返回后检查 SA、 SB 和 SAB 标志，以判断是否发生了饱和。

c) 在变换运算中用到的输入和输出复数矢量*必须*存放在 Y 数据空间中。变换因子可以存放在 X 数据空间或程序存储器中。

d) 因为位反转寻址需要矢量集合按模数对齐，明确或隐含地使用 `BitReverseComplex` 函数的运算中的输入和输出复数矢量必须正确地分配。

e) 返回一个目标复数矢量的运算可以是嵌套的，例如，如果：

   a = Op1 (b, c)，且 b = Op2 (d)， c = Op3 (e, f)，则

   a = Op1 (Op2 (d), Op3 (e, f))。

2.7.4 各个函数

下面将描述实现变换及其反变换运算的各函数。

BitReverseComplex

描述： BitReverseComplex 函数以位反转顺序重新组织复数矢量的元素。

头文件： dsp.h

函数原型：

```
extern fractcomplex* BitReverseComplex (
    int log2N,
    fractcomplex* srcCV
);
```

参数：

log2N 以 2 为底的 N 的对数 （N 为源矢量中复数元素的数目）
srcCV 指向源复数矢量的指针

返回值： 指向源复数矢量基地址的指针。

说明：

N *必须*是 2 的整数次幂。
srcCV 矢量必须以 N 为模对齐分配。
该函数 “原址”运算。

源文件： bitrev.asm

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
MODCON	保存，使用，恢复
XBREV	保存，使用，恢复

DO 和 REPEAT 指令的使用：
一级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
27

周期数 （包括 C 函数的调用和返回开销）：

变换大小	复数元素数	周期数
32 点	32	245
64 点	64	485
128 点	128	945
256 点	256	1905

CosFactorInit

描述： CosFactorInit 函数的功能是：生成离散余弦变换 （类型 II）需要的余弦因子集合的前半部分，并将结果存放在复数目标矢量中。实际上，参数集合包含以下值：

$CN(k) = e^{j\frac{\pi k}{2N}}$，其中， $0 \le k < N/2$。

头文件： dsp.h

函数原型：
```
extern fractcomplex* CosFactorInit (
   int log2N,
   fractcomplex* cosFactors
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为 DCT 所需要的复数因子的数目）
cosFactors	指向复数余弦因子的指针

返回值： 指向余弦因子基地址的指针。

说明： N *必须*是 2 的整数次幂。
只生成前面 N/2 个余弦因子。
在调用函数之前，大小为 N/2 的复数矢量*必须*已经分配并指定给 cosFactors。复数矢量*必须*存放在 X 数据空间中。
因子以浮点运算进行计算，并转换为 “1.15” 格式的复数小数。

源文件： initcosf.c

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W14	保存，使用，恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括函数的调用和返回）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

DCT

描述： DCT 函数的功能是：对源矢量进行离散余弦变换，并将结果存放在目标矢量中。

头文件： dsp.h

函数原型：
```
extern fractional* DCT (
   int log2N,
   fractional* dstV,
   fractional* srcV,
   fractcomplex* cosFactors,
   fractcomplex* twidFactors,
   int factPage
);
```

DCT （续）

参数：

log2N	以 2 为底的 N 的对数 （N 为源矢量中复数元素的数目）
dstCV	指向目标矢量的指针
srcCV	指向源矢量的指针
cosFactors	指向余弦因子的指针
twidFactors	指向旋转因子的指针
factPage	变换因子的存储器页

返回值：

指向目标矢量基地址的指针。

说明：

N *必须*是 2 的整数次幂。
该函数非 “原址”运算。大小为 2N 个元素的矢量*必须*已经分配并指定给 dstV。
dstV 矢量必须以 N 为模对齐分配。
计算的结果存放在目标矢量的前 N 个元素中。
为了避免在计算过程出现饱和 （溢出），源矢量的值*应该*在 [-0.5, 0.5] 范围内。
仅需要前面 N/2 个余弦因子。
仅需要前面 N/2 个旋转因子。
如果变换因子存放在 X 数据空间中， cosFactors 和 twidFactors 指向因子分配的实际地址。如果变换因子存放在程序存储器中，cosFactors 和 twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果变换因子存放在 X 数据空间中， factPage 必须设置为 0xFF00 （定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中，factPage 为因子所在的程序页号。后面这个值可以利用行内汇编操作符 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为一个不等于零的值。
仅需要前面 N/2 个余弦因子。
输出要乘以因子 $1/(\sqrt{2N})$ 进行定标。

源文件：

dctoop.asm

资源使用情况：

系统资源的使用：
W0..W5 使用，不恢复
以及 VectorZeroPad 和 DCTIP 使用的系统资源。

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令
以及 VectorZeroPad 和 DCTIP 使用的 DO/REPEAT 指令。

程序字 （24 位指令）：
16
加上 VectorZeroPad 和 DCTIP 的程序字。

周期数 （包括 C 函数的调用和返回开销）：
22
加上 VectorZeroPad 和 DCTIP 的周期数。

注：在 VectorZeroPad 的描述中，周期数包括 C 函数调用开销的四个周期。所以， DCT 中由于 VectorZeroPad 所增加的实际周期数比单独的 ectorZeroPad 所用的周期数少 4 个周期。同样地， DCT 中由于 DCTIP 所增加的实际周期数比单独的 DCTIP 所用的周期数少 3 个周期。

DCTIP

描述： DCTIP 函数的功能是：计算源矢量的离散余弦变换，计算结果丢回“原址”（In Place）。

头文件： dsp.h

函数原型：

```
extern fractional* DCTIP (
   int log2N,
   fractional* srcV,
   fractcomplex* cosFactors,
   fractcomplex* twidFactors,
   int factPage
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为源矢量中复数元素的数目）
srcCV	指向源矢量的指针
cosFactors	指向余弦因子的指针
twidFactors	指向旋转因子的指针
factPage	变换因子的存储器页号

返回值： 指向目标矢量基地址的指针。

说明：
N *必须*为 2 的整数次幂。
该函数要求源矢量填补零至 2N 长度。
srcV 矢量必须以 N 为模对齐分配。
计算的结果存放在源矢量的前面 N 个元素内。
为了避免在计算过程中出现饱和 （溢出），源矢量的值应该在 [-0.5, 0.5] 范围内。
仅仅需要前面 N/2 个余弦因子。
仅仅需要前面 N/2 个旋转因子。
如果变换因子存放在 X 数据空间中， cosFactors 和 twidFactors 指向因子分配的实际地址。如果变换因子存放在程序存储器中，cosFactors 和 twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果变换因子存放在 X 数据空间中， factPage 必须被设定为 0xFF00（定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中，factPage 为因子所在的程序页号。后面这个值可以利用行内汇编操作符 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为一个不等于零的值。
输出要乘以因子 $1/(\sqrt{2N})$ 进行定标。

源文件： dctoop.asm

DCTIP （续）

资源使用情况： 系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复 （仅当因子在程序存储器中时）

DO 和 REPEAT 指令的使用：
一级 DO 指令
一级 REPEAT 指令
以及 IFFTComplexIP 使用的 DO/REPEAT 指令。

程序字 （24 位指令）：
92
加上 IFFTComplexIP 的程序字。

周期数 （包括 C 函数的调用和返回开销）：
71 + 10N，
或者，当因子存放在程序存储器中，为 73 + 11N ，
加上 IFFTComplexIP 的周期数。

注：在 IFFTComplexIP 的描述开销中，周期数包括 C 函数调用的 4 个周期。因此， DCTIP 中因 IFFTComplexIP 增加的实际周期数比单独的 IFFTComplexIP 所用的周期数少 4 个周期。

FFTComplex

描述： FFTComplex 函数对源复数矢量进行离散傅里叶变换，并将结果存放在目标复数矢量中。

头文件： dsp.h

函数原型：

```
extern fractcomplex* FFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为源矢量中复数元素的数目）
dstCV	指向目标复数矢量的指针
srcCV	指向源复数矢量的指针
twidFactors	旋转因子的基地址
factPage	变换因子的存储器页

返回值： 指向目标复数矢量基地址的指针。

FFTComplex （续）

说明：

N *必须*为 2 的整数次幂。
该函数不 “原址”运算。一个足够大以接收运算结果的复数矢量*必须*已经分配并指定给 dstCV。
dstCV 矢量必须以 N 为模对齐分配。
要求源复数矢量中的元素按照自然顺序存储。
目标复数矢量中的元素按照自然顺序生成。
为了避免在计算过程中出现饱和 （溢出），源复数矢量的幅值*应该*处于 [-0.5, 0.5] 范围内。
仅需要前面 N/2 个旋转因子。
如果旋转因子存放在 X 数据空间内， twidFactors 指向因子所分配的实际地址。如果旋转因子存放在程序存储器中， twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果旋转因子存放在 X 数据空间中， factPage 必须被设置为 0xFF00（定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中，factPage 为因子所在的程序页号。后面这个值可以利用行内汇编操作符 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为零。
输出必须乘以因子 1/N 来定标。

源文件：

fftoop.asm

资源使用情况：

系统资源的使用：

W0..W4	使用，不恢复

以及 VectorCopy、 FFTComplexIP 和 BitReverseComplex 所使用的系统资源。

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令
以及 VectorCopy、 FFTComplexIP 和 BitReverseComplex 使用的 DO/REPEAT 指令。

程序字 （24 位指令）：
17
加上 VectorCopy， FFTComplexIP 和 BitReverseComplex 的程序字。

周期数 （包括 C 函数的调用和返回开销）：
23
加上 VectorCopy、 FFTComplexIP 和 BitReverseComplex 的周期数。

注：在 VectorCopy 的描述中，周期数包括 C 函数调用开销的 3 个周期。因此， FFTComplex 中因 VectorCopy 所增加的实际周期数比单独的 VectorCopy 所用的周期数少 3 个周期。同样地， FFTComplex 中因 FFTComplexIP 所增加的周期数比单独的 FFTComplexIP 所用的周期数少 4 个周期。而因 BitReverseComplex 所增加的周期数比单独的 FFTComplex 所用的周期数少 2 个周期。

FFTComplexIP

描述： FFTComplexIP 函数的功能是：计算源复数矢量的离散傅里叶变换，，计算结果丢回 “原址”（In Place）。

头文件： dsp.h

函数原型：

```
extern fractcomplex* FFTComplexIP (
   int log2N,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

参数：

log2N	以 2 为底的 N 的对数 （N 为源矢量中复数元素的数目）
srcCV	指向源复数矢量的指针
twidFactors	旋转因子的基地址
factPage	变换因子的存储器页

返回值： 指向源复数矢量基地址的指针。

说明：
N *必须*是 2 的整数次幂。
要求源复数矢量中的元素按照自然顺序存储。
变换结果以位反转顺序存放。
为了避免在计算程中出现饱和 （溢出），源复数矢量的值的幅值应该在 [-0.5, 0.5] 范围内。
仅需要前面 N/2 个旋转因子。
如果旋转因子存放在 X 数据空间中， twidFactors 指向因子分配的实际地址。如果旋转因子存放在程序存储器中， twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果旋转因子存放在 X 数据空间中， factPage 必须被设定为 0xFF00（定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中，factPage 为因子所在的程序页号。后面这个值可以利用内嵌的算子 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为零。
输出应乘以因子 1/N 进行定标。

源文件： fft.asm

FFTComplexIP （续）

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W13	保存，使用，恢复
ACCA	使用，不恢复
ACCB	使用，不恢复
CORCON	保存，使用，恢复
PSVPAG	保存，使用，恢复 （仅当因子在程序存储器中时）

DO 和 REPEAT 指令的使用：
两级 DO 指令
无 REPEAT 指令

程序字 （24 位指令）：
59

周期数 （包括 C 函数的调用和返回开销）：

变换大小	当旋转因子在 X 数据空间中时的周期数	当旋转因子在程序存储器中时的周期数
32 点	1,633	1,795
64 点	3,739	4,125
128 点	8,485	9,383
256 点	19,055	21,105

IFFTComplex

描述： IFFTComplex 函数计算源复数矢量的离散傅里叶反变换，并将结果存放在目标复矢量中。

头文件： dsp.h

函数原型：

```
extern fractcomplex* IFFTComplex (
   int log2N,
   fractcomplex* dstCV,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为源矢量中复数元素的数目）
dstCV	指向目标复矢量的指针
srcCV	指向源复矢量的指针
twidFactors	旋转因子的基地址
factPage	转换因子的存储器页

返回值： 指向目标复矢量基地址的指针。

IFFTComplex （续）

说明：

N *必须*是 2 的整数次幂。
该函数不 “原址”运算。一个足够大以接收运算结果的复数矢量*必须*已经分配并指定给 dstCV。
dstCV 矢量必须以 N 为模对齐分配。
要求源复数矢量中的元素按照自然顺序存储。
目标复数矢量中的元素按照自然顺序生成。
为了避免在计算过程中出现饱和 （溢出），源复数矢量的值的幅值*应该*在 [-0.5, 0.5] 范围内。
如果旋转因子存放在 X 数据空间中， twidFactors 指向因子分配的实际地址。如果旋转因子存放在程序存储器中， twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果旋转因子存放在 X 数据空间间中， factPage 必须被设定为 0xFF00 （定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中， factPage 为因子所在的程序页号。后面这个值可以利用行内汇编操作符 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为一个不等于零的值。
仅需要前面 N/2 个旋转因子。

源文件：

ifftoop.asm

资源使用情况：

系统资源的使用：
W0..W4 使用，不恢复
以及 VectorCopy 和 IFFTComplexIP 使用的系统资源。

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令
以及 VectorCopy 和 IFFTComplexIP 使用的 DO/REPEAT 指令。

程序字 （24 位指令）：
12
加上 VectorCopy 和 IFFTComplexIP 的程序字。

周期数 （包括 C 函数的调用和返回开销）：
15
加上 VectorCopy 和 IFFTComplexIP 的周期数。

注：在 VectorCopy 的描述中，周期数包括 C 函数调用开销的 3 个周期。因此， FFTComplex 中因 VectorCopy 所增加的实际周期数比单独的 VectorCopy 所用的周期数少 3 个。同样地， FFTComplex 中因 IFFTComplexIP 所增加的周期数比单独的 IFFTComplexIP 所用的周期数少 4 个。

IFFTComplexIP

描述： IFFTComplexIP 函数计算源复数矢量的离散傅里叶反变换，计算结果丢回 “原址”（In Place）。

头文件： dsp.h

函数原型：

```
extern fractcomplex* IFFTComplexIP (
   int log2N,
   fractcomplex* srcCV,
   fractcomplex* twidFactors,
   int factPage
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为源矢量中复数元素的数目）
srcCV	指向源复矢量的指针
twidFactors	旋转因子的基地址
factPage	转换因子的存储器页

返回值： 指向源复矢量基地址的指针。

说明： N *必须*是 2 的整数次幂。
要求源复数矢量中按照位反转顺序排列。
变换结果按照自然顺序存放。
srcCV 矢量必须以 N 为模对齐分配。
为了避免在计算过程中出现饱和 （溢出），源复数矢量的值的幅值应该在 [-0.5, 0.5] 范围内。
如果旋转因子存放在 X 数据空间中， twidFactors 指向因子分配的实际地址。如果旋转因子存放在程序存储器中， twidFactors 为相对于因子所在页边界的偏移量。后面这个值可以利用行内汇编操作符 psvoffset() 计算出来。
如果旋转因子存放在 X 数据空间中， factPage 必须被设定为 0xFF00（定义的值 COEFFS_IN_DATA）。如果它们存放在程序存储器中，factPage 为因子所在的程序页号。后面这个值可以利用行内汇编操作符 psvpage() 计算出来。
旋转因子*必须*被初始化， conjFlag 设置为一个不等于零的值。
仅需要前面 N/2 个旋转因子。

源文件： ifft.asm

IFFTComplexIP （续）

资源使用情况：

系统资源的使用：
W0..W3 使用，不恢复
以及 FFTComplexIP 和 BitReverseComplex 使用的系统资源。

DO 和 REPEAT 指令的使用：
无 DO 指令
无 REPEAT 指令
以及 FFTComplexIP 和 BitReverseComplex 使用的 DO/REPEAT 指令。

程序字：
11
加上 FFTComplexIP 和 BitReverseComplex 的程序字。

周期数 （包括 C 函数的调用和返回开销）：
15
加上 FFTComplexIP 和 BitReverseComplex 的周期数。

注：在 FFTComplexIP 的描述中，周期数包括 C 函数调用开销的 3 个周期。因此， IFFTComplexIP 中因 FFTComplexIP 所增加的实际周期数比单独的 FFTComplexIP 所用的周期数少 3 个。同样地，IFFTComplexIP中因BitReverseComplex所增加的周期数比单独的 BitReverseComplex 所用的周期数少 2 个。

TwidFactorInit

描述：

TwidFactorInit 函数生成离散傅里叶变换或离散余弦变换所需要的旋转因子集合的前半部分，并将结果存放在复数目标矢量中。实际上，该集合包含下列值：

$WN(k) = e^{-j\frac{2\pi k}{N}}$ ，当 *conjFlag* = 0 时，其中 $0 \le k \le N/2$

$WN(k) = e^{j\frac{2\pi k}{N}}$ ，当 *conjFlag* ! = 0 时，其中 $0 \le k \le N/2$

头文件：

dsp.h

函数原型：

```
extern fractcomplex* TwidFactorInit (
   int log2N,
   fractcomplex* twidFactors,
   int conjFlag
);
```

参数：

log2N	以 2 为底的 N 的对数（N 为 DFT 需要的复数因子的数目）
twidFactors	指向复数旋转因子的指针
conjFlag	指示是否要生成共轭值的标志

返回值：

指向旋转因子基地址的指针。

TwidFactorInit （续）

说明：

N *必须*是 2 的整数次幂。
仅生成前面 N/2 个旋转因子。
conjFlag 的值确定指数函数参数的符号。对于傅里叶变换，*conjFlag* 应被设为 0，对于傅里叶反变换和离散余弦变换，*conjFlag* 应被设为 1。
在调用函数之前，一个大小为 N/2 的复数矢量必须已经分配并指定给 twidFactors。复数矢量*应该*分配到 X 数据空间中。
因子以浮点运算进行计算，并转换为 “1.15”格式的复数小数形式。

源文件：

inittwid.c

资源使用情况：

系统资源的使用：

W0..W7	使用，不恢复
W8..W14	使用，保存，恢复

DO 和 REPEAT 指令的使用：
都未使用

程序字 （24 位指令）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

周期数 （包括 C 函数的调用和返回开销）：
参阅 pic30_tools\src\dsp 目录下的 “readme.txt”。

注：

第 3 章 dsPIC 外设函数库

3.1 简介

本章介绍了 dsPIC 外设函数库中包含的函数和宏，并提供了使用示例。

在 pic30_tools\src\peripheral 中的 readme.txt 文件中可以查到每个库函数或宏的代码大小。

3.1.1 汇编代码应用程序

从 Microchip 网站上可以获得这些库和相关的头文件的免费版本，其中包括源代码。

3.1.2 C 代码应用程序

与库相关的文件存放在 MPLAB C30 C 编译器安装目录 （c:\pic30_tools）的如下子目录中：

- lib——dsPIC 外设库文件
- src\peripheral——库函数的源代码以及用于重建库的批处理文件
- support\h——库的头文件

3.1.3 章节组织

本章的结构如下：

- 使用 dsPIC 外设函数库

软件函数

- 外部 LCD 函数

硬件函数

- CAN 函数
- ADC12 函数
- ADC10 函数
- 定时器函数
- 复位 / 控制函数
- I/O 端口函数
- 输入捕捉函数
- 输出比较函数
- UART 函数
- DCI 函数
- SPI 函数
- QEI 函数
- PWM 函数
- I2C 函数

3.2 使用 dsPIC 外设函数库

构建利用 dsPIC 外设函数库的应用程序需要特定处理器的库文件和每个外设模块的头文件。

对于每个外设，相应的头文件都提供了所有函数原型、以及库使用的 `#define` 和 `typedef`。归档库文件包含了每个库函数的目标文件。

头文件的形式为 *`peripheral`*`.h`，其中 *`peripheral`* *为*使用的特定外设的名称（如， `can.h` 对应 CAN）。

库文件的形式为 `libp`*`Device-omf`*`.a`，其中 *`Device`* 为 dsPIC 器件编号 （如，`libp30F6014-coff.a` 对应 dsPIC30F6014 器件）。关于特定于 OMF 的库的更多信息，请参阅**第 1.2 节 “特定于 OMF 的库 / 启动模块”**。

当编译应用程序时，调用库中函数或者使用其符号或 `typedef` 的所有源文件必须都（使用 `#include`）引用头文件。当链接应用程序时，库文件必须作为链接器的一个输入 （使用 `--library` 或 `-l` 链接器开关），这样应用程序使用的函数就会被链接到应用程序。

批处理文件 `makeplib.bat` 可以用来重建库。默认的操作是为支持的所有目标处理器建立外设函数库；但也可以在命令行中指定特定的处理器。例如：

```
makeplib.bat 30f6014
```

或

```
makeplib.bat 30F6014
```

将为 dsPIC30F6014 器件重建库。

3.3 外部 LCD 函数

本节给出了用于与 P-tec PCOG1602B LCD 控制器接口的各个函数以及使用这些函数的例子。函数也可以用宏来实现。

仅下列器件支持外部 LCD 函数：

- dsPIC30F5011
- dsPIC30F5013
- dsPIC30F6010
- dsPIC30F6011
- dsPIC30F6012
- dsPIC30F6013
- dsPIC30F6014

3.3.1 各个函数

BusyXLCD

描述： 该函数检查 P-tec PCOG1602B LCD 控制器的忙标志。

头文件： xlcd.h

函数原型：
```
char BusyXLCD(void);
```

参数： 无

返回值： 如果返回 “1”，表明 LCD 控制器忙，无法接收任何命令。
如果返回 “0”，表明 LCD 控制器准备接收下一个命令。

说明： 该函数返回 P-tec PCOG1602B LCD 控制器忙标志的状态。

源文件： BusyXLCD.c

代码示例：
```
while(BusyXLCD());
```

OpenXLCD

描述： 该函数配置 I/O 引脚并初始化 P-tec PCOG1602B LCD 控制器。

头文件： xlcd.h

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

参数： *lcdtype* 包含如下定义的要配置的 LCD 控制器参数：

接口类型
FOUR_BIT
EIGHT_BIT

行数
SINGLE_LINE
TWO_LINE

段数据传输方向
SEG1_50_SEG51_100
SEG1_50_SEG100_51
SEG100_51_SEG50_1
SEG100_51_SEG1_50

COM 数据传输方向
COM1_COM16
COM16_COM1

返回值： 无

说明： 该函数对用来控制 P-tec PCOG1602B LCD 控制器的 I/O 引脚进行配置。同时初始化 LCD 控制器。为确保外部 LCD 正确工作，必须进行如下 I/O 引脚定义：

OpenXLCD （续）

控制 I/O 引脚定义

RW_PIN	PORTxbits.Rx?
TRIS_RW	TRISxbits.Rx?
RS_PIN	PORTxbits.Rx?
TRIS_RS	TRISxbits.Rx?
E_PIN	PORTxbits.Rx?
TRIS_E	TRISxbits.Rx?

其中 x 是端口，? 是引脚号

数据引脚定义

DATA_PIN_?	PORTxbits.RD?
TRIS_DATA_PIN_?	TRISxbits.TRISD?

其中 x 是端口，? 是引脚号

数据引脚可以来自一个或多个端口。

控制引脚可以来自任意端口而不必来自同一个端口。数据接口必须定义为 4 位或 8 位。当头文件 xlcd.h 中包含 #define EIGHT_BIT_INTERFACE 时，则定义了 8 位接口。如果不包含这个定义，则默认定义了 4 位接口。

这些定义完成后，用户必须将应用代码编译成要链接的目标文件。

该函数还需要三个特定的外部延时子程序：

DelayFor18TCY()	延时 18Tcy
DelayPORXLCD()	延时 15ms
DelayXLCD()	延时 5ms
Delay100XLCD()	延时 100Tcy

源文件： openXLCD.c

代码示例：

```
OpenXLCD(EIGHT_BIT & TWO_LINE
                    & SEG1_50_SEG51_100 &COM1_COM16);
```

putsXLCD putrsXLCD

描述： 该函数向 P-tec PCOG1602B 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。

返回值： 无

说明： 这些函数将 buffer 中的字符串写入 P-tec PCOG1602B LCD 控制器中，直到在字符串中遇到空字符为止。
为了连续显示写入 P-tec PCOG1602B LCD 控制器的数据，必须将显示设置成移位模式。

源文件： PutsXLCD.c
PutrsXLCD.c

代码示例：

```
char display_char[13];
putsXLCD(display_char);
```

ReadAddrXLCD

描述： 该函数从 P-tec PCOG1602B LCD 控制器中读出一个地址字节。

头文件： xlcd.h

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

参数： 无

返回值： 该函数返回一个 8 位值，这个字节的低 7 位是 7 位地址，第 8 位是 BUSY 状态标志。

说明： 该函数从 P-tec PCOG1602B LCD 控制器中读出地址字节。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。
从控制器中读出的地址是字符发生器 （CG） RAM 还是显示数据（DD） RAM 中的地址，取决于前面调用的 Set??RamAddr() 函数，其中 ?? 指 CG 或 DD。

源文件： ReadAddrXLCD.c

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

ReadDataXLCD

描述： 该函数从 P-tec PCOG1602B LCD 控制器中读出一个数据字节。

头文件： xlcd.h

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

参数： 无

说明： 该函数从 P-tec PCOG1602B LCD 控制器中读出一个数据字节。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。
从控制器中读出的数据是来自于字符发生器 （CG） RAM 还是显示数据（DD） RAM，取决于前面调用的 Set??RamAddr() 函数，其中 ?? 指 CG 或 DD。

返回值： 该函数返回地址指定的 8 位数据值。

源文件： ReadDataXLCD.c

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

SetCGRamAddr

描述： 该函数设定字符发生器的地址。

头文件： xlcd.h

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

参数： *CGaddr* 字符发生器地址。

返回值： 无

说明： 该函数设定 P-tec PCOG1602B LCD 控制器的字符发生器地址。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。

源文件： SetCGRamAddr.c

代码示例：

```
char cgaddr = 0x1F;
while (BusyXLCD());
SetCGRamAddr(cgaddr);
```

SetDDRamAddr

描述： 该函数设定显示数据的地址。

头文件： xlcd.h

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

参数： *DDaddr* 显示数据的地址。

返回值： 无

说明： 该函数设定 P-tec PCOG1602B LCD 控制器显示数据的地址。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。

源文件： SetDDRamAddr.c

代码示例：

```
char ddaddr = 0x10;
while (BusyXLCD());
SetDDRamAddr(ddaddr);
```

WriteDataXLCD

描述： 该函数向 P-tec PCOG1602B LCD 控制器写入一个数据字节 （一个字符）。

头文件： xlcd.h

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

参数： *data* 数据的值可以是任意 8 位的值，但应与 P-tec PCOG1602B LCD 控制器的字符 RAM 表相对应。

返回值： 无

说明： 该函数向 P-tec PCOG1602B LCD 控制器写入一个数据字节。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。
写入控制器的数据是存放到字符发生器 （CG） RAM 还是显示数据 （DD） RAM，取决于前面调用的 Set??RamAddr() 函数，其中 ?? 指 CG 或 DD。

源文件： WriteDataXLCD.c

代码示例：

```
WriteDataXLCD(0x30);
```

WriteCmdXLCD

描述： 该函数向 P-tec PCOG1602B LCD 控制器写入一个命令。

头文件： xlcd.h

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

参数： *cmd* 包含需配置的 LCD 控制器参数，定义如下：

接口类型
FOUR_BIT
EIGHT_BIT

行数
SINLE_LINE
TWO_LINE

段数据传输方向
SEG1_50_SEG51_100
SEG1_50_SEG100_51
SEG100_51_SEG50_1
SEG100_51_SEG1_50

COM 数据传输方向
COM1_COM16
COM16_COM1

显示开 / 关控制
DON
DOFF
CURSOR_ON
CURSOR_OFF
BLINK_ON
BLINK_OFF

光标或显示移位定义
SHIFT_CUR_LEFT
SHIFT_CUR_RIGHT
SHIFT_DISP_LEFT
SHIFT_DISP_RIGHT

返回值： 无

说明： 该函数向 P-tec PCOG1602B LCD 控制器写入命令字节。用户必须首先通过调用 BusyXLCD() 函数来检查 LCD 控制器是否正忙。

源文件： WriteCmdXLCD.c

代码示例：

```
while(BusyXLCD());
WriteCmdXLCD(EIGHT_BIT & TWO_LINE);
WriteCmdXLCD(DON);
WriteCmdXLCD(SHIFT_DISP_LEFT);
```

3.3.2 使用示例

```
#define __dsPIC30F6014__
#include <p30fxxxx.h>
#include<xlcd.h>
/* holds the address of message */
char * buffer;
char data ;
char mesg1[] = {'H','A','R','D','W','A','R','E','\0'};
char mesg2[] = {'P','E','R','I','P',’H’,’E’,’R’,’A’,’L’
                ‘ ‘, ‘L’,’I’,’B’,’ ‘,'\0'};
int main(void)
{
/* Set 8bit interface and two line display */
    OpenXLCD(EIGHT_BIT & TWO_LINE & SEG1_50_SEG51_100
             & COM1_COM16);
/* Wait till LCD controller is busy */
    while(BusyXLCD());
/* Turn on the display */
    WriteCmdXLCD(DON & CURSOR_ON & BLINK_OFF);
    buffer = mesg1;
    PutsXLCD(buffer);
    while(BusyXLCD());
/* Set DDRam address to 0x40 to dispaly data in the second line */
    SetDDRamAddr(0x40);
    while(BusyXLCD());
    buffer = mesg2;
    PutsXLCD(buffer);
    while(BusyXLCD());
    return 0;
}
```

3.4 CAN 函数

本节给出了有关 CAN 的各个函数以及使用这些函数的例子。函数也可以用宏来实现。

3.4.1 各函数

CAN1AbortAll
CAN2AbortAll

描述：	该函数中止所有等待的发送。
头文件：	can.h
函数原型：	void CAN1AbortAll(void); void CAN2AbortAll(void);
参数：	无
返回值：	无
说明：	该函数对 CiCTRL 寄存器中的 ABAT 位置位，从而中止等待的发送。但是，正在进行的发送不会中止。当消息发送成功中止时，该位由硬件清零。
源文件：	CAN1AbortAll.c CAN2AbortAll.c
代码示例：	CAN1AbortAll();

CAN1GetRXErrorCount
CAN2GetRXErrorCount

描述：	该函数返回接收错误计数值。
头文件：	can.h
函数原型：	unsigned char CAN1GetRXErrorCount(void); unsigned char CAN2GetRXErrorCount(void);
参数：	无
返回值：	CiRERRCNT 的内容，长度为 8 位。
说明：	该函数返回 CiRERRCNT （CiEC 寄存器的低字节）的内容，其值表明接收错误计数。
源文件：	CAN1GetRXErrorCount.c CAN2GetRXErrorCount.c
代码示例：	unsigned char rx_error_count; rx_error_count = CAN1GetRXErrorCount();

CAN1GetTXErrorCount
CAN2GetTXErrorCount

描述： 该函数返回发送错误计数值。

头文件： can.h

函数原型：
```
unsigned char CAN1GetTXErrorCount(void);
unsigned char CAN2GetTXErrorCount(void);
```

参数： 无

返回值： CiTERRCNT 中的值，长度为 8 位。

说明： 该函数返回 CiTERRCNT （CiEC 寄存器的高字节）的内容，其值表明发送错误计数。

源文件：
```
CAN1GetTXErrorCount.c
CAN2GetTXErrorCount.c
```

代码示例：
```
unsigned char tx_error_count;
tx_error_count = CAN1GetTXErrorCount();
```

CAN1IsBusOff
CAN2IsBusOff

描述： 该函数确定 CAN 节点是否处于总线关闭模式。

头文件： can.h

函数原型：
```
char CAN1IsBusOff(void);
char CAN2IsBusOff(void);
```

参数： 无

返回值： 如果 TXBO 的值为 “1”，那么返回 “1”，表明总线由于发送错误而关闭了。
如果 TXBO 的值为 “0”，那么返回 “0”，表明总线没有关闭。

说明： 该函数返回 CiINTF 寄存器中 TXBO 位的状态。

源文件：
```
CAN1IsBusOff.c
CAN2IsBusOff.c
```

代码示例：
```
while(CAN1IsBusOff());
```

CAN1IsRXReady
CAN2IsRXReady

描述： 该函数返回接收缓冲器是否为满状态。

头文件： can.h

函数原型：
```
char CAN1IsRXReady(char);
char CAN2IsRXReady(char);
```

参数： *buffno* buffno 的值，指明需要知道其状态的接收缓冲器。

返回值： 如果 RXFUL 为 1，表明接收缓冲器中有一个接收到的消息。
如果 RXFUL 为 0，表明接收缓冲器是空的，可以接收新消息。

说明： 该函数返回接收控制寄存器的 RXFUL 位的状态。

源文件：
```
CAN1IsRXReady.c
CAN2IsRXReady.c
```

代码示例：
```
char rx_1_status;
rx_1_status = CAN1IsRXReady(1);
```

CAN1IsRXPassive
CAN2IsRXPassive

描述： 该函数确定接收器是否处于错误被动状态。

头文件： can.h

函数原型：
```
char CAN1IsRXPassive(void);
char CAN2IsRXPassive(void);
```

参数： 无

返回值： 如果 RXEP 的值为 “1”，那么返回 “1”，表明节点由于接收错误变为被动。
如果 RXEP 的值为 “0”，那么返回 “0”，表明总线上没有错误。

说明： 该函数返回 CiINTF 寄存器的 RXEP 位的状态。

源文件：
```
CAN1IsRXPassive.c
CAN2IsRXPassive.c
```

代码示例：
```
char rx_bus_status;
rx_bus_status = CAN1IsRXPassive();
```

CAN1IsTXPassive
CAN2IsTXPassive

描述： 该函数确定发送器是否处于错误被动状态。

头文件： can.h

函数原型：
```
char CAN1IsTXPassive(void);
char CAN2IsTXPassive(void);
```

参数： 无

返回值： 如果 TXEP 的值为 “1”，那么返回 “1”，表明发送总线上有错误且总线变为被动。
如果 TXEP 的值为 “0”，那么返回 “0”，表明发送总线上没有错误。

说明： 该函数返回 CiINTF 寄存器的 TXEP 位的状态。

源文件：
```
CAN1IsTXPassive.c
CAN2IsTXPassive.c
```

代码示例：
```
char tx_bus_status;
tx_bus_status = CAN1IsTXPassive();
```

CAN1IsTXReady
CAN2IsTXReady

描述： 该函数返回发送器的状态，指明 CAN 节点是否准备好下一次发送。

头文件： can.h

函数原型：
```
char CAN1IsTXReady(char);
char CAN2IsTXReady(char);
```

参数： *buffno* buffno 的值，指明需要知道其状态的发送缓冲器。

返回值： 如果 TXREQ 为 “1”，返回 “0”，表明发送缓冲器不是空的。
如果 TXREQ 为 “0”，返回 “1”，表明发送缓冲器是空的并准备好进行下一次发送。

说明： 该函数返回发送控制寄存器中的 TXREQ 状态位的反码。

源文件：
CAN1IsTXReady.c
CAN2IsTXReady.c

代码示例：
```
char tx_2_status;
tx_2_status = CAN1IsTXReady(2);
```

CAN1ReceiveMessage
CAN2ReceiveMessage

描述： 该函数从接收缓冲器中读出数据。

头文件： can.h

函数原型：
```
void CAN1ReceiveMessage(unsigned char *
 data, unsigned char datalen, char MsgFlag);
void CAN2ReceiveMessage(unsigned char *
 data, unsigned char datalen,char MsgFlag);
```

参数：

data 指向存储接收到的数据的地址的指针。

datalen 要读的数据字节数。

MsgFlag 接收数据的缓冲器编号。
如果为 “1”，读出从 CiRX1B1 到 CiRX1B4 的数据。
如果为 “0”或其他，读出从 CiRX0B1 到 CiRX0B4 的数据。

说明： 该函数将接收到的数据读入由输入参数 *data* 所指向的地址。

返回值： 无。

源文件：
CAN1ReceiveMessage.c
CAN2ReceiveMessage.c

代码示例：
```
unsigned char*rx_data;
CAN1ReceiveMessage(rx_data, 5, 0);
```

CAN1SendMessage
CAN2SendMessage

描述： 该函数把将要发送的数据写入 TX 寄存器中，设置数据长度并启动发送。

头文件： can.h

函数原型：

```
void CAN1SendMessage(unsigned int sid,
 unsigned long eid, unsigned char *data,
 unsigned char datalen, char MsgFlag);
void CAN2SendMessage(unsigned int sid,
 unsigned long eid, unsigned char *data,
 unsigned char datalen, char MsgFlag);
```

参数：

sid 要写入 CiTXnSID 寄存器中的 16 位值。
CAN_TX_SID(x) x 是所需的 SID 值。
替代远程请求
CAN_SUB_REM_TX_REQ
CAN_SUB_NOR_TX_REQ
消息 ID 类型
CAN_TX_EID_EN
CAN_TX_EID_DIS

eid 要写入 CiTXnEID 和 CiTXnDLC 寄存器的 32 位值。
CAN_TX_EID(x) x 是所需的 EID 值。
替代远程请求
CAN_REM_TX_REQ
CAN_NOR_TX_REQ

data 指向要发送数据的储存地址的指针。

datalen 要发送数据的字节数。

MsgFlag 从中发送数据的缓冲器编号（“0”、“1”或“2”）。
如果是“1”，数据写入 CiTX1B1 到 CiTX1B4。
如果是“2”，数据写入 CiTX2B1 到 CiTX2B4。
如果是“0”或其他数，数据写入 CiTX0B1 到 CiTX0B4。

返回值： 无

说明： 该函数将标识值写入 SID 和 EID 寄存器，将要发送的数据写入 TX 寄存器，设置数据长度并通过置位 TXREQ 位来启动发送。

源文件：
CAN1SendMessage.c
CAN2SendMessage.c

代码示例：

```
CAN1SendMessage((CAN_TX_SID(1920)) &
(CAN_TX_EID_EN) & (CAN_SUB_NOR_TX_REQ),
(CAN_TX_EID(12344)) & (CAN_NOR_TX_REQ),
Txdata, datalen, tx_rx_no);
```

CAN1SetFilter
CAN2SetFilter

描述： 该函数为指定的过滤器设置接收过滤器值 （SID 和 EID）。

头文件： can.h

函数原型：
```
void CAN1SetFilter(char filter_no, unsigned int sid,
 unsigned long eid);
void CAN2SetFilter(char filter_no, unsigned int sid,
 unsigned long eid);
```

参数：

filter_no	必须配置新过滤器值的过滤器 （0， 1， 2， 3， 4 或 5）。
sid	要写入 CiRXFnSID 寄存器的 16 位值。 CAN_FILTER_SID(x) x 是所需的 SID 值。 要接收的消息类型 CAN_RX_EID_EN CAN_RX_EID_DIS
eid	要写入 CiRXFnEIDH 和 CiRXFnEIDL 寄存器的 32 位值。 CAN_FILTER_EID(x) x 是所需的 EID 值。

返回值： 无

说明： 该函数将 *sid* 中的 16 位值写入 CiRXFnSID 寄存器中，或将 *eid* 中的 32 位值写入与由 *filter_no* 指定的过滤器相对应的 CiRXFnEIDH 和 CiRXFnEIDL 寄存器中。
默认为过滤器 0。

源文件：
CAN1SetFilter.c
CAN2SetFilter.c

代码示例：
```
CAN1SetFilter(1, CAN_FILTER_SID(7) &
CAN_RX_EID_EN, CAN_FILTER_EID(3));
```

CAN1SetMask
CAN2SetMask

描述： 该函数为指定的屏蔽器设置接收屏蔽器值 （SID 和 EID）。

头文件： can.h

函数原型：
```
void CAN1SetMask(char mask_no, unsigned int sid,
 unsigned long eid);
void CAN2SetMask(char mask_no, unsigned int sid,
 unsigned long eid);
```

参数：

mask_no	要配置屏蔽值的屏蔽器 （“0”或 “1”）。
sid	要写入 CiRXMnSID 寄存器的 16 位值。 CAN_MASK_SID(x) x 是所需的 SID 值。 过滤器中指定的匹配 / 忽略消息类型 CAN_MATCH_FILTER_TYPE CAN_IGNORE_FILTER_TYPE
eid	要写入 CiRXMnEIDH 和 CiRXMnEIDL 寄存器的 32 位值。 CAN_MASK_EID(x) x 是所需的 EID 值。

返回值： 无

CAN1SetMask（续）
CAN2SetMask

说明： 该函数将 *sid* 的 16 位值写入 CiRXFnSID 寄存器，或者将 *eid* 的 32 位值写入与由 *mask_no* 指定的屏蔽器相对应的 CiRXFnEIDH 和 CiRXFnEIDL 寄存器。
默认为屏蔽器 0。

源文件：
```
CAN1SetMask.c
CAN2SetMask.c
```

代码示例：
```
CAN1SetMask(1, CAN_MASK_SID(7) &
CAN_MATCH_FILTER_TYPE, CAN_MASK_EID(3));
```

CAN1SetOperationMode
CAN2SetOperationMode

描述： 该函数配置 CAN 模块

头文件：
```
can.h
```

函数原型：
```
void CAN1SetOperationMode(unsigned int config);
void CAN2SetOperationMode(unsigned int config);
```

参数： *config* 要装入 CiCTRL 寄存器的 16 位值，为以下定义的组合。

```
CAN_IDLE_CON    在空闲模式下 CAN 启用
CAN_IDLE_STOP   在空闲模式下 CAN 停止

CAN_MASTERCLOCK_1 FCAN 为 FCY
CAN_MASTERCLOCK_0 FCAN 为 4 FCY

CAN 操作模式
CAN_REQ_OPERMODE_NOR
CAN_REQ_OPERMODE_DIS
CAN_REQ_OPERMODE_LOOPBK
CAN_REQ_OPERMODE_LISTENONLY
CAN_REQ_OPERMODE_CONFIG
CAN_REQ_OPERMODE_LISTENALL

CAN 捕捉使能 / 禁止
CAN_CAPTURE_EN
CAN_CAPTURE_DIS
```

返回值： 无

说明： 该函数对 CiCTRL 寄存器中下面的位进行配置：CSIDL、 REQOP<2:0> 和 CANCKS。

源文件：
```
CAN1SetOperationMode.c
CAN2SetOperationMode.c
```

代码示例：
```
CAN1SetOperationMode(CAN_IDLE_STOP &
CAN_MASTERCLOCK_0 & CAN_REQ_OPERMODE_DIS &
CAN_CAPTURE_DIS);
```

CAN1SetOperationModeNoWait
CAN2SetOperationModeNoWait

描述： 该函数中止等待的发送并对 CAN 模块进行配置。

头文件： can.h

函数原型：
```
void CAN1SetOperationModeNoWait(
 unsigned int config);
void CAN2SetOperationModeNoWait(
 unsigned int config);
```

参数： *config* 要装入 CiCTRL 寄存器的 16 位值，为以下定义的组合。

CAN_IDLE_CON_NO_WAIT 在空闲模式下 CAN 启用
CAN_IDLE_STOP_NO_WAIT 在空闲模式下 CAN 停止

CAN_MASTERCLOCK_1_NO_WAIT FCAN 为 FCY
CAN_MASTERCLOCK_0_NO_WAIT FCAN 为 4 FCY

CAN 操作模式
CAN_REQ_OPERMODE_NOR_NO_WAIT
CAN_REQ_OPERMODE_DIS_NO_WAIT
CAN_REQ_OPERMODE_LOOPBK_NO_WAIT
CAN_REQ_OPERMODE_LISTENONLY_NO_WAIT
CAN_REQ_OPERMODE_CONFIG_NO_WAIT
CAN_REQ_OPERMODE_LISTENALL_NO_WAIT

CAN 捕捉使能 / 禁止
CAN_CAPTURE_EN_NO_WAIT
CAN_CAPTURE_DIS_NO_WAIT

返回值： 无

说明： 该函数对 ABAT 位进行置位来中止所有等待的发送，并对 CiCTRL 寄存器中下面的位进行配置：CSIDL、 REQOP<2:0> 和 CANCKS。

源文件：
CAN1SetOperationModeNoWait.c
CAN2SetOperationModeNoWait.c

代码示例：
```
CAN1SetOperationModeNoWait(CAN_IDLE_CON &
CAN_MASTERCLOCK_1 & CAN_REQ_OPERMODE_LISTEN &
CAN_CAPTURE_DIS_NO_WAIT);
```

CAN1SetRXMode
CAN2SetRXMode

描述： 该函数对 CAN 接收器进行配置。

头文件： can.h

函数原型：
```
void CAN1SetRXMode(char buffno, unsigned int
 config);
void CAN2SetRXMode(char buffno, unsigned int
 config);
```

参数： *buffno* buffno 指明要配置的控制寄存器。

config 要写入 CiRXnCON 寄存器的值，为以下定义的组合。

清零 RXFUL 位
CAN_RXFUL_CLEAR

双缓冲器使能 / 禁止
CAN_BUF0_DBLBUFFER_EN
CAN_BUF0_DBLBUFFER_DIS

CAN1SetRXMode （续）
CAN2SetRXMode

返回值： 无

说明： 该函数对 CiRXnCON 寄存器下面的位进行配置：
RXRTR、 RXFUL （只能设置为 0）、 RXM<1:0> 和 DBEN。

源文件：
```
CAN1SetRXMode.c
CAN2SetRXMode.c
```

代码示例：
```
CAN1SetRXMode(0,CAN_RXFUL_CLEAR &
CAN_BUF0_DBLBUFFER_EN);
```

CAN1SetTXMode (function)
CAN2SetTXMode

描述： 该函数配置 CAN 发送器模块。

头文件：
```
can.h
```

函数原型：
```
void CAN1SetTXMode(char buffno, unsigned int
 config);
void CAN2SetTXMode(char buffno, unsigned int
 config);
```

参数：

buffno buffno 指明要配置的控制寄存器。

config 要写入 CiTXnCON 寄存器的值，为以下定义的组合。

消息发送请求
```
CAN_TX_REQ
CAN_TX_STOP_REQ
```
消息发送优先级
```
CAN_TX_PRIORITY_HIGH
CAN_TX_PRIORITY_HIGH_INTER
CAN_TX_PRIORITY_LOW_INTER
CAN_TX_PRIORITY_LOW
```

返回值： 无

说明： 该函数对 CiTXnCON 寄存器下面的位进行配置：
TXRTR、 TXREQ、 DLC 和 TXPRI<1:0>。

源文件：
```
CAN1SetTXMode.c
CAN2SetTXMode.c
```

代码示例：
```
CAN1SetTXMode(1, CAN_TX_STOP_REQ &
CAN_TX_PRIORITY_HIGH);
```

CAN1Initialize
CAN2Initialize

描述： 该函数配置 CAN 模块。

头文件： can.h

函数原型：
```
void CAN1Initialize(unsigned int config1,
 unsigned int config2);
void CAN2Initialize(unsigned int config1,
 unsigned int config2);
```

参数： *config1* 要写入 CiCFG1 寄存器的值，为以下定义的组合。

同步跳转宽度
CAN_SYNC_JUMP_WIDTH1
CAN_SYNC_JUMP_WIDTH2
CAN_SYNC_JUMP_WIDTH3
CAN_SYNC_JUMP_WIDTH4

波特率预分频比
CAN_BAUD_PRE_SCALE(x) (((x-1) & 0x3f) | 0xC0)

config2 要写入 CiCFG2 寄存器的值，为以下定义的组合。

选择 CAN 总线滤波器用于唤醒
CAN_WAKEUP_BY_FILTER_EN
CAN_WAKEUP_BY_FILTER_DIS

CAN 传播段长度
CAN_PROPAGATIONTIME_SEG_TQ(x)
(((x-1) & 0x7) | 0xC7F8)

CAN 相位段 1 的长度
CAN_PHASE_SEG1_TQ(x)
((((x-1) & 0x7) *0x8) | 0xC7C7)

CAN 相位段 2 的长度
CAN_PHASE_SEG2_TQ(x)
((((x-1) & 0x7) *0x100) | 0xC0FF)

CAN 相位段 2 模式
CAN_SEG2_FREE_PROG
CAN_SEG2_TIME_LIMIT_SET

CAN 总线采样
CAN_SAMPLE3TIMES
CAN_SAMPLE1TIME

返回值： 无

说明： 该函数对寄存器 CiCFG1 和 CiCFG2 的以下位进行配置：
SJW<1:0>、 BRP<5:0>、 CANCAP、 WAKEFIL、 SEG2PH<2:0>、 SEGPHTS、 SAM、 SEG1PH<2:0> 和 PRSEG<2:0> 。

源文件： CAN1Initialize.c
CAN2Initialize.c

代码示例：
```
CAN1Initialize(CAN_SYNC_JUMP_WIDTH2 &
CAN_BAUD_PRE_SCALE(2),
CAN_WAKEUP_BY_FILTER_DIS &
CAN_PHASE_SEG2_TQ(5) &
CAN_PHASE_SEG1_TQ(4) &
CAN_PROPAGATIONTIME_SEG_TQ(4) &
CAN_SEG2_FREE_PROG &
CAN_SAMPLE1TIME);
```

ConfigIntCAN1
ConfigIntCAN2

描述： 该函数对 CAN 中断进行配置。

头文件： can.h

函数原型：
```
void ConfigIntCAN1(unsigned int config1,
unsigned int config2);
void ConfigIntCAN2(unsigned int config1,
unsigned int config2);
```

参数：

config1 定义如下的各个中断允许 / 禁止信息：
用户必须选择对所有各中断进行允许或禁止。

中断允许
```
CAN_INDI_INVMESS_EN
CAN_INDI_WAK_EN
CAN_INDI_ERR_EN
CAN_INDI_TXB2_EN
CAN_INDI_TXB1_EN
CAN_INDI_TXB0_EN
CAN_INDI_RXB1_EN
CAN_INDI_RXB0_EN
```

中断禁止
```
CAN_INDI_INVMESS_DIS
CAN_INDI_WAK_DIS
CAN_INDI_ERR_DIS
CAN_INDI_TXB2_DIS
CAN_INDI_TXB1_DIS
CAN_INDI_TXB0_DIS
CAN_INDI_RXB1_DIS
CAN_INDI_RXB0_DIS
```

config2 定义如下的 CAN 中断优先级和允许 / 禁止信息：

CAN 中断允许 / 禁止
```
CAN_INT_ENABLE
CAN_INT_DISABLE
```

CAN 中断优先级
```
CAN_INT_PRI_0
CAN_INT_PRI_1
CAN_INT_PRI_2
CAN_INT_PRI_3
CAN_INT_PRI_4
CAN_INT_PRI_5
CAN_INT_PRI_6
CAN_INT_PRI_7
```

返回值： 无

说明： 该函数对 CAN 中断进行配置。它允许 / 禁止各个 CAN 中断。它还可以允许 / 禁止 CAN 中断并设置优先级。

源文件：
```
ConfigIntCAN1.c
ConfigIntCAN2.c
```

ConfigIntCAN1 （续）
ConfigIntCAN2

代码示例：

```
ConfigIntCAN1(CAN_INDI_INVMESS_EN &
              CAN_INDI_WAK_DIS &
              CAN_INDI_ERR_DIS &
              CAN_INDI_TXB2_DIS  &
              CAN_INDI_TXB1_DIS  &
              CAN_INDI_TXB0_DIS &
              CAN_INDI_RXB1_DIS &
              CAN_INDI_RXB0_DIS ,
              CAN_INT_PRI_3 &
              CAN_INT_ENABLE);
```

3.4.2 各个宏

EnableIntCAN1
EnableIntCAN2

描述： 该宏允许 CAN 中断。

头文件： can.h

参数： 无

说明： 该宏置位中断允许控制寄存器的 CAN 中断允许位。

代码示例：

```
EnableIntCAN1;
```

DisableIntCAN1
DisableIntCAN2

描述： 该宏禁止 CAN 中断。

头文件： can.h

参数： 无

说明： 该宏清零中断允许控制寄存器的 CAN 中断允许位。

代码示例：

```
DisableIntCAN2;
```

SetPriorityIntCAN1
SetPriorityIntCAN2

描述： 该宏设置 CAN 中断的优先级。

头文件： can.h

参数： *priority*

说明： 该宏对中断优先级控制寄存器的 CAN 中断优先位进行设置。

代码示例：

```
SetPriorityIntCAN1(2);
```

3.4.3 使用示例

```
#define __dsPIC30F6014__
#include<p30fxxxx.h>
#include<can.h>
#define  dataarray 0x1820
int main(void)
{
    /* Length of data to be transmitted/read  */
    unsigned char datalen;
    unsigned char Txdata[] =
     {'M','I','C','R','O','C','H','I','P','\0'};
    unsigned int TXConfig, RXConfig;
    unsigned long MaskID,MessageID;
    char FilterNo,tx_rx_no;
    unsigned char * datareceived = (unsigned char *)
        dataarray;  /* Holds the data received */
    /* Set request for configuration mode */
    CAN1SetOperationMode(CAN_IDLE_CON  &
                         CAN_MASTERCLOCK_1 &
                         CAN_REQ_OPERMODE_CONFIG &
                         CAN_CAPTURE_DIS);
    while(C1CTRLbits.OPMODE <=3);
    /* Load configuration register */
    CAN1Initialize(CAN_SYNC_JUMP_WIDTH2 &
                   CAN_BAUD_PRE_SCALE(2),
                   CAN_WAKEUP_BY_FILTER_DIS  &
                   CAN_PHASE_SEG2_TQ(5) &
                   CAN_PHASE_SEG1_TQ(4) &
                   CAN_PROPAGATIONTIME_SEG_TQ(4) &
                   CAN_SEG2_FREE_PROG &
                   CAN_SAMPLE1TIME);
    /* Load Acceptance filter register */
    FilterNo = 0;
    CAN1SetFilter(FilterNo, CAN_FILTER_SID(1920) &
                  CAN_RX_EID_EN, CAN_FILTER_EID(12345));
    /* Load mask filter register */
    CAN1SetMask(FilterNo, CAN_MASK_SID(1920) &
                CAN_MATCH_FILTER_TYPE, CAN_MASK_EID(12344));
    /* Set transmitter and receiver mode */
    tx_rx_no = 0;
    CAN1SetTXMode(tx_rx_no,
                  CAN_TX_STOP_REQ &
                  CAN_TX_PRIORITY_HIGH );
    CAN1SetRXMode(tx_rx_no,
                  CAN_RXFUL_CLEAR &
                  CAN_BUF0_DBLBUFFER_EN);
    /* Load message ID , Data into transmit buffer and set
       transmit request bit */
    datalen = 8;
    CAN1SendMessage((CAN_TX_SID(1920)) & CAN_TX_EID_EN &
                    CAN_SUB_NOR_TX_REQ,
                   (CAN_TX_EID(12344)) & CAN_NOR_TX_REQ,
                    Txdata,datalen,tx_rx_no);
```

```
    /* Set request for Loopback mode */
     CAN1SetOperationMode(CAN_IDLE_CON & CAN_CAPTURE_DIS &
                          CAN_MASTERCLOCK_1 &
                          CAN_REQ_OPERMODE_LOOPBK);
     while(C1CTRLbits.OPMODE !=2);
     /* Wait till message is transmitted completely */
     while(!CAN1IsTXReady(0))
     /* Wait till receive buffer contain valid  message */
     while(!CAN1IsRXReady(0));
     /* Read received data from receive buffer and store it into
       user defined dataarray */
     CAN1ReceiveMessage(datareceived, datalen, tx_rx_no);
     while(1);
     return 0;
}
```

3.5 ADC12 函数

本节给出了关于 12 位 ADC 的各个函数及其使用示例。这些函数也可以用宏来实现。

3.5.1 各个函数

BusyADC12

描述： 该函数返回 ADC 转换的状态。

头文件： adc12.h

函数原型：
```
char BusyADC12(void);
```

参数： 无

返回值： 如果 DONE 的值为 “0”，那么返回 “1 “，表明 ADC 正忙于转换。如果 DONE 的值为 “1”，那么返回 “0”，表明 ADC 已经完成了转换。

说明： 该函数返回 ADCON1 <DONE> 位状态的反码，表明 ADC 是否正忙于转换。

源文件： BusyADC12.c

代码示例：
```
while(BusyADC12());
```

CloseADC12

描述： 该函数关闭 ADC 模块并禁止 ADC 中断。

头文件： adc12.h

函数原型：
```
void CloseADC12(void);
```

参数： 无

返回值： 无

说明： 该函数首先禁止 ADC 中断，然后关闭 ADC 模块。同时清除中断标志位（ADIF）。

源文件： CloseADC12.c

代码示例：
```
CloseADC12();
```

ConfigIntADC12

描述： 该函数配置 ADC 中断。

头文件： adc12.h

函数原型：
```
void ConfigIntADC12(unsigned int config);
```

参数： *config* 如下定义的 ADC 中断优先级和允许 / 禁止信息：

ADC 中断允许 / 禁止
ADC_INT_ENABLE
ADC_INT_DISABLE

ConfigIntADC12 （续）

ADC 中断优先级
ADC_INT_PRI_0
ADC_INT_PRI_1
ADC_INT_PRI_2
ADC_INT_PRI_3
ADC_INT_PRI_4
ADC_INT_PRI_5
ADC_INT_PRI_6
ADC_INT_PRI_7

返回值： 无

说明： 该函数清除中断标志位 （ADIF），然后设置中断优先级并允许 / 禁止中断。

源文件： ConfigIntADC12.c

代码示例：
```
ConfigIntADC12(ADC_INT_PRI_6 &
               ADC_INT_ENABLE);
```

ConvertADC12

描述： 该函数启动 A/D 转换。

头文件： adc12.h

函数原型：
```
void ConvertADC12(void);
```

参数： 无

返回值： 无

说明： 该函数通过清除 ADCON1 的 <SAMP> 位来停止采样并启动转换。
这只有在通过清除 ADCON1 的 <SSRC> 位选择 A/D 转换触发源为手动时才发生。

源文件： ConvertADC12.c

代码示例：
```
ConvertADC12();
```

OpenADC12

描述： 该函数对 ADC 进行配置。

头文件： adc12.h

函数原型：
```
void OpenADC12(unsigned int config1,
               unsigned int config2,
               unsigned int config3,
               unsigned int configport,
               unsigned int configscan)
```

参数： *config1* 包含 ADCON1 寄存器中要配置的参数，定义如下：

模块启用 / 关闭
ADC_MODULE_ON
ADC_MODULE_OFF

空闲模式工作
ADC_IDLE_CONTINUE
ADC_IDLE_STOP

OpenADC12 （续）

结果输出格式
ADC_FORMAT_SIGN_FRACT
ADC_FORMAT_FRACT
ADC_FORMAT_SIGN_INT
ADC_FORMAT_INTG

转换触发源
ADC_CLK_AUTO
ADC_CLK_TMR
ADC_CLK_INT0
ADC_CLK_MANUAL

自动采样选择
ADC_AUTO_SAMPLING_ON
ADC_AUTO_SAMPLING_OFF

采样使能
ADC_SAMP_ON
ADC_SAMP_OFF

config2 包含 ADCON2 寄存器中要配置的参数，定义如下：

参考电压
ADC_VREF_AVDD_AVSS
ADC_VREF_EXT_AVSS
ADC_VREF_AVDD_EXT
ADC_VREF_EXT_EXT

扫描选择
ADC_SCAN_ON
ADC_SCAN_OFF

中断之间的采样次数
ADC_SAMPLES_PER_INT_1
ADC_SAMPLES_PER_INT_2
.....
ADC_SAMPLES_PER_INT_15
ADC_SAMPLES_PER_INT_16

缓冲器模式选择
ADC_ALT_BUF_ON
ADC_ALT_BUF_OFF

备用输入采样模式选择
ADC_ALT_INPUT_ON
ADC_ALT_INPUT_OFF

config3 包含 ADCON3 寄存器中要配置的参数，定义如下：

自动采样时间位
ADC_SAMPLE_TIME_0
ADC_SAMPLE_TIME_1
.....
ADC_SAMPLE_TIME_30
ADC_SAMPLE_TIME_31

转换时钟源选择
ADC_CONV_CLK_INTERNAL_RC
ADC_CONV_CLK_SYSTEM

OpenADC12 （续）

转换时钟选择
ADC_CONV_CLK_Tcy2
ADC_CONV_CLK_Tcy
ADC_CONV_CLK_3Tcy2
.....
ADC_CONV_CLK_32Tcy

configport 包含 ADPCFG 寄存器中要配置的引脚选择参数，定义如下：

ENABLE_ALL_ANA
ENABLE_ALL_DIG
ENABLE_AN0_ANA
ENABLE_AN1_ANA
ENABLE_AN2_ANA
.....
ENABLE_AN15_ANA

configscan 包含 ADCSSL 寄存器中要配置的扫描选择参数，定义如下：

SCAN_NONE
SCAN_ALL
SKIP_SCAN_AN0
SKIP_SCAN_AN1
.....
SKIP_SCAN_AN15

返回值： 无

说明： 该函数对 ADC 的以下参数进行配置：
工作模式、休眠模式下的操作、数据输出格式、采样时钟源、参考电压源 VREF、采样 / 中断数、缓冲器填充模式、备用输入采样模式、自动采样时间、转换时钟源、转换时钟选择位、端口配置控制位。

源文件： OpenADC12.c

代码示例：

```
OpenADC12(ADC_MODULE_OFF &
          ADC_IDLE_CONTINUE &
          ADC_FORMAT_INTG &
          ADC_AUTO_SAMPLING_ON,
          ADC_VREF_AVDD_AVSS &
          ADC_SCAN_OFF &
          ADC_BUF_MODE_OFF &
          ADC_ALT_INPUT_ON &
          ADC_SAMPLES_PER_INT_15,
          ADC_SAMPLE_TIME_4 &
          ADC_CONV_CLK_SYSTEM &
          ADC_CONV_CLK_Tcy,
          ENABLE_AN0_ANA,
          SKIP_SCAN_AN1 &
          SKIP_SCAN_AN2 &
          SKIP_SCAN_AN5 &
          SKIP_SCAN_AN7);
```

ReadADC12

描述： 该函数读取 ADC 缓冲寄存器中包含的转换值。

头文件： adc12.h

函数原型： unsigned int ReadADC12(unsigned char *bufIndex*);

参数： *bufIndex* 要读取的 ADC 缓冲器的编号。

返回值： 无

说明： 该函数返回 ADC 缓冲寄存器的内容。用户提供 *bufIndex* 的值（0-15）来确保正确读取 ADCBUF0 到 ADCBUFF 寄存器的内容。

源文件： ReadADC12.c

代码示例：

```
unsigned int result;
result = ReadADC12(5);
```

StopSampADC12

描述： 该函数等同于 ConvertADC12。

源文件： dc12.h 中包含 ConvertADC12 的宏定义。

SetChanADC12

描述： 该函数为采样多路开关 A 和 B 设置正、负输入。

头文件： adc12.h

函数原型： void SetChanADC12(unsigned int *channel*);

参数： *channel* 包括 ADCHS 寄存器中要配置的输入选择参数，定义如下：

采样 A 选择为 A/D 通道 0 正输入
ADC_CH0_POS_SAMPLEA_AN0
ADC_CH0_POS_SAMPLEA_AN1
.....
ADC_CH0_POS_SAMPLEA_AN15

采样 A 选择为 A/D 通道 0 负输入
ADC_CH0_NEG_SAMPLEA_AN1
ADC_CH0_NEG_SAMPLEA_NVREF

采样 B 选择为 A/D 通道 0 正输入
ADC_CH0_POS_SAMPLEB_AN0
ADC_CH0_POS_SAMPLEB_AN1
.....
ADC_CH0_POS_SAMPLEB_AN15

采样 B 选择为 A/D 通道 0 负输入
ADC_CH0_NEG_SAMPLEB_AN1
ADC_CH0_NEG_SAMPLEB_NVREF

返回值： 无

说明： 该函数通过写 ADCHS 寄存器来配置采样多路开关 A 和 B 的输入。

源文件： SetChanADC12.c

代码示例：

```
SetChanADC12(ADC_CH0_POS_SAMPLEA_AN4 &
             ADC_CH0_NEG_SAMPLEA_NVREF);
```

3.5.2 各个宏

EnableIntADC

描述： 该宏允许 ADC 中断。
头文件： adc12.h
参数： 无
说明： 该宏置位中断允许控制寄存器的 ADC 中断允许位。
代码示例：
```
EnableIntADC;
```

DisableIntADC

描述： 该宏禁止 ADC 中断。
头文件： adc12.h
参数： 无
说明： 该宏清除中断允许控制寄存器的 ADC 中断允许位。
代码示例：
```
DisableIntADC;
```

SetPriorityIntADC

描述： 该宏设置 ADC 中断的优先级。
头文件： adc12.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的 ADC 中断优先级位。
代码示例：
```
SetPriorityIntADC(6);
```

3.5.3 使用示例

```
#define __dsPIC30F6014__
#include <p30fxxxx.h>
#include<adc12.h>
unsigned int Channel, PinConfig, Scanselect, Adcon3_reg, Adcon2_reg,
Adcon1_reg;
int  main(void)
{
    unsigned int result[20], i;
    ADCON1bits.ADON = 0;         /* turn off ADC */
    Channel = ADC_CH0_POS_SAMPLEA_AN4 &
              ADC_CH0_NEG_SAMPLEA_NVREF &
              ADC_CH0_POS_SAMPLEB_AN2&
              ADC_CH0_NEG_SAMPLEB_AN1;
    SetChanADC12(Channel);
    ConfigIntADC12(ADC_INT_DISABLE);
    PinConfig  = ENABLE_AN4_ANA;
    Scanselect = SKIP_SCAN_AN2 & SKIP_SCAN_AN5 &
                 SKIP_SCAN_AN9 & SKIP_SCAN_AN10 &
                 SKIP_SCAN_AN14 & SKIP_SCAN_AN15 ;

    Adcon3_reg = ADC_SAMPLE_TIME_10 &
                 ADC_CONV_CLK_SYSTEM &
                 ADC_CONV_CLK_13Tcy;

    Adcon2_reg = ADC_VREF_AVDD_AVSS &
                 ADC_SCAN_OFF &
                 ADC_ALT_BUF_OFF &
                 ADC_ALT_INPUT_OFF &
                 ADC_SAMPLES_PER_INT_16;
    Adcon1_reg = ADC_MODULE_ON &
                 ADC_IDLE_CONTINUE &
                 ADC_FORMAT_INTG &
                 ADC_CLK_MANUAL &
                 ADC_AUTO_SAMPLING_OFF;
    OpenADC12(Adcon1_reg, Adcon2_reg,
              Adcon3_reg,PinConfig, Scanselect);
    i = 0;
    while( i <16 )
    {
        ADCON1bits.SAMP = 1;
        while(!ADCON1bits.SAMP);
        ConvertADC12();
        while(ADCON1bits.SAMP);
        while(!BusyADC12());
        while(BusyADC12());
        result[i] = ReadADC12(i);
        i++;
    }
}
```

3.6 ADC10 函数

本节给出了关于 10 位 ADC 的各个函数及其使用示例。函数也可以用宏来实现。

3.6.1 各个函数

BusyADC10

描述： 该函数返回 ADC 转换的状态。

头文件： adc10.h

函数原型： `char BusyADC10(void);`

参数： 无

返回值： 如果 DONE 的值为 “0”，则返回 “1”，表明 ADC 正忙于转换。
如果 DONE 的值为 “1”，则返回 “0”，表明 ADC 已完成转换。

说明： 该函数返回 ADCON1 <DONE> 位状态的反码，表明 ADC 是否正忙于转换。

源文件： BusyADC10.c

代码示例：
```
while(BusyADC10());
```

CloseADC10

描述： 该函数关闭 ADC 模块并禁止 ADC 中断。

头文件： adc10.h

函数原型： `void CloseADC10(void);`

参数： 无

返回值： 无

说明： 该函数首先禁止 ADC 中断，然后关闭 ADC 模块。同时清除中断标志位（ADIF）。

源文件： CloseADC10.c

代码示例：
```
CloseADC10();
```

ConfigIntADC10

描述： 该函数配置 ADC 中断。

头文件： adc10.h

函数原型： `void ConfigIntADC10(unsigned int config);`

参数： *config* 如下定义的 ADC 中断优先级和允许 / 禁止信息：

ADC 中断允许 / 禁止
ADC_INT_ENABLE
ADC_INT_DISABLE

ADC 中断优先级
ADC_INT_PRI_0
ADC_INT_PRI_1
ADC_INT_PRI_2
ADC_INT_PRI_3
ADC_INT_PRI_4
ADC_INT_PRI_5
ADC_INT_PRI_6
ADC_INT_PRI_7

ConfigIntADC10 （续）

返回值： 无

说明： 该函数清除中断标志位 （ADIF），然后设置中断优先级并允许 / 禁止中断。

源文件： ConfigIntADC10.c

代码示例：
```
ConfigIntADC10(ADC_INT_PRI_3 &
               ADC_INT_DISABLE);
```

ConvertADC10

描述： 该函数启动 A/D 转换。

头文件： adc10.h

函数原型：
```
void ConvertADC10(void);
```

参数： 无

返回值： 无

说明： 该函数通过清除 ADCON1 的 <SAMP> 位来停止采样并启动转换。
这只有在通过清除 ADCON1 的 <SSRC> 位来选择 A/D 转换的触发源为手动时才发生。

源文件： ConvertADC10.c

代码示例：
```
ConvertADC10();
```

OpenADC10

描述： 该函数对 ADC 进行配置。

头文件： adc10.h

函数原型：
```
void OpenADC10(unsigned int config1,
               unsigned int config2,
               unsigned int config3,
               unsigned int configport,
               unsigned int configscan)
```

参数： *config1* 包含 ADCON1 寄存器中要配置的参数，定义如下：

模块启用 / 关闭
ADC_MODULE_ON
ADC_MODULE_OFF

空闲模式工作
ADC_IDLE_CONTINUE
ADC_IDLE_STOP

结果输出格式
ADC_FORMAT_SIGN_FRACT
ADC_FORMAT_FRACT
ADC_FORMAT_SIGN_INT
ADC_FORMAT_INTG

转换触发源
ADC_CLK_AUTO
ADC_CLK_MPWM
ADC_CLK_TMR
ADC_CLK_INT0
ADC_CLK_MANUAL

OpenADC10 （续）

自动采样选择
ADC_AUTO_SAMPLING_ON
ADC_AUTO_SAMPLING_OFF

同时采样
ADC_SAMPLE_SIMULTANEOUS
ADC_SAMPLE_INDIVIDUAL

采样使能
ADC_SAMP_ON
ADC_SAMP_OFF

config2 包含 ADCON2 寄存器中要配置的参数，定义如下：

参考电压
ADC_VREF_AVDD_AVSS
ADC_VREF_EXT_AVSS
ADC_VREF_AVDD_EXT
ADC_VREF_EXT_EXT

扫描选择
ADC_SCAN_ON
ADC_SCAN_OFF

使用的 A/D 通道
ADC_CONVERT_CH0123
ADC_CONVERT_CH01
ADC_CONVERT_CH0

中断之间的采样次数
ADC_SAMPLES_PER_INT_1
ADC_SAMPLES_PER_INT_2
.....
ADC_SAMPLES_PER_INT_15
ADC_SAMPLES_PER_INT_16

缓冲器模式选择
ADC_ALT_BUF_ON
ADC_ALT_BUF_OFF

备用输入采样模式选择
ADC_ALT_INPUT_ON
ADC_ALT_INPUT_OFF

config3 包含 ADCON3 寄存器中要配置的参数，定义如下：

自动采样时间位
ADC_SAMPLE_TIME_0
ADC_SAMPLE_TIME_1
.....
ADC_SAMPLE_TIME_30
ADC_SAMPLE_TIME_31

转换时钟源选择
ADC_CONV_CLK_INTERNAL_RC
ADC_CONV_CLK_SYSTEM

转换时钟选择
ADC_CONV_CLK_Tcy2
ADC_CONV_CLK_Tcy
ADC_CONV_CLK_3Tcy2
.....
ADC_CONV_CLK_32Tcy

OpenADC10 （续）

configport 包含 ADPCFG 寄存器中要配置的引脚选择参数，定义如下：

```
ENABLE_ALL_ANA
ENABLE_ALL_DIG
ENABLE_AN0_ANA
ENABLE_AN1_ANA
ENABLE_AN2_ANA
.....
ENABLE_AN15_ANA
```

configscan 包含 ADCSSL 寄存器中要配置的扫描选择参数，定义如下：

```
SCAN_NONE
SCAN_ALL
SKIP_SCAN_AN0
SKIP_SCAN_AN1
.....
SKIP_SCAN_AN15
```

返回值： 无

说明： 该函数对 ADC 的以下参数进行配置：
工作模式、休眠模式下的工作、数据输出格式、采样时钟源、参考电压源 VREF、采样 / 中断数、缓冲器填充模式、备用输入采样模式、自动采样时间、转换时钟源、转换时钟选择位、端口配置控制位。

源文件： OpenADC10.c

代码示例：

```
OpenADC10(ADC_MODULE_OFF &
          ADC_IDLE_STOP &
          ADC_FORMAT_SIGN_FRACT &
          ADC_CLK_INT0 &
          ADC_SAMPLE_INDIVIDUAL &
          ADC_AUTO_SAMPLING_ON,
          ADC_VREF_AVDD_AVSS &
          ADC_SCAN_OFF &
          ADC_BUF_MODE_OFF &
          ADC_ALT_INPUT_ON &
          ADC_CONVERT_CH0 &
          ADC_SAMPLES_PER_INT_10,
          ADC_SAMPLE_TIME_4 &
          ADC_CONV_CLK_SYSTEM &
          ADC_CONV_CLK_Tcy,
          ENABLE_AN1_ANA,
          SKIP_SCAN_AN0 &
          SKIP_SCAN_AN3 &
          SKIP_SCAN_AN4 &
          SKIP_SCAN_AN5);
```

ReadADC10

描述： 该函数读取 ADC 缓冲寄存器中包含的转换值。

头文件： adc10.h

函数原型： unsigned int ReadADC10(unsigned char *bufIndex*);

参数： *bufIndex* 要读取的 ADC 缓冲器的编号。

返回值： 无

说明： 该函数返回 ADC 缓冲寄存器的内容。用户要提供 *bufIndex* 的值（0-15）来确保正确读取 ADCBUF0 到 ADCBUFF 寄存器的内容。

源文件： ReadADC10.c

代码示例：

```
unsigned int result;
result = ReadADC10(3);
```

StopSampADC10

描述： 该函数等同于 ConvertADC10。

源文件： adc10.h 中包含对 ConvertADC10 的宏定义。

SetChanADC10

描述： 该函数为采样多路开关 A 和 B 设置正、负输入。

头文件： adc10.h

函数原型： void SetChanADC10(unsigned int *channel*);

参数： *channel* 包括 ADCHS 寄存器中要配置的输入选择参数，定义如下：

选择采样 A 为 A/D 通道 1，2，3 负输入
ADC_CHX_NEG_SAMPLEA_AN9AN10AN11
ADC_CHX_NEG_SAMPLEA_AN6AN7AN8
ADC_CHX_NEG_SAMPLEA_NVREF

选择采样 B 为 A/D 通道 1，2，3 负输入
ADC_CHX_NEG_SAMPLEB_AN9AN10AN11
ADC_CHX_NEG_SAMPLEB_AN6AN7AN8
ADC_CHX_NEG_SAMPLEB_NVREF

选择采样 A 为 A/D 通道 1，2，3 正输入
ADC_CHX_POS_SAMPLEA_AN3AN4AN5
ADC_CHX_POS_SAMPLEA_AN0AN1AN2

选择采样 B 为 A/D 通道 1，2，3 正输入
ADC_CHX_POS_SAMPLEA_AN3AN4AN5
ADC_CHX_POS_SAMPLEB_AN0AN1AN2

选择采样 A 为 A/D 通道 0 正输入
ADC_CH0_POS_SAMPLEA_AN0
ADC_CH0_POS_SAMPLEA_AN1
.....
ADC_CH0_POS_SAMPLEA_AN15

选择采样 A 为 A/D 通道 0 负输入
ADC_CH0_NEG_SAMPLEA_AN1
ADC_CH0_NEG_SAMPLEA_NVREF

SetChanADC10 （续）

选择采样 B 为 A/D 通道 0 正输入
ADC_CH0_POS_SAMPLEB_AN0
ADC_CH0_POS_SAMPLEB_AN1
.....
ADC_CH0_POS_SAMPLEB_AN15

选择采样 B 为 A/D 通道 0 负输入
ADC_CH0_NEG_SAMPLEB_AN1
ADC_CH0_NEG_SAMPLEB_NVREF

返回值： 无

说明： 该函数通过写入 ADCHS 寄存器来对采样多路开关 A 和 B 的输入进行配置。

源文件： SetChanADC10.c

代码示例：
```
SetChanADC10(ADC_CH0_POS_SAMPLEA_AN0 &
             ADC_CH0_NEG_SAMPLEA_NVREF);
```

3.6.2 各个宏

EnableIntADC

描述： 该宏允许 ADC 中断。

头文件： adc10.h

参数： 无

说明： 该宏中文中断允许控制寄存器的 ADC 中断允许位。

代码示例：
```
EnableIntADC;
```

DisableIntADC

描述： 该宏禁止 ADC 中断。

头文件： adc10.h

参数： 无

说明： 该宏清除中断允许控制寄存器的 ADC 中断允许位。

代码示例：
```
DisableIntADC；
```

SetPriorityIntADC

描述： 该宏设置 ADC 中断的优先级。

头文件： adc10.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器中的 ADC 中断优先级位。

代码示例：
```
SetPriorityIntADC(2);
```

3.6.3 使用示例

```
#define __dsPIC30F6010__
#include <p30fxxxx.h>
#include<adc10.h>
unsigned int Channel, PinConfig, Scanselect, Adcon3_reg, Adcon2_reg,
Adcon1_reg;
int  main(void)
{
    unsigned int result[20], i;
    ADCON1bits.ADON = 0;         /* turn off ADC */
    Channel = ADC_CH0_POS_SAMPLEA_AN4 &
              ADC_CH0_NEG_SAMPLEA_NVREF &
              ADC_CH0_POS_SAMPLEB_AN2 &
              ADC_CH0_NEG_SAMPLEB_AN1;
    SetChanADC1(Channel);

    ConfigIntADC10(ADC_INT_DISABLE);
    PinConfig  = ENABLE_AN4_ANA;
    Scanselect = SKIP_SCAN_AN2 & SKIP_SCAN_AN5 &
                 SKIP_SCAN_AN9 & SKIP_SCAN_AN10 &
                 SKIP_SCAN_AN14 & SKIP_SCAN_AN15;

    Adcon3_reg = ADC_SAMPLE_TIME_10 &
                 ADC_CONV_CLK_SYSTEM &
                 ADC_CONV_CLK_13Tcy;

    Adcon2_reg = ADC_VREF_AVDD_AVSS &
                 ADC_SCAN_OFF &
                 ADC_ALT_BUF_OFF &
                 ADC_ALT_INPUT_OFF &
                 ADC_CONVERT_CH0123 &
                 ADC_SAMPLES_PER_INT_16;
    Adcon1_reg = ADC_MODULE_ON &
                 ADC_IDLE_CONTINUE &
                 ADC_FORMAT_INTG &
                 ADC_CLK_MANUAL &
                 ADC_SAMPLE_SIMULTANEOUS &
                 ADC_AUTO_SAMPLING_OFF;
    OpenADC10(Adcon1_reg, Adcon2_reg,
              Adcon3_reg,PinConfig, Scanselect);
    i = 0;
    while(i <16 )
    {
        ADCON1bits.SAMP = 1;
        while(!ADCON1bits.SAMP);
        ConvertADC10();
        while(ADCON1bits.SAMP);
        while(!BusyADC10());
        while(BusyADC10());
        result[i] = ReadADC10(i);
        i++;
    }
}
```

3.7 定时器函数

本节给出关于定时器的各个函数及其使用示例。函数也可以用宏来实现。

3.7.1 各个函数

CloseTimer1
CloseTimer2
CloseTimer3
CloseTimer4
CloseTimer5

描述： 该函数关闭 16 位定时器模块。

头文件： `timer.h`

函数原型：
```
void CloseTimer1(void);
void CloseTimer2(void);
void CloseTimer3(void);
void CloseTimer4(void);
void CloseTimer5(void);
```

参数： 无

返回值： 无

说明： 该函数首先禁止 16 位定时器中断，然后关闭定时器模块。同时清除中断标志位 （TxIF）。

源文件：
```
CloseTimer1.c
CloseTimer2.c
CloseTimer3.c
CloseTimer4.c
CloseTimer5.c
```

代码示例：
```
CloseTimer1();
```

CloseTimer23
CloseTimer45

描述： 该函数关闭 32 位定时器模块。

头文件： `timer.h`

函数原型：
```
void CloseTimer23 (void)
void CloseTimer45 (void)
```

参数： 无

返回值： 无

说明： 该函数禁止 32 位定时器中断，然后关闭定时器模块。同时清除中断标志位 （TxIF）。
`CloseTimer23` 关闭 Timer2，并禁止 Timer3 中断。
`CloseTimer45` 关闭 Timer4，并禁止 Timer5 中断。

源文件：
```
CloseTimer23.c
CloseTimer45.c
```

代码示例：
```
CloseTimer23();
```

ConfigIntTimer1
ConfigIntTimer2
ConfigIntTimer3
ConfigIntTimer4
ConfigIntTimer5

描述： 该函数对 16 位定时器进行配置。

头文件： timer.h

函数原型：

```
void ConfigIntTimer1(unsigned int config);
void ConfigIntTimer2(unsigned int config);
void ConfigIntTimer3(unsigned int config);
void ConfigIntTimer4(unsigned int config);
void ConfigIntTimer5(unsigned int config);
```

参数： *config* 如下定义的定时器中断优先级和允许 / 禁止信息：

```
Tx_INT_PRIOR_7
Tx_INT_PRIOR_6
Tx_INT_PRIOR_5
Tx_INT_PRIOR_4
Tx_INT_PRIOR_3
Tx_INT_PRIOR_2
Tx_INT_PRIOR_1
Tx_INT_PRIOR_0

Tx_INT_ON
Tx_INT_OFF
```

返回值： 无

说明： 该函数清除 16 位中断标志位 （TxIF），然后设置中断优先级并允许 / 禁止中断。

源文件：

```
ConfigIntTimer1.c
ConfigIntTimer2.c
ConfigIntTimer3.c
ConfigIntTimer4.c
ConfigIntTimer5.c
```

代码示例：

```
ConfigIntTimer1(T1_INT_PRIOR_3 & T1_INT_ON);
```

ConfigIntTimer23
ConfigIntTimer45

描述： 该函数对 32 位定时器中断进行配置。

头文件： timer.h

函数原型：
```
void ConfigIntTimer23(unsigned int config);
void ConfigIntTimer45(unsigned int config);
```

参数： *config* 如下定义的定时器中断优先级和允许 / 禁止信息：

```
Tx_INT_PRIOR_7
Tx_INT_PRIOR_6
Tx_INT_PRIOR_5
Tx_INT_PRIOR_4
Tx_INT_PRIOR_3
Tx_INT_PRIOR_2
Tx_INT_PRIOR_1
Tx_INT_PRIOR_0

Tx_INT_ON
Tx_INT_OFF
```

返回值： 无

说明： 该函数清除 32 位中断标志位 （TxIF），然后设置中断优先级并允许 / 禁止中断。

源文件：
```
ConfigIntTimer23.c
ConfigIntTimer45.c
```

代码示例：
```
ConfigIntTimer23(T3_INT_PRIOR_5 & T3_INT_ON);
```

OpenTimer1
OpenTimer2
OpenTimer3
OpenTimer4
OpenTimer5

描述： 该函数对 16 位定时器模块进行配置。

头文件： timer.h

函数原型：
```
void OpenTimer1(unsigned int config,
                unsigned int period)
void OpenTimer2(unsigned int config,
                unsigned int period)
void OpenTimer3(unsigned int config,
                unsigned int period)
void OpenTimer4(unsigned int config,
                unsigned int period)
void OpenTimer5(unsigned int config,
                unsigned int period)
```

参数： *config* 包括 TxCON 寄存器中要配置的参数，定义如下：

定时器模块启用 / 关闭
```
Tx_ON
Tx_OFF
```
定时器模块在空闲模式启用 / 关闭
```
Tx_IDLE_CON
Tx_IDLE_STOP
```

OpenTimer1 （续）
OpenTimer2
OpenTimer3
OpenTimer4
OpenTimer5

定时器选通时间累加使能
Tx_GATE_ON
Tx_GATE_OFF

定时器预分频比
Tx_PS_1_1
Tx_PS_1_8
Tx_PS_1_64
Tx_PS_1_128

定时器同步时钟使能
Tx_SYNC_EXT_ON
Tx_SYNC_EXT_OFF

定时器时钟源
Tx_SOURCE_EXT
Tx_SOURCE_INT

period 要存储到 PR 寄存器中的周期匹配值

返回值： 无

说明： 该函数对 16 位定时器控制寄存器进行配置，并设置 PR 寄存器中的周期匹配值。

源文件：
```
OpenTimer1.c
OpenTimer2.c
OpenTimer3.c
OpenTimer4.c
OpenTimer5.c
```

代码示例：
```
OpenTimer1(T1_ON & T1_GATE_OFF &
           T1_PS_1_8 & T1_SYNC_EXT_OFF &
           T1_SOURCE_INT, 0xFF);
```

OpenTimer23
OpenTimer45

描述： 该函数对 32 位定时器模块进行配置。

头文件： timer.h

函数原型：
```
void OpenTimer23(unsigned int config,
                 unsigned long period);
void OpenTimer45(unsigned int config,
                 unsigned long period);
```

参数： *config* 包括 TxCON 寄存器中要配置的参数，定义如下：

定时器模块启用 / 关闭
Tx_ON
Tx_OFF

定时器模块空闲模式下启用 / 关闭
Tx_IDLE_CON
Tx_IDLE_STOP

定时器选通时间累加使能
Tx_GATE_ON
Tx_GATE_OFF

OpenTimer23 （续）
OpenTimer45

	定时器预分频比 Tx_PS_1_1 Tx_PS_1_8 Tx_PS_1_64 Tx_PS_1_128 定时器同步时钟使能 Tx_SYNC_EXT_ON Tx_SYNC_EXT_OFF 定时器时钟源 Tx_SOURCE_EXT Tx_SOURCE_INT
	period 要存储到 32 位 PR 寄存器中的周期匹配值。
返回值：	无
说明：	该函数对 32 位定时器控制寄存器进行配置，并设置 PR 寄存器中的周期匹配值。
源文件：	OpenTimer23.c OpenTimer45.c
代码示例：	OpenTimer23(T2_ON & T2_GATE_OFF & T2_PS_1_8 & T2_32BIT_MODE_ON & T2_SYNC_EXT_OFF & T2_SOURCE_INT, 0xFFFF);

ReadTimer1
ReadTimer2
ReadTimer3
ReadTimer4
ReadTimer5

描述：	该函数读出 16 位定时器寄存器的内容。
头文件：	timer.h

函数原型：	unsigned int ReadTimer1(void);
	unsigned int ReadTimer2(void);
	unsigned int ReadTimer3(void);
	unsigned int ReadTimer4(void);
	unsigned int ReadTimer5(void);

参数：	无
返回值：	无
说明：	该函数返回 16 位 TMR 寄存器的内容。
源文件：	ReadTimer1.c ReadTimer2.c ReadTimer3.c ReadTimer4.c ReadTimer5.c
代码示例：	unsigned int timer1_value; timer1_value = ReadTimer1();

ReadTimer23
ReadTimer45

描述： 该函数读出 32 位定时器寄存器的内容。

头文件： timer.h

函数原型：
```
unsigned long ReadTimer23(void);
unsigned long ReadTimer45(void);
```

参数： 无

返回值： 无

说明： 该函数返回 32 位 TMR 寄存器的内容。

源文件：
```
ReadTimer23.c
ReadTimer45.c
```

代码示例：
```
unsigned long timer23_value;
timer23_value = ReadTimer23();
```

WriteTimer1
WriteTimer2
WriteTimer3
WriteTimer4
WriteTimer5

描述： 该函数将 16 位值写入定时器寄存器中。

头文件： timer.h

函数原型：
```
void WriteTimer1(unsigned int timer);
void WriteTimer2(unsigned int timer);
void WriteTimer3(unsigned int timer);
void WriteTimer4(unsigned int timer);
void WriteTimer5(unsigned int timer);
```

参数： *timer* 要存储到 TMR 寄存器中的 16 位值。

返回值： 无

说明： 无

源文件：
```
WriteTimer1.c
WriteTimer2.c
WriteTimer3.c
WriteTimer4.c
WriteTimer5.c
```

代码示例：
```
unsigned int timer_init = 0xAB;
WriteTimer1(timer_init);
```

WriteTimer23
WriteTimer45

描述：	该函数将 32 位值写入定时器寄存器中。
头文件：	timer.h
函数原型：	void WriteTimer23(unsigned long timer); void WriteTimer45(unsigned long timer);
参数：	*timer* 要存储到 TMR 寄存器中的 32 位值。
返回值：	无
说明：	无
源文件：	WriteTimer23.c WriteTimer45.c
代码示例：	unsigned long timer23_init = 0xABCD; WriteTimer23(timer23_init);

3.7.2 各个宏

EnableIntT1
EnableIntT2
EnableIntT3
EnableIntT4
EnableIntT5

描述：	该宏允许定时器中断。
头文件：	timer.h
参数：	无
说明：	该宏置位中断允许控制寄存器中的定时器中断允许位。
代码示例：	EnableIntT1;

DisableIntT1
DisableIntT2
DisableIntT3
DisableIntT4
DisableIntT5

描述：	该宏禁止定时器中断。
头文件：	timer.h
参数：	无
说明：	该宏清除中断允许控制寄存器中的定时器中断允许位。
代码示例：	DisableIntT2;

SetPriorityIntT1
SetPriorityIntT2
SetPriorityIntT3
SetPriorityIntT4
SetPriorityIntT5

描述：	该宏设置定时器中断的优先级。
头文件：	timer.h
参数：	*priority*
说明：	该宏设置中断优先级控制寄存器中的定时器中断优先级位。
代码示例：	SetPriorityIntT4(7);

3.7.3 使用示例

```
#define __dsPIC30F6014__
#include <p30fxxxx.h>
#include<timer.h>
unsigned int timer_value;
void __attribute__((__interrupt__)) _T1Interrupt(void)
{
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
    WriteTimer1(0);
    IFS0bits.T1IF = 0;    /* Clear Timer interrupt flag */
}
int main(void)
{
    unsigned int match_value;
    TRISDbits.TRISD1 = 0;
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
/* Enable Timer1 Interrupt and Priority to "1" */
    ConfigIntTimer1(T1_INT_PRIOR_1 & T1_INT_ON);
    WriteTimer1(0);
    match_value = 0xFFF;
    OpenTimer1(T1_ON & T1_GATE_OFF & T1_IDLE_STOP &
               T1_PS_1_1 & T1_SYNC_EXT_OFF &
               T1_SOURCE_INT, match_value);
/* Wait till the timer matches with the period value */
    while(1)
    {
        timer_value = ReadTimer1();
        if(timer_value >= 0x7FF)
        {
            PORTDbits.RD1 = 0; /* turn on LED on RD1 */
        }
    }
    CloseTimer1();
}
```

3.8 复位 / 控制函数

本节给出了有关复位 / 控制的各个函数。函数也可以用宏来实现。

3.8.1 各个函数

isBOR

描述： 该函数检查复位是否是由于掉电检测引起的。

头文件： reset.h

函数原型：
```
char isBOR(void);
```

参数： 无

返回值： 该函数返回 RCON<BOR> 位的状态。
如果返回值为 1，那么复位是由于掉电引起的。
如果返回值为 0，那么没有发生掉电。

说明： 无

源文件： isBOR.c

代码示例：
```
char reset_state;
reset_state = isBOR();
```

isPOR

描述： 该函数检查复位是否是由于上电复位引起的。

头文件： reset.h

函数原型：
```
char isPOR(void);
```

参数： 无

返回值： 该函数返回 RCON<POR> 位的状态。
如果返回值为 1，那么复位是由于上电引起的。
如果返回值为 0，那么没有发生上电复位。

说明： 无

源文件： isPOR.c

代码示例：
```
char reset_state;
reset_state = isPOR();
```

isLVD

描述： 该函数检查低电压检测中断标志是否被置位。

头文件： reset.h

函数原型：
```
char isLVD(void);
```

参数： 无

返回值： 该函数返回 IFS2<LVDIF> 位的状态。
如果返回值为 1，那么产生低电压检测中断。
如果返回值为 0，那么没有产生低电压检测中断。

说明： 无

源文件： isLVD.c

代码示例：
```
char lvd;
lvd = isLVD();
```

isMCLR

描述： 该函数检查复位是否是由于 $\overline{\text{MCLR}}$ 引脚变为低电平引起的。

头文件： reset.h

函数原型：
```
char isMCLR(void);
```

参数： 无

返回值： 该函数返回 RCON<EXTR> 位的状态。
如果返回值为 1，那么复位是由于 $\overline{\text{MCLR}}$ 引脚变为低电平引起的。
如果返回值为 0，那么复位不是由于 $\overline{\text{MCLR}}$ 引脚变为低电平引起的。

说明： 无

源文件： isMCLR.c

代码示例：
```
char reset_state;
reset_state = isMCLR();
```

isWDTTO

描述： 该函数检查复位是否是由于 WDT 超时溢出引起的。

头文件： reset.h

函数原型：
```
char isWDTTO(void);
```

参数： 无

返回值： 该函数返回 RCON<WDTO> 位的状态。
如果返回值为 1，那么复位是由于 WDT 超时溢出引起的。
如果返回值为 0，那么复位不是由于 WDT 超时溢出引起的。

说明： 无

源文件： isWDTTO.c

代码示例：
```
char reset_state;
reset_state = isWDTTO();
```

isWDTWU

描述： 该函数检查是否是由于 WDT 超时溢出将器件从休眠状态唤醒。

头文件： reset.h

函数原型：
```
char isWDTWU(void);
```

参数： 无

返回值： 该函数返回 RCON<WDTO> 和 RCON<SLEEP> 位的状态。
如果返回值为 “1”，那么是由于 WDT 的超时溢出导致从休眠状态唤醒。
如果返回值为 “0”，那么不是由于 WDT 的超时溢出导致从休眠状态唤醒。

说明： 无

源文件： isWDTWU.c

代码示例：
```
char reset_state;
reset_state = isWDTWU();
```

isWU

描述： 该函数检查是否是由于 MCLR、 POR、 BOR 或任何中断导致从休眠状态唤醒。

头文件： reset.h

函数原型：
```
char isWU(void);
```

参数： 无

返回值： 该函数检查器件是否从休眠状态唤醒。
如果是，则检查唤醒的原因。
如果为 “1”，唤醒是由于中断的产生而引起的。
如果为 “2”，是由于 MCLR 引脚变为低电平引起的。
如果为 “3”，是由于 POR 引起的。
如果为 “4”，是由于 BOR 引起的。
如果没有从休眠状态唤醒，则返回 “0”。

说明： 无

源文件： isWU.c

代码示例：
```
char reset_state;
reset_state = isWU();
```

3.8.2 各个宏

DisableInterrupts

描述： 该宏在指定的指令周期数内禁止所有外设中断。

头文件： reset.h

参数： cycles

说明： 该宏执行 DISI 指令，将所有外设中断禁止指定的指令周期数。

代码示例：
```
DisableInterrupts(15);
```

PORStatReset

描述： 该宏将 RCON 寄存器的 POR 位设置成复位状态。

头文件： reset.h

参数： 无

说明： 无

代码示例：
```
PORStatReset;
```

BORStatReset

描述： 该宏将 RCON 寄存器的 BOR 位设置成复位状态。

头文件： reset.h

参数： 无

说明： 无

代码示例：
```
BORStatReset;
```

WDTSWEnable

描述： 该宏开启看门狗定时器。
头文件： reset.h
参数： 无
说明： 该宏置位 RCON 寄存器的软件 WDT 使能位 （SWDTEN）。
代码示例：
```
WDTSWEnable;
```

WDTSWDisable

描述： 该宏清除 RCON 寄存器的软件 WDT 使能位 （SWDTEN）。
头文件： reset.h
参数： 无
说明： 如果 FWDTEN 位为 “0”，该宏禁止 WDT。
代码示例：
```
WDTSWDisable;
```

3.9 I/O 端口函数

本节给出了关于 I/O 端口的各个函数。函数也可以用宏来实现。

3.9.1 各个函数

CloseINT0
CloseINT1
CloseINT2
CloseINT3
CloseINT4

描述： 该函数禁止 INT 引脚上的外部中断。
头文件： ports.h
函数原型：
```
void CloseINT0(void);
void CloseINT1(void);
void CloseINT2(void);
void CloseINT3(void);
void CloseINT4(void);
```
参数： 无
返回值： 无
说明： 该函数禁止 INT 引脚上的外部中断并清除相应的中断标志。
源文件：
```
CloseInt0.c
CloseInt1.c
CloseInt2.c
CloseInt3.c
CloseInt4.c
```
代码示例：
```
CloseINT0();
```

ConfigINT0
ConfigINT1
ConfigINT2
ConfigINT3
ConfigINT4

描述： 该函数对 INT 引脚上的中断进行配置。

头文件： ports.h

函数原型：
```
void ConfigINT0(unsigned int config);
void ConfigINT1(unsigned int config);
void ConfigINT2(unsigned int config);
void ConfigINT3(unsigned int config);
void ConfigINT4(unsigned int config);
```

参数： *config* 如下定义的中断边沿、优先级以及允许 / 禁止信息：

中断边沿选择
RISING_EDGE_INT
FALLING_EDGE_INT

中断允许
INT_ENABLE
INT_DISABLE

中断优先级
INT_PRI_0
INT_PRI_1
INT_PRI_2
INT_PRI_3
INT_PRI_4
INT_PRI_5
INT_PRI_6
INT_PRI_7

返回值： 无

说明： 该函数清除 INTx 引脚上相应的中断标志并选择边沿检测的极性。然后设置中断优先级并允许 / 禁止中断。

源文件：
ConfigInt0.c
ConfigInt1.c
ConfigInt2.c
ConfigInt3.c
ConfigInt4.c

代码示例：
```
ConfigINT0(RISING_EDGE_INT & EXT_INT_PRI_5 &
EXT_INT_ENABLE);
```

ConfigCNPullups

描述： 该函数配置 CN 引脚上的上拉电阻。

头文件： ports.h

函数原型： void ConfigCNPullups(long int *config*);

参数： *config* 用于配置上拉电阻的 32 位值。低字存储在 CNPU1 寄存器中，高字存储在 CNPU2 寄存器中。CNPU2 寄存器中的高 8 位未使用。

返回值： 无

说明： 无

源文件： ConfigCNPullups.c

代码示例：
```
ConfigCNPullups(0xFFF);
```

ConfigIntCN

描述： 该函数配置 CN 中断。

头文件： ports.h

函数原型： void ConfigIntCN(long int *config*);

参数： *config* 用于配置 CN 中断的 32 位值。

低 24 位中包括了各个允许 / 禁止 CN 中断的信息。置位 bit x（x=0，1，...，23）将允许 CNx 中断。

config 的最高字节包含中断优先级和允许 / 禁止位。低字存储在 CNEN1 寄存器中，次高字节存储在 CNEN2 寄存器中，最高字节用来设置中断优先级并允许 / 禁止 CN 中断。

返回值： 无

说明： 该函数清除 CN 中断标志并允许 / 禁止 CN 引脚上的各个中断。同时该函数还配置中断优先级并允许 / 禁止 CN 中断允许位。

源文件： ConfigIntCN.c

代码示例：
```
// This would enable CN0, CN1, CN2 and CN7 only.
ConfigIntCN(CHANGE_INT_OFF & CHANGE_INT_PRI_4 &
0xFF000087);
```

3.9.2 各个宏

EnableCN0
EnableCN1
EnableCN2
.....
EnableCN23

描述： 该宏允许各个电平变化通知中断。
头文件： ports.h
参数： 无
说明： 无
代码示例：
```
EnableCN6;
```

DisableCN0
DisableCN1
DisableCN2
.....
DisableCN23

描述： 该宏禁止各个电平变化通知中断。
头文件： ports.h
参数： 无
说明： 无
代码示例：
```
DisableCN14;
```

EnableINT0
EnableINT1
EnableINT2
EnableINT3
EnableINT4

描述： 该宏允许各个外部中断。
头文件： ports.h
参数： 无
说明： 无
代码示例：
```
EnableINT2;
```

DisableINT0
DisableINT1
DisableINT2
DisableINT3
DisableINT4

描述： 该宏禁止各个外部中断。

头文件： ports.h

参数： 无

说明： 无

代码示例：
```
DisableINT2;
```

SetPriorityInt0
SetPriorityInt1
SetPriorityInt2
SetPriorityInt3
SetPriorityInt4

描述： 该宏设置外部中断的优先级。

头文件： ports.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器的外部中断优先级位。

代码示例：
```
SetPriorityInt4(6);
```

3.10 输入捕捉函数

本节给出了关于输入捕捉模块的各个函数及其使用示例。函数也可以用宏来实现。

3.10.1 各个函数

CloseCapture1
CloseCapture2
CloseCapture3
CloseCapture4
CloseCapture5
CloseCapture6
CloseCapture7
CloseCapture8

描述：	该函数关闭输入捕捉模块。
头文件：	InCap.h

函数原型：	void CloseCapture1(void);
	void CloseCapture2(void);
	void CloseCapture3(void);
	void CloseCapture4(void);
	void CloseCapture5(void);
	void CloseCapture6(void);
	void CloseCapture7(void);
	void CloseCapture8(void);

参数：	无
返回值：	无
说明：	该函数禁止输入捕捉中断，然后关闭输入捕捉模块。同时清除中断标志位。
源文件：	CloseCapture1.c CloseCapture2.c CloseCapture3.c CloseCapture4.c CloseCapture5.c CloseCapture6.c CloseCapture7.c CloseCapture8.c
代码示例：	CloseCapture1();

ConfigIntCapture1
ConfigIntCapture2
ConfigIntCapture3
ConfigIntCapture4
ConfigIntCapture5
ConfigIntCapture6
ConfigIntCapture7
ConfigIntCapture8

描述： 该函数对输入捕捉中断进行配置。

头文件： InCap.h

函数原型：

```
void ConfigIntCapture1(unsigned int config);
void ConfigIntCapture2(unsigned int config);
void ConfigIntCapture3(unsigned int config);
void ConfigIntCapture4(unsigned int config);
void ConfigIntCapture5(unsigned int config);
void ConfigIntCapture6(unsigned int config);
void ConfigIntCapture7(unsigned int config);
void ConfigIntCapture8(unsigned int config);
```

参数： *config* 如下定义的输入捕捉中断优先级和允许 / 禁止信息：

中断允许 / 禁止
IC_INT_ON
IC_INT_OFF

中断优先级
IC_INT_PRIOR_0
IC_INT_PRIOR_1
IC_INT_PRIOR_2
IC_INT_PRIOR_3
IC_INT_PRIOR_4
IC_INT_PRIOR_5
IC_INT_PRIOR_6
IC_INT_PRIOR_7

返回值： 无

说明： 该函数清除中断标志位，然后设置中断优先级并允许 / 禁止中断。

源文件：
ConfigIntCapture1.c
ConfigIntCapture2.c
ConfigIntCapture3.c
ConfigIntCapture4.c
ConfigIntCapture5.c
ConfigIntCapture6.c
ConfigIntCapture7.c
ConfigIntCapture8.c

代码示例：

```
ConfigIntCapture1(IC_INT_ON & IC_INT_PRIOR_1);
```

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4
OpenCapture5
OpenCapture6
OpenCapture7
OpenCapture8

描述： 该函数对输入捕捉模块进行配置。

头文件： InCap.h

函数原型：

```
void OpenCapture1(unsigned int config);
void OpenCapture2(unsigned int config);
void OpenCapture3(unsigned int config);
void OpenCapture4(unsigned int config);
void OpenCapture5(unsigned int config);
void OpenCapture6(unsigned int config);
void OpenCapture7(unsigned int config);
void OpenCapture8(unsigned int config);
```

参数： *config* 包括 ICxCON 寄存器中要配置的参数，定义如下：

空闲模式下的工作
IC_IDLE_CON
IC_IDLE_STOP

时钟选择
IC_TIMER2_SRC
IC_TIMER3_SRC

每次中断的捕捉数
IC_INT_4CAPTURE
IC_INT_3CAPTURE
IC_INT_2CAPTURE
IC_INT_1CAPTURE
IC_INTERRUPT

IC 模式选择
IC_EVERY_EDGE
IC_EVERY_16_RISE_EDGE
IC_EVERY_4_RISE_EDGE
IC_EVERY_RISE_EDGE
IC_EVERY_FALL_EDGE
IC_INPUTCAP_OFF

返回值： 无

说明： 该函数对输入捕捉模块控制寄存器 （ICxCON）中的以下参数进行设置：时钟选择、每次中断的捕捉数和捕捉工作模式。

源文件：

```
OpenCapture1.c
OpenCapture2.c
OpenCapture3.c
OpenCapture4.c
OpenCapture5.c
OpenCapture6.c
OpenCapture7.c
OpenCapture8.c
```

代码示例：

```
OpenCapture1(IC_IDLE_CON & IC_TIMER2_SRC &
IC_INT_1CAPTURE & IC_EVERY_RISE_EDGE);
```

ReadCapture1
ReadCapture2
ReadCapture3
ReadCapture4
ReadCapture5
ReadCapture6
ReadCapture7
ReadCapture8

描述： 该函数读取所有等待读取的输入捕捉缓冲器。

头文件： InCap.h

函数原型：

```
void ReadCapture1(unsigned int *buffer);
void ReadCapture2(unsigned int *buffer);
void ReadCapture3(unsigned int *buffer);
void ReadCapture4(unsigned int *buffer);
void ReadCapture5(unsigned int *buffer);
void ReadCapture6(unsigned int *buffer);
void ReadCapture7(unsigned int *buffer);
void ReadCapture8(unsigned int *buffer);
```

参数： *buffer* 指向将从输入捕捉缓冲器中读取的数据存储到的地址的指针。

返回值： 无

说明： 该函数读取所有等待读取的输入捕捉缓冲器，直到缓冲器为空为止，缓冲器是否为空由 ICxCON 的 <ICBNE> 位是否清零来指示。

源文件：

```
ReadCapture1.c
ReadCapture2.c
ReadCapture3.c
ReadCapture4.c
ReadCapture5.c
ReadCapture6.c
ReadCapture7.c
ReadCapture8.c
```

代码示例：

```
unsigned int *buffer = 0x1900;
ReadCapture1(buffer);
```

3.10.2 各个宏

EnableIntIC1
EnableIntIC2
EnableIntIC3
EnableIntIC4
EnableIntIC5
EnableIntIC6
EnableIntIC7
EnableIntIC8

描述：	该宏允许捕捉事件的中断。
头文件：	InCap.h
参数：	无
说明：	该宏置位中断允许控制寄存器的输入捕捉中断允许位。
代码示例：	`EnableIntIC7;`

DisableIntIC1
DisableIntIC2
DisableIntIC3
DisableIntIC4
DisableIntIC5
DisableIntIC6
DisableIntIC7
DisableIntIC8

描述：	该宏禁止捕捉事件的中断。
头文件：	InCap.h
参数：	无
说明：	该宏清除中断允许控制寄存器的输入捕捉中断允许位。
代码示例：	`DisableIntIC7;`

SetPriorityIntIC1
SetPriorityIntIC2
SetPriorityIntIC3
SetPriorityIntIC4
SetPriorityIntIC5
SetPriorityIntIC6
SetPriorityIntIC7
SetPriorityIntIC8

描述：	该宏设置输入捕捉中断的优先级。
头文件：	InCap.h
参数：	*priority*
说明：	该宏设置中断优先级控制寄存器的输入捕捉中断优先级位。
代码示例：	`SetPriorityIntIC4(1);`

3.10.3 使用示例

```
#define __dsPIC30F6014__
#include <p30fxxxx.h>
#include<InCap.h>
int Interrupt_Count = 0 , Int_flag, count;
unsigned int timer_first_edge, timer_second_edge;
void __attribute__((__interrupt__)) _IC1Interrupt(void)
{
    Interrupt_Count++;
    if(Interrupt_Count == 1)
        ReadCapture1(&timer_first_edge);
    else if(Interrupt_Count == 2)
        ReadCapture1(&timer_second_edge);
    Int_flag = 1;
    IFS0bits.IC1IF = 0;
}
int main(void)
{
    unsigned int period;
    Int_flag = 0;
    TRISDbits.TRISD0 = 0; /* Alarm output on RD0 */
    PORTDbits.RD0 = 1;
/* Enable Timer1 Interrupt and Priority to '1' */
    ConfigIntCapture1(IC_INT_PRIOR_1 & IC_INT_ON);
    T3CON = 0x8000; /* Timer 3 On */
/* Configure the InputCapture in stop in idle mode , Timer
   3 as source , interrupt on capture 1, I/C on every fall
   edge */

    OpenCapture1(IC_IDLE_STOP & IC_TIMER3_SRC &
                 IC_INT_1CAPTURE & IC_EVERY_FALL_EDGE);
    while(1)
    {
        while(!Int_flag); /* wait here till first capture event */
        Int_flag = 0;
        while(!Int_flag); /* wait here till next capture event */
/* calculate time count between two capture events */
        period = timer_second_edge -  timer_first_edge;
/* if the time count between two capture events is more than
   0x200 counts, set alarm on RD0 */
       if(period >= 0x200)
       {
/* set alarm and wait for sometime and clear alarm */
            PORTDbits.RD0 = 0;
            while(count <= 0x10)
            {
                count++;
            }
            PORTDbits.RD0 = 1;
        }
        Interrupt_Count = 0;
        count = 0;
    }
    CloseCapture1();
}
```

3.11 输出比较函数

本节给出了关于输出比较模块的各个函数及其使用示例。函数也可以用宏来实现。

3.11.1 各个函数

CloseOC1
CloseOC2
CloseOC3
CloseOC4
CloseOC5
CloseOC6
CloseOC7
CloseOC8

描述：	该函数关闭输出比较模块。
头文件：	outcompare.h

函数原型：	void CloseOC1(void);
	void CloseOC2(void);
	void CloseOC3(void);
	void CloseOC4(void);
	void CloseOC5(void);
	void CloseOC6(void);
	void CloseOC7(void);
	void CloseOC8(void);

参数：	无
返回值：	无
说明：	该函数禁止输出比较中断，然后关闭输出比较模块。同时清除中断标志位。
源文件：	CloseOC1.c CloseOC2.c CloseOC3.c CloseOC4.c CloseOC5.c CloseOC6.c CloseOC7.c CloseOC8.c
代码示例：	CloseOC1();

ConfigIntOC1
ConfigIntOC2
ConfigIntOC3
ConfigIntOC4
ConfigIntOC5
ConfigIntOC6
ConfigIntOC7
ConfigIntOC8

描述： 该函数对输出比较中断进行配置。

头文件： outcompare.h

函数原型：

```
void ConfigIntOC1(unsigned int config);
void ConfigIntOC2(unsigned int config);
void ConfigIntOC3(unsigned int config);
void ConfigIntOC4(unsigned int config);
void ConfigIntOC5(unsigned int config);
void ConfigIntOC6(unsigned int config);
void ConfigIntOC7(unsigned int config);
void ConfigIntOC8(unsigned int config);
```

参数： *config* 如下定义的输出比较中断优先级和允许 / 禁止信息：

中断允许 / 禁止
OC_INT_ON
OC_INT_OFF

中断优先级
OC_INT_PRIOR_0
OC_INT_PRIOR_1
OC_INT_PRIOR_2
OC_INT_PRIOR_3
OC_INT_PRIOR_4
OC_INT_PRIOR_5
OC_INT_PRIOR_6
OC_INT_PRIOR_7

返回值： 无

说明： 该函数清除中断标志位，然后设置中断优先级并允许 / 禁止中断。

源文件：
ConfigIntOC1.c
ConfigIntOC2.c
ConfigIntOC3.c
ConfigIntOC4.c
ConfigIntOC5.c
ConfigIntOC6.c
ConfigIntOC7.c
ConfigIntOC8.c

代码示例：

```
ConfigIntOC1(OC_INT_ON & OC_INT_PRIOR_2);
```

OpenOC1
OpenOC2
OpenOC3
OpenOC4
OpenOC5
OpenOC6
OpenOC7
OpenOC8

描述： 该函数对输出比较模块进行配置。

头文件： outcompare.h

函数原型：

```
void OpenOC1(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC2(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC3(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC4(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC5(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC6(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC7(unsigned int config,
  unsigned int value1, unsigned int value2);
void OpenOC8(unsigned int config,
  unsigned int value1, unsigned int value2);
```

参数：

config 包括 OCxCON 寄存器中要配置的参数，定义如下：

空闲模式下的工作
OC_IDLE_STOP
OC_IDLE_CON

时钟选择
OC_TIMER2_SRC
OC_TIMER3_SRC

输出比较模式的工作
OC_PWM_FAULT_PIN_ENABLE
OC_PWM_FAULT_PIN_DISABLE
OC_CONTINUE_PULSE
OC_SINGLE_PULSE
OC_TOGGLE_PULSE
OC_HIGH_LOW
OC_LOW_HIGH
OC_OFF

value1 包含要存储到 OCxRS 辅助寄存器中的值。

value2 包含要存储到 OCxR 主寄存器中的值。

返回值： 无

OpenOC1 （续）
OpenOC2
OpenOC3
OpenOC4
OpenOC5
OpenOC6
OpenOC7
OpenOC8

说明： 该函数对输出比较模块控制寄存器 （OCxCON）中的如下参数进行设置：

时钟选择、工作模式和空闲模式下的工作。
同时，它还对 OCxRS 和 OCxR 寄存器进行配置。

源文件：
```
OpenOC1.c
OpenOC2.c
OpenOC3.c
OpenOC4.c
OpenOC5.c
OpenOC6.c
OpenOC7.c
OpenOC8.c
```

代码示例：
```
OpenOC1(OC_IDLE_CON & OC_TIMER2_SRC &
OC_PWM_FAULT_PIN_ENABLE, 0x80, 0x60);
```

ReadDCOC1PWM
ReadDCOC2PWM
ReadDCOC3PWM
ReadDCOC4PWM
ReadDCOC5PWM
ReadDCOC6PWM
ReadDCOC7PWM
ReadDCOC8PWM

描述： 该函数从输出比较辅助寄存器中读取占空比。

头文件： outcompare.h

函数原型：
```
unsigned int ReadDCOC1PWM(void);
unsigned int ReadDCOC2PWM(void);
unsigned int ReadDCOC3PWM(void);
unsigned int ReadDCOC4PWM(void);
unsigned int ReadDCOC5PWM(void);
unsigned int ReadDCOC6PWM(void);
unsigned int ReadDCOC7PWM(void);
unsigned int ReadDCOC8PWM(void);
```

参数： 无

返回值： 当输出比较模块处于 PWM 模式时，该函数返回 OCxRS 寄存器的内容。否则返回 “-1”。

说明： 当输出比较模块处于 PWM 模式时，该函数从输出比较辅助寄存器（OCxRS）中读取占空比。
如果不工作在 PWM 模式，该函数返回 “-1”。

源文件：
ReadDCOC1PWM.c
ReadDCOC2PWM.c
ReadDCOC3PWM.c
ReadDCOC4PWM.c
ReadDCOC5PWM.c
ReadDCOC6PWM.c
ReadDCOC7PWM.c
ReadDCOC8PWM.c

代码示例：
```
unsigned int compare_reg;
compare_reg = ReadDCOC1PWM();
```

ReadRegOC1
ReadRegOC2
ReadRegOC3
ReadRegOC4
ReadRegOC5
ReadRegOC6
ReadRegOC7
ReadRegOC8

描述： 当输出比较模块不工作在 PWM 模式时，该函数读取占空比寄存器。

头文件： outcompare.h

函数原型：

```
unsigned int ReadRegOC1(char reg);
unsigned int ReadRegOC2(char reg);
unsigned int ReadRegOC3(char reg);
unsigned int ReadRegOC4(char reg);
unsigned int ReadRegOC5(char reg);
unsigned int ReadRegOC6(char reg);
unsigned int ReadRegOC7(char reg);
unsigned int ReadRegOC8(char reg);
```

参数： *reg* 表明是从输出比较模块的主占空比寄存器还是辅助占空比寄存器中读取数据。
如果 *reg* 为 “1”，则读取主占空比寄存器 （OCxR）中的数据。
如果 *reg* 为 “0”，则读取辅助占空比寄存器 （OCxRS）中的数据。

返回值： 如果 *reg* 为 “1”，则读取主占空比寄存器 （OCxR）中的数据。
如果 *reg* 为 “0”，则读取辅助占空比寄存器 （OCxRS）中的数据。
如果输出比较模块工作在 PWM 模式，返回 “-1”。

说明： 只有输出比较模块不工作在 PWM 模式时才能读取占空比寄存器。否则返回 “-1”。

源文件：

```
ReadRegOC1.c
ReadRegOC2.c
ReadRegOC3.c
ReadRegOC4.c
ReadRegOC5.c
ReadRegOC6.c
ReadRegOC7.c
ReadRegOC8.c
```

代码示例：

```
unsigned int dutycycle_reg;
dutycycle_reg = ReadRegOC1(1);
```

SetDCOC1PWM
SetDCOC2PWM
SetDCOC3PWM
SetDCOC4PWM
SetDCOC5PWM
SetDCOC6PWM
SetDCOC7PWM
SetDCOC8PWM

描述： 当模块工作在 PWM 模式时，该函数对输出比较辅助占空比寄存器（OCxRS）进行配置。

头文件： outcompare.h

函数原型：

```
void SetDCOC1PWM(unsigned int dutycycle);
void SetDCOC2PWM(unsigned int dutycycle);
void SetDCOC3PWM(unsigned int dutycycle);
void SetDCOC4PWM(unsigned int dutycycle);
void SetDCOC5PWM(unsigned int dutycycle);
void SetDCOC6PWM(unsigned int dutycycle);
void SetDCOC7PWM(unsigned int dutycycle);
void SetDCOC8PWM(unsigned int dutycycle);
```

参数：

dutycycle 要存储到输出比较辅助占空比寄存器（OCxRS）中的占空比值。

返回值： 无

说明： 只有模块工作在 PWM 模式时才能对输出比较辅助占空比寄存器（OCxRS）配置新的值。

源文件：

SetDCOC1PWM.c
SetDCOC2PWM.c
SetDCOC3PWM.c
SetDCOC4PWM.c
SetDCOC5PWM.c
SetDCOC6PWM.c
SetDCOC7PWM.c
SetDCOC8PWM.c

代码示例：

```
SetDCOC1PWM(dutycycle);
```

SetPulseOC1
SetPulseOC2
SetPulseOC3
SetPulseOC4
SetPulseOC5
SetPulseOC6
SetPulseOC7
SetPulseOC8

描述： 当模块不工作在 PWM 模式时，该函数对输出比较主和辅助寄存器（OCxR 和 OCxRS）进行配置。

头文件： outcompare.h

函数原型：

```
void SetPulseOC1(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC2(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC3(unsigned int pulse_start,
 u nsigned int pulse_stop);
void SetPulseOC4(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC5(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC6(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC7(unsigned int pulse_start,
  unsigned int pulse_stop);
void SetPulseOC8(unsigned int pulse_start,
  unsigned int pulse_stop);
```

参数：

pulse_start 要存储到输出比较主寄存器（OCxR）中的值。

pulse_stop 要存储到输出比较辅助寄存器（OCxRS）中的值。

返回值： 无

说明： 只有模块不工作在 PWM 模式时，才能对输出比较占空比寄存器（OCxR 和 OCxRS）配置新的值。

源文件：

SetPulseOC1.c
SetPulseOC2.c
SetPulseOC3.c
SetPulseOC4.c
SetPulseOC5.c
SetPulseOC6.c
SetPulseOC7.c
SetPulseOC8.c

代码示例：

```
pulse_start = 0x40 ;
pulse_stop  = 0x60 ;
SetPulseOC1(pulse_start, pulse_stop);
```

3.11.2 各个宏

EnableIntOC1
EnableIntOC2
EnableIntOC3
EnableIntOC4
EnableIntOC5
EnableIntOC6
EnableIntOC7
EnableIntOC8

描述：	该宏允许输出比较匹配中断。
头文件：	outcompare.h
参数：	无
说明：	该宏置位中断允许控制寄存器的输出比较 （OC）中断允许位。
代码示例：	`EnableIntOC8;`

DisableIntOC1
DisableIntOC2
DisableIntOC3
DisableIntOC4
DisableIntOC5
DisableIntOC6
DisableIntOC7
DisableIntOC8

描述：	该宏禁止输出比较匹配中断。
头文件：	outcompare.h
参数：	无
说明：	该宏清除中断允许控制寄存器的 OC 中断允许位。
代码示例：	`DisableIntOC7;`

SetPriorityIntIC1
SetPriorityIntIC2
SetPriorityIntIC3
SetPriorityIntIC4
SetPriorityIntIC5
SetPriorityIntIC6
SetPriorityIntIC7
SetPriorityIntIC8

描述：	该宏设置输出比较中断的优先级。
头文件：	outcompare.h
参数：	*priority*
说明：	该宏设置中断优先级控制寄存器的 OC 中断优先级位。
代码示例：	`SetPriorityIntOC4(0);`

3.11.3 使用示例

```
#define __dsPIC30F6014__
#include<p30fxxxx.h>
#include<outcompare.h>
/* This is ISR corresponding to OC1 interrupt */
void __attribute__((__interrupt__)) _OC1Interrupt(void)
{
  IFS0bits.OC1IF = 0;
}
int main(void)
{
/* Holds the value at which OCx Pin to be driven high */
unsigned int pulse_start ;
/* Holds the value at which OCx Pin to be driven low */
unsigned int pulse_stop;
/* Turn off OC1 module */
    CloseOC1();
/* Configure output compare1 interrupt */
ConfigIntOC1(OC_INT_OFF & OC_INT_PRIOR_5);
/* Configure OC1 module for required pulse width */
    pulse_start = 0x40;
    pulse_stop = 0x60;
    PR3 = 0x80 ;
    PR1 = 0xffff;
    TMR1  = 0x0000;
    T3CON = 0x8000;
    T1CON = 0X8000;
/* Configure Output Compare module to 'initialise OCx pin
low and generate continuous pulse'mode */
    OpenOC1(OC_IDLE_CON & OC_TIMER3_SRC &
            OC_CONTINUE_PULSE,
            pulse_stop, pulse_start);
/* Generate continuous pulse till TMR1 reaches 0xff00 */
    while(TMR1<= 0xff00);
    asm("nop");
    CloseOC1();
    return 0;
}
```

3.12 UART 函数

本节给出了关于 UART 模块的各个函数及其使用示例。函数也可以用宏来实现。

3.12.1 各个函数

BusyUART1
BusyUART2

描述：	该函数返回 UART 的发送状态。
头文件：	uart.h
函数原型：	char BusyUART1(void); char BusyUART2(void);
参数：	无
返回值：	如果返回 “1”，表明 UART 正忙于发送且 UxSTA 的 <TRMT> 位为 “0”。 如果返回 “0”，表明 UART 处于空闲状态且 UxSTA 的 <TRMT> 位为 “1”。
说明：	该函数返回 UART 的状态。UxSTA 的 <TRMT> 位表明 UART 是否正忙于发送。
源文件：	BusyUART1.c BusyUART2.c
代码示例：	while(BusyUART1());

CloseUART1
CloseUART2

描述：	该函数关闭 UART 模块。
头文件：	uart.h
函数原型：	void CloseUART1(void); void CloseUART2(void);
参数：	无
返回值：	无
说明：	该函数首先关闭 UART 模块，然后禁止 UART 发送和接收中断。同时清除中断标志位。
源文件：	CloseUART1.c CloseUART2.c
代码示例：	CloseUART1();

ConfigIntUART1
ConfigIntUART2

描述： 该函数对 UART 中断进行配置。

头文件： uart.h

函数原型：
```
void ConfigIntUART1(unsigned int config);
void ConfigIntUART2(unsigned int config);
```

参数： *config* 如下定义的各个中断允许 / 禁止信息：

接收中断允许
UART_RX_INT_EN
UART_RX_INT_DIS

接收中断优先级
UART_RX_INT_PR0
UART_RX_INT_PR1
UART_RX_INT_PR2
UART_RX_INT_PR3
UART_RX_INT_PR4
UART_RX_INT_PR5
UART_RX_INT_PR6
UART_RX_INT_PR7

发送中断允许
UART_TX_INT_EN
UART_TX_INT_DIS

发送中断优先级
UART_TX_INT_PR0
UART_TX_INT_PR1
UART_TX_INT_PR2
UART_TX_INT_PR3
UART_TX_INT_PR4
UART_TX_INT_PR5
UART_TX_INT_PR6
UART_TX_INT_PR7

返回值： 无

说明： 该函数允许 / 禁止 UART 发送和接收中断并设置中断优先级。

源文件：
ConfigIntUART1.c
ConfigIntUART2.c

代码示例：
```
ConfigIntUART1(UART_RX_INT_EN & UART_RX_INT_PR5 &
  UART_TX_INT_EN & UART_TX_INT_PR3);
```

DataRdyUART1
DataRdyUART2

描述： 该函数返回 UART 接收缓冲器的状态。

头文件： uart.h

函数原型：
```
char DataRdyUART1(void);
char DataRdyUART2(void);
```

参数： 无

返回值： 如果返回 “1”，表明接收缓冲器中有数据要读取。
如果返回 “0”，表明接收缓冲器中没有可读取的新数据。

说明： 该函数返回 UART 接收缓冲器的状态。
通过 UxSTA 的 <URXDA> 位表明 UART 接收缓冲器中是否有可读取的新数据。

源文件：
```
DataRdyUART1.c
DataRdyUART2.c
```

代码示例：
```
while(DataRdyUART1());
```

OpenUART1
OpenUART2

描述： 该函数对 UART 模块进行配置。

头文件： uart.h

函数原型：
```
void OpenUART1(unsigned int config1,
  unsigned int config2, unsigned int ubrg);
void OpenUART2(unsigned int config1,
  unsigned int config2, unsigned int ubrg);
```

参数： *config1* 包括 UxMODE 寄存器中要配置的参数，定义如下：

UART 使能 / 禁止
```
UART_EN
UART_DIS
```
UART 空闲模式的工作
```
UART_IDLE_CON
UART_IDLE_STOP
```
UART 使用备用引脚通信
```
UART_ALTRX_ALTTX
UART_RX_TX
```
仅部分器件的 UART 具有使用备用引脚进行通信的功能，应参考相应的数据手册。

UART 起始位唤醒
```
UART_EN_WAKE
UART_DIS_WAKE
```
UART 环回模式的使能 / 禁止
```
UART_EN_LOOPBACK
UART_DIS_LOOPBACK
```
输入到捕捉模块
```
UART_EN_ABAUD
UART_DIS_ABAUD
```

OpenUART1 （续）
OpenUART2

奇偶位和数据位选择
UART_NO_PAR_9BIT
UART_ODD_PAR_8BIT
UART_EVEN_PAR_8BIT
UART_NO_PAR_8BIT

停止位的位数
UART_2STOPBITS
UART_1STOPBIT

config2 包括 UxSTA 寄存器中要设置的参数，定义如下：

UART 发送模式中断选择
UART_INT_TX_BUF_EMPTY
UART_INT_TX

UART 发送间隔位
UART_TX_PIN_NORMAL
UART_TX_PIN_LOW

UART 发送使能 / 禁止
UART_TX_ENABLE
UART_TX_DISABLE

UART 接收中断模式选择
UART_INT_RX_BUF_FUL
UART_INT_RX_3_4_FUL
UART_INT_RX_CHAR

UART 地址检测使能 / 禁止
UART_ADR_DETECT_EN
UART_ADR_DETECT_DIS

UART 溢出位清零
UART_RX_OVERRUN_CLEAR

ubrg 要写入 UxBRG 寄存器来设置波特率的值。

返回值： 无

说明： 该函数对 UART 发送和接收进行配置并设置通信波特率。

源文件：
OpenUART1.c
OpenUART2.c

代码示例：

```
baud = 5;
UMODEvalue = UART_EN & UART_IDLE_CON &
             UART_DIS_WAKE & UART_EN_LOOPBACK &
             UART_EN_ABAUD & UART_NO_PAR_8BIT &
             UART_1STOPBIT;
U1STAvalue = UART_INT_TX_BUF_EMPTY &
             UART_TX_PIN_NORMAL &
             UART_TX_ENABLE &
             UART_INT_RX_3_4_FUL &
             UART_ADR_DETECT_DIS &
             UART_RX_OVERRUN_CLEAR ;
OpenUART1(U1MODEvalue, U1STAvalue, baud);
```

ReadUART1
ReadUART2

描述： 该函数返回 UART 接收缓冲寄存器 （UxRXREG）的内容。

头文件： uart.h

函数原型：
```
unsigned int ReadUART1(void);
unsigned int ReadUART2(void);
```

参数： 无

返回值： 该函数返回 UART 接收缓冲寄存器 （UxRXREG）的内容。

说明： 该函数返回 UART 接收缓冲寄存器的内容。
如果使能接收 9 位数据，则返回整个寄存器中的数据。
如果使能接收 8 位数据。则读取寄存器并屏蔽第 9 位数据。

源文件：
```
ReadUART1.c
ReadUART2.c
```

代码示例：
```
unsigned int RX_data;
RX_data = ReadUART1();
```

WriteUART1
WriteUART2

描述： 该函数将要发送的数据写入发送缓冲寄存器 （UxTXREG）中。

头文件： uart.h

函数原型：
```
void WriteUART1(unsigned int data);
void WriteUART2(unsigned int data);
```

参数： *data* 要发送的数据。

返回值： 无

说明： 该函数将要发送的数据写入发送缓冲寄存器。
如果使能发送 9 位数据，则将 9 位的值写入发送缓冲器中。
如果使能发送 8 位数据，则将高字节屏蔽，然后写入发送缓冲器。

源文件：
```
WriteUART1.c
WriteUART2.c
```

代码示例：
```
WriteUART1(0xFF);
```

getsUART1
getsUART2

描述： 该函数读取指定长度的数据串并将其存储到指定的缓冲器地址中。

头文件： uart.h

函数原型：
```
unsigned int getsUART1(unsigned int length,
  unsigned int *buffer, unsigned int
  uart_data_wait);
unsigned int getsUART2(unsigned int length,
  unsigned int *buffer, unsigned int
  uart_data_wait);
```

参数：

length 要接收的字符串的长度。

buffer 指向存储接收到的数据的地址的指针。

uart_data_wait 模块在返回前必须等待的延时计数。如果该值为 “N”，实际的延时大约为（19*N – 1）个指令周期。

返回值： 该函数返回尚未接收的字节数。
如果返回值为 “0”，表明整个字符串已被接收。
如果返回值不是零，表明还未接收完字符串。

说明： 无

源文件：
```
getsUART1.c
getsUART2.c
```

代码示例：
```
Datarem = getsUART1(6, Rxdata_loc, 40);
```

putsUART1
putsUART2

描述： 该函数将要发送的数据串写入 UART 发送缓冲器中。

头文件： uart.h

函数原型：
```
void putsUART1(unsigned int *buffer);
void putsUART2(unsigned int *buffer);
```

参数： *buffer* 指向要发送的数据串的指针。

返回值： 无

说明： 该函数将要发送的数据写入发送缓冲器，直到遇到空字符为止。
一旦发送缓冲器满，要等到数据发送完毕再将下一个数据写入发送寄存器。

源文件：
```
putsUART1.c
putsUART2.c
```

代码示例：
```
putsUART1(Txdata_loc);
```

getcUART1
getcUART2

描述： 该函数等同于 ReadUART1 和 ReadUART2。

源文件： uart.h 中包含对 ReadUART1 和 ReadUART2 的宏定义。

putcUART1
putcUART2

描述： 该函数等同于 WriteUART1 和 WriteUART2.

源文件： uart.h 中包含对 WriteUART1 和 WriteUART2 的宏定义。

3.12.2 各个宏

EnableIntU1RX
EnableIntU2RX

描述： 该宏允许 UART 接收中断。

头文件： uart.h

参数： 无

说明： 该宏置位中断允许控制寄存器的 UART 接收中断允许位。

代码示例：
```
EnableIntU2RX;
```

EnableIntU1TX
EnableIntU2TX

描述： 该宏允许 UART 发送中断。

头文件： uart.h

参数： 无

说明： 该宏置位中断允许控制寄存器的 UART 发送中断允许位。

代码示例：
```
EnableIntU2TX;
```

DisableIntU1RX
DisableIntU2RX

描述： 该宏禁止 UART 接收中断。

头文件： uart.h

参数： 无

说明： 该宏清除中断允许控制寄存器的 UART 接收中断允许位。

代码示例：
```
DisableIntU1RX;
```

DisableIntU1TX
DisableIntU2TX

描述： 该宏禁止 UART 发送中断。

头文件： uart.h

参数： 无

说明： 该宏清除中断允许控制寄存器的 UART 发送中断允许位。

代码示例：
```
DisableIntU1TX;
```

SetPriorityIntU1RX
SetPriorityIntU2RX

描述： 该宏设置 UART 接收中断的优先级。

头文件： uart.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器的 UART 接收中断优先级位。

代码示例：
```
SetPriorityIntU1RX(6);
```

SetPriorityIntU1TX
SetPriorityIntU2TX

描述： 该宏设置 UART 发送中断的优先级。

头文件： uart.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器的 UART 发送中断优先级位。

代码示例：
```
SetPriorityIntU1TX(5);
```

3.12.3 使用示例

```
#define __dsPIC30F6014__
#include<p30fxxxx.h>
#include<uart.h>
/* Received data is stored in array Buf  */
char Buf[80];
char * Receivedddata = Buf;
/* This is UART1 transmit ISR */
void __attribute__((__interrupt__)) _U1TXInterrupt(void)
{
   IFS0bits.U1TXIF = 0;
}
/* This is UART1 receive ISR */
void __attribute__((__interrupt__)) _U1RXInterrupt(void)
{
    IFS0bits.U1RXIF = 0;
/* Read the receive buffer till atleast one or more character can be
read */
    while( DataRdyUART1())
    {
        ( *( Receiveddata)++) = ReadUART1();
    }
}
int main(void)
{
/* Data to be transmitted using UART communication module */
char Txdata[] = {'M','i','c','r','o','c','h','i','p','
                  ','I','C','D','2','\0'};
/* Holds the value of baud register   */
unsigned int baudvalue;
/* Holds the value of uart config reg */
unsigned int U1MODEvalue;
/* Holds the information regarding uart
TX & RX interrupt modes */
unsigned int U1STAvalue;
/* Turn off UART1module */
    CloseUART1();
/* Configure uart1 receive and transmit interrupt */
    ConfigIntUART1(UART_RX_INT_EN & UART_RX_INT_PR6 &
                   UART_TX_INT_DIS & UART_TX_INT_PR2);
/* Configure UART1 module to transmit 8 bit data with one stopbit.
Also Enable loopback mode  */
    baudvalue = 5;
    U1MODEvalue = UART_EN & UART_IDLE_CON &
                  UART_DIS_WAKE & UART_EN_LOOPBACK  &
                  UART_EN_ABAUD & UART_NO_PAR_8BIT  &
                  UART_1STOPBIT;
    U1STAvalue  = UART_INT_TX_BUF_EMPTY  &
                  UART_TX_PIN_NORMAL &
                  UART_TX_ENABLE & UART_INT_RX_3_4_FUL &
                  UART_ADR_DETECT_DIS &
                  UART_RX_OVERRUN_CLEAR;
    OpenUART1(U1MODEvalue, U1STAvalue, baudvalue);
```

```
/* Load transmit buffer and transmit the same till null character is
encountered */
    putsUART1 ((unsigned int *)Txdata);
/* Wait for  transmission to complete */
    while(BusyUART1());
/* Read all the data remaining in receive buffer which are unread */
    while(DataRdyUART1())
    {
        (*( Receiveddata)++) = ReadUART1() ;
    }
/* Turn off UART1 module */
    CloseUART1();
    return 0;
}
```

3.13 DCI 函数

本节给出了关于 DCI 模块的各个函数及其使用示例。函数也可以用宏来实现。

3.13.1 各个函数

CloseDCI

描述：	该函数关闭 DCI 模块。
头文件：	dci.h
函数原型：	void CloseDCI(void);
参数：	无
返回值：	无
说明：	该函数首先关闭 DCI 模块，然后禁止 DCI 中断。同时清除中断标志位。
源文件：	CloseDCI.c
代码示例：	CloseDCI();

BufferEmptyDCI

描述：	该函数返回 DCI 发送缓冲器的满状态。
头文件：	dci.h
函数原型：	char BufferEmptyDCI(void);
参数：	无
返回值：	如果 TMPTY 的值为 “1”，则返回 “1”，表明发送缓冲器是空的。 如果 TMPTY 的值为 “0”，则返回 “0”，表明发送缓冲器不是空的。
说明：	该函数返回 DCISTAT 的 <TMPTY> 位的状态。这个位表明发送缓冲器是否为空。
源文件：	BufferEmptyDCI.c
代码示例：	while(!BufferEmptyDCI());

ConfigIntDCI

描述： 该函数对 DCI 中断进行配置。

头文件： dci.h

函数原型： void ConfigIntDCI(unsigned int *config*);

参数： *config* 如下定义的 DCI 中断优先级和允许 / 禁止信息：

DCI 中断允许 / 禁止
DCI_INT_ON
DCI_INT_OFF

DCI 中断优先级
DCI_INT_PRI_0
DCI_INT_PRI_1
DCI_INT_PRI_2
DCI_INT_PRI_3
DCI_INT_PRI_4
DCI_INT_PRI_5
DCI_INT_PRI_6
DCI_INT_PRI_7

返回值： 无

说明： 该函数清除中断标志位 （DCIIF），然后设置中断优先级并允许 / 禁止中断。

源文件： ConfigIntDCI.c

代码示例： ConfigIntDCI(DCI_INT_PRI_6 & DCI_INT_ENABLE);

DataRdyDCI

描述： 该函数返回 DCI 接收缓冲器的状态。

头文件： dci.h

函数原型： char DataRdyDCI(void);

参数： 无

返回值： 如果 RFUL 的值为 “1”，则返回 “1”，表明接收缓冲器中有数据可读。
如果 RFUL 的值为 “0”，则返回 “0”，表明接收缓冲器是空的。

说明： 该函数返回 DCISTAT 的 <RFUL> 位的状态。这个位表明接收缓冲器中是否有数据。

源文件： DataRdyDCI.c

代码示例： while(!DataRdyDCI());

OpenDCI

描述： 该函数对 DCI 进行配置。

头文件： dci.h

函数原型：
```
void OpenDCI(unsigned int config1,
             unsigned int config2,
             unsigned int config3,
             unsigned int trans_mask,
             unsigned int recv_mask)
```

参数： *config1* 包含 DCION1 寄存器中要设置的参数，定义如下：

OpenDCI （续）

模块开启 / 关闭
DCI_EN
DCI_DIS

空闲模式下的工作
DCI_IDLE_CON
DCI_IDLE_STOP

DCI 环回模式使能
DCI_DIGI_LPBACK_EN
DCI_DIGI_LPBACK_DIS

CSCK 引脚方向选择
DCI_SCKD_INP
DCI_SCKD_OUP

DCI 采样边沿选择
DCI_SAMP_CLK_RIS
DCI_SAMP_CLK_FAL

FS 引脚方向选择
DCI_FSD_INP
DCI_FSD_OUP

下溢期间要发送的数据
DCI_TX_LASTVAL_UNF
DCI_TX_ZERO_UNF

发送禁止期间 SDO 引脚的状态
DCI_SDO_TRISTAT
DCI_SDO_ZERO

数据对齐控制
DCI_DJST_ON
DCI_DJST_OFF

帧同步模式选择
DCI_FSM_ACLINK_20BIT
DCI_FSM_ACLINK_16BIT
DCI_FSM_I2S
DCI_FSM_MULTI

config2 包含 DCICON2 寄存器中要设置的参数，定义如下：

缓冲器长度
DCI_BUFF_LEN_4
DCI_BUFF_LEN_3
DCI_BUFF_LEN_2
DCI_BUFF_LEN_1

DCI 帧同步发生器控制
DCI_FRAME_LEN_16
DCI_FRAME_LEN_15
DCI_FRAME_LEN_14
.....
DCI_FRAME_LEN_1

DCI 数据字大小
DCI_DATA_WORD_16
DCI_DATA_WORD_15
DCI_DATA_WORD_14
.....
DCI_DATA_WORD_5
DCI_DATA_WORD_4

OpenDCI （续）

	config3	包含 DCICON3 寄存器中要设置的位时钟发生器值。
	trans_mask / *recv_mask*	包含 TSCON/RSCON 寄存器中要配置的发送 / 接收时隙使能位，定义如下： DCI_DIS_SLOT_15 DCI_DIS_SLOT_14 ..... DCI_DIS_SLOT_1 DCI_DIS_SLOT_0 DCI_EN_SLOT_ALL DCI_DIS_SLOT_ALL

返回值： 无

说明： 这个函数对以下参数进行配置：

1. DCICON1 寄存器：
   使能位，
   帧同步模式，
   数据对齐，
   采样时钟方向，
   采样时钟，
   边沿控制，
   输出帧同步方向控制，
   连续发送 / 接收模式，
   下溢模式。
2. DCICON2 寄存器：
   帧同步发生器控制，
   数据字大小位，
   缓冲器长度控制位。
3. DCICON3 寄存器： 时钟发生器控制位
4. TSCON 寄存器： 发送时隙使能控制位。
5. RSCON 寄存器： 接收时隙使能控制位。

源文件： OpenDCI.c

代码示例：

```
DCICON1value = DCI_EN &
               DCI_IDLE_CON &
               DCI_DIGI_LPBACK_EN &
               DCI_SCKD_OUP &
               DCI_SAMP_CLK_FAL &
               DCI_FSD_OUP &
               DCI_TX_LASTVAL_UNF &
               DCI_SDO_TRISTAT &
               DCI_DJST_OFF &
               DCI_FSM_ACLINK_16BIT ;
DCICON2value = DCI_BUFF_LEN_4 &
               DCI_FRAME_LEN_2&
               DCI_DATA_WORD_16 ;
DCICON3value = 0x02 ;
RSCONvalue   = DCI_EN_SLOT_ALL &
               DCI_DIS_SLOT_15 &
               DCI_DIS_SLOT_9 &
               DCI_DIS_SLOT_2;
TSCONvalue   = DCI_EN_SLOT_ALL &
               DCI_DIS_SLOT_14 &
               DCI_DIS_SLOT_8 &
               DCI_DIS_SLOT_1;
OpenDCI(DCICON1value, DCICON2value,  DCICON3value,
TSCONvalue, RSCONvalue);
```

ReadDCI

描述： 该函数读取 DCI 接收缓冲器的内容。

头文件： dci.h

函数原型：
```
unsigned int ReadDCI(unsigned char buffer);
```

参数： *buffer* 要读取的 DCI 缓冲器的编号。

返回值： 无

说明： 该函数返回由 *buffer* 指定的 DCI 接收缓冲器的内容。

源文件： ReadDCI.c

代码示例：
```
unsigned int DCI_buf0;
DCI_buf0 = ReadDCI(0);
```

WriteDCI

描述： 该函数将要发送的数据写入 DC 发送缓冲器。

头文件： dci.h

函数原型：
```
void WriteDCI(unsigned int data_out,
              unsigned char buffer);
```

参数：

data_out 要发送的数据。

buffer 要写入的 DCI 缓冲器的编号。

返回值： 无

说明： 该函数将 *data_out* 装载到由 *buffer* 指定的发送缓冲器中。

源文件： WriteDCI.c

代码示例：
```
unsigned int DCI_tx0 = 0x60;
WriteDCI(DCI_tx0, 0);
```

3.13.2 各个宏

EnableIntDCI

描述： 该宏允许 DCI 中断。
头文件： dci.h
参数： 无
说明： 该宏置位中断允许控制寄存器中的 DCI 中断允许位。
代码示例：
```
EnableIntDCI;
```

DisableIntDCI

描述： 该宏禁止 DCI 中断。
头文件： dci.h
参数： 无
说明： 该宏清除中断允许控制寄存器中的 DCI 中断允许位。
代码示例：
```
DisableIntDCI;
```

SetPriorityIntDCI

描述： 该宏设置 DCI 中断的优先级。
头文件： dci.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的 DCI 中断优先级位。
代码示例：
```
SetPriorityIntDCI(4);
```

3.13.3 使用示例

```
#define __dsPIC30F6014__
#include<p30fxxxx.h>
#include<dci.h>
/* Received data is stored from 0x1820 onwards. */
unsigned int *  Receiveddata = ( unsigned int *)0x1820;
void __attribute__((__interrupt__)) _DCIInterrupt(void)
{
    IFS2bits.DCIIF = 0;
}
int main(void)
{
/* Data to be transmitted using DCI module */
    unsigned int data16[] = {0xabcd, 0x1234, 0x1578,
                             0xfff0, 0xf679};
/* Holds configuration information   */
    unsigned int DCICON1value;
/* Holds the value of framelength, wordsize and buffer length */
    unsigned int DCICON2value;
/* Holds the information reagarding bit clock
   generator */
    unsigned int DCICON3value ;
/* Holds the information reagarding data to be received
   or ignored during this time slot */
   unsigned int RSCONvalue ;
/* Holds the information reagarding transmit buffer
   contents are sent during the timeslot */
   unsigned int TSCONvalue ;
   int i ;
    CloseDCI();
/* Configure DCI receive / transmit interrupt */
    ConfigIntDCI( DCI_INT_ON & DCI_INT_PRI_6);
/* Configure DCI module to transmit 16 bit data with multichannel mode
*/
    DCICON1value = DCI_EN &  DCI_IDLE_CON &
                   DCI_DIGI_LPBACK_EN &
                   DCI_SCKD_OUP  &
                   DCI_SAMP_CLK_FAL &
                   DCI_FSD_OUP &
                   DCI_TX_ZERO_UNF &
                   DCI_SDO_TRISTAT &
                   DCI_DJST_OFF &
                   DCI_FSM_MULTI;
    DCICON2value = DCI_BUFF_LEN_4 & DCI_FRAME_LEN_4 &
                     DCI_DATA_WORD_16 ;
    DCICON3value = 0x00;
    RSCONvalue   = DCI_EN_SLOT_ALL & DCI_DIS_SLOT_11 &
                   DCI_DIS_SLOT_4 & DCI_DIS_SLOT_5;
    TSCONvalue   = DCI_EN_SLOT_ALL & DCI_DIS_SLOT_11 &
                   DCI_DIS_SLOT_4 &DCI_DIS_SLOT_5;
    OpenDCI(DCICON1value, DCICON2value, DCICON3value,
         TSCONvalue, RSCONvalue);
```

```
/* Load transmit buffer and transmit the same */
    i = 0;
    while( i<= 3)
    {
        WriteDCI(data16[i],i);
        i++;
    }
/* Start generating serial clock by DCI module */
    DCICON3 = 0X02;
 /* Wait for  transmit buffer to get empty  */
    while(!BufferEmptyDCI());
/* Wait till new data is available in RX buffer */
    while(!DataRdyDCI());
/*  Read all the data remaining in receive buffer which
    are unread into user defined data buffer*/
    i = 0;
    while( i<=3)
    {
        (*( Receiveddata)++) = ReadDCI(i);
        i++;
    }
/* Turn off DCI module and clear IF bit  */
    CloseDCI();
    return 0;
}
```

3.14 SPI 函数

本节给出了关于 SPI 模块的各个函数及其使用示例。函数也可以用宏来实现。

3.14.1 各个函数

ConfigIntSPI1
ConfigIntSPI2

描述： 该函数对 SPI 中断进行配置。

头文件： spi.h

函数原型：
```
void ConfigIntSPI1(unsigned int config);
void ConfigIntSPI2(unsigned int config);
```

参数： *config* 如下定义的 SPI 中断优先级和允许 / 禁止信息：

中断允许 / 禁止
SPI_INT_EN
SPI_INT_DIS

中断优先级
SPI_INT_PRI_0
SPI_INT_PRI_1
SPI_INT_PRI_2
SPI_INT_PRI_3
SPI_INT_PRI_4
SPI_INT_PRI_5
SPI_INT_PRI_6
SPI_INT_PRI_7

返回值： 无

说明： 该函数清除中断标志位，设置中断优先级并允许 / 禁止中断。

源文件：
ConfigIntSPI1.c
ConfigIntSPI2.c

代码示例：
```
ConfigIntSPI1(SPI_INT_PRI_3 & SPI_INT_EN);
```

CloseSPI1
CloseSPI2

描述： 该函数关闭 SPI 模块。

头文件： spi.h

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

参数： 无

返回值： 无

说明： 该函数禁止 SPI 中断，然后关闭该模块。同时清除中断标志位。

源文件：
CloseSPI1.c
CloseSPI2.c

代码示例：
```
CloseSPI1();
```

DataRdySPI1
DataRdySPI2

描述： 该函数确定 SPI 缓冲器中是否包含要读取的数据。

头文件： spi.h

函数原型：
```
char DataRdySPI1(void);
char DataRdySPI2(void);
```

参数： 无

返回值： 如果返回 “1”，表明数据已接收到接收缓冲器中等待读取。
如果返回 “0”，表明接收未完成，接收缓冲器是空的。

说明： 该函数返回 SPI 接收缓冲器的状态。表明 SPI 接收缓冲器中是否包含未读取的新数据，这通过 SPIxSTAT 的 <SPIRBF> 位来指示。当从缓冲器中读取数据后，该位由硬件清除。

源文件：
```
DataRdySPI1.c
DataRdySPI2.c
```

代码示例：
```
while(DataRdySPI1());
```

ReadSPI1
ReadSPI2

描述： 该函数读取 SPI 接收缓冲寄存器 （SPIxBUF）的内容。

头文件： spi.h

函数原型：
```
unsigned int ReadSPI1(void);
unsigned int ReadSPI2(void);
```

参数： 无

返回值： 该函数返回接收缓冲寄存器 （SPIxBUF）的内容。
如果返回值为 “-1”，表明 SPI 缓冲器中没有要读取的数据。

说明： 该函数返回接收缓冲寄存器的内容。
如果使能 16 位字通信模式，则返回 SPIxBUF 中的数据。
如果使能 8 位字节通信模式，则返回 SPIxBUF 中的低字节。
SPIxBUF 只有在包含数据时才能被读取，是否包含数据由 SPISTAT 的 <RBF> 位指示。否则返回 “-1”。

源文件：
```
ReadSPI1.c
ReadSPI2.c
```

代码示例：
```
unsigned int RX_data;
RX_data = ReadSPI1();
```

WriteSPI1
WriteSPI2

描述： 该函数将要发送的数据写入发送缓冲寄存器 （SPIxBUF）中。

头文件： spi.h

函数原型：
```
void WriteSPI1(unsigned int data);
void WriteSPI2(unsigned int data);
```

参数： *data* 要发送的数据，存储到 SPI 缓冲器中。

返回值： 该函数将要发送的数据 （字节 / 字）写入发送缓冲器中。
如果使能 16 位字通信模式，则将 16 位值写入发送缓冲器中。
如果使能 8 位字节通信模式，则将高字节屏蔽后写入发送缓冲器。

说明： 无

源文件：
```
WriteSPI1.c
WriteSPI2.c
```

代码示例：
```
WriteSPI1(0x3FFF);
```

OpenSPI1
OpenSPI2

描述： 该函数对 SPI 模块进行配置。

头文件： spi.h

函数原型：
```
void OpenSPI1(unsigned int config1,
              unsigned int config2);
void OpenSPI2(unsigned int config1,
              unsigned int config2);
```

参数： *config1* 包含 SPIxCON 寄存器中要配置的参数，定义如下：

帧 SPI 支持使能 / 禁止
```
FRAME_ENABLE_ON
FRAME_ENABLE_OFF
```
帧同步脉冲方向控制
```
FRAME_SYNC_INPUT
FRAME_SYNC_OUTPUT
```
SDO 引脚控制位
```
DISABLE_SDO_PIN
ENABLE_SDO_PIN
```
字 / 字节通信模式
```
SPI_MODE16_ON
SPI_MODE16_OFF
```
SPI 数据输入采样相位
```
SPI_SMP_ON
SPI_SMP_OFF
```
SPI 时钟边沿选择
```
SPI_CKE_ON
SPI_CKE_OFF
```
SPI 从动选择模式使能
```
SLAVE_SELECT_ENABLE_ON
SLAVE_SELECT_ENABLE_OFF
```

OpenSPI1 （续）
OpenSPI2

SPI 时钟极性选择
CLK_POL_ACTIVE_LOW
CLK_POL_ACTIVE_HIGH
SPI 模式选择位
MASTER_ENABLE_ON
MASTER_ENABLE_OFF
辅助预分频比选择
SEC_PRESCAL_1_1
SEC_PRESCAL_2_1
SEC_PRESCAL_3_1
SEC_PRESCAL_4_1
SEC_PRESCAL_5_1
SEC_PRESCAL_6_1
SEC_PRESCAL_7_1
SEC_PRESCAL_8_1
主预分频比选择
PRI_PRESCAL_1_1
PRI_PRESCAL_4_1
PRI_PRESCAL_16_1
PRI_PRESCAL_64_1

config2 包含 SPIxSTAT 寄存器中要配置的参数，定义如下：

SPI 使能 / 禁止
SPI_ENABLE
SPI_DISABLE
SPI 在空闲模式下的工作
SPI_IDLE_CON
SPI_IDLE_STOP
清除接收溢出标志位
SPI_RX_OVFLOW_CLR

返回值： 无

说明： 该函数初始化 SPI 模块并设置空闲模式下的工作。

源文件：
OpenSPI1.c
OpenSPI2.c

代码示例：

```
config1 = FRAME_ENABLE_OFF &
          FRAME_SYNC_OUTPUT &
          ENABLE_SDO_PIN &
          SPI_MODE16_ON &
          SPI_SMP_ON &
          SPI_CKE_OFF &
          SLAVE_SELECT_ENABLE_OFF &
          CLK_POL_ACTIVE_HIGH &
          MASTER_ENABLE_ON &
          SEC_PRESCAL_7_1 &
          PRI_PRESCAL_64_1;
config2 = SPI_ENABLE &
          SPI_IDLE_CON &
          SPI_RX_OVFLOW_CLR OpenSPI1(config1,
           config2);
```

putsSPI1
putsSPI2

描述： 该函数将要发送的数据串写入 SPI 发送缓冲器中。

头文件： spi.h

函数原型：
```
void putsSPI1(unsigned int length,
              unsigned int *wrptr);
void putsSPI2(unsigned int length,
              unsigned int *wrptr);
```

参数：
length 要发送的数据字 / 字节数。
wrptr 指向要发送的数据串的指针。

返回值： 无

说明： 该函数将要发送的指定长度的数据字 / 字节写入发送缓冲器中。
一旦发送缓冲器为满，要等到数据发送后才能将下一个数据写入发送寄存器中。
当 SPITBF 位置位时，如果 SPI 模块禁止，控制仍保持在这一功能。

源文件： putsSPI1.c
putsSPI2.c

代码示例：
```
putsSPI1(10,Txdata_loc);
```

getsSPI1
getsSPI2

描述： 该函数读取指定长度的数据串并将它们存储到指定地址。

头文件： spi.h

函数原型：
```
unsigned int getsSPI1(
            unsigned int length,
            unsigned int *rdptr,
            unsigned int  spi_data_wait);
unsigned int getsSPI2(
            unsigned int length,
            unsigned int *rdptr,
            unsigned int  spi_data_wait);
```

参数：
length 要接收的字符串长度。
rdptr 指向存储接收到的数据的地址的指针。
spi_data_wait 模块在返回前必须等待的延时计数。
如果延时计数为 “N”，那么实际的延时时间大约为 （19*N–1）个指令周期。

返回值： 该函数返回未接收的字节数。
如果返回值为 “0”，表明已接收完整个数据串。
如果返回值不是零，表明整个数据串还未接收完。

说明： 无

源文件： getsSPI1.c
getsSPI2.c

代码示例：
```
Datarem = getsSPI1(6, Rxdata_loc, 40);
```

getcSPI1
getcSPI2

描述： 该函数等同于 ReadSPI1 和 ReadSPI2。

源文件： spi.h 中包含对 ReadSPI1 和 ReadSPI2 的宏定义。

putcSPI1
putcSPI2

描述： 该函数等同于 WriteSPI1 和 WriteSPI2。

源文件： spi.h 中包含对 WriteSPI1 和 WriteSPI2 的宏定义。

3.14.2 各个宏

EnableIntSPI1
EnableIntSPI2

描述： 该宏允许 SPI 中断。

头文件： spi.h

参数： 无

说明： 该宏中文中断允许控制寄存器的 SPI 中断允许位。

代码示例：
```
EnableIntSPI1;
```

DisableIntSPI1
DisableIntSPI2

描述： 该宏禁止 SPI 中断。

头文件： spi.h

参数： 无

说明： 该宏清除中断允许控制寄存器的 SPI 中断允许位。

代码示例：
```
DisableIntSPI2;
```

SetPriorityIntSPI1
SetPriorityIntSPI2

描述： 该宏设置 SPI 的中断优先级。

头文件： spi.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器的 SPI 中断优先级位。

代码示例：
```
SetPriorityIntSPI2(2);
```

3.14.3 使用示例

```
#define __dsPIC30F6014__
#include<p30fxxxx.h>
#include<spi.h>
/* Data received at SPI2 */
unsigned int datard ;
void __attribute__((__interrupt__)) _SPI1Interrupt(void)
{
    IFS0bits.SPI1IF = 0;
}
void __attribute__((__interrupt__)) _SPI2Interrupt(void)
{
    IFS1bits.SPI2IF = 0;
    SPI1STATbits.SPIROV = 0;  /* Clear SPI1 receive overflow
                                 flag if set */
}
int main(void)
{
/* Holds the information about SPI configuartion */
   unsigned int SPICONValue;
/* Holds the information about SPI Enable/Disable */
   unsigned int SPISTATValue;
/*Timeout value during which timer1 is ON */
   int timeout;
/* Turn off SPI modules */
    CloseSPI1();
    CloseSPI2();
    TMR1 = 0 ;
    timeout = 0;
    TRISDbits.TRISD0 = 0;
/* Configure SPI2 interrupt */
    ConfigIntSPI2(SPI_INT_EN &  SPI_INT_PRI_6);
/* Configure SPI1 module to transmit 16 bit timer1 value
   in master mode  */
    SPICONValue  =  FRAME_ENABLE_OFF & FRAME_SYNC_OUTPUT &
                    ENABLE_SDO_PIN & SPI_MODE16_ON &
                    SPI_SMP_ON & SPI_CKE_OFF &
                    SLAVE_SELECT_ENABLE_OFF &
                    CLK_POL_ACTIVE_HIGH &
                    MASTER_ENABLE_ON &
                    SEC_PRESCAL_7_1 &
                    PRI_PRESCAL_64_1;
    SPISTATValue  = SPI_ENABLE & SPI_IDLE_CON &
                    SPI_RX_OVFLOW_CLR;
    OpenSPI1(SPICONValue,SPISTATValue );
```

```
/* Configure SPI2 module to receive 16 bit timer value in
   slave mode */
    SPICONValue  =  FRAME_ENABLE_OFF & FRAME_SYNC_OUTPUT &
                    ENABLE_SDO_PIN  & SPI_MODE16_ON &
                    SPI_SMP_OFF & SPI_CKE_OFF &
                    SLAVE_SELECT_ENABLE_OFF  &
                    CLK_POL_ACTIVE_HIGH &
                    MASTER_ENABLE_OFF &
                    SEC_PRESCAL_7_1 &
                    PRI_PRESCAL_64_1;
    SPISTATValue =  SPI_ENABLE & SPI_IDLE_CON &
                    PI_RX_OVFLOW_CLR;
    OpenSPI2(SPICONValue,SPISTATValue );
    T1CON = 0X8000;
    while(timeout< 100 )
    {
        timeout = timeout+2 ;
    }
    T1CON = 0;
    WriteSPI1(TMR1);
    while(SPI1STATbits.SPITBF);
    while(!DataRdySPI2());
    datard =  ReadSPI2();
    if(datard <= 600)
    {
        PORTDbits.RD0 = 1;
    }

/* Turn off SPI module and clear IF bit  */
    CloseSPI1();
    CloseSPI2();
    return 0;
}
```

3.15 QEI 函数

本节给出了关于 QEI 模块的各个函数及其使用示例。函数也可以用宏来实现。

3.15.1 各个函数

CloseQEI

描述： 该函数关闭 QEI 模块。

头文件： qei.h

函数原型： void closeQEI(void);

参数： 无

返回值： 无

说明： 该函数禁止 QEI 模块并清除 QEI 中断允许和标志位。

源文件： CloseQEI.c

代码示例： CloseQEI();

ConfigIntQEI

描述： 该函数对 QEI 中断进行配置。

头文件： qei.h

函数原型： void ConfigIntQEI(unsigned int *config*);

参数： *config* 定义如下的 QEI 中断优先级和允许 / 禁止信息：

QEI 中断允许 / 禁止
QEI_INT_ENABLE
QEI_INT_DISABLE

QEI 中断优先级
QEI_INT_PRI_0
QEI_INT_PRI_1
QEI_INT_PRI_2
QEI_INT_PRI_3
QEI_INT_PRI_4
QEI_INT_PRI_5
QEI_INT_PRI_6
QEI_INT_PRI_7

返回值： 无

说明： 该函数清除中断标志位，设置中断优先级并允许 / 禁止中断。

源文件： ConfigIntQEI.c

代码示例： ConfigIntQEI(QEI_INT_ENABLE & QEI_INT_PRI_1);

OpenQEI

描述： 该函数对 QEI 进行配置。

头文件： qei.h

函数原型： void OpenQEI(unsigned int *config1*, unsigned int *config2*);

参数：

config1 包括 QEIxCON 寄存器中要设置的参数，定义如下：

位置计数器方向选择控制
QEI_DIR_SEL_QEB
QEI_DIR_SEL_CNTRL

定时器时钟选择位
QEI_EXT_CLK
QEI_INT_CLK

位置计数器复位使能
QEI_INDEX_RESET_ENABLE
QEI_INDEX_RESET_DISABLE

定时器输入时钟预分频比选择位
QEI_CLK_PRESCALE_1
QEI_CLK_PRESCALE_8
QEI_CLK_PRESCALE_64
QEI_CLK_PRESCALE_256

定时器选通时间累加使能
QEI_GATED_ACC_ENABLE
QEI_GATED_ACC_DISABLE

位置计数器方向状态输出使能
QEI_LOGIC_CONTROL_IO
QEI_NORMAL_IO

A 相和 B 相输入交换选择位
QEI_INPUTS_SWAP
QEI_INPUTS_NOSWAP

QEI 工作模式选择
QEI_MODE_x4_MATCH
QEI_MODE_x4_PULSE
QEI_MODE_x2_MATCH
QEI_MODE_x2_PULSE
QEI_MODE_TIMER
QEI_MODE_OFF

位置计数器方向状态
QEI_UP_COUNT
QEI_DOWN_COUNT

空闲模式下的工作
QEI_IDLE_STOP
QEI_IDLE_CON

config2 包括 DFLTxCON 寄存器中要配置的参数。

在 4x 正交计数模式下：
为与索引脉冲相匹配所需的
A 相输入信号状态
MATCH_INDEX_PHASEA_HIGH
MATCH_INDEX_PHASEA_LOW

为与索引脉冲相匹配所需的
B 相输入信号状态
MATCH_INDEX_PHASEB_HIGH
MATCH_INDEX_PHASEB_LOW

OpenQEI （续）

在 2x 计数模式下：

与索引状态匹配的相输入信号

```
MATCH_INDEX_INPUT_PHASEA
MATCH_INDEX_INPUT_PHASEB
```

与索引脉冲匹配的相输入信号状态

```
MATCH_INDEX_INPUT_HIGH
MATCH_INDEX_INPUT_LOW
```

由位置计数事件引起的中断允许 / 禁止

```
POS_CNT_ERR_INT_ENABLE
POS_CNT_ERR_INT_DISABLE
```

QEA/QEB 数字滤波器时钟分频选择位

```
QEI_QE_CLK_DIVIDE_1_1
QEI_QE_CLK_DIVIDE_1_2
QEI_QE_CLK_DIVIDE_1_4
QEI_QE_CLK_DIVIDE_1_16
QEI_QE_CLK_DIVIDE_1_32
QEI_QE_CLK_DIVIDE_1_64
QEI_QE_CLK_DIVIDE_1_128
QEI_QE_CLK_DIVIDE_1_256
```

QEA/QEB 数字滤波器输出使能

```
QEI_QE_OUT_ENABLE
QEI_QE_OUT_DISABLE
```

返回值： 无

说明： 该函数对 QEI 模块的 QEICON 和 DFLTCON 寄存器进行配置。该函数同时清除 QEICON 的 <CNTERR> 位。

源文件： OpenQEI.c

代码示例：

```
OpenQEI(QEI_DIR_SEL_QEB & QEI_INT_CLK &
        QEI_INDEX_RESET_ENABLE &
        QEI_CLK_PRESCALE_1 & QEI_NORMAL_IO &
        QEI_MODE_TIMER & QEI_UP_COUNT,0);
```

ReadQEI

描述： 该函数从 POSCNT 寄存器中读取位置计数值。

头文件： qei.h

函数原型：

```
unsigned int ReadQEI(void);
```

参数： 无

返回值： 无

说明： 该函数返回 POSCNT 寄存器的内容。

源文件： ReadQEI.c

代码示例：

```
unsigned int pos_count;
pos_count = ReadQEI();
```

WriteQEI

描述： 该函数设置 QEI 的最大计数值。

头文件： qei.h

函数原型： void WriteQEI(unsigned int *position*);

参数： *position* 要存储到 MAXCNT 寄存器中的值。

返回值： 无

说明： 无

源文件： WriteQEI.c

代码示例：
```
unsigned int position = 0x3FFF;
WriteQEI(position);
```

3.15.2 各个宏

EnableIntQEI

描述： 该宏允许 QEI 中断。

头文件： qei.h

参数： 无

说明： 该宏置位中断允许控制寄存器中的 QEI 中断允许位。

代码示例：
```
EnableIntQEI;
```

DisableIntQEI

描述： 该宏禁止 QEI 中断。

头文件： qei.h

参数： 无

说明： 该宏清除中断允许控制寄存器中的 QEI 中断允许位。

代码示例：
```
DisableIntQEI;
```

SetPriorityIntQEI

描述： 该宏设置 QEI 中断的优先级。

头文件： qei.h

参数： *priority*

说明： 该宏设置中断优先级控制寄存器中的 QEI 中断优先级位。

代码示例：
```
SetPriorityIntQEI(7);
```

3.15.3 使用示例

```
#define __dsPIC30F6010__
#include <p30fxxxx.h>
#include<qei.h>
unsigned int pos_value;

void __attribute__((__interrupt__)) _QEIInterrupt(void)
{
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */
    POSCNT = 0;
    IFS2bits.QEIIF = 0;    /* Clear QEI interrupt flag */
}
int main(void)
{
    unsigned int max_value;
    TRISDbits.TRISD1 = 0;
    PORTDbits.RD1 = 1;    /* turn off LED on RD1 */

/* Enable QEI Interrupt and Priority to "1" */
    ConfigIntQEI(QEI_INT_PRI_1 & QEI_INT_ENABLE);

    POSCNT  = 0;
    MAXCNT = 0xFFFF;
    OpenQEI(QEI_INT_CLK & QEI_INDEX_RESET_ENABLE &
            QEI_CLK_PRESCALE_256 &
            QEI_GATED_ACC_DISABLE & QEI_INPUTS_NOSWAP &
            QEI_MODE_TIMER & QEI_DIR_SEL_CNTRL &
            QEI_IDLE_CON, 0);
    QEICONbits.UPDN = 1;
    while(1)
    {
        pos_value = ReadQEI();
        if(pos_value >= 0x7FFF)
        {
            PORTDbits.RD1 = 0; /* turn on LED on RD1 */
        }
    }
    CloseQEI();
}
```

3.16 PWM 函数

本节给出了 PWM 模块的各个函数及其使用示例。函数也可以用宏来实现。

3.16.1 各个函数

CloseMCPWM

描述： 该函数关闭电机控制 PWM 模块。

头文件： pwm.h

函数原型：
```
void closeMCPWM(void);
```

参数： 无

返回值： 无

说明： 该函数禁止电机控制 PWM 模块并清除 PWM、故障 A 和故障 B 中断允许及其标志位。
该函数同时清除 PTCON、 PWMCON1 和 PWMCON2 寄存器。

源文件： CloseMCPWM.c

代码示例：
```
CloseMCPWM();
```

ConfigIntMCPWM

描述： 该函数对 PWM 中断进行配置。

头文件： pwm.h

函数原型：
```
void ConfigIntMCPWM(unsigned int config);
```

参数： *config* 定义如下的 PWM 中断优先级和允许 / 禁止信息：

PWM 中断允许 / 禁止
```
PWM_INT_EN
PWM_INT_DIS
```
PWM 中断优先级
```
PWM_INT_PR0
PWM_INT_PR1
PWM_INT_PR2
PWM_INT_PR3
PWM_INT_PR4
PWM_INT_PR5
PWM_INT_PR6
PWM_INT_PR7
```
故障 A 中断允许 / 禁止
```
PWM_FLTA_EN_INT
PWM_FLTA_DIS_INT
```
故障 A 中断优先级
```
PWM_FLTA_INT_PR0
PWM_FLTA_INT_PR1
PWM_FLTA_INT_PR2
PWM_FLTA_INT_PR3
PWM_FLTA_INT_PR4
PWM_FLTA_INT_PR5
PWM_FLTA_INT_PR6
PWM_FLTA_INT_PR7
```
故障 B 中断允许 / 禁止
```
PWM_FLTB_EN_INT
PWM_FLTB_DIS_INT
```

ConfigIntMCPWM （续）

故障 B 中断优先级
PWM_FLTB_INT_PR0
PWM_FLTB_INT_PR1
PWM_FLTB_INT_PR2
PWM_FLTB_INT_PR3
PWM_FLTB_INT_PR4
PWM_FLTB_INT_PR5
PWM_FLTB_INT_PR6
PWM_FLTB_INT_PR7

返回值： 无

说明： 该函数清除中断标志位，设置中断优先级并允许 / 禁止中断。

源文件： ConfigIntMCPWM.c

代码示例：
```
ConfigIntMCPWM(PWM_INT_EN & PWM_INT_PR5 &
               PWM_FLTA_EN_INT &
               PWM_FLTA_INT_PR6 &
               PWM_FLTB_EN_INT &
               PWM_FLTB_INT_PR7);
```

OpenMCPWM

描述： 该函数对电机控制 PWM 模块进行配置。

头文件： pwm.h

函数原型：
```
void OpenMCPWM(unsigned int period,
               unsigned int sptime,
               unsigned int config1,
               unsigned int config2,
               unsigned int config3);
```

参数：

period 包括要存储到 PTPER 寄存器中的 PWM 时基周期值。

sptime 包括要存储到 SEVTCMP 寄存器中的特殊事件比较值。

config1 包括 PTCON 寄存器中要配置的参数，定义如下：

PWM 模块使能 / 禁止
PWM_EN
PWM_DIS

空闲模式使能 / 禁止
PWM_IDLE_STOP
PWM_IDLE_CON

输出后分频比选择
PWM_OP_SCALE1
PWM_OP_SCALE2
.....
PWM_OP_SCALE15
PWM_OP_SCALE16

输入预分频比选择
PWM_IPCLK_SCALE1
PWM_IPCLK_SCALE4
PWM_IPCLK_SCALE16
PWM_IPCLK_SCALE64

OpenMCPWM （续）

PWM 工作模式
PWM_MOD_FREE
PWM_MOD_SING
PWM_MOD_UPDN
PWM_MOD_DBL

config2 包括 PWMCON1 寄存器中要配置的参数，定义如下：

PWM I/O 引脚对
PWM_MOD4_COMP
PWM_MOD3_COMP
PWM_MOD2_COMP
PWM_MOD1_COMP
PWM_MOD4_IND
PWM_MOD3_IND
PWM_MOD2_IND
PWM_MOD1_IND

PWM H/L I/O 使能 / 禁止选择
PWM_PEN4H
PWM_PDIS4H
PWM_PEN3H
PWM_PDIS3H
PWM_PEN2H
PWM_PDIS2H
PWM_PEN1H
PWM_PDIS1H
PWM_PEN4L
PWM_PDIS4L
PWM_PEN3L
PWM_PDIS3L
PWM_PEN2L
PWM_PDIS2L
PWM_PEN1L
PWM_PDIS1L

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

config3 包括 PWMCON2 寄存器中要配置的参数，定义如下：

特殊事件后分频比
PWM_SEVOPS1
PWM_SEVOPS2
.....
PWM_SEVOPS15
PWM_SEVOPS16

输出改写同步选择
PWM_OSYNC_PWM
PWM_OSYNC_Tcy

PWM 更新使能 / 禁止
PWM_UDIS
PWM_UEN

返回值： 无

说明： 该函数对 PTPER、 SEVTCMP、 PTCON、 PWMCON1 和 PWMCON2 寄存器进行配置。

OpenMCPWM （续）

源文件： OpenMCPWM.c

代码示例：

```
period = 0x7fff;
sptime = 0x0;
config1 = PWM_EN & PWM_PTSIDL_DIS &
          PWM_OP_SCALE16 &
          PWM_IPCLK_SCALE16 &
          PWM_MOD_UPDN;
config2 = PWM_MOD1_COMP & PWM_PDIS4H &
          PWM_PDIS3H & PWM_PDIS2H &
          PWM_PEN1H &  PWM_PDIS4L &
          PWM_PDIS3L & PWM_PDIS2L &
          PWM_PEN1L;
config3 = PWM_SEVOPS1 & PWM_OSYNC_PWM &
          PWM_UEN;
OpenMCPWM(period, sptime, config1,
          config2, config3);
```

OverrideMCPWM

描述： 该函数对 OVDCON 寄存器进行配置。

头文件： pwm.h

函数原型： `void OverrideMCPWM(unsigned int config);`

参数： *config* 包括 OVDCON 寄存器中要配置的参数，定义如下：

由 PWM 发生器控制的输出

PWM_GEN_4H
PWM_GEN_3H
PWM_GEN_2H
PWM_GEN_1H
PWM_GEN_4L
PWM_GEN_3L
PWM_GEN_2L
PWM_GEN_1L

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

由 POUT 位控制的输出

PWM_POUT_4H
PWM_POUT_4L
PWM_POUT_3H
PWM_POUT_3L
PWM_POUT_2H
PWM_POUT_2L
PWM_POUT_1H
PWM_POUT_1L

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

OverrideMCPWM （续）

PWM 手动输出位
PWM_POUT4H_ACT
PWM_POUT4H_INACT
PWM_POUT4L_ACT
PWM_POUT4L_INACT
PWM_POUT3H_ACT
PWM_POUT3H_INACT
PWM_POUT3L_ACT
PWM_POUT3L_INACT
PWM_POUT2H_ACT
PWM_POUT2H_INACT
PWM_POUT2L_ACT
PWM_POUT2L_INACT
PWM_POUT1H_ACT
PWM_POUT1H_INACT
PWM_POUT1L_ACT
PWM_POUT1L_INACT

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

返回值： 无

说明： 该函数对 PWM 输出改写和 OVDCON 寄存器的手动控制位进行配置。

源文件： OverrideMCPWM.c

代码示例：

```
config = PWM_GEN_1L &
         PWM_GEN_1H &
         PWM_POUT1L_INACT &
         PWM_POUT3L_INACT;
OverrideMCPWM(config);
```

SetDCMCPWM

描述： 该函数对占空比寄存器进行配置并更新 PWMCON2 寄存器中的“PWM 更新禁止”位。

头文件： pwm.h

函数原型：

```
void SetDCMCPWM(
            unsigned int dutycyclereg,
            unsigned int dutycycle,
            char updatedisable);
```

参数：

dutycyclereg 指向占空比寄存器的指针。

dutycycle 要存储到占空比寄存器中的值。

updatedisable 要装载到 PWMCON2 寄存器的“更新禁止”位的值。

返回值： 无

说明： 无

源文件： SetDCMCPWM.c

代码示例：

```
dutycyclereg = 1;
dutycycle = 0xFFF;
updatedisable = 0;
SetDCMCPWM(dutycyclereg, dutycycle,updatedisable);
```

SetMCPWMDeadTimeAssignment

描述： 该函数对 PWM 输出对的死区单元分配进行配置。

头文件： pwm.h

函数原型：
```
void SetMCPWMDeadTimeAssignment (unsigned int config);
```

参数： *config* 包括 DTCON2 寄存器中要配置的参数，定义如下：

PWM4 信号死区选择位
```
PWM_DTS4A_UA
PWM_DTS4A_UB
PWM_DTS4I_UA
PWM_DTS4I_UB
```
与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

PWM3 信号死区选择位
```
PWM_DTS3A_UA
PWM_DTS3A_UB
PWM_DTS3I_UA
PWM_DTS3I_UB
```
PWM2 信号死区选择位
```
PWM_DTS2A_UA
PWM_DTS2A_UB
PWM_DTS2I_UA
PWM_DTS2I_UB
```
PWM1 信号死区选择位
```
PWM_DTS1A_UA
PWM_DTS1A_UB
PWM_DTS1I_UA
PWM_DTS1I_UB
```

返回值： 无

说明： 无

源文件： SetMCPWMDeadTimeAssignment.c

代码示例：
```
SetMCPWMDeadTimeAssignment(PWM_DTS3A_UA &
PWM_DTS2I_UA & PWM_DTS1I_UA);
```

SetMCPWMDeadTimeGeneration

描述： 该函数配置死区值和时钟预分频比。

头文件： pwm.h

函数原型：
```
void SetMCPWMDeadTimeGeneration(
          unsigned int config);
```

参数： *config* 包括 DTCON1 寄存器中要配置的参数，定义如下：

死区单元 B 预分频比选择位
PWM_DTBPS8
PWM_DTBPS4
PWM_DTBPS2
PWM_DTBPS1

死区单元 A 预分频比选择常数
PWM_DTA0
PWM_DTA1
PWM_DTA2
.....
PWM_DTA62
PWM_DTA63

死区单元 B 预分频比选择常数
PWM_DTB0
PWM_DTB1
PWM_DTB2
.....
PWM_DTB62
PWM_DTB63

死区单元 A 预分频比选择位
PWM_DTAPS8
PWM_DTAPS4
PWM_DTAPS2
PWM_DTAPS1

返回值： 无

说明： 无

源文件： SetMCPWMDeadTimeGeneration.c

代码示例：
```
SetMCPWMDeadTimeGeneration(PWM_DTBPS16 &
PWM_DT54 & PWM_DTAPS8);
```

SetMCPWMFaultA

描述： 该函数对 PWM 的故障 A 改写位、故障 A 模式位和故障输入 A 使能位进行配置。

头文件： pwm.h

函数原型： void SetMCPWMFaultA(unsigned int *config*);

参数： *config* 包括 FLTACON 寄存器中要配置的参数，定义如下：

故障输入 A 的 PWM 改写值位

PWM_OVA4H_ACTIVE
PWM_OVA3H_ACTIVE
PWM_OVA2H_ACTIVE
PWM_OVA1H_ACTIVE
PWM_OVA4L_ACTIVE
PWM_OVA3L_ACTIVE
PWM_OVA2L_ACTIVE
PWM_OVA1L_ACTIVE
PWM_OVA4H_INACTIVE
PWM_OVA3H_INACTIVE
PWM_OVA2H_INACTIVE
PWM_OVA1H_INACTIVE
PWM_OVA4L_INACTIVE
PWM_OVA3L_INACTIVE
PWM_OVA2L_INACTIVE
PWM_OVA1L_INACTIVE

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

故障 A 模式位

PWM_FLTA_MODE_CYCLE
PWM_FLTA_MODE_LATCH

故障输入 A 使能位

PWM_FLTA4_EN
PWM_FLTA4_DIS
PWM_FLTA3_EN
PWM_FLTA3_DIS
PWM_FLTA2_EN
PWM_FLTA2_DIS
PWM_FLTA1_EN
PWM_FLTA1_DIS

与 PWM4 相关的位定义仅适用于某些器件，应参考相应的数据手册。

返回值： 无

说明： 无

源文件： SetMCPWMFaultA.c

代码示例：

```
SetMCPWMFaultA(PWM_OVA3L_INACTIVE &
               PWM_FLTA_MODE_LATCH &
               PWM_FLTA1_DIS);
```

SetMCPWMFaultB

描述： 该函数对 PWM 的故障 B 改写位、故障 B 模式位和故障输入 B 使能位进行配置。

头文件： pwm.h

函数原型： void SetMCPWMFaultB(unsigned int *config*);

参数： *config* 包括 FLTBCON 寄存器中要配置的参数，定义如下：
只有某些器件具有 FLTBCON 寄存器，应参考相应的数据手册。

故障输入 B 的 PWM 改写值位
PWM_OVB4H_ACTIVE
PWM_OVB3H_ACTIVE
PWM_OVB2H_ACTIVE
PWM_OVB1H_ACTIVE
PWM_OVB4L_ACTIVE
PWM_OVB3L_ACTIVE
PWM_OVB2L_ACTIVE
PWM_OVB1L_ACTIVE
PWM_OVB4H_INACTIVE
PWM_OVB3H_INACTIVE
PWM_OVB2H_INACTIVE
PWM_OVB1H_INACTIVE
PWM_OVB4L_INACTIVE
PWM_OVB3L_INACTIVE
PWM_OVB2L_INACTIVE
PWM_OVB1L_INACTIVE

故障 B 模式位
PWM_FLTB_MODE_CYCLE
PWM_FLTB_MODE_LATCH

故障输入 B 使能位
PWM_FLTB4_EN
PWM_FLTB4_DIS
PWM_FLTB3_EN
PWM_FLTB3_DIS
PWM_FLTB2_EN
PWM_FLTB2_DIS
PWM_FLTB1_EN
PWM_FLTB1_DIS

返回值： 无

说明： 无

源文件： SetMCPWMFaultB.c

代码示例：

```
SetMCPWMFaultB(PWM_OVB3L_INACTIVE &
               PWM_FLTB_MODE_LATCH &
               PWM_FLTB2_DIS);
```

3.16.2 各个宏

EnableIntMCPWM

描述： 该宏允许 PWM 中断。
头文件： pwm.h
参数： 无
说明： 该宏置位中断允许控制寄存器中的 PWM 中断允许位。
代码示例：
```
EnableIntMCPWM;
```

DisableIntMCPWM

描述： 该宏禁止 PWM 中断。
头文件： pwm.h
参数： 无
说明： 该宏清除中断允许控制寄存器中的 PWM 中断允许位。
代码示例：
```
DisableIntMCPWM;
```

SetPriorityIntMCPWM

描述： 该宏设置 PWM 中断的优先级。
头文件： pwm.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的 PWM 中断优先级位。
代码示例：
```
SetPriorityIntMCPWM(7);
```

EnableIntFLTA

描述： 该宏允许 FLTA 中断。
头文件： pwm.h
参数： 无
说明： 该宏置位中断允许控制寄存器中的 FLTA 中断允许位。
代码示例：
```
EnableIntFLTA;
```

DisableIntFLTA

描述： 该宏禁止 FLTA 中断。
头文件： pwm.h
参数： 无
说明： 该宏清除中断允许控制寄存器中的 FLTA 中断允许位。
代码示例：
```
DisableIntFLTA;
```

SetPriorityIntFLTA

描述： 该宏设置 FLTA 中断的优先级。
头文件： pwm.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的 FLTA 中断优先级位。
代码示例：
```
SetPriorityIntFLTA(7);
```

EnableIntFLTB

描述： 该宏允许 FLTB 中断。
头文件： pwm.h
参数： 无
说明： 该宏置位中断允许控制寄存器中的 FLTB 中断允许位。
代码示例：
```
EnableIntFLTB;
```

DisableIntFLTB

描述： 该宏禁止 FLTB 中断。
头文件： pwm.h
参数： 无
说明： 该宏清除中断允许控制寄存器中的 FLTB 中断允许位。
代码示例：
```
DisableIntFLTB;
```

SetPriorityIntFLTB

描述： 该宏设置 FLTB 中断的优先级。
头文件： pwm.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的 FLTB 中断优先级位。
代码示例：
```
SetPriorityIntFLTB(1);
```

3.16.3 使用示例

```
#define __dsPIC30F6010__
#include <p30fxxxx.h>
#include<pwm.h>
void __attribute__((__interrupt__)) _PWMInterrupt(void)
{
   IFS2bits.PWMIF = 0;
}
int main()
{
/* Holds the  PWM interrupt configuration value*/
    unsigned int config;
/* Holds the value to be loaded into dutycycle register */
    unsigned int period;
/* Holds the value to be loaded into special event compare register */
    unsigned int sptime;
/* Holds PWM configuration value  */
    unsigned int config1;
/* Holds the value be loaded into PWMCON1 register */
    unsigned int config2;
/* Holds the value to configure the special event trigger
   postscale and dutycycle */
    unsigned int config3;
/* The value of 'dutycyclereg' determines the duty cycle
   register(PDCx) to be written */
    unsigned int dutycyclereg;
    unsigned int dutycycle;
    unsigned char updatedisable;

/* Configure pwm interrupt enable/disable and set interrupt
   priorties */
    config = (PWM_INT_EN & PWM_FLTA_DIS_INT & PWM_INT_PR1
             & PWM_FLTA_INT_PR0
             & PWM_FLTB_DIS_INT & PWM_FLTB_INT_PR0);
    ConfigIntMCPWM( config );
/* Configure PWM to generate square wave of 50% duty  cycle */
    dutycyclereg  = 1;
    dutycycle     = 0x3FFF;
    updatedisable = 0;

    SetDCMCPWM(dutycyclereg,dutycycle,updatedisable);
    period = 0x7fff;
    sptime = 0x0;
    config1 = (PWM_EN & PWM_PTSIDL_DIS & PWM_OP_SCALE16
               & PWM_IPCLK_SCALE16 &
                 PWM_MOD_UPDN);
    config2 = (PWM_MOD1_COMP & PWM_PDIS4H & PWM_PDIS3H &
               PWM_PDIS2H & PWM_PEN1H & PWM_PDIS4L &
               PWM_PDIS3L & PWM_PDIS2L & PWM_PEN1L);
    config3  = (PWM_SEVOPS1 & PWM_OSYNC_PWM &  PWM_UEN);
    OpenMCPWM(period,sptime,config1,config2,config3);
    while(1);
}
```

3.17 I²C 函数

本节给出了关于 I²C 模块的各个函数及其使用示例。函数也可以用宏来实现。

CloseI2C

描述： 该函数关闭 I²C 模块。

头文件： i2c.h

函数原型： void CloseI2C(void);

参数： 无

返回值： 无

说明： 该函数禁止 I²C 模块并清除主从中断允许以及标志位。

源文件： CloseI2C.c

代码示例： CloseI2C();

ConfigIntI2C

描述： 该函数对 I²C 中断进行配置。

头文件： i2c.h

函数原型： void ConfigIntI2C(unsigned int *config*);

参数： *config* 定义如下的 I²C 中断优先级和允许 / 禁止信息：

I²C 主中断允许 / 禁止

MI2C_INT_ON
MI2C_INT_OFF
I2C slave Interrupt enable/disable
SI2C_INT_ON
SI2C_INT_OFF

I²C 主中断优先级

MI2C_INT_PRI_7
MI2C_INT_PRI_6
MI2C_INT_PRI_5
MI2C_INT_PRI_4
MI2C_INT_PRI_3
MI2C_INT_PRI_2
MI2C_INT_PRI_1
MI2C_INT_PRI_0

I²C 从中断优先级

SI2C_INT_PRI_7
SI2C_INT_PRI_6
SI2C_INT_PRI_5
SI2C_INT_PRI_4
SI2C_INT_PRI_3
SI2C_INT_PRI_2
SI2C_INT_PRI_1
SI2C_INT_PRI_0

返回值： 无

ConfigIntI2C （续）

说明： 该函数清除中断标志位，设置主从中断优先级并允许 / 禁止中断。

源文件： ConfigIntI2C.c

代码示例：
```
ConfigIntI2C(MI2C_INT_ON & MI2C_INT_PRI_3
& SI2C_INT_ON & SI2C_INT_PRI_5);
```

AckI2C

描述： 产生 I^2C 总线应答条件。

头文件： i2c.h

函数原型：
```
void AckI2C(void);
```

参数： 无

返回值： 无

说明： 该函数产生 I^2C 总线应答条件。

源文件： AckI2C.c

代码示例：
```
AckI2C();
```

DataRdyI2C

描述： 如果 I2CRCV 寄存器中有数据，该函数向用户返回状态。

头文件： i2c.h

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

参数： 无

返回值： 如果 I2CRCV 寄存器中有数据，该函数返回 “1”；否则返回 “0”表明 I2CRCV 寄存器中没有数据。

说明： 该函数确定 I2CRCV 寄存器中是否有数据可读。

源文件： DataRdyI2C.c

代码示例：
```
if(DataRdyI2C());
```

IdleI2C

描述： 该函数产生等待条件直到 I^2C 总线空闲为止。

头文件： i2c.h

函数原型：
```
void IdleI2C(void);
```

参数： 无

返回值： 无

说明： 该函数将处于等待状态直到 I^2C 控制寄存器中的启动条件使能位、停止条件使能位、接收使能位、应答序列使能位和 I^2C 条件寄存器中的发送状态位被清零为止。由于硬件 I^2C 外设不允许队列缓冲总线序列，所以需要 IdleI2C 函数。在 I^2C 操作启动之前或写冲突发生之前， I^2C 外设必须处于空闲状态。

源文件： IdleI2C.c

代码示例：
```
IdleI2C();
```

MastergetsI2C

描述： 该函数从 I²C 总线上读取预定义长度的数据串。

头文件： i2c.h

函数原型：
```
unsigned int MastergetsI2C(unsigned int length,
unsigned char *rdptr, unsigned int i2c_data_wait);
```

参数：

length	从 I²C 器件读取的字节数。
rdptr	指向存储从I²C器件中读取的数据的dsPIC RAM的字符型指针。
i2c_data_wait	模块在返回前必须等待的延时计数。 如果延时计数为 “N”，那么实际的延时时间为 (20*N-1) 个指令周期。

返回值： 如果所有字节都已发送，则该函数返回 0；如果在指定的 *i2c_data_wait* 延时时间内无法读完数据，则返回从I²C总线上读取的字节数。

说明： 该函数从 I²C 总线上读取一个预定义长度的数据串。

源文件： MastergetsI2C.c

代码示例：
```
unsigned char string[10];
unsigned char *rdptr;
unsigned int length, i2c_data_wait;
length = 9;
rdptr = string;
i2c_data_wait = 152;
MastergetsI2C(length, rdptr, i2c_data_wait);
```

MasterputsI2C

描述： 该函数用于向 I²C 总线写一个数据串。

头文件： i2c.h

函数原型：
```
unsigned int MasterputsI2C(unsigned char *wrptr);
```

参数：

wrptr	指向 dsPIC RAM 中的数据对象的字符型指针。数据对象将被写到 I²C 器件中。

返回值： 如果发生写冲突，该函数将返回 -3。如果数据串中遇到空字符，该函数将返回 0。

说明： 该函数将一个字符串写到 I²C 总线，直到遇到空字符为止。通过调用 MasterputcI2C 函数来写每个字节。实际调用的函数主体名为 MasterWriteI2C。通过 i2c.h 中的一条宏定义语句，MasterWriteI2C 和 MasterputcI2C 指同一个函数。

源文件： MasterputsI2C.c

代码示例：
```
unsigned char string[] = " MICROCHIP ";
unsigned char *wrptr;
wrptr = string;
MasterputsI2C( wrptr);
```

MasterReadI2C

描述： 该函数用来从 I²C 总线上读取一个字节。

头文件： i2c.h

函数原型：
```
unsigned char MasterReadI2C(void);
```

参数： 无

返回值： 返回值为从 I²C 总线上读取的数据字节。

说明： 该函数从 I²C 总线上读取一个字节。
该函数与 MastergetcI2C 具有相同的功能。

源文件： MasterReadI2C.c

代码示例：
```
unsigned char value;
value = MasterReadI2C();
```

MasterWriteI2C

描述： 该函数用来向 I²C 器件写一个数据字节。

头文件： i2c.h

函数原型：
```
unsigned char MasterWriteI2C(unsigned char
data_out);
```

参数： *data_out* 要写到 I²C 总线器件的一个数据字节。

返回值： 如果发生写冲突，该函数返回 -1，否则返回 0。

说明： 该函数向 I²C 总线器件写一个数据字节。该函数与 MasterputcI2C 具有相同的功能。

源文件： MasterWriteI2C.c

代码示例：
```
MasterWriteI2C('a');
```

NotAckI2C

描述： 产生 I²C 总线无应答条件。

头文件： i2c.h

函数原型：
```
void NotAckI2C(void);
```

参数： 无

返回值： 无

说明： 该函数产生一个 I²C 总线*无应答*条件。

源文件： NotAckI2C.c

代码示例：
```
NotAckI2C();
```

OpenI2C

描述： 配置 I²C 模块。

头文件： i2c.h

函数原型： void OpenI2C(unsigned int *config1*, unsigned int *config2*);

参数：

config1 包括 I2CCON 寄存器中要配置的参数。

I²C 使能位
I2C_ON
I2C_OFF

I²C 空闲模式停止位
I2C_IDLE_STOP
I2C_IDLE_CON

SCL 释放控制位
I2C_CLK_REL
I2C_CLK_HLD

智能外设管理接口使能位
I2C_IPMI_EN
I2C_IPMI_DIS

10 位从地址位
I2C_10BIT_ADD
I2C_7BIT_ADD

禁止压摆率控制位
I2C_SLW_DIS
I2C_SLW_EN

SMBus 输入级别位
I2C_SM_EN
I2C_SM_DIS

全局呼叫使能位
I2C_GCALL_EN
I2C_GCALL_DIS

SCL 时钟延长使能位
I2C_STR_EN
I2C_STR_DIS

应答数据位
I2C_ACK
I2C_NACK

应答序列使能位
I2C_ACK_EN
I2C_ACK_DIS

接收使能位
I2C_RCV_EN
I2C_RCV_DIS

停止状态使能位
I2C_STOP_EN
I2C_STOP_DIS

重复启动条件使能位
I2C_RESTART_EN
I2C_RESTART_DIS

启动条件使能位
I2C_START_EN
I2C_START_DIS

config2 波特率发生器的计算值。

OpenI2C （续）

返回值： 无

说明： 该函数配置 I²C 控制寄存器和 I²C 波特率发生器寄存器。

源文件： OpenI2C.c

代码示例：
```
OpenI2C();
```

RestartI2C

描述： 产生 I²C 总线重新启动条件。

头文件： i2c.h

函数原型：
```
void RestartI2C(void);
```

参数： 无

返回值： 无

说明： 该函数产生一个 I²C 总线重新启动条件。

源文件： RestartI2C.c

代码示例：
```
RestartI2C();
```

SlavegetsI2C

描述： 该函数从 I²C 总线上读取预定义长度的数据串。

头文件： i2c.h

函数原型：
```
unsigned int SlavegetsI2C(unsigned char *rdptr,
unsigned int i2c_data_wait);
```

参数：

rdptr 指向存储从 I²C 器件中读取的数据的 dsPIC RAM 的字符型指针。

i2c_data_wait 模块在返回前必须等待的延时计数。
如果延时计数为 “N”，那么实际的延时时间大约为 (20*N-1) 个指令周期。

返回值： 返回从 I²C 总线上接收到的字节数。

说明： 该函数从 I²C 总线上读取预定义长度的数据串。

源文件： SlavegetsI2C.c

代码示例：
```
unsigned char string[12];
unsigned char *rdptr;
rdptr = string;
i2c_data_out = 0x11;
SlavegetsI2C(rdptr, i2c_data_wait);
```

SlaveputsI2C

描述： 该函数用来向 I²C 总线写一个数据串。

头文件： i2c.h

函数原型： unsigned int SlaveputsI2C(unsigned char *wrptr);

参数： *wrptr* 指向 dsPIC RAM 中数据对象的字符型指针。将数据对象写到 I²C 器件。

返回值： 如果在数据串中遇到空字符，则该函数返回 “0”。

说明： 该函数向 I²C 总线写一个数据串，直到遇到空字符为止。

源文件： SlaveputsI2C.c

代码示例：

```
unsigned char string[] =”MICROCHIP”;
unsigned char *rdptr;
rdptr = string;
SlaveputsI2C(rdptr);
```

SlaveReadI2C

描述： 该函数用来从 I²C 总线上读取一个字节。

头文件： i2c.h

函数原型： unsigned char SlaveReadI2C(void);

参数： 无

返回值： 返回值为从 I²C 总线上读取的数据字节。

说明： 该函数从 I²C 总线上读取单个字节。该函数与 SlavegetcI2C 具有相同的功能。

源文件： SlaveReadI2C.c

代码示例：

```
unsigned char value;
value = SlaveReadI2C();
```

SlaveWriteI2C

描述： 该函数用来向 I²C 总线写一个字节。

头文件： i2c.h

函数原型： void SlaveWriteI2C(unsigned char *data_out*);

参数： *data_out* 要写入 I²C 总线的一个数据字节。

返回值： 无

说明： 该函数向 I²C 总线写一个数据字节。该函数与 SlaveputcI2C 具有相同的功能。

源文件： SlaveWriteI2C.c

代码示例： SlaveWriteI2C(‘a’);

StartI2C

描述： 产生 I^2C 总线启动条件。
头文件： i2c.h
函数原型： void StartI2C(void);
参数： 无
返回值： 无
说明： 该函数产生一个 I^2C 总线启动条件。
源文件： StartI2C.c
代码示例： StartI2C();

StopI2C

描述： 产生 I^2C 总线停止条件。
头文件： i2c.h
函数原型： void StopI2C(void);
参数： 无
返回值： 无
说明： 该函数产生一个 I^2C 总线停止条件。
源文件： StopI2C.c
代码示例： StopI2C();

3.17.1 各个宏

EnableIntMI2C

描述： 该宏允许主 I^2C 中断。
头文件： i2c.h
参数： 无
说明： 该宏置位中断允许控制寄存器中的主 I^2C 中断允许位。
代码示例： EnableIntMI2C;

DisableIntMI2C

描述： 该宏禁止主 I^2C 中断。
头文件： i2c.h
参数： 无
说明： 该宏清除中断允许控制寄存器中的主 I^2C 中断允许位。
代码示例： DisableIntMI2C;

SetPriorityIntMI2C

描述： 该宏设置主 I^2C 中断的优先级。
头文件： i2c.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器中的主 I^2C 中断优先级位。
代码示例：
```
SetPriorityIntMI2C(1);
```

EnableIntSI2C

描述： 该宏允许从 I^2C 中断。
头文件： i2c.h
参数： 无
说明： 该宏置位中断允许控制寄存器的从 I^2C 中断允许位。
代码示例：
```
EnableIntSI2C;
```

DisableIntSI2C

描述： 该宏禁止从 I^2C 中断。
头文件： i2c.h
参数： 无
说明： 该宏清除中断允许控制寄存器的从 I^2C 中断允许位。
代码示例：
```
DisableIntSI2C;
```

SetPriorityIntSI2C

描述： 该宏设置主 I^2C 中断的优先级。
头文件： i2c.h
参数： *priority*
说明： 该宏设置中断优先级控制寄存器的主 I^2C 中断优先级位。
代码示例：
```
SetPriorityIntSI2C(4);
```

3.17.2 使用示例

```
#define __dsPIC30F6014__
#include <p30fxxxx.h>
#include<i2c.h>

void main(void )
{
    unsigned int config2, config1;
    unsigned char *wrptr;
    unsigned char tx_data[] =
    {'M','I','C','R','O','C','H','I','P','\0'};
    wrptr   = tx_data;
/* Baud rate is set for 100 Khz */
    config2 = 0x11;
/* Configure I2C for 7 bit address mode */
    config1 = (I2C_ON & I2C_IDLE_CON & I2C_CLK_HLD
               & I2C_IPMI_DIS & I2C_7BIT_ADD
               & I2C_SLW_DIS & I2C_SM_DIS &
               I2C_GCALL_DIS & I2C_STR_DIS &
               I2C_NACK & I2C_ACK_DIS & I2C_RCV_DIS &
               I2C_STOP_DIS & I2C_RESTART_DIS
               & I2C_START_DIS);
    OpenI2C(config1,config2);
    IdleI2C();
    StartI2C();
/* Wait till Start sequence is completed */
    while(I2CCONbits.SEN );
/* Write Slave address and set master for transmission */
    MasterWriteI2C(0xE);
/* Wait till address is transmitted */
    while(I2CSTATbits.TBF);
    while(I2CSTATbits.ACKSTAT);
/* Transmit string of data */
    MasterputsI2C(wrptr);
    StopI2C();
/* Wait till stop sequence is completed */
    while(I2CCONbits.PEN);
    CloseI2C();
}
```

第 4 章 标准 C 函数库和数学函数

4.1 简介

标准 ANSI C 库函数包含在 libc-*omf*.a 库和 libm-*omf*.a 库 （数学函数）中，其中 *omf* 为 coff 或 elf，这取决于所选择的目标模块格式。

另外，某些 dsPIC 标准 C 函数库的辅助函数和一些必须修改后才能适用于 dsPIC 器件的标准函数都在 libpic30-*omf*.a 库中。

4.1.1 汇编代码应用程序

从 Microchip 网站上可以获得数学函数库和头文件的免费版本，但免费版本中不包含源代码。

4.1.2 C 代码应用程序

MPLAB C30 C 编译器安装在目录 （c:\pic30_tools）下，这个目录中包含下列子目录以及与库相关的文件：

- lib — 标准 C 库文件
- src\libm — 数学库函数的源代码，以及用于重建库的批处理文件
- support\h — 库的头文件

4.1.3 章节结构

本章的结构如下：

- 使用标准 C 函数库

libc-*omf*.a

- <assert.h> 诊断
- <ctype.h> 字符处理
- <errno.h> 错误
- <float.h> 浮点特征
- <limits.h> 实现定义的限制 （ implementation-defined limits)
- <locale.h> 语言环境
- <setjmp.h> 与语言环境无关的跳转
- <signal.h> 信号处理
- <stdarg.h> 可变参数列表
- <stddef.h> 公共定义
- <stdio.h> 输入和输出
- <stdlib.h> 实用函数
- <string.h> 字符串函数
- <time.h> 日期和时间函数

libm-*omf*.a

- <math.h> 数学函数

libpic30-*omf*.a

- pic30 函数库

4.2 使用标准 C 函数库

利用标准 C 函数库构建应用程序需要两种类型的文件：头文件和库文件。

4.2.1 头文件

在一个或多个标准头文件 （参见**第 4.1.3 节 “章节结构”**）中声明或定义所有标准 C 函数库实体。为了在程序中使用一个库实体，要写一条 include 伪指令来指定相关的标准头文件 。

通过在 include 伪指令中指定一个标准头文件来包含标准头文件的内容，如：

```
#include <stdio.h> /* include I/O facilities */
```

可以用任何顺序包含标准头文件。但是不要在声明中包含标准头文件。在包含标准头文件之前，不要定义与关键字同名的宏。

一个标准头文件不能包含另一个标准头文件。

4.2.2 库文件

归档库文件包含每个库函数的目标文件。

当链接应用程序时，库文件必须作为链接器的一个输入（使用 `--library` 或 `-l` 链接器开关），这样应用程序使用的函数将被链接到应用程序。

一个典型的 C 应用程序需要三个库文件：`libc-`*`omf`*`.a`、`libm-`*`omf`*`.a` 和 `libpic30-`*`omf`*`.a`（更多 OMF 特定库的信息，请参阅**第 1.2 节“特定于 OMF 的库 / 启动模块”**。）如果采用 MPLAB C30 编译器来链接的话，这些库将自动被包含。

注：	一些标准库函数需要使用堆，这些函数包括打开文件的标准 I/O 函数和存储空间分配函数。关于堆的更多信息，请参阅 *MPLAB ASM30, MPLAB LINK30 and Utilities User’s Guide* 和 *《MPLAB C30 C 编译器用户指南》*。

4.3 <ASSERT.H> 诊断

头文件 assert.h 由单个宏组成，用于调试程序中的逻辑错误。在某些条件应为真的程序关键位置使用 assert 语句，可以测试程序的逻辑。

通过在包含 <assert.h> 前定义 NDEBUG，可以在不删除代码的情况下关闭断言测试。如果定义了宏 NDEBUG， assert() 将被忽略，也不会产生任何代码。

assert

描述： 如果表达式的值为假，将打印一条断言消息到 stdeer 并中止程序。

头文件： <assert.h>

函数原型： void assert(int *expression*);

参数： *expression*要测试的表达式。

说明： 表达式求值为零或非零。如果为零，则断言失败，并打印一条消息到 stdeer。消息包括源文件名 （__FILE__）、源代码行号（__LINE__）、被求值的表达式和消息。然后宏调用函数 abort()。如果定义了宏 _VERBOSE_DEBUGGING，则每次调用 assert() 时都会打印一条消息到 stdeer。

示例：

```
#include <assert.h> /* for assert */

int main(void)
{
  int a;

  a = 2 * 2;
  assert(a == 4); /* if true-nothing prints */
  assert(a == 6); /* if false-print message */
                  /* and abort */
}
```

输出：

```
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

如果定义了宏 _VERBOSE_DEBUGGING，则输出如下：

```
sampassert.c:8 a == 4 -- OK
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

4.4 <CTYPE.H> 字符处理

头文件 ctype.h 由用来对字符进行排序和映射的函数组成。按照标准 C 语言环境来解释字符。

isalnum

描述： 测试字符是否为字母或数字字符。

头文件： <ctype.h>

函数原型： `int isalnum(int c);`

参数： *c* 要测试的字符。

返回值： 如果该字符是字母数字字符，则返回非零的整数值；否则返回零。

说明： 字母数字字符包含在 A-Z、 a-z 或 0-9 的范围内。

示例：

```
#include <ctype.h> /* for isalnum */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isalnum(ch))
    printf("3 is an alphanumeric\n");
  else
    printf("3 is NOT an alphanumeric\n");

  ch = '#';
  if (isalnum(ch))
    printf("# is an alphanumeric\n");
  else
    printf("# is NOT an alphanumeric\n");
}
```

输出：

```
3 is an alphanumeric
# is NOT an alphanumeric
```

isalpha

描述： 测试字符是否为字母字符。

头文件： <ctype.h>

函数原型： `int isalpha(int c);`

参数： *c* 要测试的字符。

返回值： 如果该字符是字母字符，则返回非零的整数值；否则返回零。

说明： 字母字符包含在 A-Z 或 a-z 的范围内。

isalpha （续）

示例：

```
#include <ctype.h> /* for isalpha */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isalpha(ch))
    printf("B is alphabetic\n");
  else
    printf("B is NOT alphabetic\n");

  ch = '#';
  if (isalpha(ch))
    printf("# is alphabetic\n");
  else
    printf("# is NOT alphabetic\n");
}
```

输出：

```
B is an alphanumeric
# is NOT an alphanumeric
```

iscntrl

描述： 测试字符是否为控制字符。

头文件： <ctype.h>

函数原型： `int iscntrl(int c);`

参数： *c* 要测试的字符。

返回值： 如果字符是控制字符，则返回非零的整数值；否则返回零。

说明： 如果字符的 ASCII 值是在 0x00 ～ 0x1F 之间或是 0x7F，那么该字符被认为是控制字符。

示例：

```
#include <ctype.h> /* for iscntrl */
#include <stdio.h> /* for printf */

int main(void)
{
  char ch;

  ch = 'B';
  if (iscntrl(ch))
    printf("B is a control character\n");
  else
    printf("B is NOT a control character\n");

  ch = '\t';
  if (iscntrl(ch))
    printf("A tab is a control character\n");
  else
    printf("A tab is NOT a control character\n");
}
```

输出：

```
B is NOT a control character
A tab is a control character
```

isdigit

描述： 测试字符是否为十进制数。

头文件： <ctype.h>

函数原型： int isdigit(int *c*);

参数： *c* 要测试的字符。

返回值： 如果字符是一个数字，则返回非零的整数值；否则返回零。

说明： 如果字符在 0-9 范围内，那么该字符被认为是数字字符。

示例：

```
#include <ctype.h> /* for isdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isdigit(ch))
    printf("3 is a digit\n");
  else
    printf("3 is NOT a digit\n");

  ch = '#';
  if (isdigit(ch))
    printf("# is a digit\n");
  else
    printf("# is NOT a digit\n");
}
```

输出：

```
3 is a digit
# is NOT a digit
```

isgraph

描述： 测试字符是否为图形字符。

头文件： <ctype.h>

函数原型： int isgraph (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个图形字符，那么返回非零的整数值；否则返回零。

说明： 如果字符是除空格外的任意可打印字符，那么该字符被认为是图形字符。

示例：

```
#include <ctype.h> /* for isgraph */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '3';
  if (isgraph(ch))
    printf("3 is a graphical character\n");
  else
    printf("3 is NOT a graphical character\n");
```

isgraph （续）

```
  ch = '#';
  if (isgraph(ch))
    printf("# is a graphical character\n");
  else
    printf("# is NOT a graphical character\n");

  ch = ' ';
  if (isgraph(ch))
    printf("a space is a graphical character\n");
  else
    printf("a space is NOT a graphical character\n");
}
```

输出：

```
3 is a graphical character
# is a graphical character
a space is NOT a graphical character
```

islower

描述： 测试字符是否为小写字母字符。

头文件： <ctype.h>

函数原型： `int islower (int c);`

参数： *c* 要测试的字符

返回值： 如果字符是一个小写字母字符，那么返回非零的整数值；否则返回零。

说明： 如果字符是在 a - z 范围内，那么该字符被认为是小写字母字符。

示例：

```
#include <ctype.h> /* for islower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (islower(ch))
    printf("B is lower case\n");
  else
    printf("B is NOT lower case\n");

  ch = 'b';
  if (islower(ch))
    printf("b is lower case\n");
  else
    printf("b is NOT lower case\n");
}
```

输出：

```
B is NOT lower case
b is lower case
```

isprint

描述： 测试字符是否为可打印字符 （包括空格）。

头文件： <ctype.h>

函数原型： int isprint (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个可打印字符，那么返回非零的整数值；否则返回零。

说明： 如果字符是在 0x20 ～ 0x7e 之间，那么该字符被认为是可打印字符。

示例：

```
#include <ctype.h> /* for isprint */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isprint(ch))
    printf("& is a printable character\n");
  else
    printf("& is NOT a printable character\n");

  ch = '\t';
  if (isprint(ch))
    printf("a tab is a printable character\n");
  else
    printf("a tab is NOT a printable character\n");
}
```

输出：

```
& is a printable character
a tab is NOT a printable character
```

ispunct

描述： 测试字符是否为标点符号。

头文件： <ctype.h>

函数原型： int ispunct (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个标点符号，那么返回非零的整数值；否则返回零。

说明： 如果字符是既非空格也非字母数字字符的可打印字符，那么该字符被认为是标点符号字符，标点符号字符由下列符号组成：

! " # $ % & ' ( ) ; < = > ? @ [\ ] * + , - . / : ^ _ { | } ~

ispunct （续）

示例：

```
#include <ctype.h> /* for ispunct */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = '&';
  if (ispunct(ch))
    printf("& is a punctuation character\n");
  else
    printf("& is NOT a punctuation character\n");

  ch = '\t';
  if (ispunct(ch))
    printf("a tab is a punctuation character\n");
  else
    printf("a tab is NOT a punctuation character\n");
}
```

输出：

```
& is a punctuation character
a tab is NOT a punctuation character
```

isspace

描述： 测试字符是否为空白字符。

头文件： <ctype.h>

函数原型： int isspace (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个空白字符，那么返回非零的整数值；否则返回零。

说明： 如果字符是下列字符之一，那么该字符被认为是空白字符：空格（“ ”）、换页符 （“\f”）、换行符 （“\n”）、回车符 （“\r”）、水平制表符 （“\t”）或垂直制表符 （“\v”）。

示例：

```
#include <ctype.h> /* for isspace */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = '&';
  if (isspace(ch))
    printf("& is a white-space character\n");
  else
    printf("& is NOT a white-space character\n");

  ch = '\t';
  if (isspace(ch))
    printf("a tab is a white-space character\n");
  else
    printf("a tab is NOT a white-space character\n");
}
```

输出：

```
& is NOT a white-space character
a tab is a white-space character
```

isupper

描述： 测试字符是否为大写字母。

头文件： <ctype.h>

函数原型： int isupper (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个大写字母，那么返回非零的整数值；否则返回零。

说明： 如果字符是在 A-Z 范围内，那么该字符被认为是大写字母。

示例：

```
#include <ctype.h> /* for isupper */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isupper(ch))
    printf("B is upper case\n");
  else
    printf("B is NOT upper case\n");

  ch = 'b';
  if (isupper(ch))
    printf("b is upper case\n");
  else
    printf("b is NOT upper case\n");
}
```

输出：

```
B is upper case
b is NOT upper case
```

isxdigit

描述： 测试字符是否为十六进制数。

头文件： <ctype.h>

函数原型： int isxdigit (int *c*);

参数： *c* 要测试的字符

返回值： 如果字符是一个十六进制数，那么返回非零的整数值；否则返回零。

说明： 如果字符是在 0-9、A-F 或 a-f 范围内，那么该字符被认为是十六进制数。注：该列表不包含 0x 字符，因为 0x 字符是十六进制数的前缀，而不是一个真正的十六进制数。

isxdigit （续）

示例：

```
#include <ctype.h> /* for isxdigit */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  if (isxdigit(ch))
    printf("B is a hexadecimal digit\n");
  else
    printf("B is NOT a hexadecimal digit\n");

  ch = 't';
  if (isxdigit(ch))
    printf("t is a hexadecimal digit\n");
  else
    printf("t is NOT a hexadecimal digit\n");
}
```

输出：

```
B is a hexadecimal digit
t is NOT a hexadecimal digit
```

tolower

描述： 将一个字符转换为小写字母字符。

头文件： <ctype.h>

函数原型： `int tolower (int c);`

参数： *c* 要转换为小写字母的字符。

返回值： 如果参数原本是大写字母，那么返回相应的小写字母字符；否则返回原始字符。

说明： 只有大写字母字符才能转换为小写字母。

示例：

```
#include <ctype.h> /* for tolower */
#include <stdio.h> /* for printf */

int main(void)
{
  int ch;

  ch = 'B';
  printf("B changes to lower case %c\n",
         tolower(ch));

  ch = 'b';
  printf("b remains lower case %c\n",
         tolower(ch));

  ch = '@';
  printf("@ has no lower case, ");
  printf("so %c is returned\n", tolower(ch));
}
```

输出：

```
B changes to lower case b
b remains lower case b
@ has no lower case, so @ is returned
```

toupper

描述： 将一个字符转换为大写字母字符。

头文件： <ctype.h>

函数原型： int toupper (int *c*);

参数： *c* 要转换为大写字母的字符。

返回值： 如果参数原本是小写字母字符，那么返回相应的大写字母；否则返回原始字符。

说明： 只有小写字母字符才能转换为大写字母。

示例：

```
#include <ctype.h> /* for toupper */
#include <stdio.h> /* for printf */

int main(void)
{

  int ch;

  ch = 'b';
  printf("b changes to upper case %c\n",
          toupper(ch));

  ch = 'B';
  printf("B remains upper case %c\n",
          toupper(ch));

  ch = '@';
  printf("@ has no upper case, ");
  printf("so %c is returned\n", toupper(ch));
}
```

输出：

```
b changes to upper case B
B remains upper case B
@ has no upper case, so @ is returned
```

4.5 <ERRNO.H> 错误

头文件 errno.h 由提供某些库函数 （参见各个函数）报告的错误码的宏组成。变量 errno 可以返回大于零的任何值。为了测试库函数是否遇到错误，程序应在马上调用库函数之前先将值零保存到变量 error 中。应在另一个函数调用改变这个值之前检查这个值。程序启动时， errno 为零。库函数不会将 errno 设置为零。

EDOM

描述： 表示定义域错误。

头文件： <errno.h>

说明： EDOM 表示一个定义域错误，当输入参数超出函数的定义域时会产生这种错误。

ERANGE

描述： 表示溢出或下溢错误。

头文件： <errno.h>

说明： ERANGE 表示溢出或下溢错误，当要结果太大或太小不能存储时会产生这种错误。

errno

描述： 当函数中发生错误时，包含这个错误的值。

头文件： <errno.h>

说明： 产生错误时，库函数将变量 errno 设置为一个非零的整数值。程序启动时，变量 errno 设置为零。变量 Errno 应该在调用改变其值的函数之前复位为零。

4.6 <FLOAT.H> 浮点特征

头文件 float.h 由指定浮点型的各种属性的宏组成。这些属性包括有效数字位数，大小限制以及使用何种舍入模式。

DBL_DIG

描述： 双精度浮点型值的精度 （十进制数的位数）。

头文件： <float.h>

值： 默认为 6 位，如果使用了 -fno-short-double 选项，为 15 位。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_EPSILON

描述： 1.0 与比它大的最小双精度浮点型值之间的差值。

头文件： <float.h>

值： 默认情况下为 1.192093e-07，如果使用 -fno-short-double 选项，则为 2.220446e-16。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MANT_DIG

描述： 以 FLT_RADIX 为底的双精度浮点型值尾数的位数。

头文件： <float.h>

值： 默认情况下为 24，如果使用 -fno-short-double 选项，则为 53。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MAX

描述： 有限双精度浮点型值的最大值。

头文件： <float.h>

值： 默认情况下为 3.402823e+38，如果使用 -fno-short-double 选项，则为 1.797693e+308。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MAX_10_EXP

描述： 以 10 为底的双精度浮点型数指数的最大整数值。

头文件： <float.h>

值： 默认情况下为 38，如果使用 -fno-short-double 选项，则为 308。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MAX_EXP

描述： 以 FLT_RADIX 为底的双精度浮点型数指数的最大整数值。

头文件： <float.h>

值： 默认情况下为 128，如果使用 -fno-short-double 选项，则为 1024。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MIN

描述： 双精度浮点型数的最小值。

头文件： <float.h>

值： 默认情况下为 1.175494e-38，如果使用 -fno-short-double 选项，则为 2.225074e-308。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

DBL_MIN_10_EXP

描述： 以 10 为底的双精度浮点型数指数的最小负整数值。

头文件： <float.h>

值： 默认情况下为 -37，如果使用 -fno-short-double 选项，则为 -307。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。。

DBL_MIN_EXP

描述： 以 FLT_RADIX 为底的双精度浮点型数指数的最小负整数值。

头文件： <float.h>

值： 默认情况下为 -125，如果使用 -fno-short-double 选项，则为 -1021。

说明： 默认情况下，双精度和浮点型值的位数相同 （32 位表示）。如果使用了 -fno-short-double 选项，则双精度浮点型值可以使用 IEEE 64 位表示。

FLT_DIG

描述： 单精度浮点型值的精度 （十进制数的位数）。

头文件： <float.h>

值： 6

FLT_EPSILON

描述： 1.0 与比它大的最小单精度浮点型数之间的差值。

头文件： <float.h>

值： 1.192093e-07

FLT_MANT_DIG

描述： 以 -FLT_RADIX 为底的单精度浮点型数尾数的位数。

头文件： <float.h>

值： 24

FLT_MAX

描述： 有限单精度浮点型数的最大值。

头文件： <float.h>

值： 3.402823e+38

FLT_MAX_10_EXP

描述： 以 10 为底的单精度浮点型数指数的最大整数值。

头文件： <float.h>

值： 38

FLT_MAX_EXP

描述： 以 FLT_RADIX 为底的单精度浮点型数指数的最大整数值。
头文件： <float.h>
值： 128

FLT_MIN

描述： 单精度浮点型数的最小值。
头文件： <float.h>
值： 1.175494e-38

FLT_MIN_10_EXP

描述： 以 10 为底的单精度浮点型数指数的最小负整数值。
头文件： <float.h>
值： -37

FLT_MIN_EXP

描述： 以 FLT_RADIX 为底的单精度浮点型数指数的最小负整数值。
头文件： <float.h>
值： -125

FLT_RADIX

描述： 指数表示的基数。
头文件： <float.h>
值： 2
说明： 指数的基数表示为以 2 为基或二进制。

FLT_ROUNDS

描述： 浮点型数运算的舍入模式。
头文件： <float.h>
值： 1
说明： 舍入到最接近的可表示值。

LDBL_DIG

描述： 长双精度浮点型值的精度 （十进制数的位数）。
头文件： <float.h>
值： 15

LDBL_EPSILON

描述： 1.0 与比它大的最小可表示长双精度浮点型数之间的差值。
头文件： <float.h>
值： 2.220446e-16

LDBL_MANT_DIG

描述： 以 FLT_RADIX 为底的长双精度浮点型数尾数的位数。
头文件： <float.h>
值： 53

LDBL_MAX

描述： 有限长双精度浮点型数的最大值。
头文件： <float.h>
值： 1.797693e+308

LDBL_MAX_10_EXP

描述： 以 10 为底的长双精度浮点型数指数的最大整数值。
头文件： <float.h>
值： 308

LDBL_MAX_EXP

描述： 以 FLT_RADIX 为底的长双精度浮点型数指数的最大整数值。
头文件： <float.h>
值： 1024

LDBL_MIN

描述： 长双精度浮点型数的最小值。
头文件： <float.h>
值： 2.225074e-308

LDBL_MIN_10_EXP

描述： 以 10 为底的长双精度浮点型数指数的最小负整数值。
头文件： <float.h>
值： -307

LDBL_MIN_EXP

描述： 以 FLT_RADIX 为底的长双精度浮点型数的指数的最小负整数值
头文件： <float.h>
值： -1021

4.7 <LIMITS.H> 实现定义的限制 （ IMPLEMENTATION-DEFINED LIMITS)

头文件 limits.h 里包含了很多宏，这些宏定义了整型数据 （Integer）所能表示的最小值和最大值。每一个宏都可以用在 #if 预处理伪指令中。

CHAR_BIT

描述： 表示 char 型的位数。
头文件： <limits.h>
值： 8

CHAR_MAX

描述： char 型的最大值。
头文件： <limits.h>
值： 127

CHAR_MIN

描述： char 型的最小值。
头文件： <limits.h>
值： -128

INT_MAX

描述： int 型的最大值。
头文件： <limits.h>
值： 32767

INT_MIN

描述： int 型的最小值。
头文件： <limits.h>
值： -32768

LLONG_MAX

描述： long long int 型的最大值。
头文件： <limits.h>
值： 9223372036854775807

LLONG_MIN

描述： long long int 型的最小值。
头文件： <limits.h>
值： -9223372036854775808

LONG_MAX

描述： long int 型的最大值。
头文件： <limits.h>
值： 2147483647

LONG_MIN

描述： long int 型的最小值。
头文件： <limits.h>
值： -2147483648

MB_LEN_MAX

描述： 多字节字符的最大字节数。
头文件： <limits.h>
值： 1

SCHAR_MAX

描述： signed char 型的最大值。
头文件： <limits.h>
值： 127

SCHAR_MIN

描述： signed char 型的最小值。
头文件： <limits.h>
值： -128

SHRT_MAX

描述： short int 型的最大值。
头文件： <limits.h>
值： 32767

SHRT_MIN

描述： short int 型的最小值。
头文件： <limits.h>
值： -32768

UCHAR_MAX

描述： unsigned char 型的最大值。
头文件： <limits.h>
值： 255

UINT_MAX

描述： unsigned int 型的最大值。
头文件： <limits.h>
值： 65535

ULLONG_MAX

描述： long long unsigned int 型的最大值。
头文件： <limits.h>
值： 18446744073709551615

ULONG_MAX

描述： long unsigned int 型的最大值。
头文件： <limits.h>
值： 4294967295

USHRT_MAX

描述： unsigned short int 型的最大值。
头文件： <limits.h>
值： 65535

4.8 <LOCALE.H> 语言环境

编译器默认为仅支持 C 语言环境，不支持其他语言；因此不支持头文件 locale.h。一般可在这个文件中找到下述关键字：

- struct lconv
- NULL
- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME
- localeconv
- setlocale

4.9 <SETJMP.H> 与语言环境无关的跳转

头文件 setjmp.h 由一个类型 （ type）、一个宏 （macro）和一个函数 (function）组成，跳过使用这些类型、宏和函数可以允许发生控制转换 （control transfer），这样可以跳过通常必需的函数调用和返回处理。

jmp_buf

描述： setjmp 和 longjmp 使用的一个数组类型，用来保存和恢复程序环境。

头文件： <setjmp.h>

函数原型： typedef int jmp_buf[_NSETJMP];

说明： _NSETJMP 定义为 16+2，表示 16 个寄存器和一个 32 位返回地址。

setjmp

描述： 一个宏，用来保存程序的当前状态供 longjmp 以后使用。

头文件： <setjmp.h>

函数原型： #define setjmp(jmp_buf *env*)

参数： *env* 存储环境的变量

返回值： 如果返回是直接调用的返回，那么 setjmp 返回零。如果返回是调用 longjmp 的返回，那么 setjmp 返回一个非零值。
注： 如果 longjmp 提供的参数 *val* 为 0，那么 setjmp 返回 1。

示例： 参见 longjmp。

longjmp

描述： 该函数的功能是：恢复由 setjmp 所保存的环境参数。

头文件： <setjmp.h>

函数原型： void longjmp(jmp_buf *env*, int *val*);

参数： *env* 该变量用来存储环境参数

val 要返回给 setjmp 的值。

说明： 参数 *val* 应为非零值。如果从嵌套的信号处理函数中调用 longjmp（即作为处理一个信号过程中产生的另外一个信号的结果来调用），则操作未定义。

4.10 <SIGNAL.H> 信号处理

头文件 signal.h 由一个类型 （ type）、几个宏 （macro）和两个函数 （function）组成，这些类型、宏和函数可以指示出程序在执行过程中是如何处理信号的。信号是程序执行过程中报告的状态。信号是同步发生的，通过 raise 函数由软件控制。

信号可以通过以下方式处理：

- 默认处理方式 （SIG_DFL）：信号看作是致命的错误，程序执行停止。
- 忽略信号 （SIG_IGN）：忽略信号，控制返回到用户应用程序。
- 通过 signal 指定的函数处理信号。

默认情况下，所有信号都由默认的处理函数来处理，这通过 SIG_DFL 来识别。

类型 sig_atomic_t 是程序以不可分割的方式访问的整型类型。当这个类型和关键字 volatile 一起使用时，信号处理函数可与程序的其他部分共享数据对象。

sig_atomic_t

描述： 信号处理函数使用的类型。

头文件： <signal.h>

函数原型： typedef int sig_atomic_t;

SIG_DFL

描述： 用作 signal 的第二个参数和 / 或返回值，用来指定应对特定的信号使用默认处理函数。

头文件： <signal.h>

SIG_ERR

描述： 当由于发生错误 signal 不能完成请求时，用作 signal 的返回值。

头文件： <signal.h>

SIG_IGN

描述： 用作 signal 的第二个参数和 / 或返回值，用来指定应忽略的信号。

头文件： <signal.h>

SIGABRT

描述： 异常中止信号的名字。

头文件： <signal.h>

函数原型： #define SIGABRT

说明： SIGABRT 表示异常中止信号，与 raise 或 signal 一起使用。默认的 raise 操作（通过 SIG_DFL 识别）是输出到标准错误流：

```
abort - terminating
```

参见 signal 函数所附的例子来了解信号名称和信号处理的一般用法。

示例：

```
#include <signal.h> /* for raise, SIGABRT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGABRT);
  printf("Program never reaches here.");
}
```

输出：

```
ABRT
```

说明：
ABRT 代表 abort（中止）。

SIGFPE

描述： 表明浮点运算错误，如除以零或者结果超出值域。

头文件： <signal.h>

函数原型： #define SIGFPE

说明： SIGFPE 用作 raise 和/或 signal 的参数。在使用时，默认操作是打印算术错误消息并终止调用程序。可通过定义信号处理函数操作的用户函数改写。参见 signal 中用户定义函数的例子。

示例：

```
#include <signal.h> /* for raise, SIGFPE */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGFPE);
  printf("Program never reaches here");
}
```

输出：

```
FPE
```

说明：
FPE 代表 floating-point error（浮点运算错误）。

SIGILL

描述： 表明非法指令。

头文件： <signal.h>

函数原型： #define SIGILL

说明： SIGILL 用作 raise 和 / 或 signal 的参数。在使用时，默认操作是打印无效可执行代码消息并终止调用程序。可通过定义信号处理函数操作的用户函数改写。参见 signal 中用户定义函数的例子。

示例：

```
#include <signal.h> /* for raise, SIGILL */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGILL);
  printf("Program never reaches here");
}
```

输出：

```
ILL
```

说明：
ILL 代表 illegal instruction （非法指令）。

SIGINT

描述： 中断信号。

头文件： <signal.h>

函数原型： #define SIGINT

说明： SIGINT 用作 raise 和 / 或 signal 的参数。在使用时，默认操作是打印中断消息并终止调用程序。可通过定义信号处理函数操作的用户函数改写。参见 signal 中用户定义函数的例子。

示例：

```
#include <signal.h> /* for raise, SIGINT */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGINT);
  printf("Program never reaches here.");
}
```

输出：

```
INT
```

说明：
INT 代表 interruption （中断）。

SIGSEGV

描述： 表明非法访问存储。

头文件： <signal.h>

函数原型： #define SIGSEGV

说明： SIGSEGV用作raise和/或signal的参数。在使用时，默认操作是打印一个非法存储请求（invalid storage request）消息并终止调用程序。这可以被一个用户函数接管，该用户函数定义了信号处理动作（signal handler action）。参见 signal 中用户定义函数的例子。

示例：

```
#include <signal.h> /* for raise, SIGSEGV */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGSEGV);
  printf("Program never reaches here.");
}
```

输出：

```
SEGV
```

说明：
SEGV 代表 “非法存储访问”（invalid storage access）。

SIGTERM

描述： 表明终止请求。

头文件： <signal.h>

函数原型： #define SIGTERM

说明： SIGTERM 用作raise和/或signal的参数。可由定义信号处理动作的用户函数接管。参见 signal 中用户定义函数的例子。

示例：

```
#include <signal.h> /* for raise, SIGTERM */
#include <stdio.h> /* for printf */

int main(void)
{
  raise(SIGTERM);
  printf("Program never reaches here.");
}
```

输出：

```
TERM
```

说明：
TERM 代表 “终止请求”（termination request）。

raise

描述： 报告一个同步信号 （synchronous signal）。

头文件： <signal.h>

函数原型： int raise(int *sig*);

参数： *sig* 信号名称

返回值： 如果成功返回零，否则返回非零值。

说明： raise 将由 sig 标识的信号发送到正在执行的程序。

示例：

```
#include <signal.h>   /* for raise, signal, */
                      /* SIGILL, SIG_DFL    */
#include <stdlib.h>   /* for div, div_t     */
#include <stdio.h>    /* for printf         */
#include <p30f6014.h> /* for INTCON1bits */

void __attribute__((__interrupt__))
_MathError(void)
{
  raise(SIGILL);
  INTCON1bits.MATHERR = 0;
}

void illegalinsn(int idsig)
{
  printf("Illegal instruction executed\n");
  exit(1);
}
int main(void)
{
  int x, y;
  div_t z;

  signal(SIGILL, illegalinsn);
  x = 7;
  y = 0;
  z = div(x, y);
  printf("Program never reaches here");
}
```

输出：

```
Illegal instruction executed
```

说明：

这个例子需要链接描述文件 p30f6014.gld。这个例子包括三个部分：
第一部分是为中断向量 _MathError 编写的中断处理函数，通过发送非法指令 signal (SIGILL) 到正在执行的程序来处理数学错误。中断处理函数的最后一条语句清零了异常标志。
第二部分是函数 illegalinsn，将打印错误消息并调用 exit。
第三部分，在 main 中， signal (SIGILL, illegalinsn) 将 SIGILL 的处理函数设置为函数 illegalinsn。
由于除数为零，会发生数学错误，此时将调用 _MathError 中断向量，因此会产生一个将调用 SIGILL 的处理函数 （即 illegalinsn）的信号。从而打印出错误消息并终止程序。

signal

描述： 控制中断信号处理。

头文件： <signal.h>

函数原型： void (*signal(int *sig*, void(**func*)(int)))(int);

参数： *sig* 信号名称

func 要执行的函数

返回值： 返回 *func* 原来的值。

示例：

```
#include <signal.h> /* for signal, raise, */
                    /* SIGINT, SIGILL, */
                    /* SIG_IGN, and SIGFPE */
#include <stdio.h>  /* for printf */

/* Signal handler function */
void mysigint(int id)
{
  printf("SIGINT received\n");
}

int main(void)
{
  /* Override default with user defined function */
  signal(SIGINT, mysigint);
  raise(SIGINT);

  /* Ignore signal handler */
  signal(SIGILL, SIG_IGN);
  raise(SIGILL);
  printf("SIGILL was ignored\n");

  /* Use default signal handler */
  raise(SIGFPE);
  printf("Program never reaches here.");
}
```

输出：

```
SIGINT received
SIGILL was ignored
FPE
```

说明：

函数 mysigint 是 SIGINT 的用户定义信号处理函数，在主程序中，调用函数函数 signal 为信号 SIGINT 建立信号处理函数（mysigint），这将改写默认操作。调用函数 raise 来报告信号 SIGINT。这将使 SIGINT 的信号处理函数使用用户定义函数（mysigint）作为信号处理函数，从而打印 “SIGINT received” 消息。

接着，调用函数 signal 为信号 SIGILL 建立信号处理函数 SIG_IGN。常量 SIG_IGN 用来表示信号应该被忽略。调用函数 raise 来报告被忽略的信号 SIGILL。

最后，调用函数 raise 来报告信号 SIGFPE。由于 SIGFPE 没有用户定义的函数，因此使用默认的信号处理函数，从而打印 “FPE” 消息（“FPE” 代表 “arithmetic error - terminating”）。然后调用程序终止。 Printf 语句不会被执行到。

4.11 <STDARG.H> 可变参数列表

头文件 stdarg.h 支持带有可变参数列表的函数。这允许函数带有没有相应参数声明的参数。参数列表中必须至少包含一个指定的参数。可变参数用省略号 （...）表示。必须在函数内部声明类型为 va_list 的对象来保存参数。 va_start 初始化参数列表的变量， va_arg 用来访问参数列表， va_end 结束参数的使用。

va_list

描述： 类型 va_list 声明一个变量，以引用可变长度参数列表（variable-length argument list）中的每个参数。

头文件： <stdarg.h>

示例： 参见 va_arg。

va_arg

描述： 获取当前参数。

头文件： <stdarg.h>

函数原型： #define va_arg(va_list *ap*, *Ty*)

参数： *ap* 指向参数列表的指针

Ty 要获取的参数的类型

返回值： 返回当前参数

说明： 必须在 va_arg 之前调用 va_start。

示例：

```
#include <stdio.h>   /* for printf */
#include <stdarg.h>  /* for va_arg, va_start,
                        va_list, va_end */

void tprint(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  while (*fmt)
  {
    switch (*fmt)
    {
```

va_arg （续）

```
            case '%':
                  fmt++;
                  if (*fmt == 'd')
                  {
                    int d = va_arg(ap, int);
                    printf("<%d> is an integer\n",d);
                  }
                  else if (*fmt == 's')
                  {
                    char *s = va_arg(ap, char*);
                    printf("<%s> is a string\n", s);
                  }
                  else
                  {
                    printf("%%%c is an unknown format\n",
                         *fmt);
                  }
                  fmt++;
                  break;
            default:
                  printf("%c is unknown\n", *fmt);
                  fmt++;
                  break;
        }
    }
    va_end(ap);
}

int main(void)
{
  tprint("%d%s.%c", 83, "This is text.", 'a');
}
```

输出：

```
<83> is an integer
<This is text.> is a string
. is unknown
%c is an unknown format
```

va_end

描述： 结束使用 *ap*。

头文件： <stdarg.h>

函数原型： #define va_end(va_list *ap*)

参数： *ap* 指向参数列表的指针

说明： 调用 va_end 后，参数列表指针 ***ap*** 被视为是无效的。在遇到下一个 va_start 之前，不应该再调用 va_arg 。在 MPLAB C30 中，va_end 不做任何事情，因此没有必要调用它，但可将其用于提高程序的可读性和可移植性。

示例： 参见 va_arg。

va_start

描述： 设置参数指针 *ap* 指向可变长度参数列表中的第一个可选参数。

头文件： <stdarg.h>

函数原型： #define va_start(va_list *ap*, *last_arg*)

参数： *ap* 指向参数列表的指针

说明： *last_arg* 可选参数前最后一个指定的参数

示例： 参见 va_arg。

4.12 <STDDEF.H> 公共定义

头文件 stddef.h 由程序中通用的几个类型和宏组成。

ptrdiff_t

描述： 两个指针相减的结果的类型。

头文件： <stddef.h>

size_t

描述： sizeof 运算符的结果的类型。

头文件： <stddef.h>

wchar_t

描述： 保存宽字符值的类型。

头文件： <stddef.h>

NULL

描述： 空指针常量的值。

头文件： <stddef.h>

offsetof

描述： 给出结构成员自结构开始的偏移量。

头文件： <stddef.h>

函数原型： #define offsetof(*T*, *mbr*)

参数： *T* 结构名

mbr 结构 T 中成员的名字

返回值： 返回指定成员 （*mbr*）自结构开始的偏移量 （以字节为单位）。

说明： 对于位域，宏 offsetof 未定义。如果使用位域，将发生错误消息。

示例：

```
#include <stddef.h> /* for offsetof */
#include <stdio.h>  /* for printf */

struct info {
  char item1[5];
  int item2;
  char item3;
  float item4;
};

int main(void)
{
  printf("Offset of item1 = %d\n",
          offsetof(struct info,item1));
  printf("Offset of item2 = %d\n",
          offsetof(struct info,item2));
  printf("Offset of item3 = %d\n",
          offsetof(struct info,item3));
  printf("Offset of item4 = %d\n",
          offsetof(struct info,item4));
}
```

输出：

```
Offset of item1 = 0
Offset of item2 = 6
Offset of item3 = 8
Offset of item4 = 10
```

说明：

本程序给出了每个结构成员自结构开始的偏移量 （以字节为单位）。虽然 item1 只有 5 个字节 （char item1[5]），但进行了填充，因此 item2 的地址在偶数边界上。item3 也是这样，它是 1 个字节（char item3）加 1 个字节的填充。

4.13 <STDIO.H> 输入和输出

头文件 stdio.h 由由若干类型 （ type）、宏 （macro）和函数 （function）组成，这些类型、宏和函数为文件 （file）和流 （stream）提供输入和输出操作的支持。当打开一个文件时，它就和一个流相关联。流是文件中数据输入和输出的管道（pipeline）。由于不同的系统使用不同的属性，“流”提供更为统一的属性来读写文件。

流可以是文本流或二进制流。文本流由分成行的字符序列组成，每行以换行符（‘\n’）结尾。可以在字符的内部表示中更改字符，尤其是在行结束的时候。二进制流由信息的字节序列组成，发送到二进制流的字节不能改变，文件仅仅是一系列字节，没有行的概念。

在启动时有三个流会自动打开：stdin、 stdout 和 stderr。 Stdin 提供标准输入流， stdout 是标准输出流，而 stderr 是标准错误流。其他流可由 fopen 函数创建。关于允许的不同文件访问类型，可参见 fopen 的内容。这些访问类型由 fopen 和 freopen 使用。

类型 FILE 用来存储每个打开的文件流的信息，包括错误指示符、文件结束指示符、文件位置指示符和控制一个流所需的其他内部状态信息。 stdio 中的许多函数使用 FILE 作为参数。

缓冲的类型有三种：非缓冲、行缓冲和全缓冲。非缓冲意味着每次传输一个字符或一个字节。行缓冲每次收集并传输一个完整的行 （即换行符表示一行结束）。全缓冲允许传输任意大小的块。函数 setbuf 和 setvbuf 用来控制文件的缓冲。

文件 stdio.h 中还包含使用输入和输出格式的函数。输入格式或扫描格式，用于读取数据的，在 scanf 下可以找到有关它们的描述，但 fscanf 和 sscanf 也使用这些输入格式。输出格式或打印格式用于写数据。在 printf 下可以找到有关它们的描述，fprintf、 sprintf、 vfprintf、 vprintf 和 vsprintf 也使用这些打印格式。

某些编译器选项会影响标准 I/O 的执行。为了提供格式 I/O 函数的定制版本，工具链可将对 printf 或 scanf 型函数的调用转换成不同的调用。这些选项总结如下：

- 当使能 -msmart-io 选项时，将试图将 printf、 scanf 和其他使用输入输出格式的函数转换成仅支持整型的形式。转换后其功能与 C 标准形式相同，但不支持浮点型数输出。选项 -msmart-io=0 将禁止这个功能，不会进行转换。选项 -msmart-io=1 或者 -msmart-io （默认），如果能够保证 I/O 函数不会进行浮点型转换的话，将转换函数调用。 -msmart-io=2 比默认方式更好，假设非常量格式字符串或者未知的格式字符串不会包含浮点型格式。当将 -msmart-io=2 用于浮点型格式时，格式字母将以文本出现，而且它相应的参数不会使用。
- -fno-short-double 将使编译器产生对将double类型视作long double类型来支持的格式 I/O 函数的调用。

使用这些选项来编译混合模块将增加可执行代码的量，或者如果各模块共用 double 和 long double 型数据的话，程序可能会执行不正确。

FILE

描述： 存储文件流的信息。
头文件： <stdio.h>

fpos_t

描述： 用于存储文件位置的变量类型。
头文件： <stdio.h>

size_t

描述： sizeof 运算符的结果类型。
头文件： <stdio.h>

_IOFBF

描述： 表明是全缓冲。
头文件： <stdio.h>
说明： 由函数 setvbuf 使用。

_IOLBF

描述： 表明是行缓冲。
头文件： <stdio.h>
说明： 由函数 setvbuf 使用。

_IONBF

描述： 表明是非缓冲。
头文件： <stdio.h>
说明： 由函数 setvbuf 使用。

BUFSIZ

描述： 定义函数 setbuf 使用的缓冲区大小。
头文件： <stdio.h>
值： 512

EOF

描述： 负数字表明已经达到文件结束，或者报告一个错误条件。

头文件： <stdio.h>

说明： 如果遇到了文件结束，则置位文件结束指示符。如果遇到了一个错误条件，则置位错误指示符。错误条件包括写错误和输入或读错误。

FILENAME_MAX

描述： 在文件名中的最大字符数 （包括空终止符）。

头文件： <stdio.h>

值： 260

FOPEN_MAX

描述： 定义可同时打开的最大文件数。

头文件： <stdio.h>

值： 8

说明： stderr、 stdin 和 stdout 包含在 FOPEN_MAX 计数中。

L_tmpnam

描述： 定义函数 tmpnam 创建的最长临时文件名的字符数。

头文件： <stdio.h>

值： 16

说明： L_tmpnam 用于定义 tmpnam 使用的数组的大小。

NULL

描述： 空指针常量的值。

头文件： <stdio.h>

SEEK_CUR

描述： 表明 fseek 应该从文件指针的当前位置查找。

头文件： <stdio.h>

示例： 参见 fseek 的示例。

SEEK_END

描述： 表明 fseek 应该从文件结束开始查找。
头文件： <stdio.h>
示例： 参见 fseek 的示例。

SEEK_SET

描述： 表明 fseek 应该从文件开头开始查找。
头文件： <stdio.h>
示例： 参见 fseek 的示例。

stderr

描述： 指向标准错误流的文件指针。
头文件： <stdio.h>

stdin

描述： 指向标准输入流的文件指针。
头文件： <stdio.h>

stdout

描述： 指向标准输出流的文件指针。
头文件： <stdio.h>

TMP_MAX

描述： 函数 tmpnam 可产生的唯一文件名的最大数目。
头文件： <stdio.h>
值： 32

clearerr

描述： 将 “流”的错误指示符复位。

头文件： <stdio.h>

函数原型： void clearerr(FILE *stream);

参数： *stream* 要复位错误指示符的流

说明： 函数的功能是：将一个指定的 “流”的文件结束符 （end-of-file）和错误指示符 （error indicator）清零 （在调用 clearerr 函数之后，feof 和 ferror 返回 “假”）。

示例：

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator. The function clearerr is */
/* used to reset the error indicator so the next */
/* time ferror is called it will not report an   */
/* error.                                        */
#include <stdio.h> /* for ferror, clearerr,  */
                   /* printf, fprintf, fopen,*/
                   /* fclose, FILE, NULL     */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

输出：

```
Error
Error indicator reset
```

fclose

描述： 关闭一个流。

头文件： <stdio.h>

函数原型： int fclose(FILE *stream);

参数： *stream* 指向要关闭的流的指针

返回值： 如果成功返回 0；如果检测到任何错误，则返回 EOF。

说明： fclose 向文件写任何缓冲的输出。

描述：

```
#include <stdio.h> /* for fopen, fclose,     */
                   /* printf,FILE, NULL, EOF */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }
}
```

输出：

```
afile1 was opened
afile1 was closed
```

feof

描述： 检测文件的结束。

头文件： <stdio.h>

函数原型： int feof(FILE *stream);

参数： *stream* 要检测文件结束的流

返回值： 如果流在文件结束，返回非零值；否则返回零。

示例：

```
#include <stdio.h> /* for feof, fgetc, fputc, */
                   /* fopen, fclose, FILE,   */
                   /* NULL */

int main(void)
{
  FILE *myfile;
  int y = 0;

  if( (myfile = fopen( "afile.txt", "rb" )) == NULL )
    printf( "Cannot open file\n" );
  else
  {
    for (;;)
    {
      y = fgetc(myfile);
      if (feof(myfile))
        break;
      fputc(y, stdout);
    }
    fclose( myfile );
  }
}
```

输入：

afile.txt 的内容（作为输入）：

```
This is a sentence.
```

输出：

```
This is a sentence.
```

ferror

描述： 检查是否设置了错误指示符。

头文件： <stdio.h>

函数原型： int ferror(FILE *stream);

参数： *stream* FILE 结构的指针

返回值： 如果设置了错误指示符，返回非零值；否则返回零。

示例：

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator and find the error. The   */
/* function clearerr is used to reset the error  */
/* indicator so the next time ferror is called   */
/* it will not report an error.                  */

#include <stdio.h> /* for ferror, clearerr, */
                   /* printf, fprintf,      */
                   /* fopen, fclose,        */
                   /* FILE, NULL            */


int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the "
            "file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

输出：

```
Error
Error indicator reset
```

fflush

描述： 在指定的流中刷新缓冲区。

头文件： <stdio.h>

函数原型： int fflush(FILE *stream);

参数： *stream* 指向要刷新的流的指针。

返回值： 如果发生写错误，返回 EOF；否则，成功的话返回零。

说明： 如果“流”是一个空指针（null pointer），则所有输出缓冲区的内容都被写入到文件中。 fflush 对非缓冲流（unbuffered stream）不起作用。

fgetc

描述： 从流中获取一个字符。

头文件： <stdio.h>

函数原型： int fgetc(FILE *stream);

参数： *stream* 指向打开的流的指针

返回值： 返回读取的字符，或者如果发生读错误或达到文件结束，返回 EOF。

说明： 该函数从输入流读取下一个字符，文件位置指示符前移，并将 unsigned char 转换为 int 返回字符。

示例：

```
#include <stdio.h> /* for fgetc, printf, */
                   /* fclose, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

输入：

afile.txt 的内容（作为输入）：

```
Short
Longer string
```

输出：

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

fgetpos

描述： 获取流的文件位置。

头文件： <stdio.h>

函数原型： int fgetpos(FILE *stream, fpos_t *pos);

参数： *stream* 目标流

pos 存储位置指示符的指针

返回值： 如果成功返回 0；否则返回非零值。

说明： 如果成功，函数将给定流的文件位置指示符存储到 *pos 中；否则 fgetpos 设置为 errno。

示例：

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are    */
/* read then 18 bytes are read. Next the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) ==
       NULL)
    printf("Cannot open file\n");
  else
  {
   fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

输出：

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

fgets

描述： 从流中获取一个字符串。

头文件： <stdio.h>

函数原型： char *fgets(char *s, int n, FILE *stream);

参数：

s	指向存储字符串的指针
n	要读取的最大字符数
stream	指向打开流的指针

返回值： 如果成功，返回指向字符串 *s* 的指针；否则返回空指针。

说明： 该函数从输入流中读取字符，并将它们存储到由 *s* 指向的字符串中，直到读取 n-1 个字符，存储换行符或者设置文件结束符或错误指示符。如果存储完所有字符，则立即在数组的下一个元素中最后一个读取的字符后存储一个空字符。如果 fgets 设置了错误指示符，则数组的内容不确定。

示例：

```
#include <stdio.h> /* for fgets, printf, */
                   /* fopen, fclose,     */
                   /* FILE, NULL         */
#define MAX 50

int main(void)
{
  FILE *buf;
  char s[MAX];

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    while (fgets(s, MAX, buf) != NULL)
    {
      printf("%s|", s);
    }
    fclose(buf);
  }
}
```

输入：

afile.txt 的内容（作为输入）：

```
Short
Longer string
```

输出：

```
Short
|Longer string
|
```

fopen

描述： 打开一个文件。

头文件： <stdio.h>

函数原型： FILE *fopen(const char *filename, const char *mode);

参数

filename 文件名

mode 允许的访问类型

返回值： 返回指向打开流的指针；如果函数失败则返回空指针。

说明： 下列是文件访问的类型：

r - 打开一个现有文本文件来读取。

w - 打开一个空文本文件来写入（现有的文件将被覆写）。

a - 打开一个文本文件来附加（如果该文件不存在，则创建文件）。

rb - 打开一个现有二进制文件来读取。

wb - 打开一个空二进制文件来写入（现有的文件将被覆写）。

ab - 打开一个二进制文件来附加（如果该文件不存在，则创建文件）。

r+ - 打开一个现有文本文件来读写。

w+ - 打开一个空文本文件来读写（现有的文件将被覆写）。

a+ - 打开一个文本文件来读取和附加（如果该文件不存在，则创建文件）。

r+b 或 rb+ - 打开一个现有二进制文件来读写。

w+b 或 wb+ - 打开一个空二进制文件来读写（现有的文件将被覆写）。

a+b 或 ab+ - 打开一个二进制文件来读取和附加（如果该文件不存在，则创建文件）。

示例：

```
#include <stdio.h> /* for fopen, fclose, */
                   /* printf, FILE,      */
                   /* NULL, EOF          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;
```

fopen （续）

```
  if ((myfile1 = fopen("afile1", "r")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Second try, cannot open afile1\n");
  else
  {
    printf("Second try, afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }

  if ((myfile2 = fopen("afile2", "w+")) == NULL)
    printf("Cannot open afile2\n");
  else
  {
    printf("afile2 was opened\n");
    y = fclose(myfile2);
    if (y == EOF)
      printf("afile2 was not closed\n");
   else
      printf("afile2 was closed\n");
  }

}
```

输出：

```
Cannot open afile1
Second try, afile1 was opened
afile1 was closed
afile2 was opened
afile2 was closed
```

说明：

afile1 在被打开来读取（r）之前必须存在，否则 fopen 函数将失败。如果 fopen 函数打开一个文件来写入 （w+），则不要求文件一定要存在。如果这个文件不存在，则将创建文件，并打开它。

fprintf

描述： 打印格式化的数据到一个流。

头文件： <stdio.h>

函数原型： int fprintf(FILE *stream, const char *format, ...);

参数：

stream 指向向其输出数据的流的指针

format 格式控制字符串

... 可选参数

返回值： 返回生成的字符数；或者如果发生错误，则返回一个负数字。

说明： 格式参数具有相同的语法，使用它在 print 中的语法。

示例：

```
#include <stdio.h> /* for fopen, fclose, */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  int y;
  char s[]="Print this string";
  int x = 1;
  char a = '\n';

  if ((myfile = fopen("afile", "w")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    y = fprintf(myfile, "%s %d time%c", s, x, a);

    printf("Number of characters printed "
           "to file = %d",y);

    fclose(myfile);
  }
}
```

输出：

```
Number of characters printed to file = 25
afile 的内容：
Print this string 1 time
```

fputc

描述： 将一个字符写到流中。

头文件： <stdio.h>

函数原型： int fputc(int *c*, FILE **stream*);

参数： *c* 要写的字符

stream 指向打开流的指针

返回值： 返回被写的字符；或者，如果发生写入错误，则返回 EOF。

说明： 函数向输出流写入字符，前移文件位置指示符，并将 unsigned char 转换为 int 返回字符。

示例：

```
#include <stdio.h> /* for fputc, EOF, stdout */

int main(void)
{
  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

  for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = fputc(*y, stdout);
    fputc('|', stdout);
  }
}
```

输出：

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

fputs

描述： 将一个字符串写到流中。

头文件： <stdio.h>

函数原型： int fputs(const char **s*, FILE **stream*);

参数： *s* 要写的字符串

stream 指向打开流的指针

返回值： 如果成功返回非负值；否则返回 EOF。

说明： 该函数向输出流写字符直到空字符 （但不写空字符）。

示例：

```
#include <stdio.h> /* for fputs, stdout */

int main(void)
{
  char buf[] = "This is text\n";

  fputs(buf,stdout);
  fputs("|",stdout);
}
```

输出：

```
This is text
|
```

fread

描述： 从流中读取数据。

头文件： <stdio.h>

函数原型： size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);

参数：

ptr	指向存储缓冲区的指针
size	项的大小
nelem	要读的项的最大数目
stream	指向流的指针

返回值： 返回读取的全部元素的数目（最多为 *nelem*），元素大小由 *size* 确定。

说明： 该函数从给定的流中读取字符到由 *ptr* 指向的缓冲区，直到函数存储了 *size* * *nelem* 个字符，或者置位了文件结束符或错误指示符。 Fread 返回 n/size 的值，其中 n 是读取的字符数。如果 n 不是 size 的整数倍，则最后一个元素的值不确定。如果函数置位了错误指示符，则文件位置指示符不确定。

示例：

```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);

 }
  else
    printf("Cannot open afile.out\n");
```

fread （续）

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f\n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out\n");
}
```

输出：

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

说明：

本程序使用 `fwrite` 来保存 10 个数字到一个二进制格式的文件中。这允许采用与程序使用的相同的位模式来保存这些数字，从而获得更高的精确性和一致性。而如果使用 `fprintf` 会将这些数字保存为文本字符串，可能导致数字被截断。用每个数字除 10 从而得到多个数字。用 `fread` 将这些数字读到一个新数组中，然后将这些数字乘以原来的数字，结果表明这些数字在保存过程中没有被截断。

freopen

描述： 将一个现有的流重新指定到一个新文件。

头文件： <stdio.h>

函数原型： FILE *freopen(const char *filename, const char *mode, FILE *stream);

参数：

filename	新文件名
mode	允许的服务类型
stream	指向当前打开流的指针

返回值： 返回指向新打开文件的指针。如果函数失败，则返回空指针。

说明： 该函数关闭与流相关联的文件，如同调用了 fclose 函数。然后函数打开新的文件，如同调用了 fopen 函数。如果指定的流没打开，函数 freopen 将失败。可能的文件访问类型请参见 fopen。

示例：

```
#include <stdio.h> /* for fopen, freopen, */
                   /* printf, fclose,     */
                   /* FILE, NULL          */

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    if ((myfile2 = freopen("afile2", "w+",
         myfile1)) == NULL)
    {
      printf("Cannot open afile2\n");
      fclose(myfile1);
    }
    else
    {
      printf("afile2 was opened\n");
      fclose(myfile2);
    }
  }
}
```

输出：

```
afile1 was opened
afile2 was opened
```

说明：
当调用 freopen 函数时，程序使用 myfile2 指向流，如果发生错误，myfile1 仍然指向流并能正确关闭。如果调用 freopen 函数成功，myfile2 可正确地关闭。

fscanf

描述： 从流中扫描格式化文本。

头文件： <stdio.h>

fscanf （续）

函数原型： `int fscanf(FILE *stream, const char *format, ...);`

参数：

stream 是一个指针，它指向一个即将打开的流，以便从中读取数据

format 格式控制字符串

... 可选参数

返回值： 返回成功转换和指定的项数。如果没有指定项，则返回 0。如果在第一次转换之前遇到文件结束或发生错误，则返回 EOF。

说明： 格式参数具有相同的语法，使用它在 scanf 中的语法。

示例：

```
#include <stdio.h> /* for fopen, fscanf,   */
                   /* fclose, fprintf,     */
                   /* fseek, printf, FILE, */
                   /* NULL, SEEK_SET       */

int main(void)
{
  FILE *myfile;
  char s[30];
  int x;
  char a;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%s %d times%c",
            "Print this string", 100, '\n');

    fseek(myfile, 0L, SEEK_SET);

    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%d", &x);
    printf("%d\n", x);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%c", a);
    printf("%c\n", a);

    fclose(myfile);
  }
}
```

输入：

afile 的内容：

```
Print this string 100 times
```

输出：

```
Print
this
string
100
times
```

fseek

描述： 将文件指针 （file pointer）移动到指定位置。

头文件： <stdio.h>

函数原型： int fseek(FILE *stream, long offset, int mode);

参数：
stream 在其中移动文件指针的流
offset 要向当前位置增加的值
mode 要执行的查找类型

返回值： 如果成功情况返回 0 ；否则，返回非零的值并设置 errno。

说明： 模式可以是下列之一：
SEEK_SET – 从文件开头开始查找
SEEK_CUR – 从文件指针的当前位置开始查找
SEEK_END – 从文件结束开始查找

示例：

```
#include <stdio.h> /* for fseek, fgets,      */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* SEEK_SET, SEEK_CUR,    */
                   /* SEEK_END               */

int main(void)
{
  FILE *myfile;
  char s[70];
  int  y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile, "This is the beginning, "
                    "this is the middle and "
                    "this is the end.");

    y = fseek(myfile, 0L, SEEK_SET);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 22, myfile);
      printf("\"%s\"\n\n", s);
    }

    y = fseek(myfile, 2L, SEEK_CUR);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n\n", s);
    }
```

fseek （续）

```
    y = fseek(myfile, -16L, SEEK_END);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n", s);
    }
    fclose(myfile);
  }
}
```

输出：

```
"This is the beginning"

"this is the middle and this is the end."

"this is the end."
```

说明：

文件 afile.out 与文本“This is the beginning, this is the middle and this is the end”一起创建。

函数 fseek 使用偏移量零和 SEEK_SET 来设置文件指针到文件开头。然后 fgets 读取 “This is the beginning,”这 22 个字符，并添加一个空字符到字符串中。

接着， fseek 使用偏移量 2 和 SEEK_CURRENT 来设置文件指针到当前位置加 2 的位置上 （跳过逗号和空格）。然后 fgets 读取接下来的 70 个字符。前 39 个字符是 “this is the middle and this is the end.”。当它读到 EOF 时就停止了，并添加一个空字符到字符串中。

最后， fseek 使用负 16 个字符的偏移量和 SEEK_END 从文件末尾来设置文件指针到 16 个字符，然后 Fgets 读取 70 个字符。在读取完 “this is the end.” 16 个字符后读到 EOF 就停止了，并添加一个空字符到字符串。

fsetpos

描述： 设置流的文件位置。

头文件： <stdio.h>

函数原型： int fsetpos(FILE *stream, const fpos_t *pos);

参数：

stream 目标流

pos 存储先前调用 fgetpos 返回的位置指示符的指针

返回值： 如果成功返回 0 ；否则返回非零值。

说明： 如果成功，函数在 *pos 中为给定流设置文件位置指示符；否则 fsetpos 设置 *errno*。

fsetpos （续）

示例：

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are    */
/* read then 18 bytes are read. Next the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h> /* for fgetpos, fread,    */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, perror,    */
                   /* fpos_t, sizeof         */

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) ==
                        NULL)
    printf("Cannot open file\n");
  else
  {
    fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

输出：

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

ftell

描述： 获取文件指针 （file pointer）当前所在的位置。

头文件： <stdio.h>

函数原型： long ftell(FILE *stream);

参数： *stream* 要获取其中当前文件位置的流

返回值： 如果成功返回文件指针的位置；否则返回 -1。

示例：

```
#include <stdio.h> /* for ftell, fread,      */
                   /* fprintf, printf,       */
                   /* fopen, fclose, sizeof, */
                   /* FILE, NULL */

int main(void)
{
  FILE *myfile;
  char s[75];
  long y;

  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile,"This is a very long sentence "
                    "for input into the file named "
                  "afile.out for testing.");

    fclose(myfile);

    if ((myfile = fopen("afile.out", "rb")) != NULL)
    {
      printf("Read some characters:\n");
      fread(s, sizeof(char), 29, myfile);
      printf("\t\"%s\"\n", s);

      y = ftell(myfile);
      printf("The current position of the "
             "file pointer is %ld\n", y);
      fclose(myfile);
    }
  }
}
```

输出：

```
Read some characters:
        "This is a very long sentence "
The current position of the file pointer is 29
```

fwrite

描述： 写数据到流中。

头文件： <stdio.h>

函数原型：
```
size_t fwrite(const void *ptr, size_t size,
 size_t nelem, FILE *stream);
```

参数：

ptr	指向存储缓冲区的指针
size	项的大小
nelem	要读的项的最大数目
stream	指向打开流的指针

返回值： 如果写入成功，返回全部元素的数目，只有当遇到写入错误时，元素数才少于 *nelem* 。

说明： 该函数从由 *ptr* 指定的缓冲区中写最多 *nelem* 个元素到给定的流，元素的大小由 *size* 指定。文件位置指示符前移成功写入的字符数。如果函数设置了错误指示符，则文件位置指示符不确定。

示例：
```
#include <stdio.h> /* for fread, fwrite,     */
                   /* printf, fopen, fclose, */
                   /* sizeof, FILE, NULL     */

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double),
                      10, buf);
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);

  }
  else
    printf("Cannot open afile.out\n");
```

fwrite （续）

```
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double),
                    10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f\n", x+1, readnums[x],
             (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out\n");
}
```

输出：

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

说明：

本程序使用 `fwrite` 来保存 10 个数字到一个二进制格式的文件中。这允许采用与程序使用的相同的位模式来保存这些数字，从而获得更高的精确性和一致性。而如果使用 `fprintf` 会将这些数字保存为文本字符串，可能导致数字被截断。用每个数字除 10 从而得到多个数字。用 `fread` 将这些数字读到一个新数组中，然后将这些数字乘以原来的数字，结果表明这些数字在保存过程中没有被截断。

getc

描述： 从流中获取一个字符。

头文件： <stdio.h>

函数原型： int getc(FILE *stream);

参数： *stream* 指向打开流的指针

返回值： 正常情况下返回读取的字符；或者如果发生读取错误或到达文件结束，返回 EOF。

说明： 函数 getc 和函数 fgetc 相同。

示例：

```
#include <stdio.h> /* for getc, printf, */
                   /* fopen, fclose,    */
                   /* FILE, NULL, EOF   */

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = getc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = getc(buf);
    }
    fclose(buf);
  }
}
```

输入：

afile.txt 的内容（作为输入）：

```
Short
Longer string
```

输出：

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

getchar

描述： 从 stdin 中获取一个字符。

头文件： <stdio.h>

函数原型： int getchar(void);

参数： 正常情况下返回读取的字符；或者如果发生读取错误或到达文件结束，返回 EOF。

返回值： 与以 stdin 为参数的函数 fgetc 具有相同的作用。

示例：

```
#include <stdio.h> /* for getchar, printf */

int main(void)
{
  char y;

    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
}
```

输入：

UartIn.txt 的内容 （用于模拟 stdin 输入）：

```
Short
Longer string
```

输出：

```
S|h|o|r|t|
```

gets

描述： 从 stdin 中获取一个字符串。

头文件： <stdio.h>

函数原型： char *gets(char *s);

参数： *s* 指向存储字符串的指针

返回值： 如果成功，返回指向字符串 s 指针；否则返回空指针。

说明： 该函数从流 stdin 中读取字符，并将它们存储到由 *s* 指向的字符串中，直到读到一个换行符 （此换行符没有被存储）或者置位了文件结束符或错误指示符。如果读取完所有字符，则立即在数组的下一个元素中最后一个读取的字符后存储一个空字符。如果 gets 置位了错误指示符，则数组的内容不确定。

gets （续）

示例：

```
#include <stdio.h> /* for gets, printf */

int main(void)
{
  char y[50];

  gets(y) ;
  printf("Text: %s\n", y);
}
```

输入：

UartIn.txt 的内容 （用于模拟 stdin 输入）：

```
Short
Longer string
```

输出：

```
Text: Short
```

perror

描述： 打印错误信息到 stderr。

头文件： <stdio.h>

函数原型：

```
void perror(const char *s);
```

参数： *s* 要打印的字符串

返回值： 无。

说明： 打印字符串 s，后面接着打印冒号和空格。接着根据 errno 打印一条错误消息，然后换行。

示例：

```
#include <stdio.h> /* for perror, fopen, */
                   /* fclose, printf,    */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    perror("Cannot open samp.fil");
  else
    printf("Success opening samp.fil\n");

  fclose(myfile);
}
```

输出：

```
Cannot open samp.fil:  file open error
```

printf

描述： 打印格式化文本到 stdout。

头文件： <stdio.h>

函数原型： int printf(const char *format, ...);

参数： *format* 格式控制字符串

... 可选参数

返回值： 返回生成的字符数；或者如果发生错误，则返回负的数字。

说明： 参数的数目必须与格式说明符数目完全一致。如果参数比格式说明符少，那么输出不确定；如果参数比说明符多，那么多余的参数将被丢弃。每个格式说明符以百分号开头，后跟可选的字段和一个必需的类型，具体如下：

```
%[flags][width][.precision][size]type
```

flags

- **-** 在给出的字段宽度中左对齐值
- 0 填充字符使用 0 而不是空格 （默认为空格）
- **+** 为正的有符号值生成加号
- space 生成空格或没有加号也没有减号的有符号值
- **#** 在八进制转换前以 0 作为前缀，十六进制转换前以 0x 或 0X 作为前缀，或者生成在浮点型转换中禁止的十进制小数点和小数位。

width

指定要为转换生成的字符数，如果使用星号 （*）而不是十进制数，那么下一个参数 （该参数必须是 int 型）将用于指定字段宽度。如果结果小于字段宽度，将从左边开始用填充字符来填充字段。如果结果大于字段宽度，则不填充而增大字段宽度来适应结果的值。

precision

字段宽度后可跟点 （.）和十进制整数表示精度，来指定下列之一：

- 在整型转换中生成的数据位数
- 在 e、E 或 f 转换中生成的小数位数的数目
- 在 g 或 G 转换中生成的最大有效位数
- 在 s 转换中从 C 字符串生成的最大字符数

如果仅出现了点 （.）而没有出现整数，那么整数就假定为零。如果使用了用星号 （*）而不是十进制数，那么下一个参数 （该参数必需是 int 型）将用来代表精度。

printf （续）

size
- h修饰符 — 用于类型d、i、o、u、x和 X，将值转换为short int 或 unsigned short int
- h 修饰符 — 用于 n，指定指向 short int 的指针
- l修饰符 — 用于类型 d、i、o、u、x 和 X，将值转换为 long int unsigned long int
- l 修饰符 — 用于 n，指定指向 long int 的指针
- l 修饰符 — 用于 c，指定宽字符
- l 修饰符 — 用于类型 e、E、f、F、g 和 G，将值转换为 double
- ll修饰符 — 用于类型 d、i、o、u、x和X，将值转换为long long int 或 unsigned long long int
- ll 修饰符 — 用于 n，指定指向 long long int 的指针
- L 修饰符 — 用于 e、E、f、g 和 G，将值转换为 long double

type
- d, i　signed int
- o　八进制 unsigned int
- u　十进制 unsigned int
- x　小写十六进制 unsigned int
- X　大写十六进制 unsigned int
- e, E　在科学记数法中的 double
- f　十进制表示法的 double
- g, G　double （根据情况选择 e、 E 或 f 形式）
- c　char - 单个字符
- s　string
- p　指针的值
- n　相关的参数应该是整型指针，在这个指针中存放写的字符数，不打印字符。
- %　打印 % 字符

示例：

```
#include <stdio.h> /* for printf */

int main(void)
{
  /* print a character right justified in a 3  */
  /* character space.                          */
  printf("%3c\n", 'a');

  /* print an integer, left justified (as      */
  /* specified by the minus sign in the format */
  /* string) in a 4 character space. Print a   */
  /* second integer that is right justified in */
  /* a 4 character space using the pipe (|) as */
  /* a separator between the integers.         */
  printf("%-4d|%4d\n", -4, 4);

  /* print a number converted to octal in 4    */
  /* digits.                                   */
  printf("%.4o\n", 10);
```

printf （续）

```
  /* print a number converted to hexadecimal   */
  /* format with a 0x prefix.                  */
  printf("%#x\n", 28);

  /* print a float in scientific notation      */
  printf("%E\n", 1.1e20);

  /* print a float with 2 fraction digits      */
  printf("%.2f\n", -3.346);

  /* print a long float with %E, %e, or %f     */
  /* whichever is the shortest version         */
  printf("%Lg\n", .02L);
}
```

输出：

```
  a
-4  |   4
0012
0x1c
1.100000E+20
-3.35
0.02
```

putc

描述： 写一个字符到流中。

头文件： <stdio.h>

函数原型： int putc(int *c*, FILE **stream*);

参数：

c 要写的字符

stream 指向 FILE 结构的指针

返回值： 返回字符；或者如果发生错误或到达文件结束则返回 EOF。

说明： 函数 putc 与 fputc 相同。

示例：

```
#include <stdio.h> /* for putc, EOF, stdout */

int main(void)
{

  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = putc(*y, stdout);
    putc('|', stdout);
  }
}
```

输出：

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

putchar

描述： 写一个字符到 stdout 中。

头文件： <stdio.h>

函数原型： int putchar(int *c*);

参数： *c* 要写的字符

返回值： 返回字符；或者如果发生错误或到达文件结束，则返回 EOF。

说明： 与以 stdout 作为参数的 fputc 函数具有相同的作用。

示例：

```
#include <stdio.h> /* for putchar, printf, */
                   /* EOF, stdout          */

int main(void)
{

  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;


  for (y = buf; (x != EOF) && (*y != '\0'); y++)
    x = putchar(*y);
}
```

输出：

```
This is text
```

puts

描述： 写字符串到 stdout 中。

头文件： <stdio.h>

函数原型： int puts(const char **s*);

参数： *s* 要写的字符串

返回值： 如果成功返回非负值；否则返回 EOF。

说明： 该函数写一个字符到流 stdout 中。添加一个换行符。终止空字符不写到流中。

示例：

```
#include <stdio.h> /* for puts */

int main(void)
{
  char buf[] = "This is text\n";

  puts(buf);
  puts("|");
}
```

输出：

```
This is text

|
```

remove

描述： 删除指定的文件。

头文件： <stdio.h>

函数原型： int remove(const char *filename);

参数： *filename* 要删除的文件的名字

返回值： 如果成功返回 0，否则返回 -1。

说明： 如果文件名不存在或打开了， remove 将失败。

示例：

```
#include <stdio.h> /* for remove, printf */

int main(void)
{
  if (remove("myfile.txt") != 0)
    printf("Cannot remove file");
  else
    printf("File removed");
}
```

输出：

```
File removed
```

rename

描述： 重命名指定的文件。

头文件： <stdio.h>

函数原型： int rename(const char *old, const char *new);

参数：

old 指向旧文件名的指针

new 指向新文件名的指针。

返回值： 如果成功返回 0，否则返回非零值。

说明： 新文件名在当前工作目录下不一定要已经存在，旧文件名在当前工作目录下必须存在。

示例：

```
#include <stdio.h> /* for rename, printf */

int main(void)
{
  if (rename("myfile.txt","newfile.txt") != 0)
    printf("Cannot rename file");
  else
    printf("File renamed");
}
```

输出：

```
File renamed
```

rewind

描述： 将文件指针重新设置到文件的开头位置。

头文件： <stdio.h>

函数原型： void rewind(FILE *stream);

参数： *stream* 要对其复位文件指针的流

说明： 该函数调用 fseek(stream, 0L, SEEK_SET) 函数，然后清零给定流的错误指示符。

示例：

```
#include <stdio.h> /* for rewind, fopen, */
                   /* fscanf, fclose,    */
                   /* fprintf, printf,   */
                   /* FILE, NULL         */

int main(void)
{
  FILE *myfile;
  char s[] = "cookies";
  int x = 10;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%d %s", x, s);
    printf("I have %d %s.\n", x, s);

    /* set pointer to beginning of file */
    rewind(myfile);
    fscanf(myfile, "%d %s", &x, &s);
    printf("I ate %d %s.\n", x, s);

    fclose(myfile);
  }
}
```

输出：

```
I have 10 cookies.
I ate 10 cookies.
```

scanf

描述： 从 stdin 中扫描格式化文本。

头文件： <stdio.h>

函数原型： int scanf(const char *format, ...);

参数： *format* 格式控制字符串

... 可选参数

返回值： 返回值为成功转换和指定的项（item）的数目。如果没有指定任何项，则返回 0。如果在第一次转换前遇到输入错误，则返回 EOF。

说明： 每个格式化说明符都以百分号开头，后跟可选字段和必需的类型，具体如下：

```
%[*][width][modifier]type
```

*****

表示指定被禁止，这将导致输入字段被跳过而不指定。

width

指定要匹配转换的最大输入字符数，不包括可跳过的空白字符。

modifier

h修饰符 — 用于类型d、i、o、u、x和 X，将值转换为short int 或 unsigned short int

h 修饰符 — 用于 n，指定指向 short int 的指针

l 修饰符 — 用于类型 d、i、o、u、x 和 X，将值转换为 long int unsigned long int

l 修饰符 — 用于 n，指定指向 long int 的指针

l 修饰符 — 用于 c，指定宽字符

l 修饰符 — 用于类型 e、E、f、F、g 和 G，将值转换为 double

ll修饰符 — 用于类型 d、i、o、u、x和X，将值转换为long long int 或 unsigned long long int

ll 修饰符 — 用于 n，指定指向 long long int 的指针

L 修饰符 — 用于 e、E、f、g 和 G，将值转换为 long double

scanf （续）

type

d, i	signed int
o	八进制 unsigned int
u	十进制 unsigned int
x	小写十六进制 unsigned int
X	大写十六进制 unsigned int
e, E	在科学记数法中的 double
f	十进制表示法的 double
g, G	double （根据情况选择 e、E 或 f 形式）
c	char - 单个字符
s	string
p	指针的值
n	相关的参数应该是整型指针，在这个指针中存放写的字符数，不扫描字符。
[...]	字符数组。脱字号 （^）紧跟在方括号 （[）后面将颠倒扫描集并允许除方括号中间指定的字符外的任意 ASCII 字符。破折号字符 （-）可用于指定以破折号前的字符开头，以破折号后的字符结尾的范围。空字符不能作为扫描集的一部分。
%	扫描一个 % 字符

示例：

```
#include <stdio.h> /* for scanf, printf */

int main(void)
{
  int number, items;
  char letter;
  char color[30], string[30];
  float salary;

  printf("Enter your favorite number, "
         "favorite letter, ");
  printf("favorite color desired salary "
         "and SSN:\n");
  items = scanf("%d %c %[A-Za-z] %f %s", &number,
  &letter, &color, &salary, &string);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d, ", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s, ", color);
  printf("Desired salary = $%.2f\n", salary);
  printf("Social Security Number = %s, ", string);
}
```

输入：

UartIn.txt 的内容 （用于模拟 stdin 输入）：

```
5 T Green 300000 123-45-6789
```

输出：

```
Enter your favorite number, favorite letter,
favorite color, desired salary and SSN:
Number of items scanned = 5
Favorite number = 5, Favorite letter = T
Favorite color = Green, Desired salary = $300000.00
Social Security Number = 123-45-6789
```

setbuf

描述： 对流的缓冲进行定义。

头文件： <stdio.h>

函数原型： void setbuf(FILE *stream, char *buf);

参数：

stream 指向打开的流的指针

buf 用户分配的缓冲区

说明： 必须在 fopen 之后，在对流进行操作的任何其他函数调用之前调用 setbuf。如果 *buf* 是一个空指针，则 setbuf 以无缓冲方式调用函数 setvbuf(stream, 0, _IONBF, BUFSIZ)；否则 setbuf 以全缓冲方式调用 etvbuf(stream, buf, _IOFBF, BUFSIZ)，缓冲区大小为 BUFSIZ。参见 setvbuf。

示例：

```
#include <stdio.h> /* for setbuf, printf, */
                   /* fopen, fclose,      */
                   /* FILE, NULL, BUFSIZ  */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[BUFSIZ];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    setbuf(myfile1, NULL);
    printf("myfile1 has no buffering\n");
    fclose(myfile1);
  }

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    setbuf(myfile2, buf);
    printf("myfile2 has full buffering");
    fclose(myfile2);
  }

}
```

输出：

```
myfile1 has no buffering
myfile2 has full buffering
```

setvbuf

描述： 定义流的缓冲和缓冲区的大小。

头文件： <stdio.h>

函数原型：
```
int setvbuf(FILE *stream, char *buf, int mode,
size_t size);
```

参数：

stream 指向打开的流的指针

buf 用户分配的缓冲区

mode 缓冲类型

size 缓冲区大小

返回值： 如果成功则返回 0。

说明： 必须在 fopen 之后，在对流进行操作的任何其他函数调用之前调用 setbuf。可使用如下模式之一：
_IOFBF – 表示全缓冲
_IOLBF – 表示行缓冲
_IONBF – 表示非缓冲

示例：
```
#include <stdio.h> /* for setvbuf, fopen, */
                   /* printf, FILE, NULL, */
                   /* _IONBF, _IOFBF      */

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[256];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    if (setvbuf(myfile1, NULL, _IONBF, 0) == 0)
      printf("myfile1 has no buffering\n");
    else
      printf("Unable to define buffer stream "
             "and/or size\n");
  }
  fclose(myfile1);

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    if (setvbuf(myfile2, buf, _IOFBF, sizeof(buf)) ==
        0)
      printf("myfile2 has a buffer of %d "
             "characters\n", sizeof(buf));
    else
      printf("Unable to define buffer stream "
             "and/or size\n");
  }
  fclose(myfile2);
}
```

输出：
```
myfile1 has no buffering
myfile2 has a buffer of 256 characters
```

sprintf

描述： 打印格式化文本到字符串中

头文件： <stdio.h>

函数原型： int sprintf(char *s, const char *format, ...);

参数：

s	存储输出的字符串
format	格式控制字符串
...	可选参数

返回值： 返回存储在 *s* 中的字符数，不包括终止空字符。

说明： 格式参数具有相同的语法，使用它在 printf 中的语法。

示例：

```
#include <stdio.h> /* for sprintf, printf */

int main(void)
{
  char sbuf[100], s[]="Print this string";
  int x = 1, y;
  char a = '\n';

  y = sprintf(sbuf, "%s %d time%c", s, x, a);

  printf("Number of characters printed to "
         "string buffer = %d\n", y);
  printf("String = %s\n", sbuf);
}
```

输出：

```
Number of characters printed to string buffer = 25
String = Print this string 1 time
```

sscanf

描述： 从字符串中扫描格式化文本。

头文件： <stdio.h>

函数原型： int sscanf(const char *s, const char *format, ...);

参数：

s	存储输入的字符串
format	格式控制字符串
...	可选参数

返回值： 返回成功转换和指定的项数。如果没有指定任何项，则返回 0。如果在第一次转换前遇到输入错误，则返回 EOF。

说明： 格式参数具有相同的语法，使用它在 scanf 中的语法。

sscanf （续）

示例：

```
#include <stdio.h> /* for sscanf, printf */

int main(void)
{
  char s[] = "5 T green 3000000.00";
  int number, items;
  char letter;
  char color[10];
  float salary;

  items = sscanf(s, "%d %c %s %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d\n", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s\n", color);
  printf("Desired salary = $%.2f\n", salary);

}
```

输出：

```
Number of items scanned = 4
Favorite number = 5
Favorite letter = T
Favorite color = green
Desired salary = $3000000.00
```

tmpfile

描述： 创建一个临时文件。

头文件： <stdio.h>

函数原型： FILE *tmpfile(void)

返回值： 如果成功返回流指针；否则返回空指针。

说明： tmpfile 创建一个文件名唯一的文件。临时文件以 w+b（二进制读/写）模式打开。当调用 exit 函数时，临时文件将自动被删除；否则文件将保留在目录下。

示例：

```
#include <stdio.h> /* for tmpfile, printf, */
                   /* FILE, NULL           */

int main(void)
{
  FILE *mytempfile;

  if ((mytempfile = tmpfile()) == NULL)
    printf("Cannot create temporary file");
  else
    printf("Temporary file was created");
}
```

输出：

```
Temporary file was created
```

tmpnam

描述： 创建一个唯一的临时文件名。

头文件： <stdio.h>

函数原型： char *tmpnam(char *s);

参数： *s* 指向临时文件名的指针

返回值： 返回指向生成的文件名的指针，并将文件名存储在 *s* 中。如果不能生成文件名，则返回空指针。

说明： 生成的文件名不能与现有的文件名冲突。使用 L_tmpnam 定义参数 tmpnam 指向的数组的大小。

示例：

```
#include <stdio.h> /* for tmpnam, L_tmpnam, */
                   /* printf, NULL          */

int main(void)
{
  char *myfilename;
  char mybuf[L_tmpnam];
  char *myptr = (char *) &mybuf;

  if ((myfilename = tmpnam(myptr)) == NULL)
    printf("Cannot create temporary file name");
  else
    printf("Temporary file %s was created",
           myfilename);
}
```

输出：

```
Temporary file ctm00001.tmp was created
```

ungetc

描述： 将字符写回到流中。

头文件： <stdio.h>

函数原型： int ungetc(int *c*, FILE *stream);

参数： *c* 要写回的字符

stream 指向打开流的指针

返回值： 如果成功返回写入的字符；否则返回 EOF。

说明： 随后读取流将返回写回的字符。如果多个字符被写回，则将以与写入这些字符时相反的顺序返回它们。成功调用文件定位函数（fseek、fsetpos 或 rewind）将取消任何写回的字符。只能保证一个字符的写回。多次调用 ungetc 函数而不插入读取操作或文件定位操作可能导致失败。

ungetc （续）

示例：

```
#include <stdio.h> /* for ungetc, fgetc,     */
                   /* printf, fopen, fclose, */
                   /* FILE, NULL, EOF        */

int main(void)
{
  FILE *buf;
  char y, c;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      if (y == 'r')
      {
        c = ungetc(y, buf);
        if (c != EOF)
        {
          printf("2");
          y = fgetc(buf);
        }
      }
      printf("%c", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

输入：

afile.txt 的内容（作为输入）：

```
Short
Longer string
```

输出：

```
Sho2rt
Longe2r st2ring
```

vfprintf

描述： 利用可变长度的参数列表打印格式化数据到流。

头文件：
```
<stdio.h>
<stdarg.h>
```

函数原型：
```
int vfprintf(FILE *stream, const char *format,
va_list ap);
```

参数：

stream 指向打开流的指针

format 格式控制字符串

ap 指向参数列表的指针

返回值： 返回生成的字符数；或者如果发生，返回一个负的数字。

说明： 格式参数具有相同的语法，使用它在 printf 中的语法。

为了访问可变长度的参数列表， *ap* 变量必须由宏 va_start 初始化并可被 va_arg 函数的其他调用重新初始化。这些必须在调用 vfprintf 函数之前进行。在函数返回之后调用 va_end。更多详细信息，请参见 stdarg.h。

示例：
```
#include <stdio.h>  /* for vfprintf, fopen, */
                    /* fclose, printf,      */
                    /* FILE, NULL           */

#include <stdarg.h> /* for va_start,        */
                    /* va_list, va_end      */

FILE *myfile;

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vfprintf(myfile, fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  if ((myfile = fopen("afile.txt", "w")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    errmsg("Error: The letter '%c' is not %s\n", 'a',
           "an integer value.");
    errmsg("Error: Requires %d%s%c", num,
           " or more characters.", '\n');
  }
  fclose(myfile);
}
```

输出：
```
afile.txt 的内容
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

vprintf

描述： 利用可变长度的参数列表打印格式化文本到 stdout。

头文件： <stdio.h>
<stdarg.h>

函数原型： `int vprintf(const char *format, va_list ap);`

参数： *format* 格式控制字符串
ap 指向参数列表的指针

返回值： 返回生成的字符数；或者如果发生错误，返回一个负的数字。

说明： 格式参数具有相同的语法，使用它在 printf 中的语法。

为了访问可变长度的参数列表， *ap* 变量必须由宏 va_start 初始化并可被 va_arg 函数的其他调用重新初始化。这些必须在调用 vfprintf 函数之前进行。在函数返回之后调用 va_end。更多详细信息，请参见 stdarg.h。

示例：

```
#include <stdio.h>   /* for vprintf, printf */
#include <stdarg.h>  /* for va_start,       */
                     /* va_list, va_end     */

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  printf("Error: ");
  vprintf(fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a',
         "an integer value.");
  errmsg("Requires %d%s\n", num,
         " or more characters.\n");
}
```

输出：

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

vsprintf

描述： 利用可变长度的参数列表打印格式化文本到字符串。

头文件：
```
<stdio.h>
<stdarg.h>
```

函数原型：
```
int vsprintf(char *s, const char *format, va_list
ap);
```

参数：

s	存储输出的字符串
format	格式控制字符串
ap	指向参数列表的指针

返回值： 返回存储在 *s* 中除终止空字符外的字符数。

说明： 格式参数具有相同的语法，使用它在 printf 中的语法

为了访问可变长度的参数列表， *ap* 变量必须由宏 va_start 初始化并可被 va_arg 函数的其他调用重新初始化。这些必须在调用 vfprintf 函数之前进行。在函数返回之后调用 va_end。更多详细信息，请参见 stdarg.h。

示例：
```
#include <stdio.h>   /* for vsprintf, printf */
#include <stdarg.h>  /* for va_start,        */
                     /* va_list, va_end      */

void errmsg(const char *fmt, ...)
{
  va_list ap;
  char buf[100];

  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  printf("Error: %s", buf);
}

int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a',
         "an integer value.");
  errmsg("Requires %d%s\n", num,
         " or more characters.\n");
}
```

输出：
```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

4.14 <STDLIB.H> 实用函数

头文件 stdlib.h 由提供文本转换、存储器管理、查找和排序功能以及其他一般实用功能的类型、宏和函数组成。

div_t

描述： 保存操作数为 int 型的有符号整型除法的商和余数的类型。
头文件： <stdlib.h>
函数原型： typedef struct { int quot, rem; } div_t;
说明： 这是函数 div 返回的结构类型。

ldiv_t

描述： 保存操作数为 long 型的有符号整型除法的商和余数的类型。
头文件： <stdlib.h>
函数原型： typedef struct { long quot, rem; } ldiv_t;
说明： 这是函数 ldiv 返回的结构类型。

size_t

描述： 运算符 sizeof 的结果的类型。
头文件： <stdlib.h>

wchar_t

描述： 保存宽字符值的类型。
头文件： <stdlib.h>

EXIT_FAILURE

描述： 报告不成功的终止。
头文件： <stdlib.h>
说明： EXIT_FAILURE 是 exit 函数的值，用以返回不成功的终止状态。
示例： 使用示例参见 exit。

EXIT_SUCCESS

描述： 报告成功的终止。
头文件： <stdlib.h>
说明： EXIT_SUCCESS 是 exit 函数的值，用以返回成功的终止状态。
示例： 使用示例参见 exit。

MB_CUR_MAX

描述： 多字节字符中的最大字符数。
头文件： <stdlib.h>
值： 1

NULL

描述： 空指针常量的值。
头文件： <stdlib.h>

RAND_MAX

描述： rand 函数能够返回的最大值。
头文件： <stdlib.h>
值： 32767

abort

描述： 中止当前进程。
头文件： <stdlib.h>
函数原型： void abort(void);
说明： abort 将使处理器复位。
示例：

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <stdlib.h> /* for abort          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r")) == NULL)
  {
    printf("Cannot open samp.fil\n");
    abort();
  }
  else
    printf("Success opening samp.fil\n");

  fclose(myfile);
}
```

输出：

```
Cannot open samp.fil
ABRT
```

abs

描述： 计算绝对值 。

头文件： <stdlib.h>

函数原型： int abs(int *i*);

参数： *i* 整型值

返回值： 返回 *i* 的绝对值。

说明： 对于负数，返回它的相反数；正数不变。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for abs    */

int main(void)
{
  int i;

  i = 12;
  printf("The absolute value of  %d is  %d\n",
          i, abs(i));

  i = -2;
  printf("The absolute value of  %d is   %d\n",
          i, abs(i));

  i = 0;
  printf("The absolute value of   %d is   %d\n",
          i, abs(i));
}
```

输出：

```
The absolute value of  12 is  12
The absolute value of  -2 is   2
The absolute value of   0 is   0
```

atexit

描述： 程序正常终止时，注册要调用的指定函数。

头文件： <stdlib.h>

函数原型： int atexit(void(**func*)(void));

参数： *func* 要调用的函数

返回值： 如果成功返回 0；否则返回非零值。

说明： 要调用已注册的函数，程序必须调用 exit 函数来终止。

示例：

```
#include <stdio.h>  /* for scanf, printf */
#include <stdlib.h> /* for atexit, exit  */

void good_msg(void);
void bad_msg(void);
void end_msg(void);
```

atexit （续）

```
int main(void)
{
  int number;

  atexit(end_msg);
  printf("Enter your favorite number:");
  scanf("%d", &number);
  printf(" %d\n", number);
  if (number == 5)
  {
    printf("Good Choice\n");
    atexit(good_msg);
    exit(0);
  }
  else
  {
    printf("%d!?\n", number);
    atexit(bad_msg);
    exit(0);
  }
}

void good_msg(void)
{
  printf("That's an excellent number\n");
}

void bad_msg(void)
{
  printf("That's an awful number\n");
}

void end_msg(void)
{
  printf("Now go count something\n");
}
```

输入：
UartIn.txt 的内容 （用于模拟 stdin 输入）：
5

输出：
```
Enter your favorite number: 5
Good Choice
That's an excellent number
Now go count something
```

输入：
UartIn.txt 的内容 （用于模拟 stdin 输入）：
42

输出：
```
Enter your favorite number: 42
42!?
That's an awful number
Now go count something
```

atof

描述： 将字符串转换为双精度浮点型值。

头文件： <stdlib.h>

函数原型： double atof(const char *s);

参数： *s* 指向要转换的字符串的指针

返回值： 如果成功返回转换的值；否则返回 0。

说明： 数由如下组成：

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

可选的 whitespace 后跟一个可选的 sign，然后是一个或多个 digits 项，再后面是一个或多个带十进制小数点的可选 digits 项，最后是可选的 e 或 E 和可选的有符号指数。当遇到第一个不可识别的字符时，转换就停止。该转换除了没有错误检查外，与 strtod(s,0,0) 是相同的，因此不会设置 errno。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atof   */

int main(void)
{
  char a[] = " 1.28";
  char b[] = "27.835e2";
  char c[] = "Number1";
  double x;

  x = atof(a);
  printf("String = \"%s\" float = %f\n", a, x);

  x = atof(b);
  printf("String = \"%s\" float = %f\n", b, x);

  x = atof(c);
  printf("String = \"%s\"  float = %f\n", c, x);
}
```

输出：

```
String = "1.28"     float = 1.280000
String = "27.835:e2" float = 2783.500000
String = "Number1"  float = 0.000000
```

atoi

描述： 将字符串转换为整型。

头文件： <stdlib.h>

函数原型： int atoi(const char *s);

参数： *s* 要转换的字符串

返回值： 如果成功返回转换的整数；否则返回 0。

说明： 数由如下组成：
[whitespace] [sign] digits
可选项 whitespace 后面是可选项 sign，然后是一个或多个 digits。当遇到第一个不可识别的字符时，转换就停止。该转换除了没有错误检查外，与 (int) strtol(s,0,10) 是相同的，因此不会设置 errno。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atoi   */

int main(void)
{

  char a[] = " -127";
  char b[] = "Number1";
  int x;

  x = atoi(a);
  printf("String = \"%s\"\tint = %d\n", a, x);

  x = atoi(b);
  printf("String = \"%s\"\tint = %d\n", b, x);
}
```

输出：

```
String = " -127"        int = -127
String = "Number1"      int = 0
```

atol

描述： 将字符串转换为长整型。

头文件： <stdlib.h>

函数原型： long atol(const char *s);

参数： *s* 要转换的字符串

返回值： 如果转换成功返回转换的长整型值；否则返回 0。

说明： 该数由如下格式组成：
[whitespace] [sign] digits
可选项 whitespace 后面是可选项 sign，然后是一个或多个 digits。当遇到第一个不可识别的字符时，转换就停止。该转换除了没有错误检查外，与 (int) strtol(s,0,10) 是相同的，因此不会设置 errno。

atol （续）

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for atol   */

int main(void)
{
  char a[] = " -123456";
  char b[] = "2Number";
  long x;

  x = atol(a);
  printf("String = \"%s\"  int = %ld\n", a, x);

  x = atol(b);
  printf("String = \"%s\"  int = %ld\n", b, x);
}
```

输出：

```
String = " -123456"     int = -123456
String = "2Number"      int = 2
```

bsearch

描述： 执行一个二进制搜索。

头文件： <stdlib.h>

函数原型：

```
void *bsearch(const void *key, const void *base,
 size_t nelem, size_t size,
 int (*cmp)(const void *ck, const void *ce));
```

参数：

key	要搜索的对象
base	指向查找数据开始位置的指针
nelem	元素的数目
size	元素大小
cmp	指向比较函数的指针
ck	指向要搜索的键的指针
ce	指向要与键比较的元素

返回值： 如果能够查找到，则返回指向被查找对象的指针；否则返回空指针。

说明： 如果 *ck* 小于 *ce*，则比较函数返回的值小于 0；如果 *ck* 等于 *ce*，则比较函数返回的值为 0；如果 *ck* 大于 *ce*，则比较函数返回的值大于 0。

在下面的示例中，qsort 用于在调用 bsearch 之前对列表排序。bsearch 要求根据比较函数来排序列表。该比较函数使用升序。

bsearch （续）

示例：

```
#include <stdlib.h> /* for bsearch, qsort */
#include <stdio.h>  /* for printf, sizeof */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x, y;
  int *r;

  qsort(list, NUM, sizeof(int), comp);

  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  y = 25;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);

  y = 75;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);
}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

输出：

```
Sorted List:   16  25  35  47  52  63  93
The value 25 was found

The value 75 was not found
```

calloc

描述： 在存储器中分配数组，并将元素初始化为 0。

头文件： <stdlib.h>

函数原型： void *calloc(size_t *nelem*, size_t *size*);

参数： *nelem* 元素的数目

size 每个元素的长度

返回值： 如果成功返回指向分配空间的指针；否则返回空指针。

说明： 对于任意长度的数据元素，由 calloc 返回的存储空间被正确对齐，并初始化为 0。

示例：

```
/* This program allocates memory for the      */
/* array 'i' of long integers and initializes */
/* them to zero.                              */

#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for calloc, free */

int main(void)
{
  int x;
  long *i;

  i = (long *)calloc(5, sizeof(long));
  if (i != NULL)
  {
    for (x = 0; x < 5; x++)
      printf("i[%d] = %ld\n", x, i[x]);
    free(i);
  }
  else
    printf("Cannot allocate memory\n");
}
```

输出：

```
i[0] = 0
i[1] = 0
i[2] = 0
i[3] = 0
i[4] = 0
```

div

描述： 计算两个数的商和余数。

头文件： <stdlib.h>

函数原型： div_t div(int *numer*, int *denom*);

参数： *numer* 分子

denom 分母

返回值： 返回商和余数。

说明： 返回的商与分子除以分母的符号相同，余数的符号要使商乘以分母再加上余数等于分子（quot * denom + rem = numer）。除以 0 将引起数学异常错误，默认情况下会导致复位。可编写一个数学错误处理函数来进行处理。

div （续）

示例：

```
#include <stdlib.h> /* for div, div_t */
#include <stdio.h>  /* for printf     */

void __attribute__((__interrupt__))
_MathError(void)
{
  printf("Illegal instruction executed\n");
  abort();
}

int main(void)
{
  int x, y;
  div_t z;

  x = 7;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the "
         "remainder is %d\n\n", z.quot, z.rem);
}
```

div（续）

输出：

```
For div(7, 3)
The quotient is 2 and the remainder is 1

For div(7, -3)
The quotient is -2 and the remainder is 1

For div(-5, 3)
The quotient is -1 and the remainder is -2

For div(7, 7)
The quotient is 1 and the remainder is 0

For div(7, 0)
Illegal instruction executed
ABRT
```

exit

描述： 清除后终止程序。

头文件： <stdlib.h>

函数原型： void exit(int *status*);

参数： *status* 退出状态

说明： exit 以与注册相反的顺序调用 atexit 注册的任何函数，刷新缓冲区，关闭流，关闭 tmpfile 建立的任何临时文件并复位处理器。该函数可定制。参见 pic30 函数库。

示例：

```
#include <stdio.h>  /* for fopen, printf, */
                    /* FILE, NULL         */
#include <stdlib.h> /* for exit           */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r" )) == NULL)
  {
    printf("Cannot open samp.fil\n");
    exit(EXIT_FAILURE);
  }
  else
  {
    printf("Success opening samp.fil\n");
    exit(EXIT_SUCCESS);
  }
  printf("This will not be printed");
}
```

输出：

```
Cannot open samp.fil
```

free

描述： 释放存储空间。

头文件： <stdlib.h>

函数原型： void free(void *ptr);

参数： *ptr* 指向要释放的存储空间

说明： 释放先前用 calloc、malloc 或 realloc 分配的存储空间。如果将 free 用于已释放的存储空间（通过前面调用 free 或 realloc）或者未使用 calloc、 malloc 或 realloc 分配的存储空间，则操作未定义。

示例：

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
       NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

输出：

```
Memory allocated
Memory freed
```

getenv

描述： 获取环境变量的值。

头文件： <stdlib.h>

函数原型： char *getenv(const char *name);

参数： *name* 环境变量名

返回值： 如果成功，返回指向环境变量值的指针；否则返回空指针。

说明： 要按照描述使用该函数 （参见 pic30 函数库），必须定制该函数。默认情况下， getenv 在环境变量列表中找不到环境变量。

getenv （续）

示例：

```
#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for getenv       */

int main(void)
{
  char *incvar;

  incvar = getenv("INCLUDE");
  if (incvar != NULL)
    printf("INCLUDE environment variable = %s\n",
           incvar);
  else
    printf("Cannot find environment variable "
           "INCLUDE ");
}
```

输出：

```
Cannot find environment variable INCLUDE
```

labs

描述： 计算长整型的绝对值 。

头文件： <stdlib.h>

函数原型： long labs(long *i*);

参数： *i* 长整型值

返回值： 返回 *i* 的绝对值。

说明： 对于负数，返回它的相反数；正数不变。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for labs   */

int main(void)
{
  long i;

  i = 123456;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = -246834;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = 0;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));
}
```

输出：

```
The absolute value of  123456 is 123456
The absolute value of -246834 is 246834
The absolute value of       0 is      0
```

ldiv

描述： 计算两个长整型数的商和余数。

头文件： <stdlib.h>

函数原型： ldiv_t ldiv(long *numer*, long *denom*);

参数： *numer* 分子

denom 分母

返回值： 返回商和余数。

说明： 返回的商与分子除以分母的符号相同，余数的符号要使商乘以分母再加上余数等于分子 （quot * denom + rem = numer）。除以 0 将引起数学异常错误，默认情况下会导致复位。可编写一个数学错误处理函数来进行处理。

示例：

```
#include <stdlib.h> /* for ldiv, ldiv_t */
#include <stdio.h>  /* for printf       */

int main(void)
{
  long x,y;
  ldiv_t z;

  x = 7;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n",
         z.quot, z.rem);
}
```

ldiv （续）

输出：

```
For ldiv(7, 3)
The quotient is 2 and the remainder is 1

For ldiv(7, -3)
The quotient is -2 and the remainder is 1

For ldiv(-5, 3)
The quotient is -1 and the remainder is -2

For ldiv(7, 7)
The quotient is 1 and the remainder is 0

For ldiv(7, 0)
The quotient is -1 and the remainder is 7
```

说明：
在上例的 （ldiv(7, 0)）中分母为 0，因此操作未定义。

malloc

描述： 分配存储空间。

头文件： <stdlib.h>

函数原型： void *malloc(size_t *size*);

参数： *size* 要分配的字符数

返回值： 如果成功，返回指向分配空间的指针；否则返回空指针。

说明： malloc 不初始化它返回的存储空间。

示例：

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) ==
      NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

输出：

```
Memory allocated
Memory freed
```

mblen

描述： 获取多字节字符的长度 （参见说明）。

头文件： <stdlib.h>

函数原型： int mblen(const char *s, size_t n);

参数：
s 指向多字节字符
n 要检查的字节数

返回值： 如果 s 指向空字符则返回 0 ； 否则返回 1。

说明： MPLAB C30 不支持长度大于 1 字节的多字节字符。

mbstowcs

描述： 将多字节字符串转换为宽字符串 （参见说明）。

头文件： <stdlib.h>

函数原型： size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);

参数：
wcs 指向宽字符串
s 指向多字节字符串
n 要转换的宽字符数

返回值： 返回除空字符外存储的宽字符数。

说明： mbstowcs 转换 n 个宽字符，除非遇到空宽字符。MPLAB C30 不支持长度大于 1 字节的多字节字符。

mbtowc

描述： 将多字节字符转换为宽字符 （参见说明）。

头文件： <stdlib.h>

函数原型： int mbtowc(wchar_t *pwc, const char *s, size_t n);

参数：
pwc 指向宽字符
s 指向多字节字符
n 要检查的字节数

返回值： 如果 *s* 指向空字符，则返回 0 ； 否则返回 1。

说明： 生成的宽字符存储在 pwc 中。 MPLAB C30 不支持长度大于 1 字节的多字节字符。

qsort

描述： 执行快速排序。

头文件： <stdlib.h>

函数原型：
```
void qsort(void *base, size_t nelem, size_t size,
 int (*cmp)(const void *e1, const void *e2));
```

参数：

base	指向数组起始地址的指针
nelem	元素数
size	元素的大小
cmp	指向比较函数的指针
e1	指向要搜索的键的指针
e2	指向与键相比较的元素的指针

说明： qsort 用排序好的数组覆盖原来的数组，比较函数由用户提供。在以下示例中，根据比较函数来排序列表。该比较函数采用升序。

示例：
```
#include <stdlib.h> /* for qsort  */
#include <stdio.h>  /* for printf */

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x;

  printf("Unsorted List: ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  qsort(list, NUM, sizeof(int), comp);

  printf("\n");
  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

输出：
```
Unsorted List: 35  47  63  25  93  16  52
Sorted List:   16  25  35  47  52  63  93
```

rand

描述： 生成伪随机整数 。

头文件： <stdlib.h>

函数原型： int rand(void);

返回值： 返回 0 和 RAND_MAX 之间的整数。

说明： 调用该函数返回 [0,RAND_MAX] 范围内的伪随机整数。要想有效地使用此函数，必须先使用 srand 函数给随机数发生器生成种子 （seed）。当没有使用种子 （正如下列的示例中）或者使用相同的种子值时，该函数总是返回相同序列的整数 （参见 srand 中有关种子的示例）。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for rand   */

int main(void)
{
  int x;

  for (x = 0; x < 5; x++)
    printf("Number = %d\n", rand());
}
```

输出：

```
Number = 21422
Number = 2061
Number = 16443
Number = 11617
Number = 9125
```

注意如果程序再次运行，则生成的伪随机数与第一次是一样的。参见 srand 示例中有关给随机数发生器生成种子 （seed）的内容。

realloc

描述： 允许改变大小重新分配存储空间。

头文件： <stdlib.h>

函数原型： void *realloc(void *ptr, size_t size);

参数： *ptr* 指向原先分配的存储空间

size 要分配的存储空间大小

返回值： 如果成功，返回指向分配空间的指针；否则返回空指针。

说明： 如果现有的对象比新对象小，则全部整个对象将会被复制到新对象中，而新对象中剩余的空间将不确定。如果现有的对象比新对象大，则函数从现有对象中复制与新对象大小一样的内容。如果 realloc 成功完成新对象的分配，现有对象将会被释放；否则，现有对象将会原封不动地保留下来。由于在失败的情况下 realloc 将返回空指针，因此要有一个指向现有对象的临时指针。

realloc （续）

示例：

```
#include <stdio.h> /* for printf, sizeof, NULL */
#include <stdlib.h> /* for realloc, malloc, free */

int main(void)
{
  long *i, *j;

  if ((i = (long *)malloc(50 * sizeof(long)))
       == NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    /* Temp pointer in case realloc() fails */
    j = i;

    if ((i = (long *)realloc(i, 25 * sizeof(long)))
        == NULL)
    {
      printf("Cannot reallocate memory\n");
       /* j pointed to allocated memory */
      free(j);
    }
    else
    {
      printf("Memory reallocated\n");
      free(i);
    }
  }
}
```

输出：

```
Memory allocated
Memory reallocated
```

srand

描述： 设置伪随机数序列的起始种子。

头文件： <stdlib.h>

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

参数： *seed* 伪随机数序列的起始值

返回值： 无

说明： 该函数为 rand 函数生成的伪随机数序列设置起始种子。当使用相同的种子值时， Rand 函数总是返回相同的整数序列。如果以种子值 1 来调用 rand，则生成的数字序列将与没有先调用 srand 函数而调用 rand 函数生成的数字序列相同。

示例：

```
#include <stdio.h>  /* for printf      */
#include <stdlib.h> /* for rand, srand */

int main(void)
{
  int x;

  srand(7);
  for (x = 0; x < 5; x++)
    printf("Number = %d\n", rand());
}
```

输出：

```
Number = 16327
Number = 5931
Number = 23117
Number = 30985
Number = 29612
```

strtod

描述： 将部分字符串转换为双精度浮点型数。

头文件： <stdlib.h>

函数原型： double strtod(const char *s, char **endptr);

参数： *s* 要转换的字符串

endptr 指向转换停止处字符的指针

返回值： 如果成功，返回转换的数字；否则返回 0。

说明： 该数由如下组成：

```
[whitespace] [sign] digits [.digits]
  [ { e | E }[sign]digits]
```

可选的 whitespace 后跟一个可选的 sign，然后是一个或多个 digits 项，再后面是一个或多个带十进制小数点的可选 digits 项，最后是可选的 e 或 E 和可选的有符号指数。

strtod 转换字符串直到遇到一个不可转换成数的字符。 *endptr* 将指向自第一个不可转换字符开始的字符串的其他字符。

如果出现值域错误，将设置 errno。

strtod （续）

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtod */

int main(void)
{
  char *end;
  char a[] = "1.28 inches";
  char b[] = "27.835e2i";
  char c[] = "Number1";
  double x;

  x = strtod(a, &end);
  printf("String = \"%s\"  float = %f\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtod(b, &end);
  printf("String = \"%s\" float = %f\n", b, x );
  printf("Stopped at: %s\n\n", end );

  x = strtod(c, &end);
  printf("String = \"%s\"   float = %f\n", c, x );
  printf("Stopped at: %s\n\n", end );
}
```

输出：

```
String = "1.28 inches"  float = 1.280000
Stopped at:  inches

String = "27.835e2i" float = 2783.500000
Stopped at: i

String = "Number1"   float = 0.000000
Stopped at: Number1
```

strtol

描述： 将部分字符串转换为长整型数。

头文件： <stdlib.h>

函数原型： long strtol(const char *s, char **endptr, int base);

参数：

s	要转换的字符串
endptr	指向转换停止处字符的指针
base	转换中使用的基数

返回值： 如果成功，返回转换成的数；否则返回 0。

说明： 如果 *base* 为 0， strtol 将自动确定基数。可以是八进制的 （以 0 打头），也可以是十六进制的 （以 0x 或 0X 打头），或者是十进制的。如果指定了基数， strtol 将转换数字和字母 a-z （区分大小写）序列，这里的 a-z 代表数字 10-36。当遇到超过基数所表示范围的数时转换停止。 *endptr* 将指向剩下的自第一个未转换字符起始的字符串。如果发生值域错误，将设置 errno。

示例：

```
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for strtol */

int main(void)
{
  char *end;
  char a[] = "-12BGEE";
  char b[] = "1234Number";
  long x;

  x = strtol(a, &end, 16);
  printf("String = \"%s\"  long = %ld\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtol(b, &end, 4);
  printf("String = \"%s\"  long = %ld\n", b, x );
  printf("Stopped at: %s\n\n", end );
}
```

输出：

```
String = "-12BGEE"  long = -299
Stopped at: GEE

String = "1234Number"  long = 27
Stopped at: 4Number
```

strtoul

描述： 将部分字符串转换为无符号长整型数。

头文件： <stdlib.h>

函数原型：
```
unsigned long strtoul(const char *s, char **endptr,
 int base);
```

参数：

s	要转换的字符串
endptr	指向转换停止处字符的指针
base	转换中使用的基数

返回值： 如果成功，返回转换成的数；否则返回 0。

说明： 如果 *base* 为 0， strtol 将自动确定基数。可以是八进制的 （以 0 打头），也可以是十六进制的 （以 0x 或 0X 打头），或者是十进制的。如果指定了基数， strtol 将转换数字和字母 a-z （区分大小写）序列，这里的 a-z 代表数字 10-36。当遇到超过基数所表示范围的数时转换停止。 *endptr* 将指向剩下的自第一个未转换字符起始的字符串。如果发生值域错误，将设置 errno。

示例：
```
#include <stdio.h>  /* for printf  */
#include <stdlib.h> /* for strtoul */

int main(void)
{
  char *end;
  char a[] = "12BGET3";
  char b[] = "0x1234Number";
  char c[] = "-123abc";
  unsigned long x;

  x = strtoul(a, &end, 25);
  printf("String = \"%s\"  long = %lu\n", a, x );
  printf("Stopped at: %s\n\n", end );

  x = strtoul(b, &end, 0);
  printf("String = \"%s\"  long = %lu\n", b, x );
  printf("Stopped at: %s\n\n", end );

  x = strtoul(c, &end, 0);
  printf("String = \"%s\"  long = %lu\n", c, x );
  printf("Stopped at: %s\n\n", end );
}
```

输出：
```
String = "12BGET3"  long = 429164
Stopped at: T3

String = "0x1234Number"  long = 4660
Stopped at: Number

String = "-123abc"  long = 4294967173
Stopped at: abc
```

system

描述： 执行一个命令。

头文件： <stdlib.h>

函数原型： int system(const char *s);

参数： *s* 要执行的命令

说明： 为按照描述来使用该函数，必须定制该函数 （参见 pic30 函数库）。默认情况下，如果使用除 NULL 外的任何参数调用函数， system 将导致 复位。 system(NULL) 不做任何事情。

示例：

```
/* This program uses system */
/* to TYPE its source file. */

#include <stdlib.h> /* for system */

int main(void)
{
  system("type sampsystem.c");
}
```

输出：

```
System(type sampsystem.c) called: Aborting
```

wctomb

描述： 将宽字符转换为多字节字符 （参见 “说明”）。

头文件： <stdlib.h>

函数原型： int wctomb(char *s, wchar_t wchar);

参数： *s* 指向多字节字符

wchar 要转换的宽字符

返回值： 如果 s 指向空字符，则返回 0 ；否则返回 1。

说明： 生成的多字节字符存储在 *s* 中。 MPLAB C30 不支持长度大于 1 个字符的多字节字符。

wcstombs

描述： 将宽字符串转换为多字节字符串 （参见 “说明”）。

头文件： <stdlib.h>

函数原型： size_t wcstombs(char *s, const wchar_t *wcs, size_t n);

参数： *s* 指向多字节字符串

wcs 指向宽字符串

n 要转换的字符数

返回值： 返回存储的除空字符外的字符数。

说明： wcstombs 转换 *n* 个多字节字符，除非遇到空字符。MPLAB C30 不支持长度大于 1 个字符的多字节字符。

4.15 <STRING.H> 字符串函数

头文件 string.h 由一系列类型、宏和函数组成，提供对字符串进行操作的工具。

size_t

描述： sizeof 运算符结果的类型。
头文件： <string.h>

NULL

描述： 空指针常量的值。
头文件： <string.h>

memchr

描述： 在缓冲区中定位一个字符。
头文件： <string.h>
函数原型： void *memchr(const void *s, int c, size_t n);
参数：
s 指向缓冲区的指针
c 要搜索的字符
n 要检查的字符数
返回值： 如果成功，返回指向匹配字符位置的指针；否则返回零。
说明： memchr 在第一次发现 *c* 或在搜索 *n* 个字符数后停止。

示例：

```
#include <string.h> /* for memchr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'i', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = memchr(buf1, ch1, 50);
if (ptr != NULL)
  {
   res = ptr - buf1 + 1;
   printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
```

memchr （续）

```
  printf("\n");

  ptr = memchr(buf1, ch2, 50);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

输出：

```
buf1 : What time is it?

i found at position 7

y not found
```

memcmp

描述： 比较两个缓冲区的内容。

头文件： <string.h>

函数原型： `int memcmp(const void *s1, const void *s2, size_t n);`

参数：

s1 第一个缓冲区

s2 第二个缓冲区

n 要比较的字符数

返回值： 如果 *s1* 大于 *s2*，返回正数；如果 *s1* 等于 *s2*，返回 0；如果 *s1* 小于 *s2*，返回负数。

说明： 该函数将 *s1* 中前 n 个字符与 *s2* 中的前 n 个字符相比较，返回一个值表明其中一个缓冲区小于、等于还是大于另外一个缓冲区。

示例：

```
#include <string.h> /* memcmp     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = memcmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
```

memcmp （续）

```
  printf("\n");

  res = memcmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");

  printf("\n");

  res = memcmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
  else
    printf("buf3 comes before buf1\n");
}
```

输出：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

memcpy

描述： 从一个缓冲区复制字符到另一个缓冲区中。

头文件： <string.h>

函数原型： void *memcpy(void *dst , const void *src , size_t n);

参数：
dst 将字符复制到的目的缓冲区
src 从中复制字符的源缓冲区
n 复制的字符数

返回值： 返回 *dst*。

说明： memcpy从源缓冲区 *src* 中复制 *n* 个字符到目的缓冲区 *dst*。如果两个缓冲区重叠，则操作未定义。

示例：

```
#include <string.h> /* memcpy     */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memcpy(buf1, buf2, 6);
  printf("buf1 after memcpy of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memcpy(buf1, buf3, 5);
  printf("buf1 after memcpy of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

输出：

```
buf1 :
buf2 : Where is the time?
buf3 : Why?

buf1 after memcpy of 6 chars of buf2:
        Where

buf1 after memcpy of 5 chars of buf3:
        Why?
```

memmove

描述： 从源缓冲区中复制 n 个字符到目的缓冲区，即使缓冲区重叠也进行复制。

头文件： <string.h>

函数原型： void *memmove(void *s1, const void *s2, size_t n);

参数：
dst 将字符复制到的目的缓冲区
src 从中复制字符的源缓冲区
n 从 *s2* 复制到 *s1* 的字符数

返回值： 返回指向目的缓冲区的指针

说明： 如果两个缓冲区重叠，其作用相当于先从 *s2* 中读取字符，然后写入 *s1*，因此缓冲区没有被破坏。

示例：

```
#include <string.h> /* for memmove */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "When time marches on";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memmove(buf1, buf2, 6);
  printf("buf1 after memmove of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memmove(buf1, buf3, 5);
  printf("buf1 after memmove of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

输出：

```
buf1 : When time marches on
buf2 : Where is the time?
buf3 : Why?

buf1 after memmove of 6 chars of buf2:
        Where ime marches on

buf1 after memmove of 5 chars of buf3:
        Why?
```

memset

描述： 复制指定字符到目的缓冲区。

头文件： <string.h>

函数原型： void *memset(void *s, int c, size_t n);

参数：

s 缓冲区

c 要写入缓冲区的字符

n 次数

返回值： 返回写入字符的缓冲区。

说明： 字符 *c* 被写入到缓冲区 *n* 次。

示例：

```
#include <string.h> /* for memset */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[20] = "What time is it?";
  char buf2[20] = "";
  char ch1 = '?', ch2 = 'y';
  char *ptr;
  int res;

  printf("memset(\"%s\", \'%c\',4);\n", buf1, ch1);
  memset(buf1, ch1, 4);
  printf("buf1 after memset: %s\n", buf1);

  printf("\n");
  printf("memset(\"%s\", \'%c\',10);\n", buf2, ch2);
  memset(buf2, ch2, 10);
  printf("buf2 after memset: %s\n", buf2);
}
```

输出：

```
memset("What time is it?", '?',4);
buf1 after memset: ???? time is it?

memset("", 'y',10);
buf2 after memset: yyyyyyyyyy
```

strcat

描述： 将源字符串的拷贝添加到目的字符串的末尾。

头文件： <string.h>

函数原型： char *strcat(char *s1, const char *s2);

参数：
s1 以空字符终止的目的字符串
s2 以空字符终止的源字符串

返回值： 返回指向目的字符串的指针。

说明： 该函数将源字符串（包含终止空字符）添加到目的字符串末尾，源字符串中的首字符覆盖目的字符串末尾的空字符。如果缓冲区重叠，则操作未定义。

示例：

```
#include <string.h> /* for strcat, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";

  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));

  strcat(buf1, buf2);
  printf("buf1 after strcat of buf2: \n\t%s\n",
          buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strcat(buf1, "Why?");
  printf("buf1 after strcat of \"Why?\": \n\t%s\n",
         buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

输出：

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf1 after strcat of buf2:
        We're hereWhere is the time?
        (28 characters)

buf1 after strcat of "Why?":
        We're hereWhere is the time?Why?
        (32 characters)
```

strchr

描述： 定位字符串中指定字符首次出现的位置。

头文件： <string.h>

函数原型： char *strchr(const char *s, int c);

参数： *s* 指向字符串的指针

c 要搜索的字符

返回值： 如果成功，返回指向匹配位置的指针；否则返回空指针。

说明： 该函数用来搜索字符串 *s* 中字符 *c* 首次出现的位置。

示例：

```
#include <string.h> /* for strchr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = strchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);

  printf("\n");

  ptr = strchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

输出：

```
buf1 : What time is it?

m found at position 8

y not found
```

strcmp

描述： 比较两个字符串。

头文件： <string.h>

函数原型： int strcmp(const char *s1, const char *s2);

参数： *s1* 第一个字符串

s2 第二个字符串

返回值： 如果 *s1* 大于 *s2*，返回正数；如果 *s1* 等于 *s2*，返回 0；如果 *s1* 小于 *s2*，返回负数。

说明： 该函数逐个比较 *s1* 和 *s2* 中的字符，直到遇到不相等的字符或者遇到空终止符。

示例：

```
#include <string.h> /* for strcmp */
#include <stdio.h>  /* for printf */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strcmp(buf1, buf2);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("buf1 and buf2 are equal\n");
  else
    printf("buf2 comes before buf1\n");

  printf("\n");

  res = strcmp(buf1, buf3);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("buf1 and buf3 are equal\n");
  else
    printf("buf3 comes before buf1\n");

  printf("\n");

  res = strcmp("Why?", buf3);
  if (res < 0)
    printf("\"Why?\" comes before buf3\n");
  else if (res == 0)
    printf("\"Why?\" and buf3 are equal\n");
  else
    printf("buf3 comes before \"Why?\"\n");
}
```

strcmp （续）

输出：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

buf2 comes before buf1

buf1 comes before buf3

"Why?" and buf3 are equal
```

strcoll

描述： 将一个字符串与另一个字符串相比较 （参见 “说明”）。

头文件： <string.h>

函数原型： `int strcoll(const char *s1, const char *s2);`

参数： *s1* 第一个字符串

s2 第二个字符串

返回值： 根据特定语言环境的规则，如果 *s1* 大于 *s2*，返回正数；如果 *s1* 等于 *s2*，返回 0 ；如果 *s1* 小于 *s2*，返回负数。

说明： 由于 MPLAB C30 不支持其他语言环境，因此该函数与 strcmp 函数作用相同。

strcpy

描述： 复制源字符串到目的字符串。

头文件： <string.h>

函数原型： `char *strcpy(char *s1, const char *s2);`

参数： *s1* 目的字符串

s2 源字符串

返回值： 返回指向目的字符串的指针。

说明： *s2* 中所有字符都被复制，包括空中止字符，如果字符串重叠，则操作未定义。

示例：

```
#include <string.h> /* for strcpy, strlen */
#include <stdio.h>  /* for printf         */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  strcpy(buf1, buf2);
  printf("buf1 after strcpy of buf2: \n\t%s\n\n",
         buf1);
```

strcpy （续）

```
  strcpy(buf1, buf3);
  printf("buf1 after strcpy of buf3: \n\t%s\n",
          buf1);
}
```

输出：

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?

buf1 after strcpy of buf2:
        Where is the time?

buf1 after strcpy of buf3:
        Why?
```

strcspn

描述： 计算自字符串开头起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 *s1, const char *s2);

参数：

s1 指向要被搜索的字符串的指针

s2 指向要查找的字符的指针

返回值： 返回字符串 *s1* 中自字符串开头起不包含 *s2* 中字符的字符片段的长度。

说明： 该函数将确定字符串 *s1* 中自字符串开头起不包含在 *s2* 中的连续字符数。

示例：

```
#include <string.h> /* for strcspn */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char str1[20] = "hello";
  char str2[20] = "aeiou";
  char str3[20] = "animal";
  char str4[20] = "xyz";
  int res;

  res = strcspn(str1, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strcspn(str3, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str2, res);

  res = strcspn(str3, str4);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str4, res);
}
```

输出：

```
strcspn("hello", "aeiou") = 1
strcspn("animal", "aeiou") = 0
strcspn("animal", "xyz") = 6
```

strcspn （续）

说明：
在第一个结果中， e 在 *s2* 中，因此不计入 h 之后的字符。
在第二个结果中， a 在 *s2* 中。
在第三个结果中， *s1* 中的任何字符都不在 *s2* 中，因此计入所有的字符。

strerror

描述： 获取内部错误消息。

头文件： <string.h>

函数原型： char *strerror(int *errcode*);

参数： *errcode* 错误码数字

返回值： 返回指向与指定错误码 *errcode* 相对应的内部错误消息字符串的指针。

说明： 由 strerror 指向的数组可能在后续调用该函数时被改写。

示例：

```
#include <stdio.h>  /* for fopen, fclose, */
                    /* printf, FILE, NULL */
#include <string.h> /* for strerror       */
#include <errno.h>  /* for errno          */

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    printf("Cannot open samp.fil: %s\n",
            strerror(errno));
  else
    printf("Success opening samp.fil\n");
  fclose(myfile);
}
```

输出：

```
Cannot open samp.fil: file open error
```

strlen

描述： 确定字符串的长度。

头文件： <string.h>

函数原型： size_t strlen(const char *s);

参数： *s* 字符串

返回值： 返回字符串的长度。

说明： 该函数确定字符串的长度，不包括终止空字符。

strlen （续）

示例：

```
#include <string.h> /* for strlen */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "We are here";
  char str2[20] = "";
  char str3[20] = "Why me?";

  printf("str1 : %s\n", str1);
  printf("\t(string length = %d characters)\n\n",
         strlen(str1));
  printf("str2 : %s\n", str2);
  printf("\t(string length = %d characters)\n\n",
         strlen(str2));
  printf("str3 : %s\n", str3);
  printf("\t(string length = %d characters)\n\n\n",
         strlen(str3));

}
```

输出：

```
str1 : We are here
        (string length = 11 characters)

str2 :
        (string length = 0 characters)

str3 : Why me?
        (string length = 7 characters)
```

strncat

描述： 将源字符串中指定数目的字符添加到目的字符串。

头文件： <string.h>

函数原型： char *strncat(char *s1, const char *s2, size_t n);

参数：

s1 目的字符串

s2 源字符串

n 要添加的字符数

返回值： 返回指向目的字符串的指针。

说明： 该函数从源字符串中添加 *n* 个字符（空字符及其后面的字符不会被添加）到目的字符串。如果遇到空字符，就将终止空符添加到结果。如果字符串重叠，则操作未定义。

示例：

```
#include <string.h> /* for strncat, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
```

strncat （续）

```
  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));
  printf("buf3 : %s\n", buf3);
  printf("\t(%d characters)\n\n\n", strlen(buf3));

  strncat(buf1, buf2, 6);
  printf("buf1 after strncat of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strncat(buf1, buf2, 25);
  printf("buf1 after strncat of 25 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));

  printf("\n");

  strncat(buf1, buf3, 4);
  printf("buf1 after strncat of 4 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

输出：

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf3 : Why?
        (4 characters)

buf1 after strncat of 6 characters of buf2:
        We're hereWhere
        (16 characters)

buf1 after strncat of 25 characters of buf2:
        We're hereWhere Where is the time?
        (34 characters)

buf1 after strncat of 4 characters of buf3:
   We're hereWhere Where is the time?Why?
        (38 characters)
```

strncmp

描述： 比较两个字符串，比较长度为某一指定的字符数。

头文件： <string.h>

函数原型：
```
int strncmp(const char *s1, const char *s2,
 size_t n);
```

参数：

s1 第一个字符串

s2 第二个字符串

n 要比较的字符数

返回值： 如果 *s1* 大于 *s2*，返回正数；如果 *s1* 等于 *s2*，返回 0；如果 *s1* 小于 *s2*，返回负数。

说明： strncmp 返回 *s1* 和 *s2* 之间第一个不同的字符。不比较空字符后的字符。

示例：
```
#include <string.h> /* for strncmp */
#include <stdio.h>  /* for printf  */

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strncmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
   printf("buf2 comes before buf1\n");

  printf("\n");

  res = strncmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
```

strncmp （续）

```
  printf("\n");

  res = strncmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
else
    printf("buf3 comes before buf1\n");
}
```

输出：

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

strncpy

描述： 从源字符串中复制字符到目的字符串中，直到指定的字符数。

头文件： <string.h>

函数原型： `char *strncpy(char *s1, const char *s2, size_t n);`

参数：

s1 目的字符串

s2 源字符串

n 要复制的字符数

返回值： 返回指向目的字符串的指针。

说明： 从源字符串复制 *n* 个字符到目的字符串中，如果源字符串的字符数小于 *n*，目的字符串将填充空字符直到总共 n 个字符。如果复制了 *n* 个字符，而没有遇到空字符，则目的字符串不会以空字符终止。如果字符串重叠，则操作未定义。

示例：

```
#include <string.h> /* for strncpy, strlen */
#include <stdio.h>  /* for printf          */

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
  char buf4[7]  = "Where?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n", buf3);
  printf("buf4 : %s\n", buf4);
```

strncpy （续）

```
  strncpy(buf1, buf2, 6);
  printf("buf1 after strncpy of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf2, 18);
  printf("buf1 after strncpy of 18 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf3, 5);
  printf("buf1 after strncpy of 5 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));

  printf("\n");

  strncpy(buf1, buf4, 9);
  printf("buf1 after strncpy of 9 characters "
         "of buf4: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
}
```

输出：

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?
buf4 : Where?
buf1 after strncpy of 6 characters of buf2:
        Where here
        ( 10 characters)

buf1 after strncpy of 18 characters of buf2:
        Where is the time?
        ( 18 characters)

buf1 after strncpy of 5 characters of buf3:
        Why?
        ( 4 characters)

buf1 after strncpy of 9 characters of buf4:
        Where?
        ( 6 characters)
```

strncpy （续）

说明：

每个缓冲区包含所示的字符串，加上后面的空字符，总长度为 50，利用函数 strlen 可以求出第一个空字符前的字符串的长度 （但不包含第一个空字符）。

在第一个例子中， buf2 中的 6 个字符 （“Where”）替换 buf1 的前 6 个字符 （“We're”），而 buf1 中其余字符保持不变 （“here”加空字符）。

在第二个例子中， buf2 中的 18 个字符替换 buf1 的前 18 个字符，而 buf1 中其余字符仍为空字符。

在第三个例子中， buf3 中的 5 个字符 （“Why?”加一个空终止符）替换 buf1 的前 5 个字符。 buf1 现在实际包含 （“Why?”加一个空字符，加 “ is the time?”，再加 32 个空字符）。由于遇到第一个空字符时为 4 个字符，利用 strlen 函数来求字符数时结果为 4。

在第四个例子中，由于 buf4 只有 7 个字符，函数 strncpy 增加两个空字符来替换 buf1 的前 9 个字符。这样 buf1 的结果为 6 个字符 （“Where?”），后面跟 3 个空字符，加 9 个字符 （“the time?”），再加 32 个空字符。

strpbrk

描述： 在一个字符串中搜索与指定一组字符中任何一个字符相匹配的第一个字符的位置。

头文件： <string.h>

函数原型： `char *strpbrk(const char *s1, const char *s2);`

参数：

s1 指向要被搜索的字符串的指针

s2 指向要查找字符的指针

返回值： 如果找到，返回 *s1* 中匹配的字符；否则返回空指针。

说明： 该函数在字符串 *s1* 中搜索与 *s2* 中任何一个字符相匹配的第一个字符的位置。

示例：

```
#include <string.h> /* for strpbrk, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "xyz";
  char str3[20] = "eou?";
  char *ptr;
  int res;

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str2);
  ptr = strpbrk(str1, str2);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
```

strpbrk （续）

```
  printf("\n");

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str3);
  ptr = strpbrk(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
}
```

输出：

```
strpbrk("What time is it?", "xyz")
match not found

strpbrk("What time is it?", "eou?")
match found at position 9
```

strrchr

描述： 在字符串中查找指定字符最后出现的位置。

头文件： <string.h>

函数原型： `char *strrchr(const char *s, int c);`

参数：

s 指向要被搜索的字符串的指针

c 要查找的字符

返回值： 如果找到，则返回指向该字符的指针；否则返回空指针。

说明： 该函数在字符串 *s* 中查找字符 *c* 最后出现的位置，包括终止空字符。

示例：

```
#include <string.h> /* for strrchr, NULL */
#include <stdio.h>  /* for printf        */

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = strrchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
```

strrchr （续）

```
  printf("\n");

  ptr = strrchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

输出：

```
buf1 : What time is it?

m found at position 8

y not found
```

strspn

描述： 计算自一个字符串开头起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 *s1, const char *s2);

参数：

s1 指向要被搜索的字符串的指针

s2 指向要查找的字符的指针

返回值： 返回 *s1* 中自 *s1* 开头起包含在 *s2* 中的连续字符数。

说明： 当 *s1* 中的字符不在 *s2* 中时，该函数停止搜索。

示例：

```
#include <string.h> /* for strspn */
#include <stdio.h>  /* for printf */

int main(void)
{
  char str1[20] = "animal";
  char str2[20] = "aeiounm";
  char str3[20] = "aimnl";
  char str4[20] = "xyz";
  int res;

  res = strspn(str1, str2);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strspn(str1, str3);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str3, res);

  res = strspn(str1, str4);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str4, res);
}
```

输出：

```
strspn("animal", "aeiounm") = 5
strspn("animal", "aimnl") = 6
strspn("animal", "xyz") = 0
```

strspn （续）

说明：
在第一个结果中， l 不在 *s2* 中。
在第二个结果中，终止空字符不在 *s2* 中。
在第三个结果中， a 不在 *s2* 中，因此比较停止。

strstr

描述： 查找一个字符串在另一个字符串中第一次出现的位置。

头文件： <string.h>

函数原型： char *strstr(const char *s1, const char *s2);

参数：
s1 指向要被搜索的字符串的指针
s2 指向要查找的子字符串的指针

返回值： 如果找到，返回匹配子字符串的第一个元素的位置；否则返回空指针。

说明： 该函数将查找字符串 *s2* 在字符串 *s1* 中第一次出现的位置 （除空终止符外）。如果 *s2* 是一个长度为零的字符串，则返回 *s1*。

示例：

```
#include <string.h> /* for strstr, NULL */
#include <stdio.h>  /* for printf       */

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "is";
  char str3[20] = "xyz";
  char *ptr;
  int res;

  printf("str1 : %s\n", str1);
  printf("str2 : %s\n", str2);
  printf("str3 : %s\n\n", str3);

  ptr = strstr(str1, str2);
if (ptr != NULL)
  {
   res = ptr - str1 + 1;
   printf("\"%s\" found at position %d\n",
          str2, res);
  }
  else
    printf("\"%s\" not found\n", str2);
```

strstr （续）

```
  printf("\n");

  ptr = strstr(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("\"%s\" found at position %d\n",
           str3, res);
  }
  else
    printf("\"%s\" not found\n", str3);
}
```

输出：

```
str1 : What time is it?
str2 : is
str3 : xyz

"is" found at position 11

"xyz" not found
```

strtok

描述： 通过插入空字符来代替字符串中的分隔符 （比如逗号），将字符串分成子字符串或标记 （token）。

头文件： <string.h>

函数原型： `char *strtok(char *s1, const char *s2);`

参数：

s1 指向要被搜索的以空字符终止的字符串的指针

s2 指向要查找字符的指针 （作为分隔符）

返回值： 返回指向标记的第一个字符的指针 （*s1* 中没有出现在 *s2* 中的第一个字符）；如果没有找到标记，则返回空指针。

说明： 连续调用该函数能够通过用空字符替换指定的字符将一个字符串分成多个子字符串 （或标记）。对一个特定的字符串第一次调用该函数时，该字符串应传递到 *s1* 中。在第一次调用之后，该函数可通过将一个空值传递到 *s1* 中调用该函数来自最后一个分隔符继续解析这个字符串。

如果忽略所有出现在字符串 *s2* 中的前导字符 （分隔符），接着忽略所有未出现在 *s2* 中的字符，这段字符被称为 “标记”，然后用一个空字符覆盖下一个字符，结束当前的 “标记”。函数 strtok 将会保存指向下一个字符的指针，下一次查找就从这里开始。如果 strtok 在找到分隔符之前就找到字符串的终止符，则当前的标记延伸到 *s1* 字符串的末端。如果第一次调用函数 strtok，则它不会修改字符串 *s1* （也就是不会有空字符写到 *s1* 中）。每次调用 strtok 时，传递到 *s2* 中的字符组不必相同。

如果对 *s1* 初始调用该函数之后再用一个非空参数调用 strtok 函数，则字符串变成要被搜索的新字符串。原先被搜索的旧字符串将会丢失。

strtok （续）

示例：

```
#include <string.h> /* for strtok, NULL */
#include <stdio.h> / * for printf       */

int main(void)
{
  char str1[30] = "Here, on top of the world!";
  char delim[5] = ", .";
  char *word;
  int x;

  printf("str1 : %s\n", str1);
  x = 1;
  word = strtok(str1,delim);
  while (word != NULL)
  {
    printf("word %d: %s\n", x++, word);
    word = strtok(NULL, delim);
  }
}
```

输出：

```
str1 : Here, on top of the world!
word 1: Here
word 2: on
word 3: top
word 4: of
word 5: the
word 6: world!
```

strxfrm

描述： 采用特定语言环境的规则转换字符串 （参见 “说明”）。

头文件： <string.h>

函数原型： size_t strxfrm(char *s1, const char *s2, size_t n);

参数：

s1 目的字符串

s2 要转换的源字符串

n 要转换的字符数

返回值： 返回被转换字符串的长度，不包括终止空字符。如果 *n* 为零，字符串不会被转换 （这样 *s1* 为空），返回 *s2* 的长度。

说明： 如果返回值大于或等于 *n*，则 *s1* 的内容不确定。由于 MPLAB C30 不支持其他语言环境，因此除了目的字符串的长度限制在 n-1 内之外，这个转换与 strcpy 具有相同的作用。

4.16 <TIME.H> 日期和时间函数

头文件 time.h 由一系列类型 （ type）、宏 （macro）和函数 (function）组成，对时间进行操作。

clock_t

描述： 存储处理器时间值。

头文件： <time.h>

函数原型：

```
typedef long clock_t
```

size_t

描述： sizeof 运算符结果的类型。

头文件： <time.h>

struct tm

描述： 用来保存时间和日期 （日历时间）的结构。

头文件： <time.h>

函数原型：

```
struct tm {
int tm_sec; /*seconds after the minute ( 0 to 61 ) */
            /* allows for up to two leap seconds */
int tm_min; /* minutes after the hour ( 0 to 59 ) */
int tm_hour;/* hours since midnight ( 0 to 23 ) */
int tm_mday;/* day of month ( 1 to 31 ) */
int tm_mon; /* month ( 0 to 11 where January = 0 ) */
int tm_year;/* years since 1900 */
int tm_wday;/* day of week ( 0 to 6 where Sunday = 0 ) */
int tm_yday;/* day of year ( 0 to 365 where January 1 = 0 ) */
int tm_isdst;/* Daylight Savings Time flag */
}
```

说明： 如果 tm_isdst 为正值，则夏令时是有效的；如果 tm_isdst 为零，夏令时是无效的；如果 tm_isdst 为负值，则夏令时的状态是不可知。

time_t

描述： 表示日历时间值。

头文件： <time.h>

函数原型：

```
typedef long time_t
```

CLOCKS_PER_SEC

描述： 每秒钟里包含的处理器时钟 （processor clock）数。

头文件： <time.h>

函数原型：

```
#define CLOCKS_PER_SEC
```

值： 1

说明： MPLAB C30 返回时钟节拍数 （指令周期数），而非实际时间。

NULL

描述： 空指针常数的值。

头文件： <time.h>

asctime

描述： 将时间结构转换为字符串。

头文件： <time.h>

函数原型：
```
char *asctime(const struct tm *tptr);
```

参数： *tptr* 时间 / 日期结构

返回值： 返回指向下列格式字符串的指针：
```
  DDD MMM dd hh:mm:ss YYYY
```
DDD 表示星期几
MMM 表示一年中的月份
dd 表示一月个中的第几天
hh 表示小时
mm 表示分
ss 表示秒
YYYY 表示年份

示例：
```
#include <time.h>  /* for asctime, tm */
#include <stdio.h> /* for printf      */

volatile int i;

int main(void)
{
  struct tm when;
  time_t whattime;

  when.tm_sec = 30;
  when.tm_min = 30;
  when.tm_hour = 2;
  when.tm_mday = 1;
  when.tm_mon = 1;
  when.tm_year = 103;

  whattime = mktime(&when);
  printf("Day and time is %s\n", asctime(&when));
}
```

输出：
```
Day and time is Sat Feb  1 02:30:30 2003
```

clock

描述： 计算处理器时间。

头文件： <time.h>

函数原型：
```
clock_t clock(void);
```

返回值： 返回所用的处理器时钟节拍数。

说明： 如果目标环境不能计算所用的处理器时间，函数返回 -1，采用 clock_t 表示（即 (clock_t) -1）。默认情况下，MPLAB C30 返回以指令周期表示的时间。

clock （续）

示例：

```
#include <time.h>  /* for clock  */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  clock_t start, stop;
  int ct;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
}
```

输出：

```
start = 0
stop = 317
```

ctime

描述： 将日历时间转换为当地时间的字符串表示。

头文件： <time.h>

函数原型： `char *ctime(const time_t *tod);`

参数： *tod* 指向存储时间的指针

返回值： 返回表示传递的当地时间参数的字符串的地址。

说明： 该函数与 asctime(localtime(tod)) 相同。

示例：

```
#include <time.h>  /* for mktime, tm, ctime */
#include <stdio.h> /* for printf            */

int main(void)
{
  time_t whattime;
  struct tm nowtime;

  nowtime.tm_sec = 30;
  nowtime.tm_min = 30;
  nowtime.tm_hour = 2;
  nowtime.tm_mday = 1;
  nowtime.tm_mon = 1;
  nowtime.tm_year = 103;

  whattime = mktime(&nowtime);
  printf("Day and time %s\n", ctime(&whattime));
}
```

输出：

```
Day and time Sat Feb  1 02:30:30 2003
```

difftime

描述： 找出两个时间之间的差值。

头文件： <time.h>

函数原型： double difftime(time_t *t1*, time_t *t0*);

参数： *t1* 结束时间

t0 开始时间

返回值： 返回 *t1* 和 *t0* 之间的秒数。

说明： 默认情况下， MPLAB C30 返回指令周期表示的时间，因此 difftime 返回 *t1* 和 *t0* 之间的时钟节拍数。

示例：

```
#include <time.h>  /* for clock, difftime */
#include <stdio.h> /* for printf          */

volatile int i;

int main(void)
{
  clock_t start, stop;
  double elapsed;

  start = clock();
  for (i = 0; i < 10; i++)
  stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
  elapsed = difftime(stop, start);
  printf("Elapsed time = %.0f\n", elapsed);
}
```

输出：

```
start = 0
stop = 317
Elapsed time = 317
```

gmtime

描述： 将日历时间转换为世界协调时间 （UTC），也就是格林威治标准时间（GMT）。

头文件： <time.h>

函数原型： struct tm *gmtime(const time_t **tod*);

参数： *tod* 指向存储的时间的指针

返回值： 返回时间结构的地址。

说明： 该函数将 *tod* 的值转换为 tm 型的时间结构。默认情况下， MPLAB C30 返回以指令周期表示的时间。在这种默认情况下，除了 gmtime 将返回 tm_isdst （夏令时标志）值为 0 表明夏令时无效外， gmtime 和 localtime 完全相同。

gmtime （续）

示例：

```
#include <time.h>  /* for gmtime, asctime, */
                   /* time_t, tm           */
#include <stdio.h> /* for printf           */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = gmtime(&timer);
  printf("UTC time = %s\n", asctime(newtime));
}
```

输出：

```
UTC time = Mon Oct 20 16:43:02 2003
```

localtime

描述： 将值转换为本地时间。

头文件： `<time.h>`

函数原型： `struct tm *localtime(const time_t *tod);`

参数： *tod* 指向存储的时间的指针

返回值： 返回时间结构的地址。

说明： 默认情况下， MPLAB C30 返回以指令周期表示的时间。默认情况下，MPLAB C30 返回以指令周期表示的时间。在这种默认情况下，除了 `localtime` 将返回 `tm_isdst`（夏令时标志）值为 -1 表明夏令时状态未知外， `gmtime` 和 `localtime` 完全相同。

示例：

```
#include <time.h>  /* for localtime,      */
                   /* asctime, time_t, tm */
#include <stdio.h> /* for printf          */

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = localtime(&timer);
  printf("Local time = %s\n", asctime(newtime));
}
```

输出：

```
Local time = Mon Oct 20 16:43:02 2003
```

mktime

描述： 将当地时间转换为日历时间。

头文件： <time.h>

函数原型：
```
time_t mktime(struct tm *tptr);
```

参数： *tptr* 指向时间结构的指针

返回值： 返回编码成 time_t 的值的日历时间。

说明： 如果不能表示为日历时间，函数返回为 -1，采用 time_t 表示（即 (time_t) -1）。

示例：
```
#include <time.h>  /* for localtime,   */
                   /* asctime, mktime, */
                   /* time_t, tm       */
#include <stdio.h> /* for printf       */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for struct tm */
  newtime = localtime(&timer);
  printf("Local time = %s", asctime(newtime));

  whattime = mktime(newtime);
  printf("Calendar time as time_t = %ld\n",
          whattime);
}
```

输出：
```
Local time = Mon Oct 20 16:43:02 2003
Calendar time as time_t = 1066668182
```

strftime

描述： 按照格式参数将时间结构转换为字符串。

头文件： <time.h>

函数原型：
```
size_t strftime(char *s, size_t n,
 const char *format, const struct tm *tptr);
```

参数：

s 输出字符串

n 字符串的最大长度

format 格式控制字符串

tptr 指向 tm 数据结构的指针

返回值： 如果数组 s 中包括终止符在内的总字符数不大于 *n*，则返回存放在数组 *s* 中的字符数；否则函数返回 0，且数组 *s* 的内容不确定。

说明： 格式参数如下：

%a 简写的星期几表示

%A 完整的星期几表示

%b 简写的月份名

%B 完整的月份名

%c 恰当的日期和时间表示法

%d 月份中的第几天（01-31）

%H 一天中的小时（00-23）

strftime （续）

%I 一天中的小时 （01-12）
%j 一年中的第几天 （001-366）
%m 一年中的月份 （01-12）
%M 小时中的分 （00-59）
%p AM/PM 表示法
%S 分中的秒 （00-61）
允许两个闰秒
%U 一年中的第几个星期，其中星期天为第一周的第一天 （00-53）
%w 星期几，星期天为一周的第 0 天 （0-6）
%W 一年中的第几个星期，其中星期一为第一周的第一天 （00-53）
%x 适当的日期表示法
%X 适当的时间表示法
%y 没有百年的年份表示法 （00-99）
%Y 有百年的年份表示法
%Z 时区 （可能为简写形式）或者如果无法获得时区，为空字符
%% 百份号 %

示例：

```
#include <time.h>  /* for strftime, */
                   /* localtime,    */
                   /* time_t, tm    */
#include <stdio.h> /* for printf    */

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;
  char buf[128];

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for structure */
  newtime = localtime(&timer);

  strftime(buf, 128, "It was a %A, %d days into the "
           "month of %B in the year %Y.\n", newtime);
  printf(buf);

  strftime(buf, 128, "It was %W weeks into the year "
           "or %j days into the year.\n", newtime);
  printf(buf);
}
```

输出：

```
It was a Monday, 20 days into the month of October in
the year 2003.
It was 42 weeks into the year or 293 days into the
year.
```

time

描述： 计算当前日历时间。

头文件： <time.h>

函数原型： time_t time(time_t *tod);

参数： *tod* 指向时间存储位置的指针

返回值： 返回编码为 time_t 的值的日历时间。

说明： 如果目标环境不能确定时间，函数返回 -1，用 time_t 表示。默认情况下， MPLAB C30 返回以指令周期表示的时间，这个函数可定制，参见 pic30 函数库。

示例：

```
#include <time.h>  /* for time   */
#include <stdio.h> /* for printf */

volatile int i;

int main(void)
{
  time_t ticks;

  time(0); /* start time */
  for (i = 0; i < 10; i++) /* waste time */
  time(&ticks); /* get time */
  printf("Time = %ld\n", ticks);
}
```

输出：

```
Time = 256
```

4.17 <MATH.H> 数学函数

头文件 math.h 由进行一般数学运算的宏和各种函数组成。错误条件可通过定义域错误和值域错误来处理 （参见 errno.h）。

当输入参数超出函数的定义域时将产生定义域错误。通过将 EDOM 的值存储在 errno 中并返回为每个函数定义的特定值来报告错误。

当结果超过目标精度所能表示的范围时将产生值域错误，报告错误的方式为：将 ERANGE 的值存储在 errno 中，且如果结果溢出（返回值太大）返回 HUGE_VAL；如果结果下溢 （返回值太小）则返回零。

对一些特殊的值 （如 NaN、零和无穷）的响应，可能随不同函数而有所不同。每个函数的描述中包括函数对这些值的响应的定义。

HUGE_VAL

描述： 函数发生值域错误时返回 HUGE_VAL （例如：函数试图返回的值太大，超出了目标精度所能表示的范围）。

头文件： <math.h>

说明： 如果函数的结果是负数，而且太大 （绝对值），超出了目标精度所能表示的范围，则返回 -HUGE_VAL。当打印的结果是 +/- HUGE_VAL 时，它将用 +/- inf 来表示。

acos

描述： 计算双精度浮点型值的三角反余弦函数。

头文件： <math.h>

函数原型： double acos (double *x*);

参数： *x* 要为其返回反余弦值的值，在 -1 和 1 之间。

返回值： 返回反余弦值 （单位为弧度），值的范围在 0 到 π 之间 （包括 0 和 π）。

说明： 如果 *x* 的值小于 -1 或大于 1，将发生定义域错误。

示例：

```
#include <math.h>  /* for acos           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;

  errno = 0;
  x = -2.0;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);
```

acos （续）

```
  errno = 0;
  x = 0.10;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);
}
```

输出：

```
Error: domain error
The arccosine of -2.000000 is nan

The arccosine of 0.100000 is 1.470629
```

acosf

描述： 计算单精度浮点型值的三角反余弦函数。

头文件： <math.h>

函数原型： float acosf (float *x*);

参数： *x* -1 和 1 之间的值

返回值： 返回反余弦值 （单位为弧度），值的范围在 0 到 π 之间 （包括 0 和 π）。

说明： 如果 *x* 的值小于 -1 或大于 1，将发生定义域错误。

示例：

```
#include <math.h>  /* for acosf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);
}
```

输出：

```
Error: domain error
The arccosine of 2.000000 is nan

The arccosine of 0.000000 is 1.570796
```

asin

描述： 计算双精度浮点型值的三角反正弦函数。

头文件： <math.h>

函数原型： double asin (double *x*);

参数： *x* 要为其返回反正弦值的值，在 -1 和 1 之间。

返回值： 返回反正弦值（单位为弧度），值的范围在 - π /2 到 + π /2 之间（包括 - π /2 和 + π /2）。

说明： 如果 *x* 的值小于 -1 或大于 1，将发生定义域错误。

示例：

```
#include <math.h>  /* for asin           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);
}
```

输出：

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

asinf

描述： 计算单精度浮点型值的三角反正弦函数。

头文件： <math.h>

函数原型： float asinf (float *x*);

参数： *x* -1 和 1 之间的值

返回值： 返回反正弦值（单位为弧度），值的范围在 - π /2 到 + π /2 之间（包括 - π /2 和 + π /2）。

说明： 如果 *x* 的值小于 -1 或大于 1，将发生定义域错误。

示例：

```
#include <math.h>  /* for asinf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

asinf（续）

```
  errno = 0;
  x = 2.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n\n", x, y);
}
```

输出：

```
Error: domain error
The arcsine of 2.000000 is nan

The arcsine of 0.000000 is 0.000000
```

atan

描述： 计算双精度浮点型值的三角反正切函数。

头文件： <math.h>

函数原型： double atan (double *x*);

参数： *x* 要为其返回反正切值的值

返回值： 返回反正切值（单位为弧度），值的范围在 - π /2 到 + π /2 之间（包括 - π /2 和 + π /2）。

说明： 不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for atan   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 2.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n\n", x, y);

  x = -1.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n\n", x, y);
}
```

输出：

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atanf

描述： 计算双精度浮点型值的三角反正切函数。

头文件： <math.h>

函数原型： float atanf (float *x*);

参数： *x* 要为其返回反正切值的值

返回值： 返回反正切值（单位为弧度），值的范围在 - π /2 到 + π /2 之间（包括 - π /2 和 + π /2）。

说明： 不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for atanf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = 2.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n\n", x, y);

  x = -1.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n\n", x, y);
}
```

输出：

```
The arctangent of 2.000000 is 1.107149

The arctangent of -1.000000 is -0.785398
```

atan2

描述： 计算双精度浮点型值 y/x 的三角反正切函数。

头文件： <math.h>

函数原型： double atan2 (double *y*, double *x*);

参数： *y* 要为其返回反正切函数的 y 值

x 要为其返回反正切函数的 x 值

返回值： 返回反正切值（单位为弧度），值的范围在 - π 到 + π 之间（包括 - π 和 + π），通过两个参数的符号来决定象限。

说明： 如果 *x* 和 *y* 的值都为零或者都为 +/- 无穷大，将发生定义域错误。

示例：

```
#include <math.h>  /* for atan2          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y, z;
```

atan2 （续）

```
  errno = 0;
  x = 0.0;
  y = 2.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = -1.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);
}
```

输出：

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

atan2f

描述： 计算单精度浮点型值 y/x 的三角反正切函数值。

头文件： <math.h>

函数原型： float atan2f (float *y*, float *x*);

参数： *y* 要为其返回反正切函数的 y 值

x 要为其返回反正切函数的 x 值

返回值： 返回反正切值（单位为弧度），值的范围在 - π 到 + π 之间（包括 - π 和 + π），通过两个参数的符号来决定象限。

说明： 如果 *x* 和 *y* 的值都为零或者都为 +/- 无穷大，将发生定义域错误。

示例：

```
#include <math.h>  /* for atan2f          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno           */

int main(void)
{
  float x, y, z;

  errno = 0;
  x = 2.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = -1.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);

  errno = 0;
  x = 0.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n\n",
          y, x, z);
}
```

输出：

```
The arctangent of 2.000000/0.000000 is 1.570796

The arctangent of 0.000000/-1.000000 is 3.141593

Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

ceil

描述： 计算一个双精度浮点型值的上限。

头文件： <math.h>

函数原型： double ceil(double *x*);

参数： *x* 要为其返回上限的浮点型值。

返回值： 返回大于或等于 *x* 的最小整型值。

说明： 不会发生定义域或值域错误，参见 floor。

示例：

```
#include <math.h>  /* for ceil   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceil (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

输出：

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

ceilf

描述： 计算一个单精度浮点型值的上限。

头文件： <math.h>

函数原型： float ceilf(float *x*);

参数： *x* 浮点型值。

返回值： 返回大于或等于 *x* 的最小整型值。

说明： 不会发生定义域或值域错误，参见 floorf。

示例：

```
#include <math.h>  /* for ceilf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceilf (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

输出：

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

cos

描述： 计算双精度浮点型值的三角余弦函数。

头文件： <math.h>

函数原型： double cos (double *x*);

参数： *x* 要为其返回余弦值的值

返回值： 返回 *x* 的余弦值（单位为弧度），值的范围在 -1 到 +1 之间（包括 -1 和 +1）。

说明： 如果 *x* 的值为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for cos            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;

  errno = 0;
  x = -1.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
```

cos （续）

```
  errno = 0;
  x = 0.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
}
```

输出：

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosf

描述： 计算单精度浮点型值的三角余弦函数。

头文件： <math.h>

函数原型： float cosf (float *x*);

参数： *x* 要为其返回余弦值的值

返回值： 返回 *x* 的余弦值 （单位为弧度），值的范围在 -1 到 +1 之间 （包括 -1 和 +1）。

说明： 如果 *x* 的值为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for cosf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n\n", x, y);
}
```

输出：

```
The cosine of -1.000000 is 0.540302

The cosine of 0.000000 is 1.000000
```

cosh

描述： 计算双精度浮点型值的双曲余弦函数。

头文件： <math.h>

函数原型： double cosh (double *x*);

参数： *x* 为返回的双曲余弦值

返回值： 返回 *x* 的双曲余弦值

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for cosh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
         x, y);

  errno = 0;
  x = 0.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
         x, y);

  errno = 0;
  x = 720.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
         x, y);
}
```

输出：

```
The hyperbolic cosine of -1.500000 is 2.352410

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

coshf

描述： 计算单精度浮点型值的双曲余弦函数。

头文件： <math.h>

函数原型： float coshf (float *x*);

参数： *x* 要为其返回双曲余弦值的值

返回值： 返回 *x* 的双曲余弦值

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for coshf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 720.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n\n",
          x, y);
}
```

输出：

```
The hyperbolic cosine of -1.000000 is 1.543081

The hyperbolic cosine of 0.000000 is 1.000000

Error: range error
The hyperbolic cosine of 720.000000 is inf
```

exp

描述： 计算关于 x 的指数函数 （e 的 x 次幂，其中 x 为双精度浮点型值）。

头文件： <math.h>

函数原型： double exp (double *x*);

参数： *x* 要为其返回指数函数值的值

返回值： 返回 *x* 的指数。如果溢出， exp 返回 inf ； 如果下溢， exp 返回 0。

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for exp            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = 1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = -1E3;
  y = exp (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);
}
```

输出：

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

expf

描述： 计算关于 x 的指数函数（e 的 x 次幂，其中 x 为单精度浮点型值）。

头文件： <math.h>

函数原型： float expf (float *x*);

参数： *x* 要为其返回指数函数值的值

返回值： 返回 *x* 的指数。如果溢出， exp 返回 inf ；如果下溢， exp 返回 0。

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for expf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = 1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);

  errno = 0;
  x = -1.0E3F;
  y = expf (x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n\n", x, y);
}
```

输出：

```
The exponential of 1.000000 is 2.718282

Error: range error
The exponential of 1000.000000 is inf

Error: range error
The exponential of -1000.000000 is 0.000000
```

fabs

描述： 计算一个双精度浮点型值的绝对值。

头文件： <math.h>

函数原型： double fabs(double *x*);

参数： *x* 要为其返回绝对值的双精度浮点型值

返回值： 返回 *x* 的绝对值（负数返回其相反数，正数返回其本身）。

说明： 不会发生定义域或值域错误。

示例：

```
#include <math.h>  /* for fabs   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = 1.75;
  y = fabs (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5;
  y = fabs (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

输出：

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

fabsf

描述： 计算一个单精度浮点型值的绝对值。

头文件： <math.h>

函数原型： float fabsf(float *x*);

参数： *x* 要为其返回绝对值的浮点型值

返回值： 返回 *x* 的绝对值（负数返回其相反数，正数返回其本身）。

说明： 不会发生定义域或值域错误。

示例：

```
#include <math.h>  /* for fabsf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;

  x = 1.75F;
  y = fabsf (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5F;
  y = fabsf (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

输出：

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

floor

描述： 计算一个双精度浮点型值的下限。

头文件： <math.h>

函数原型： double floor (double *x*);

参数： *x* 要为其返回下限的浮点型值。

返回值： 返回小于或等于 *x* 的最大整型值。

说明： 不会发生定义域或值域错误，参见 ceil。

示例：

```
#include <math.h>  /* for floor  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floor (x[i]);
    printf("The ceiling for %f is %f\n", x[i], y);
  }
}
```

输出：

```
The  floor for 2.000000 is 2.000000
The  floor for 1.750000 is 1.000000
The  floor for 1.500000 is 1.000000
The  floor for 1.250000 is 1.000000
The  floor for -2.000000 is -2.000000
The  floor for -1.750000 is -2.000000
The  floor for -1.500000 is -2.000000
The  floor for -1.250000 is -2.000000
```

floorf

描述： 计算一个单精度浮点型值的下限。

头文件： <math.h>

函数原型： float floorf(float *x*);

参数： *x* 浮点型值

返回值： 返回小于或等于 *x* 的最大整型值。

说明： 不会发生定义域或值域错误，参见 ceilf。

floorf （续）

示例：

```
#include <math.h>  /* for floorf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floorf (x[i]);
    printf("The floor for  %f is  %f\n", x[i], y);
  }
}
```

输出：

```
The floor for  2.000000 is  2.000000
The floor for  1.750000 is  1.000000
The floor for  1.500000 is  1.000000
The floor for  1.250000 is  1.000000
The floor for  -2.000000 is  -2.000000
The floor for  -1.750000 is  -2.000000
The floor for  -1.500000 is  -2.000000
The floor for  -1.250000 is  -2.000000
```

fmod

描述： 计算双精度浮点型值 x/y 的余数。

头文件： <math.h>

函数原型： double fmod(double *x*, double *y*);

参数： *x* 双精度浮点型值。

y 双精度浮点型值。

返回值： 返回 *x* 除以 *y* 的余数。

说明： 如果 *y* = 0，将发生定义域错误。如果 *y* 非零，则结果与 *x* 符号相同，且结果的绝对值小于 *y* 的绝对值。

示例：

```
#include <math.h>  /* for fmod           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = 7.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);
```

fmod （续）

```
  errno = 0;
  x = 7.0;
  y = 7.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = 5.0;
  y = -3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = -5.0;
  y = -5.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);

  errno = 0;
  x = 7.0;
  y = 0.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n\n",
         x, y, z);
}
```

输出：

```
For fmod(7.000000, 3.000000) the remainder is 1.000000

For fmod(7.000000, 7.000000) the remainder is 0.000000

For fmod(-5.000000, 3.000000) the remainder is -2.000000

For fmod(5.000000, -3.000000) the remainder is 2.000000

For fmod(-5.000000, -5.000000) the remainder is -0.000000

Error: domain error
For fmod(7.000000, 0.000000) the remainder is nan
```

fmodf

描述： 计算单精度浮点型值 x/y 的余数。

头文件： <math.h>

函数原型： float fmodf(float *x*, float *y*);

参数： *x* 单精度浮点型值

y 单精度浮点型值

返回值： 返回 *x* 除以 *y* 的余数。

说明： 如果 *y* = 0，将发生定义域错误；如果 *y* 为非零值，则结果与 *x* 符号相同，且结果的绝对值小于 *y* 的绝对值。

示例：

```
#include <math.h>  /* for fmodf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = 7.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);

  errno = 0;
  x = -5.0F;
  y = 3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -3.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -5.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);
```

fmodf （续）

```
  errno = 0;
  x = 7.0F;
  y = 0.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
          " %f\n\n", x, y, z);

  errno = 0;
  x = 7.0F;
  y = 7.0F;
  z = fmodf (x, y);
  if (errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is"
         " %f\n\n", x, y, z);
}
```

输出：

```
For fmodf (7.000000, 3.000000) the remainder is 1.000000

For fmodf (-5.000000, 3.000000) the remainder is -2.000000

For fmodf (5.000000, -3.000000) the remainder is 2.000000

For fmodf (5.000000, -5.000000) the remainder is 0.000000

Error: domain error
For fmodf (7.000000, 0.000000) the remainder is nan

For fmodf (7.000000, 7.000000) the remainder is 0.000000
```

frexp

描述： 获取双精度浮点型数的小数和指数。

头文件： <math.h>

函数原型： double frexp (double *x*, int **exp*);

参数：

x 要为其返回小数和指数的双精度浮点型值

**exp* 指向存储的整型指数的指针

返回值： 返回小数， *exp* 指向指数。如果 *x* 为 0，函数返回的小数和指数的值都为 0。

说明： 小数的绝对值的范围为 1/2 （包括 1/2）到 1 （不包括 1）。不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for frexp  */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y;
  int n;
```

frexp （续）

```
  x = 50.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

输出：

```
For frexp of 50.000000
  the fraction is 0.781250
   and the exponent is 6

For frexp of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexp of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

frexpf

描述： 获取单精度浮点型数的小数和指数。

头文件： <math.h>

函数原型： `float frexpf (float x, int *exp);`

参数：

x 要为其返回小数和指数的单精度浮点型值

**exp* 指向存储的整型指数的指针

返回值： 返回小数， *exp* 指向指数。如果 *x* 为 0，函数返回的小数和指数的值都为 0。

说明： 小数的绝对值的范围为 1/2 （包括 1/2）到 1 （不包括 1）。不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for frexpf */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y;
  int n;
```

frexpf （续）

```
  x = 0.15F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ",
         x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

输出：

```
For frexpf of 0.150000
  the fraction is 0.600000
   and the exponent is -2

For frexpf of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexpf of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

ldexp

描述： 计算一个双精度浮点数与 2 的指数相乘的结果。

头文件： <math.h>

函数原型： double ldexp(double *x*, int *ex*);

参数： *x* 浮点型值

ex 整型指数

返回值： 返回 *x* * 2^*ex*。如果溢出， ldexp 返回 inf ；如果下溢， ldexp 返回 0。

说明： 在溢出或下溢时会发生值域错误。

示例：

```
#include <math.h>  /* for ldexp          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y;
  int n;
```

ldexp （续）

```
  errno = 0;
  x = -0.625;
  n = 2;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 2.5;
  n = 3;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 15.0;
  n = 10000;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);
}
```

输出：

```
For a number = -0.625000 and an exponent = 2
  ldexp(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexp(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

ldexpf

描述： 计算一个单精度浮点型数与 2 的指数相乘的结果。

头文件： <math.h>

函数原型： float ldexpf(float *x*, int *ex*);

参数：

x 浮点型值

ex 整型指数

返回值： 返回 *x* * 2^*ex*。如果溢出， ldexp 返回 inf ； 如果下溢， ldexp 返回 0。

ldexpf （续）

说明： 在溢出或下溢时会发生值域错误。

示例：

```
#include <math.h>  /* for ldexpf         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y;
  int n;

  errno = 0;
  x = -0.625F;
  n = 2;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 2.5F;
  n = 3;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 15.0F;
  n = 10000;
  y = ldexpf (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexpf(%f, %d) = %f\n\n",
         x, n, y);
}
```

输出：

```
For a number = -0.625000 and an exponent = 2
  ldexpf(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexpf(2.500000, 3) = 20.000000

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexpf(15.000000, 10000) = inf
```

log

描述： 计算双精度浮点型值的自然对数。

头文件： <math.h>

函数原型： double log(double *x*);

参数： *x* 要为其返回自然对数的任意正数

返回值： 返回 *x* 的自然对数。如果 *x* 为 0，函数返回 -inf；如果 *x* 为负数，则函数返回 NaN。

说明： 如果 $x \leq 0$，将发生定义域错误。

示例：

```
#include <math.h>  /* for log            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);
}
```

输出：

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

log10

描述： 计算一个双精度浮点型值以 10 为底的对数。

头文件： <math.h>

函数原型： double log10(double *x*);

参数： *x* 任意双精度浮点型正数

返回值： 返回 *x* 以 10 为底的对数。如果 *x* 为 0，函数返回 -inf ；如果 *x* 为负数，则函数返回 NaN。

说明： 如果 x ≤ 0，将发生定义域错误。

示例：

```
#include <math.h>  /* for log10          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
          x, y);
}
```

输出：

```
The base-10 logarithm of 2.000000 is 0.301030

The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

log10f

描述： 计算一个单精度浮点型值以 10 为底的对数。

头文件： <math.h>

函数原型： float log10f(float *x*);

参数： *x* 任意单精度浮点型正数

返回值： 返回 *x* 以 10 为底的对数值。如果 *x* 为 0，函数返回 -inf；如果 *x* 为负数，则函数返回 NaN。

说明： 如果 x ≤ 0，将发生定义域错误。

示例：

```
#include <math.h>  /* for log10f         */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
         x, y);

  errno = 0;
  x = 0.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
         x, y);

  errno = 0;
  x = -2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n\n",
         x, y);
}
```

输出：

```
The base-10 logarithm of 2.000000 is 0.301030

Error: domain error
The base-10 logarithm of 0.000000 is -inf

Error: domain error
The base-10 logarithm of -2.000000 is nan
```

logf

描述： 计算一个单精度浮点型值的自然对数。

头文件： <math.h>

函数原型： float logf(float *x*);

参数： *x* 要为其返回自然对数的任意正数

返回值： 返回 *x* 的自然对数值。如果 *x* 为 0，函数返回 -inf；如果 *x* 为负数，则函数返回 NaN。

说明： 如果 x ≤ 0，将发生定义域错误。

示例：

```
#include <math.h>  /* for logf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);

  errno = 0;
  x = -2.0F;
  y = logf (x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n\n",
          x, y);
}
```

输出：

```
The natural logarithm of 2.000000 is 0.693147

The natural logarithm of 0.000000 is -inf

Error: domain error
The natural logarithm of -2.000000 is nan
```

modf

描述： 将一个双精度浮点型值分为整数和小数两部分。

头文件： <math.h>

函数原型： double modf(double *x*, double **pint*);

参数：

x 双精度浮点型值

pint 指向存储的整数部分的指针

返回值： 返回有符号的小数部分， *pint* 指向整数部分。

说明： 小数部分的绝对值的范围是 0（包括 0）到 1（不包括 1）。不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for modf   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x,y,n;

  x = 0.707;
  y = modf (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121;
  y = modf (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

输出：

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

modff

描述： 将一个单精度浮点型值分为整数和小数两部分。

头文件： <math.h>

函数原型： float modff(float *x*, float **pint*);

参数： *x* 单精度浮点型值

pint 指向存储的整数部分的指针

返回值： 返回有符号小数部分， *pint* 指向整数部分。

说明： 小数部分的绝对值的范围是 0（包括 0）到 1（不包括 1）。不会发生定义域和值域错误。

示例：

```
#include <math.h>  /* for modff  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x,y,n;

  x = 0.707F;
  y = modff (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121F;
  y = modff (x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

输出：

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

pow

描述： 计算 x 的 y 次幂值，其中 x 和 y 都为双精度浮点型数。

头文件： <math.h>

函数原型： double pow(double *x*, double *y*);

参数： *x* 底数

y 指数

返回值： 返回 *x* 的 *y* 次幂值（*x*^*y*）。

说明： 如果 *y* 的值为 0，则函数 pow 返回 1；如果 *x* 的值为 0.0，*y* 的值小于 0，则函数 pow 返回 inf，同时发生定义域错误。如果结果发生溢出或下溢，将发生值域错误。

示例：

```
#include <math.h>  /* for pow            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x,y,z;

  errno = 0;
  x = -2.0;
  y = 3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 3.0;
  y = -0.5;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

 errno = 0;
  x = 4.0;
  y = 0.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 0.0;
  y = -3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);
}
```

输出：

```
-2.000000 raised to 3.000000 is -8.000000

 3.000000 raised to -0.500000 is 0.577350

 4.000000 raised to 0.000000 is 1.000000

 Error: domain error
 0.000000 raised to -3.000000 is inf
```

powf

描述： 计算 x 的 y 次幂值，其中 x 和 y 都为单精度浮点型数。

头文件： <math.h>

函数原型： float powf(float *x*, float *y*);

参数： *x* 底数

y 指数

返回值： 返回 *x* 的 *y* 次幂值（*x*^*y*）。

说明： 如果 *y* 的值为 0，则函数 pow 返回 1；如果 *x* 的值为 0.0，*y* 的值小于 0，则函数 pow 返回 inf，同时发生定义域错误；如果结果发生溢出或下溢，将发生值域错误。

示例：

```
#include <math.h>  /* for powf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x,y,z;

  errno = 0;
  x = -2.0F;
  y = 3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 3.0F;
  y = -0.5F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);

  errno = 0;
  x = 0.0F;
  y = -3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n\n ", x, y, z);
}
```

输出：

```
-2.000000 raised to 3.000000 is -8.000000

 3.000000 raised to -0.500000 is 0.577350

 Error: domain error
 0.000000 raised to -3.000000 is inf
```

sin

描述： 计算双精度浮点型值的三角正弦函数。

头文件： <math.h>

函数原型： double sin (double *x*);

参数： *x* 要为其返回正弦值的值

返回值： 返回 *x* 的正弦值（单位为弧度），值的范围在 -1 到 +1 之间（包括 -1 和 +1）。

说明： 如果 *x* 的值为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for sin            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);
}
```

输出：

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinf

描述： 计算单精度浮点型值的三角正弦函数值。

头文件： <math.h>

函数原型： float sinf (float *x*);

参数： *x* 要为其返回正弦函值的值

返回值： 返回 *x* 的正弦值 （单位为弧度），值的范围在 -1 到 +1 之间 （包括 -1 和 +1）。

说明： 如果 *x* 的值为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for sinf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n\n", x, y);
}
```

输出：

```
The sine of -1.000000 is -0.841471

The sine of 0.000000 is 0.000000
```

sinh

描述： 计算双精度浮点型值的双曲正弦函数值。

头文件： <math.h>

函数原型： double sinh (double *x*);

参数： *x* 要为其返回双曲正弦值的值

返回值： 返回 *x* 的双曲正弦值

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for sinh           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 720.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);
}
```

输出：

```
The hyperbolic sine of -1.500000 is -2.129279

The hyperbolic sine of 0.000000 is 0.000000

Error: range error
The hyperbolic sine of 720.000000 is inf
```

sinhf

描述： 计算单精度浮点型值的双曲正弦函数值。

头文件： <math.h>

函数原型： float sinhf (float *x*);

参数： *x* 要为其返回双曲正弦值的值

返回值： 返回 *x* 的双曲正弦值

说明： 如果 *x* 的绝对值太大，将发生值域错误。

示例：

```
#include <math.h>  /* for sinhf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);

  errno = 0;
  x = 0.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n\n",
          x, y);
}
```

输出：

```
The hyperbolic sine of -1.000000 is -1.175201

The hyperbolic sine of 0.000000 is 0.000000
```

sqrt

描述： 计算双精度浮点型值的平方根。

头文件： <math.h>

函数原型： double sqrt(double *x*);

参数： *x* 非负双精度浮点型值

返回值： 返回 *x* 的非负平方根

说明： 如果 *x* 为负数，将发生定义域错误。

示例：

```
#include <math.h>  /* for sqrt           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);

  errno = 0;
  x = 9.5;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);

  errno = 0;
  x = -25.0;
  y = sqrt (x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n\n", x, y);
}
```

输出：

```
The square root of 0.000000 is 0.000000

The square root of 9.500000 is 3.082207

Error: domain error
The square root of -25.000000 is nan
```

sqrtf

描述： 计算单精度浮点型值的平方根。

头文件： <math.h>

函数原型： float sqrtf(float *x*);

参数： *x* 非负单精度浮点型值

返回值： 返回 *x* 的非负平方根

说明： 如果 *x* 为负数，将发生定义域错误。

示例：

```
#include <math.h>  /* for sqrtf          */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x;

  errno = 0;
  x = sqrtf (0.0F);
  if (errno)
    perror("Error");
  printf("The square root of 0.0F is %f\n\n", x);

  errno = 0;
  x = sqrtf (9.5F);
  if (errno)
    perror("Error");
  printf("The square root of 9.5F is %f\n\n", x);

  errno = 0;
  x = sqrtf (-25.0F);
  if (errno)
    perror("Error");
  printf("The square root of -25F is %f\n", x);
}
```

输出：

```
The square root of 0.0F is 0.000000

The square root of 9.5F is 3.082207

Error: domain error
The square root of -25F is nan
```

tan

描述： 计算双精度浮点型值的三角正切函数值。

头文件： <math.h>

函数原型： double tan (double *x*);

参数： *x* 要为其返回正切值的值

返回值： 返回 *x* 的正切值（单位为弧度）。

说明： 如果 *x* 为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for tan            */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);
}
```

输出：

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanf

描述： 计算单精度浮点型值的三角正切函数值。

头文件： <math.h>

函数原型： float tanf (float *x*);

参数： *x* 要为其返回正切值的值

返回值： 返回 *x* 的正切值（单位为弧度）。

说明： 如果 *x* 为 NaN 或无穷，将发生定义域错误。

示例：

```
#include <math.h>  /* for tanf           */
#include <stdio.h> /* for printf, perror */
#include <errno.h> /* for errno          */

int main(void)
{
  float x, y;
```

tanf（续）

```
  errno = 0;
  x = -1.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n\n", x, y);

  errno = 0;
  x = 0.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);
}
```

输出：

```
The tangent of -1.000000 is -1.557408

The tangent of 0.000000 is 0.000000
```

tanh

描述： 计算双精度浮点型值的双曲正切函数值。

头文件： <math.h>

函数原型： double tanh (double *x*);

参数： *x* 要为其返回双曲正切值的值

返回值： 返回 *x* 的双曲正切值，值的范围为 -1 到 1（包括 -1 和 1）。

说明： 不会发生定义域或值域错误。

示例：

```
#include <math.h>  /* for tanh   */
#include <stdio.h> /* for printf */

int main(void)
{
  double x, y;

  x = -1.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);

  x = 2.0;
  y = tanh (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);
}
```

输出：

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 2.000000 is 0.964028
```

tanhf

描述： 计算单精度浮点型值的双曲正切函数值。

头文件： <math.h>

函数原型： float tanhf (float *x*);

参数： *x* 要为其返回双曲正切值的值

返回值： 返回 *x* 的双曲正切值，值的范围为 -1 到 1（包括 -1 和 1）。

说明： 不会发生定义域或值域错误。

示例：

```
#include <math.h>  /* for tanhf  */
#include <stdio.h> /* for printf */

int main(void)
{
  float x, y;

  x = -1.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);

  x = 0.0F;
  y = tanhf (x);
  printf("The hyperbolic tangent of %f is %f\n\n",
          x, y);
}
```

输出：

```
The hyperbolic tangent of -1.000000 is -0.761594

The hyperbolic tangent of 0.000000 is 0.000000
```

4.18 PIC30 函数库

下列函数是标准 C 函数库辅助函数：

- _exit 终止程序执行
- brk 设置进程数据空间的末端
- close 关闭一个文件
- lseek 将文件指针移动到指定位置
- open 打开一个文件
- read 从文件中读取数据
- sbrk 通过给定的增量扩展进程的数据空间
- write 向一个文件写数据

这些函数被标准 C 函数库中的其他函数调用，且必须经过修改以适用于具体的目标应用。相应的目标模块分发到 libpic30-*omf*.a 归档文件中，源代码（适用于 MPLAB C30）包含在文件夹 src\pic30 中。

另外，一些标准 C 库函数也必须经过修改以适用于具体的目标应用。这些函数如下：

- getenv 获取环境变量的值
- remove 移除一个文件
- rename 重命名文件或目录
- system 执行一个命令
- time 获取系统时间

虽然这些函数是标准 C 函数库的组成部分，但其目标模块分发到 libpic30-*omf*.a 归档文件中，源代码 （适用于 MPLAB C30）包含在文件夹 src\pic30 中。这些模块没有分发到 libc-*omf*.a 中。

4.18.1 重建 libpic30-*omf*.a 库

默认情况下，本章所列的辅助函数编写为与 sim30 软件模拟器配合工作。头文件 simio.h 定义了函数库和软件模拟器之间的接口。因此提供的这个头文件使您可以重建函数库并继续使用软件模拟器。但由于软件模拟器不能用于嵌入式应用，因此应用不能使用这个接口。

辅助函数必须根据目标应用进行修改并重建。可通过名为 makelib.bat 的批处理文件重建 libpic30-*omf*.a 库，该批处理文件及其源代码包含在文件夹 src\pic30 中。可通过命令行窗口中执行这个批处理文件，此时要确保在 src\pic30 目录下。然后将最新编译的文件 （libpic30-*omf*.a）复制到 lib 目录中。

4.18.2 函数描述

本节描述了为使标准 C 库函数在目标环境中能正常工作，必须进行定制的函数。“默认操作”部分描述了函数分发时的功能。“描述”和 “说明”部分描述了函数一般情况下完成的功能和用法。

_exit

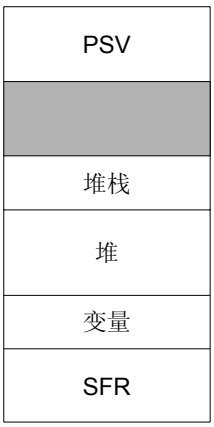
描述：	终止程序执行。
头文件：	无
函数原型：	`void _exit (int status);`
参数：	*status* 退出状态
说明：	这是一个由标准 C 库函数 `exit()` 调用的辅助函数。
默认操作：	这个函数分发时的功能是刷新 stdout 并终止程序执行。参数状态与传递到标准 C 库函数 `exit()` 的参数状态相同。
源文件：	`_exit.c`

brk

描述：	设置进程数据空间的末端。
头文件：	无
函数原型：	`int brk(void *endds)`
参数：	*endds* 指向数据段末端的指针
返回值：	如果成功返回 “0”；否则返回 “-1”。
说明：	`brk()` 用于动态改变为调用进程的数据段分配的空间的大小。通过复位进程的中断值并分配适当大小的空间来实现。中断值是数据段末端后的第一个位置的地址。分配的空间大小随着中断值的增加而增加。 新分配的空间未被初始化。 这个辅助函数由标准 C 库函数 `malloc()` 使用。

brk （续）

默认操作： 如果参数 *endds* 为 0，则函数将全局变量 __curbrk 设置为堆的起始地址，并返回 0。
如果参数 *endds* 不为 0，而且该参数的值小于堆的末地址，则函数将全局变量 __curbrk 设置为 *endds* 的值，并返回 0。
否则，全局变量 __curbrk 不变，函数返回 -1。
参数 *endds* 必须在堆的范围内 （参见下面的数据存储空间映射）。



值得注意的是，由于堆栈紧靠堆位于堆的上面，使用 brk() 或 sbrk() 对动态存储池的大小影响很小。brk() 和 sbrk() 函数主要用在堆栈向下增长而堆向上增长的运行时环境中。
如果指定了 -Wl,--heap=n 选项，则链接器为堆分配一个存储区，这里的 n 指期望的堆大小 （用字符数表示）。堆的起始和结束地址分别由变量 _heap 和 _eheap 提供。
对于 MPLAB C30，使用链接器的堆大小选项是控制堆大小的标准方式，而不是依赖 brk() 和 sbrk()。

文件： brk.c

close

描述： 关闭一个文件。

头文件： 无

函数原型： int close(int *handle*);

参数： *handle* 指向一个打开文件的句柄

返回值： 如果文件成功关闭，则返回 “0”；否则返回 “-1”，表明发生了错误。

说明： 该辅助函数由标准 C 库函数 fclose() 调用。

默认操作： 该函数分发为传递文件句柄给软件模拟器，在主文件系统中执行关闭文件的操作。

文件： close.c

getenv

描述： 获取环境变量的值。

头文件： `<stdlib.h>`

函数原型： `char *getenv(const char *s);`

参数： *s* 环境变量名

返回值： 如果成功，返回指向环境变量值的指针；否则返回空指针。

默认操作： 该函数分发为返回空指针。不支持环境变量。

文件： getenv.c

lseek

描述： 移动一个文件指针到指定的位置。

头文件： 无

函数原型： `long lseek(int handle, long offset, int origin);`

参数：

handle 指向一个打开的文件

offset 从起点到此处的字符数

origin 开始查找的位置，*origin* 可能是下列值（在 stdio.h 中定义）中的一个：
SEEK_SET – 文件的开始位置。
SEEK_CUR – 文件指针的当前位置。
SEEK_END – 文件的结束位置。

返回值： 返回从开始位置到新位置的偏移量 （字符数）。返回值 “-1L”表明发生了错误。

说明： 该辅助函数由标准 C 库函数 fgetpos()、 ftell()、 fseek()、 fsetpos 和 rewind() 调用。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统。返回值是主文件系统返回的值。

文件： lseek.c

open

描述： 打开一个文件。

头文件： 无

函数原型： `int open(const char *name, int access, int mode);`

参数：

name	要打开的文件的名字
access	打开文件的访问方法
mode	允许的访问类型

返回值： 如果成功，函数返回一个文件句柄，也就是一个小的正整数。这个句柄用于后面的低级文件 I/O 操作中。返回值 “-1”表明发生了错误。

说明： 访问标志是下列访问方法之一以及零个或多个访问限定符的联合：
0 – 表示打开一个文件来读。
1 - 表示打开一个文件来写。
2 - 表示打开一个文件来读和写。
必须支持下列访问限定符：
0x0008 - 在每个写操作之前移动文件指针到文件的末端。
0x0100 - 创建并打开一个新文件来写。
0x0200 - 打开一个文件并将其长度截为 0。
0x4000 - 以文本 （已转换）的方式打开文件。
0x8000 - 以二进制 （未转换）的方式打开文件。
模式参数可为如下之一：
0x0100 - 只允许读。
0x0080 - 可写 （隐含允许读）。
该辅助函数由标准 C 库函数 `fopen()` 和 `freopen()` 调用。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统。返回值是主文件系统返回的值。如果主文件系统返回值 “-1”，则全局变量 errno 设置为 <errno.h> 中定义的符号常量 EFOPEN 的值。

文件： open.c

read

描述： 从一个文件读取数据。

头文件： 无

函数原型：
```
int read(int handle, void * buffer,
 unsigned int len);
```

参数：
handle 指向打开文件的句柄
buffer 指向要读的数据的存储地址
len 要读的字符的最大数目

返回值： 返回读取的字符数。如果文件中的字符数小于 *len* 或者文件以文本模式打开，则字符数应该小于 *len*，此时每对回车换行符 （CR-LF）由单个回车换行符代替。只有单个回车换行符计入返回值。这种替换不影响文件指针。如果函数试图从文件末尾读，或文件被锁定，则函数返回“-1”。

说明： 该辅助函数由标准 C 库函数 `fgetc()`、 `fgets()`、 `fread()` 和 `gets()` 调用。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统，返回值是主文件系统返回的值。

文件： `read.c`

remove

描述： 移除一个文件。

头文件： `<stdio.h>`

函数原型：
```
int remove(const char *filename);
```

参数： *filename* 要移除的文件

返回值： 如果成功，函数返回“0”；否则返回“-1”。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统，返回值是主文件系统返回的值。

文件： `remove.c`

rename

描述： 重命名一个文件或目录。

头文件： `<stdio.h>`

函数原型：
```
int rename(const char *oldname, const char
 *newname);
```

参数：
oldname 指向旧文件名的指针
newname 指向新文件名的指针

返回值： 如果成功，返回“0”；发生错误时，函数返回非零值。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统，返回值是主文件系统返回的值。

文件： `rename.c`

sbrk

描述： 按照给定的增量值将进程的数据空间 （process's data space）进行扩展。

头文件： 无

函数原型： `void * sbrk(int incr);`

参数： *incr* 增加 / 减少的字符数

返回值： 函数返回新分配的空间的起始地址；如果发生错误，返回 “-1”。

说明： `sbrk()` 使中断值增加 `incr` 个字符，并相应地改变分配的空间。 `incr` 可以是负值，在这种情况下分配的空间大小将会减少。
`sbrk()` 用于动态改变为调用进程的数据段分配的空间的大小。通过复位进程的中断值并分配适当大小的空间来实现。中断值是数据段末端后的第一个位置的地址。分配的空间大小随着中断值的增加而增加。
这个辅助函数由标准 C 库函数 `malloc()` 调用。

默认操作： 如果全局变量 `__curbrk` 为 0，函数调用 `brk()` 来初始化中断值。如果 `brk()` 返回 -1，则该函数也返回 -1。
如果 *incr* 为 0，函数返回全局变量 `__curbrk` 的当前值。
如果 *incr* 为非零值，函数将检查地址（`__curbrk` + *incr*）是否小于堆的末地址。如果小于，则全局变量 `__curbrk` 更新为这个值，且函数返回 `__curbrk` 的无符号值。
否则函数返回 -1。
参见 `brk()` 的描述。

文件： `sbrk.c`

system

描述： 执行一个命令。

头文件： `<stdlib.h>`

函数原型： `int system(const char *s);`

参数： *s* 要执行的命令。

默认操作： 该函数分发为作为用户函数的桩或占位符。如果 s 不为空，则写一个错误消息到 stdout，且程序将复位；否则返回值 -1。

文件： `system.c`

time

描述： 获取系统时间。

头文件： `<time.h>`

函数原型： `time_t time(time_t *timer);`

参数： *timer* 指向时间的存储位置

返回值： 返回经过的时间 （以秒为单位）。没有错误返回。

默认操作： 该函数分发为如果 timer2 未使能，则将其使能为 32 位模式。返回 32 位 timer2 寄存器的当前值。除了在非常少的情况下，返回值不是经过的时间 （以秒为单位）。

文件： time.c

write

描述： 向一个文件写数据。

头文件： 无

函数原型： `int write(int handle, void *buffer, unsigned int count);`

参数：
handle 指向打开的文件
buffer 指向要写的数据的存储位置
count 要写的字符数

返回值： 如果成功，返回实际写的字符数；返回 “-1” 表明发生了错误。

说明： 如果磁盘上实际剩余的空间小于函数要写到磁盘的缓冲区大小，则写失败，而且也不会刷新缓冲区中的任何内容到磁盘上。如果文件以文本方式打开，则每个换行符用一个回车符代替 （输出中成对的换行符）。这种替换不影响返回值。
该辅助函数由标准 C 库函数 `fflush()` 调用。

默认操作： 该函数分发为通过软件模拟器传递参数到主文件系统，返回值是主文件系统返回的值。

文件： write.c

注：

第 5 章 MPLAB C30 内建函数

5.1 简介

本章介绍了 dsPIC 芯片专用的 MPLAB C30 内建函数。

内建函数使 C 编程人员可以访问目前只能通过行内汇编访问的汇编运算符或机器指令，这些汇编运算符或及其指令很有用，应用范围很广。内建函数的源代码用 C 语言编写，在句法上类似于函数调用，但被编译成直接实现函数的汇编代码，且不涉及函数调用或库函数。

提供内建函数比要求编程人员使用行内汇编更可取，这是有很多原因的。这些原因包括：

1. 提供专用的内建函数可以简化编码。
2. 使用行内汇编时会禁止某些优化功能。而使用内建函数则不会。
3. 对于使用专用寄存器的机器指令来说，编写行内汇编代码时要特别注意避免寄存器分配错误。内建函数使这个过程更简单，无需考虑每个机器指令的特殊寄存器要求。

本章结构如下：

- 内建函数列表
- 内建函数错误消息

5.2 内建函数列表

本节介绍了 MPLAB C30 C 编译器内建函数的编程接口。由于函数是 “内建”的，所以没有与之相关的头文件。同样，也没有与内建函数相关的命令行开关——它们总是可用的。内建函数的名称按照所属编译器的名字空间进行选择 （它们都有一个前缀 __builtin_），因此它们不会与编程名字空间中的函数或变量名冲突。

__builtin_divsd

描述： 该函数计算 *num* / *den* 的商。 如果 *den* 为 0，则出现数学错误异常。函数参数是有符号的，函数的结果也是有符号的。命令行选项 -Wconversions 可用来检测意外的符号转换。

函数原型：
```
int __builtin_divsd(const long num, const int den);
```

参数：
num 分子
den 分母

返回值： 返回商 *num* / *den* 的有符号整型值。

汇编运算符 / 机器指令：
```
div.sd
```

__builtin_divud

描述： 该函数计算 *num* / *den* 的商。 如果 *den* 为 0，则出现数学错误异常。函数参数是无符号的，函数的结果也是无符号的。命令行选项 -Wconversions 可用来检测意外的符号转换。

函数原型：
```
unsigned int __builtin_divud(const unsigned
  long num, const unsigned int den);
```

参数：
num 分子
den 分母

返回值： 返回商 *num* / *den* 的无符号整型值。

汇编运算符 / 机器指令：
```
div.ud
```

__builtin_mulss

描述： 该函数计算乘积 *p0* x *p1*。函数参数是有符号整型，函数的结果是有符号长整型。命令行选项 -Wconversions 可用来检测意外的符号转换。

函数原型：
```
signed long __builtin_mulss(const signed int p0,
const signed int p1);
```

参数：
p0 被乘数
p1 乘数

返回值： 返回乘积 *p0* x *p1* 的有符号长整型值。

汇编运算符 / 机器指令：
```
mul.ss
```

__builtin_mulsu

描述： 该函数计算乘积 *p0* x *p1* 。函数参数是混合符号整型，函数的结果是有符号长整型。命令行选项 -Wconversions 可用来检测意外的符号转换。该函数支持全部指令寻址模式，包括对操作数 *p1* 的立即寻址模式。

函数原型：
```
signed long __builtin_mulsu(const signed int p0,
const unsigned int p1);
```

参数：
p0 被乘数
p1 乘数

返回值： 返回乘积 *p0* x *p1* 的有符号长整型值。

汇编运算符 / 机器指令：
```
mul.su
```

__builtin_mulus

描述： 该函数计算乘积 *p0* x *p1* 。函数参数是混合符号整型，函数的结果是有符号长整型。命令行选项 -Wconversions 可用来检测意外的符号转换。该函数支持全部指令寻址模式。

函数原型：
```
signed long __builtin_mulus(const unsigned int p0,
const signed int p1);
```

参数：
p0 被乘数
p1 乘数

返回值： 返回乘积 *p0* x *p1* 的有符号长整型值。

汇编运算符 / 机器指令：
```
mul.us
```

__builtin_muluu

描述： 该函数计算乘积 *p0* x *p1* 。函数参数是无符号整型，函数的结果是无符号长整型。命令行选项 -Wconversions 可用来检测意外的符号转换。该函数支持全部指令寻址模式，包括对操作数 *p1* 的立即寻址模式。

函数原型：
```
unsigned long __builtin_muluu(const unsigned int p0,
const unsigned int p1);
```

参数：
p0 被乘数
p1 乘数

返回值： 返回乘积 *p0* x *p1* 的有符号长整型值。

汇编运算符 / 机器指令：
```
mul.uu
```

__builtin_tblpage

描述： 该函数返回地址作为参数给定的对象的表页码。参数 *p* 必须是 EE 数据空间、PSV 或可执行存储空间中的对象的地址；否则，会产生错误消息并导致编译失败。可参阅 *《MPLAB® C30 C 编译器用户指南》* 中的 space 属性。

函数原型：
```
unsigned int __builtin_tblpage(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给定的对象的表页码。

汇编运算符 / 机器指令：
```
tblpage
```

__builtin_tbloffset

描述： 该函数返回地址作为参数给定的对象的表页码偏移量。参数 *p* 必须是 EE 数据空间、PSV 或可执行存储空间中的对象的地址；否则，会产生错误消息并导致编译失败。可参阅 *《MPLAB® C30 C 编译器用户指南》* 中的 space 属性。

函数原型：
```
unsigned int __builtin_tbloffset(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给定的对象的表页码偏移量。

汇编运算符 / 机器指令：
```
tbloffset
```

__builtin_psvpage

描述： 该函数返回地址作为参数给定的对象的 PSV 页码。参数 *p* 必须是 EE 数据空间、PSV 或可执行存储空间中的对象的地址；否则，会产生错误消息并导致编译失败。可参阅 *《MPLAB® C30 C 编译器用户指南》* 中的 space 属性。

函数原型：
```
unsigned int __builtin_psvpage(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给定的对象的 PSV 页码。

汇编运算符 / 机器指令：
```
psvpage
```

__builtin_psvoffset

描述：	该函数返回地址作为参数给定的对象的 PSV 页码偏移量。参数 *p* 必须是 EE 数据空间、 PSV 或可执行存储空间中的对象的地址；否则，会产生错误消息并导致编译失败。可参阅 *《MPLAB® C30 C 编译器用户指南》* 中的 space 属性。
函数原型：	`unsigned int __builtin_psvoffset(const void *p);`
参数：	*p* 对象地址
返回值：	返回地址作为参数给定的对象的 PSV 页码偏移量。
汇编运算符 / 机器指令：	`psvoffset`

__builtin_return_address

描述：	该函数返回当前函数或它的一个调用函数的返回地址。对于参数 *level* ，值 0 产生当前函数的返回地址，值 1 产生当前函数的调用函数的返回地址，等等。当 *level* 超过当前的堆栈深度时，返回 0。调试时，这个函数必须带有一个非 0 的参数。
函数原型：	`int __builtin_return_address (const int level);`
参数：	*level* 扫描调用堆栈的帧数。
返回值：	返回当前函数或它的一个调用函数的返回地址。
汇编运算符 / 机器指令：	`return_address`

5.3 内建函数错误消息

当对内建函数的使用不正确时会产生以下错误消息。

Argument to *__builtin_function*() is not the address of an object in code, psv, or eedata section

以下内建函数的参数必须是一个明确的对象地址：

- `__builtin_tblpage`
- `__builtin_tbloffset`
- `__builtin_psvpage`
- `__builtin_psvoffset`

例如，如果 `obj` 是一个可执行或只读段中的对象，下面的语法就是有效的：

```
unsigned page = __builtin_function(&obj);
```

注：

附录 A ASCII 字符集

表 A-1： ASCII 字符集

最低有效字符 \ 最高有效字符	0	1	2	3	4	5	6	7
Hex								
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

注：

索引

符号

数字

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

W

X

Y

Z