MSP430 系列 C 编译器编程指南

第1章 安装和文档关系图

本章说明如何安装和运行 IAR 产品的命令行和 Windwos Workbench 版本，并给出与产品一起提供的用户指南的概述。

请注意某此产品只有命令行版本，且根据用户正在使用的产品或平台，资料可能稍有不同。

1.1 命令行版本

本节叙述怎样安装和运得 IAR 系统工具的命令版本。

1.1.1 用户需要什么

- DOS 4.X 或更高版本。此产品也和运行在 Windows95, Windows NT 3.51 或更高版本，或 Windows 3.1 X 下的 DOS 窗口相兼容。
- 至少 10M 字节的自由磁盘空间。
- 最少有 4M 字节的 RAM 可供 IAR 应用程序使用。

1.1.2 安装

1、插入第一张安装盘。

2、在 MS-DOS 提示符下健入： a:\install（回车）。

3、按屏幕上的指示操作。

当安装完成时：

4、在用户的 auuroexec.bat 文件中作下列修改：把至可执行的 IAR 系统的路径和用户接口文件加入 PATH 变量;例如:PATH=C:\dos;C:\iar\exe;C:\air\ui; 定义环境变量 C-INCLUDE 和 XLINK-DFLTDIR；它指定到 inc 和 lib 目录的路径，例如：

```
Set C-INCLUDE=C:\iar\inc
Set XLINK-DFLTDIR =C:\iar\lnb\
```

5、为了使修改有效，重新启动用户的计算机。

6、要得到任何没有包括在本指南中的资料，请阅读名为 product.doc 的 Read-Me 文件。

1.1.3 运行工具

在 MS-DOS 提示符下键入适当的命令。

要获取更多的信息，请参见《Command Line Interface Guide(命令行接口指南)》中“Getting Started (开始)”一章。

1.2 Windows Workbench 版本

本节说明怎样安装和运行嵌入式 Workbench。

1.2.1 用户需要什么

- Windows95，Windows NT 3.51 或更高版本，或 Windows 3.1 X。
- 多达 15M 字节的自由磁盘空间，用于嵌入式 Workbench。
- 最少 4M 字节的 RAM，用于 IAR 应用程序。

如果用户正在使用 C-SPY，那么用户应当在 C-SPY 之前安装 Workbench。

1.2.2 从 Windows 95 或 NT4.0 安装

1、插入第 1 张安装盘。
2、单击任务栏中 Start(开始)按钮，然后单击 Settings（设置）和 Control Panel（控制面板）。
3、双击 Control Panel（控制面板）中 Add/Remove Programs（增加/删除程序）图标。
4、单击 Install（安装），然后按照屏幕上的指示去做。

1.2.3 从 Windows 95 或 NT4.0 运行

1、单击任务栏中 Start(开始)按钮，然后单击 Programs（程序）和 IAR Embedded Workbench。
2、单击 IAR Embedded Workbench。

1.2.4 从 Windows 3.1X 或 NT3.51 安装

1、插入第 1 张安装盘。
2、双击 M ain 程序组中 File Manager（文件管理器）图标。
3、在 File Manager（文件管理器）工具栏中单击磁盘图标。
4、双击 Setup.exe 图标，然后按照屏幕上的指示操作。

1.2.5 从 Windows 3.1X 或 NT3.51 运行

进入程序管理器（Program Manager）并双击 IAR Embedded Workbench 图标。

1.2.6 运行 C-SPY

可以用下列两种方法之一：

以与启动 Embedded Workbench 相同的方法启动 C-SPY（参见上面所述）。或从 Embedded Workbench 的 Project（工程）菜单中选择 Debugger（调试器）。

1.3 UNIX 版本

本节叙述怎样安装和运行 IAR 系统工具的 UNIX 版本。

1.3.1 用户需要什么

装有 HP-UX 9.X（最小）的 HP9000/700 工作站，或装有 SunOS 4.X（最小）或 Solaris 2.X（最小）的 Sun4/SPARC 工作站。

1.3.2　安装

按照用媒体（media）提供的指示操作。

1.3.3　运行工具

在 UNIX 提示符下键入合适的命令。详情请参见《Command Line Interface Guide（命令行接口指南)》中“Getting Started（开始)”一章。

1.4　文档关系图

文档关系图如下页所示。

1.4.1　Windows Workbench 版本的文档

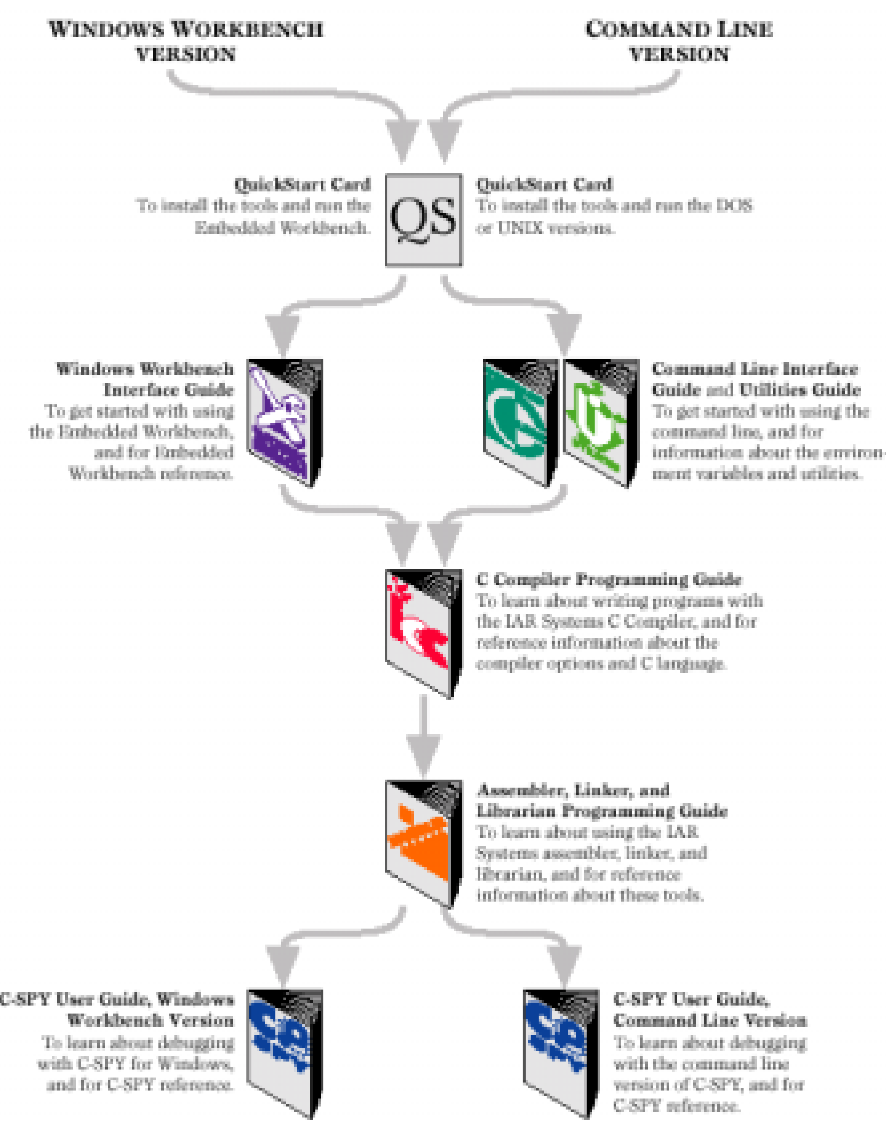
1、Quickstart Card（快速启动卡）
安装工具和运行 DOS 或 UNIX 版本
2、Windows Workbench Interface Guide（Windows Workbench 接口指南）
用嵌入式 Workbench 启动及有关嵌入式 Workbench 的参考资料。
3、C-SPY User Guide，Windows Workbench Versim(C-SPY 用户指南，Workbench 版本)学习用 C-SPY for Windows 进行调试及有关 C-SPY 的参考资料。

1.4.2　命令行版本的文档

1、 Quickstart Card（快速启动卡）
安装工具和运行 DOS 或 UNIX 版本
2、 Command Line Interface Guide（命令接口指南）和 Utilites Guide（实用程序指南）
用命令行启动及有关环境变量和实用程序的资料。
3、C-SPY User Guide，Command Line Versim(C-SPY 用户指南，命令行版本)学习用 C-SPY 的命令进行调试及有关 C-SPY 的参考资料。

1.4.3　两种版本共用的文档

1、C Compiler Programming Guide（C 编译器编程指南)。
学习使用 IAR 系统 C 编译器写程序及有关编译选项和 C 语言的参考资料。
3、 Assembler, Linker, and Librarian Programming Guide（汇编器、连接器和库编程指南)。
学习使用 IAR 系统汇编器、连接器和库，以及有关这些工具的参考资料。



文档关系图

第2章 引　言

IAR 系统 MSP430 编译器有两种版本可供使用：命令行版本以及与 IAR 系统嵌入式 Workbench 开发环境集成在一起的 Windows 版本。

本指南叙述 C 编译器的两个版本，提供从嵌入式 Workbench 或从命令行运行该编译器的有关资料。

2.1 C 编译器

运用于 MSP430 微处理器的 IAR 系统 C 编译器提供 C 语言的标准特性，再加上许多为利用 MSP430

专用工具而设计的扩展功能。编译器与 MSP430 IAR 系统汇编器一起提供，与它集成在一起，共享连接器和库管理工具。

它提供以下特性：

2.1.1 语言工具

- 与 ANSI 规格一致
- 可应用于嵌入式系统的标准函数库，具有可选用的源（代码）。
- IEEE 兼容的浮点算法。
- 对 MSP430 特殊性能的有力扩展，包括高效的 I/O。
- 用户代码与汇编子程序连接。
- 长识别符——多达 255 个有效字符。
- 多达 32000 个外部符号。

2.1.2 性能

- 快速编译。
- 避免暂时文件或覆盖的（overlays）基本存储器的设计。
- 编译时严格的模块接口类型检查。
- 程序源的 LINT-like 检查。

2.1.3 代码产生

- 可选择的代码速度或大小的最佳化。
- 综合输出选项，包括可重定位二进制，ASM，ASM+C，XREF，等等。
- 易于理解的出错和警告消息。
- 与 C-SPY 高级调试器兼容。

2.1.4 目标支持

- 灵活的变量分配。
- 不需要汇编语言的中断函数。
- 使用处理器专用扩展时保持可移植性的#Pragma 伪指令。

第3章 指 导

本章说明怎样使用 MSP430C 编译器来开发一系列典型的程序并说明编译器一些最重要的特性。

在阅读本章之前用户应当：

- 已经安装了 C 编译器软件；见 Quickstart Card（快速启动卡）或“安装和文档关系图”一章。
- 熟悉 MSP430 处理器的结构和指令集。

详情见工厂的数据手册。

如果合适的话，建议用户完成《MSP430 Windows Workbench Interface Guide(MSP430 Windows Workbench 接口指南)》或《MSP430 Command Line Interface Guide（MSP430 命令行接口指南)》中的引导性指导，以便让用户熟悉用户正在使用的接口。

指导文件摘要

下表归纳了本章中使用的指导文件：

文件	文档是什么
Tutor1	编译和运行简单的 C 程序
Tutor2	使用 I/O
Tutor3	中断处理

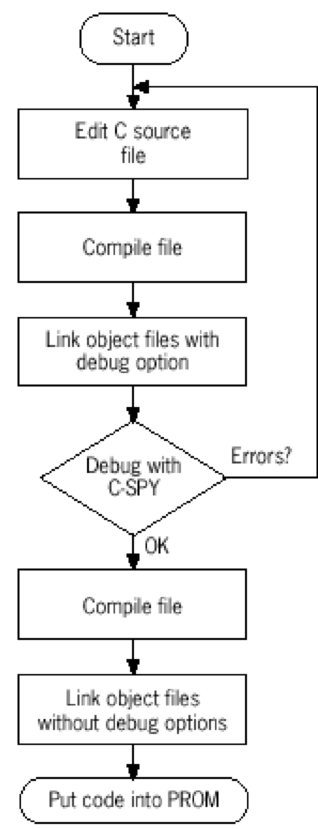
3.1 运行示例程序

本指导说明怎样使用可选的 C-SPY 模拟器运行示例程序。

用户也可以用 EPROM 仿真器和调试器在目标系统上运行示例。在此情况下，用户首先要配置 I/O 子程序。此外，用户还可以通过察看所创建的列表文件跟随本指导进行操作。.lst 和.map 文件表示监视哪一个存储器区域。

3.2 典型开发周期

开发通常遵循下图所示的流程：



下面的指导材料按照此流程进叙述。

3.3 开始

用 C 编译器开发工程（项目）的第一步是决定适合于用户的目标系统的合适的配置。

3.3.1 适合于目标系统的配置

每一个工程需要包含目标系统存储器映象细节的 XLINK 命令文件。

选择连接器命令文件

在 icc430 子目录中提供了合适的连接器命令文件 lnk430.xcl。

用合适的文本编辑器，例如嵌入式 Workbench 编辑器或 MS-DOS edit 编缉器察看 ink 430.xcl。

文件首先包含下列 XLINK 命令以便把 CPU 类型定义为 MSP430：

-cmsp 430。

然后，它包含一系列-Z 命令以定义编译器使用的段。关健的段如下：

段类型	段名	地址范围
DATA（数据）	IDATA0，UDATA0 ECSTR，CSTACK	0×0200 至 0×7FFF
CODE（数据）	RCODE，CODE， CDATA0，CONST， CSTR，CCSTR	0×8000 至 0×FFDF
CODE（数据）	INTVEC	0×FFE0 至 0×FFFF

更为详细的资料请参见“段参考”一章。

文件定义了用于 printf 和 scanf 的子程序。最后，它包含了下列行以装载合适的 C 库：cl 430

有关所提供的不同的 C 库的详细情况，请参见“运行时（Run-time）库”。

注意，这些定义并非永久性的：如果原先的选择被证明为不正确的，或不是最佳的那么以后可以改变它们以适合于用户的工程（项目）。

有关适合于目标存储器配置的详细资料，请参见“存储器映象”。有关选择堆栈大小的详细资料请参见“堆栈大小”。

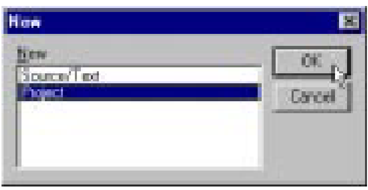
3.3.2 创建一个新的工程

第一步是创建适用于指导程序的新工程。

1、使用嵌入式 Workbench 创建新工程。

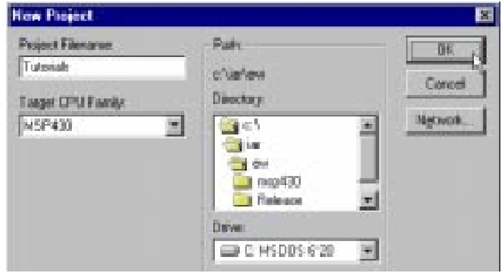
首先，运行嵌入式 Workbench，并如下所述为指导创建一个工程。

从 File（文件）菜单选择 New（新）以显示下列对话框：



选择 Prroject（工程）并选择 OK，显示 New Project（新工程）对话框。

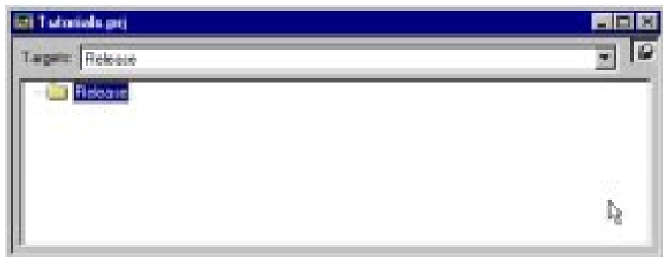
在 Prroject Filename（文件名）框中输入 Tutorials，并把 Target CPU Family（目标 CPU 类型）设置为 MSP430：



然后选择 OK 以创建新工程。

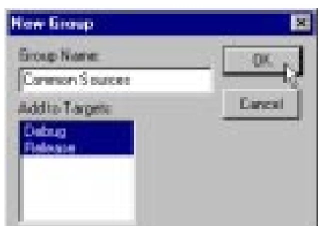
屏幕上将显示工程窗口。如果有必要的话，从 Targets（目标）下拉列表中选择 Release 以

便显示 Release 目标：



接着，如下所述创建包括指导源文件的组。

从 Prroject（工程）菜单中选择 New Group...（新组...）并输入名字 Common Sources。缺省情况下两个目标均被选择，因此将把组加至两个目标：



选择 OK 以创建组。它将被显示在工程窗口。

2、 利用命令行创建新的工程

最好把特定工程的所有文件保存在一个目录中，使它与其他工程和系统文件分开。

指导文件被安装在 icc430 目录下。通过输入下述命令来选择此目录：

cd c: \iar\icc430（回车）

在本指导中用户将工作此目标下，因此用户创建的文件将驻留在其中。

3.4 创建程序

第一个指导示例说明怎样编译、连接和运行程序。

3.4.1 键入程序

第一个程序是仅使用标准 C 工具的简单程序。它重复地调用使变量增量的函数：

```
#include <stdio.h>
int call_count;
unsigned char my_char;
const char con_char='a';

void do_foreground_process(void)
{
  call_count++;
  putchar(my_char);
}

void main(void)
{
  int my_int=0;
  call_count=0;
  my_char=con_char;
  while (my_int<100)
  {
    do_foreground_process();
    my_int++;
  }
}
```

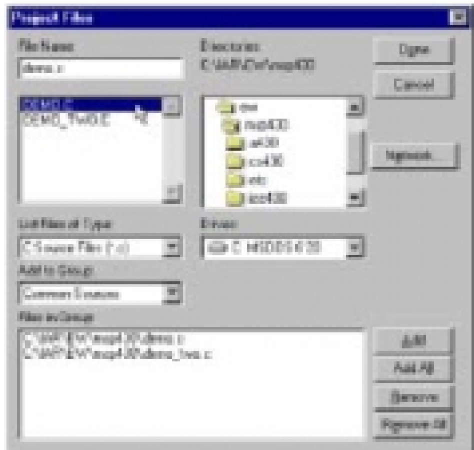
1、 }

从 File（文件）菜单中选择 New（新）以显示 New（新）对话框。
选择 Source/Text（源/文本）并选 OK 以打开新的文本文档。
键入上面给出的程序并把它保存在文件 tutor1.c 中。
另外，在 C 编译器文件目录中提供程序的副本。

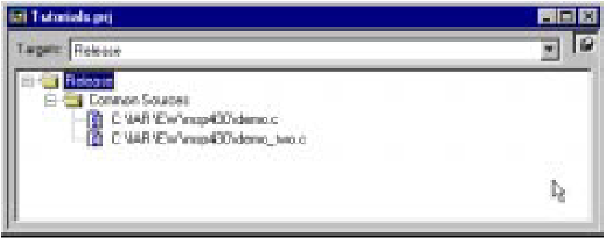
2、 用命令行写程序
用任何标准的文本编辑器，例如，MS-DOS edit 编辑器键入程序并把它保存在名为 tutor1.c 的文件中。另处，在 C 编译器文件目录中提供副本。
现在用户有了已准备好供编译的源文件。

3.4.2 编译程序

1、利用嵌入式 Workbench 编译程序
为了编译程序，首先要如下所示把它加入 Tutorials 工程中
从 Project（工程）菜单中选择 Files...（文件...）以显示 Project Files（工程文件）对话框。在对话框上半部的文件选择列表中找到文件 tutor1.c 并选择 Add(加)把它加到 Common Sources（通用源）组中：



然后单击 Done 以关闭 Project Files（工程文件）对话框。
单击+符号以显示工程窗口树中的文件：

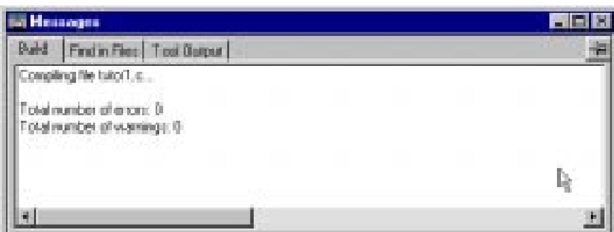


然后如下所示为工程设置编译器选项：在工程窗口中选择 Release 文件夹图标，从 Project（工程）菜单中选择 Options...（选项...），并在 Category 列表中选择 ICC430 以便显示 C 编译器选项页：

确保在 Options（选项）对话框的适当页中选择了下列选项：

页	选项
Code generation（代码产生）	Enable Language extensions（使能语言扩展）
Debug（调试）	Generate debug information（产生调试信息）
List（列表）	List file（列表文件）
	Insert mnemonics（插入助记符号）

当用户已作了这些修改时，选择 OK，以便设置用户已指定的选项。
为了编译文件，工程窗口内选择它并从 Project（工程）菜单中选择 Compile（编译）。
过程将显示在 Messages（消息）窗口中：



在文件 tutor1.lst 中创建了列表。通过从 File（文件）菜单中选择 Open…（打开），并从 release\list 目录中选择 tutor1.lst，可打开此文件。

2、从命令行编译程序

要编译程序，键入命令：

icc 430-r-l-q tutor1-l\iar\inc（回车）

有几个编译选项应用于此：

选项	说明
-r	允许用 C-SPY 调试代码
-L	创建列表文件
-q	在列表中与 C 一起包含汇编器代码
-i	指定包含文件的路径名

这将创建名为 tutor1.r43 的目标模块和名为 tutor1.lst 的列表文件。

3、察看列表

察看所产生的列表文件且观察怎样把变量分配到不同的段。

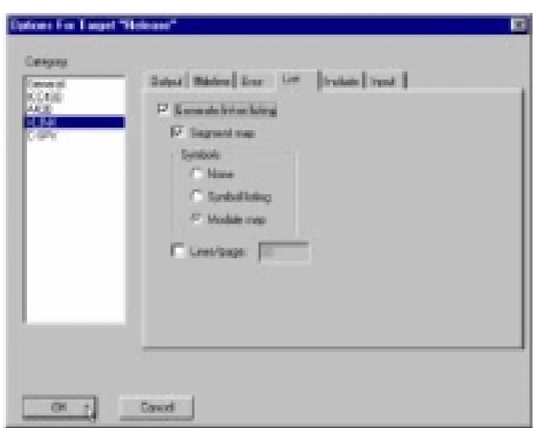
```
##########################################################################
#                                                                        #
# IAR MSP430 C-Compiler Vx.xx                                            #
# Front End Vx.xx                                                        #
# Global Optimizer Vx.xx                                                 #
#                                                                        #
#       Source file   =  tutor1.c                                        #
#       List file     =  tutor1.lst                                      #
#       Object file   =  tutor1.r43                                      #
#       Command line  =  -r -L -q tutor1 -I\iar\inc                      #
#                                                                        #
#                                     (c) Copyright IAR Systems 1996     #
##########################################################################

   \   0000                    NAME    tutor1(16)
   \   0000                    RSEG    CODE(1)
   \   0000                    RSEG    CONST(1)
   \   0000                    RSEG    UDATA0(1)
   \   0000                    PUBLIC  call_count
   \   0000                    PUBLIC  con_char
   \   0000                    PUBLIC  do_foreground_process
   \   0000                    PUBLIC  main
   \   0000                    PUBLIC  my_char
   \   0000                    EXTERN  putchar
   \   0000                    EXTERN  ?CL430_1_00_L08
   \   0000                    RSEG    CODE
   \   0000            do_foreground_process:
      1          #include <stdio.h>
      2          int call_count;
      3          unsigned char my_char;
      4          const char con_char='a';
      5
      6          void do_foreground_process(void)
      7          {
      8            call_count++;
   \   0000  92530000          ADD     #1,&call_count
      9            putchar(my_char);
   \   0004  5C420200          MOV.B   &my_char,R12
   \   0008  7CF3              AND.B   #-1,R12
   \   000A  B0120000          CALL    #putchar
     10          }
   \   000E  3041              RET
   \   0010            main:
     11
     12          void main(void)
     13          {
```

3.4.3 连接程序

1、利用嵌入式 Workbench 连接程序。

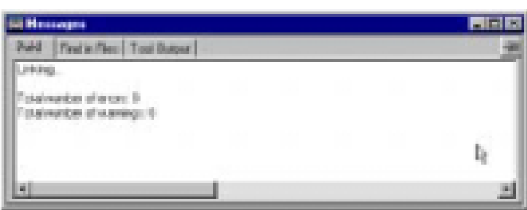
首先设置用于 XLINK 连接器的先项。在工程窗口中选择 Release 文件夹图标，从 Project（工程）菜单中选择 Options...(选项...)并在 Category 列表中选择 XLINK，以显示 XLINK 选项页。

然后单击 List（列表）以显示列表选项页。选择 Generate linker listing（产生连接器列表）和 Segment map（段映射）以产生映象文件 tutor1.map。



然后选择 OK 以便保存 XLINK 选项。

为了连接目标文件以产生可以被调试的代码，从 Project（工程）菜单中选择 Link（连接）。过程将显示在 Massage（消息）窗口中：



连接的结果是（产生）代码文件 tutorial.dbg 和映象文件 tutorial.map。

2、从命令行连接程序

为了连接目标文件和适当的库模块以产生可以被 C-SPY 调试器执行的代码，键入命令为

xlink tutor1-f lnk430-rt-x-l tutor1.map（回车）

-f 选项指定用户的 XLINK 命令文件 lnk430

-r 选项允许用 C-SPY 调试代码

-x 创建映象文件，-l filenme 给出文件名。

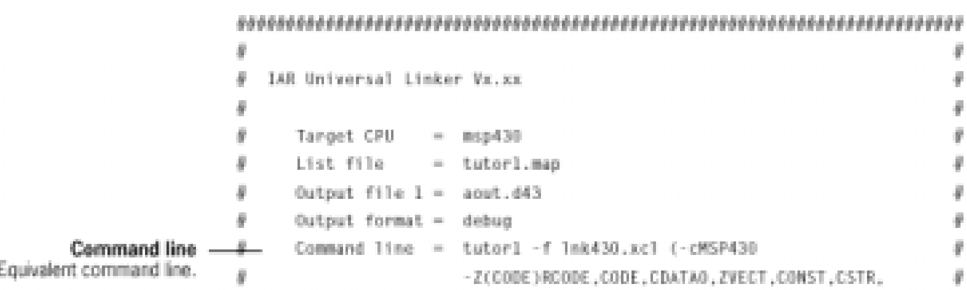
连接的结果是产生名为 aout.a43 的代码文件和名为 tutor1.map 的映象文件。

3、察看映象文件

察看映象文件以观察段定义和代码怎样被置于其实际地址。映象文件的主要点显示在下表中：

一

利:



```
################################################################
#                                                              #
#  IAR Universal Linker Vx.xx                                  #
#                                                              #
#     Target CPU    =  msp430                                  #
#     List file     =  tutor1.map                              #
#     Output file 1 =  aout.d43                                #
#     Output format =  debug                                   #
#     Command line  =  tutor1 -f lnk430.xcl (-cMSP430          #
#                      -Z(CODE)RCODE,CODE,CDATA0,ZVECT,CONST,CSTR, #
```

```
          calls direct
        LOCALS                  ADDRESS
        ?0001                    801E
        ?0000                    802C
   ------------------------------------
CONST
  Relative segment, address : 80D2 - 80D2
        ENTRIES                 ADDRESS           REF BY MODULE
        con_char                 80D2             Not referred to
   ------------------------------------
```

```
-------------------------------------
LIBRARY MODULE, NAME : 108
        ABSOLUTE ENTRIES        ADDRESS         REF BY MODULE
         ==============         =======         =============
         ?CL430_1_00_L08          0001          tutor1
                                                memcpy
                                                memset
                                                putchar
```

利;

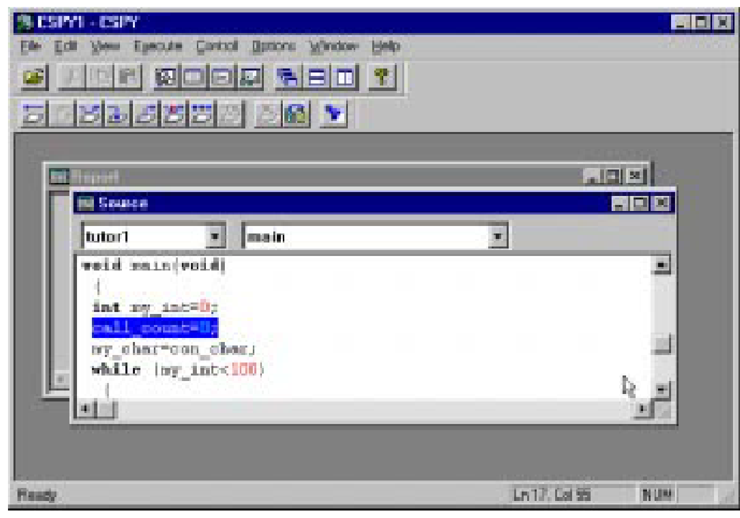
```
CODE
  Relative segment, address : 80A8 - 80CB
          ENTRIES               ADDRESS          REF BY MODULE
          putchar                80AE            tutor1
              calls direct
          LOCALS                ADDRESS
          __low_level_put        80A8

    ------------------------------------
```

注意，虽然连接文件规定了所有段的地址，但是许多段没有用。有关段的最重要的信息在末尾，在哪里给出了它们的地址和范围。
一些没有出现在原 C 代码中的入口点已被说明。? C-EXIT 入口来自 CSTARTUP 模块 putchar 入口来自库文件。

3.4.4　运行程序

1、 使用嵌入式 Workbench 运行程序

为了使用 C-SPY 调试器运行程序，从 Project（工程）菜单中选择 Debugger（调试器）。屏幕上将显示 C-SPY 窗口。从 Execute 菜单中选择 Step（单步），或在工具栏上单击 Step（单步）按钮，在源（代码）窗口中显示源（代码）：



现在使用监视（Watch）窗口监视 Call-count 的值。从 Windows（窗口）菜单中选择 Watch（监视），并单击监视窗口工具栏中 Watch（监视）按钮：

然后键入 Call-count（回车键），把此变量加入监视窗口。

从 Execute（执行）菜单中选择 Step（单步）以单步执行程序至到达 do-foreground-process()；这一行，并在监视窗口中检查变量 call-count 的值。由于变量已被初始化但沿未增量，所以其值应当为 0。执行当前行并移至循环中的下一行。再次检查 call-count-它应当显示 1，表示变量已由 do-foreground-process 增量。

2、 从命令行运行程序

执行下列命令：

cs430 aout（回车键）

这将装载模拟器并装载程序。

键入 STEP 或按 F2 功能键以显示程序和执行第 1 条命令。

然后通过键入：

call-count（回车键）

显示 call-count 的值。

这将返回答案 0。

然后通过键入 STEP（单步）或按 F2 功能键，继续执行程序行，直到 my-int++这一行被高亮显示为止。

再次检查 call-count 的值，现在它应当是 1。

如果用户也希望模拟子程序 foreground-process，用命令 ISTEP，或按 F3 功能键代替 F2 功能键。

3.5　使用 I/O

现在我们将创建一个程序，它使用处理器的 I/O 端口。所产生的代码将把 LCD 驱动器设置到

4MUX 模式，然后输出 7 数字。此代码说明使用#pragma 伪指令和头部文件。

以下是代码的列表。把它键入到合适的文件编辑器并保存为 tutor2.c。另外在 icc430 子目录中提供副本：

```
/* This example demonstrates how to display digits on the
LCD 4 MUX method */

/* enable use of extended keywords */
#pragma language=extended

/* include sfrb/sfrw definitions for I/O registers */
#include "io310.h"

char digit[10] = {
0xB7,  /* "0" LCD segments a+b+c+d+e+f */
0x12,  /* "1" */
0x8F,  /* "2" */
0x1F,  /* "3" */
0x3A,  /* "4" */
0x3D,  /* "5" */
0xBD,  /* "6" */
0x13,  /* "7" */
0xBF,  /* "8" */
0x3F   /* "9" */
};

void main(void)
{
   int i;

  /* Initialize LCD driver (4Mux mode) */
  LCDCTL = 0xFF;

  /* display "6543210" */
  for (i=0; i<7; ++i)
  LCDMEM[i] = digit[i];
}
```

程序的第 1 行是：

/*enable use of extended keywords */

pragma language = extended

缺省情况下，扩展关键字是不能使用的，所以在试用使用任何一个之前，必须包含此伪指令。# pragma 伪指令在“#pragma 伪指令参考”一章中叙述。

代码下面的行是：

/* include sfrb/sfrw definitions for I/O registers * /

#include “i0310.h”

文件 i0310.h 包含 310 处理器所有 I/O 寄存器的定义。

3.5.1 编译和连接程序

1、利用嵌入式 Workbench 编译和连接程序

从 Project（工程）菜单中选择 Files...（文件...），使用 Project Files（工程文件）对话框，从 Tutorials 工程中移走原 tutor1.c 文件并加入 tutor2.c 代替之。然后从 Project（工程）菜单中选择 Make（生成）。

2、从命令行编译和连接程序

如下所示，用标准的连接文件编译和连接程序：

icc430　　tutor2　-r　-L　-q　（回车键）

xlink　　tutor2　-r　lnk 430.xcl　-r　（回车键）

3.5.2　运行程序

利用功能键 F2 或通过键入 STEP 来单步执行程序。

在真实的目标上可能联系到 LCD 显示并监视其变化。使用 C-SPY 只能监视代码的执行。

3.6　加入中断处理程序

现在我们通过加入中断处理程序来修改原先的程序。MSP430C 编译器允许用户通过使用关键字 interrupt 直接用 C 书写中断处理程序。我们将处理的中断是定时器中断。此程序每秒一次设置定时器中断并把一个接着一个的数字送到 LCD。

下面是中断代码的列表。在样本指导文件 tutor3.C 提供了该代码。

```
/* This example demonstrates how to use the basic timer
Interrupt frequency 1 Hz */

/* enable use of extended keywords */
#pragma language=extended

/* include sfr definitions for I/O registers and
intrinsic functions (_EINT) */
#include "io310.h"
#include "in430.h"

volatile int clock;  /* count number of basic timer
                        interrupts */

char digit[10] = {
0xB7,  /* "0" LCD segments a+b+c+d+e+f */
0x12,  /* "1" */
0x8F,  /* "2" */
0x1F,  /* "3" */
0x3A,  /* "4" */
0x3D,  /* "5" */
0xBD,  /* "6" */
0x13,  /* "7" */
0xBF,  /* "8" */
0x3F   /* "9" */
};

/* Basic Timer has vector address 0xFFE2, ie offset 2 ir
INTVECT */

interrupt [0x02] void basic_timer(void)
{
  if (++clock == 10)
    clock = 0;
  /* Display 1,2,3,...,9,0,1,2,... */
  LCDMEM[0] = digit[clock];
}
void main(void)
```

为了定义-EINT 函数，必须提供内在（intrinsic）包含文件；为了定义 MSP430 I/O 寄存器必须提供 I/O 包含文件：

```
/* enable use of extended keywords */
#pragma language=extended

/* include sfr definitions for I/O registers and
intrinsic functions (_EINT) */
#include "io310.h"
#include "in430.h"
```

中断函数本身由下列各行定义：

```
interrupt [0x02] void basic_timer(void)
{
  if (++clock == 10)
    clock = 0;
  /* Display 1,2,3,....,9,0,1,2,... */
  LCDMEM[0] = digit[clock];
}
```

关键字 interrupt 在“扩展关键字参考”一章中叙述。

3.6.1 编译和连接程序

1、利用嵌入式 Workbench 编译和连接程序如前所述，通过把程序加入 Tutorials 工程并从 Project(工程)菜单中选择 Make（生成）来编译和连接程序。

2、从命令行编译和连接程序

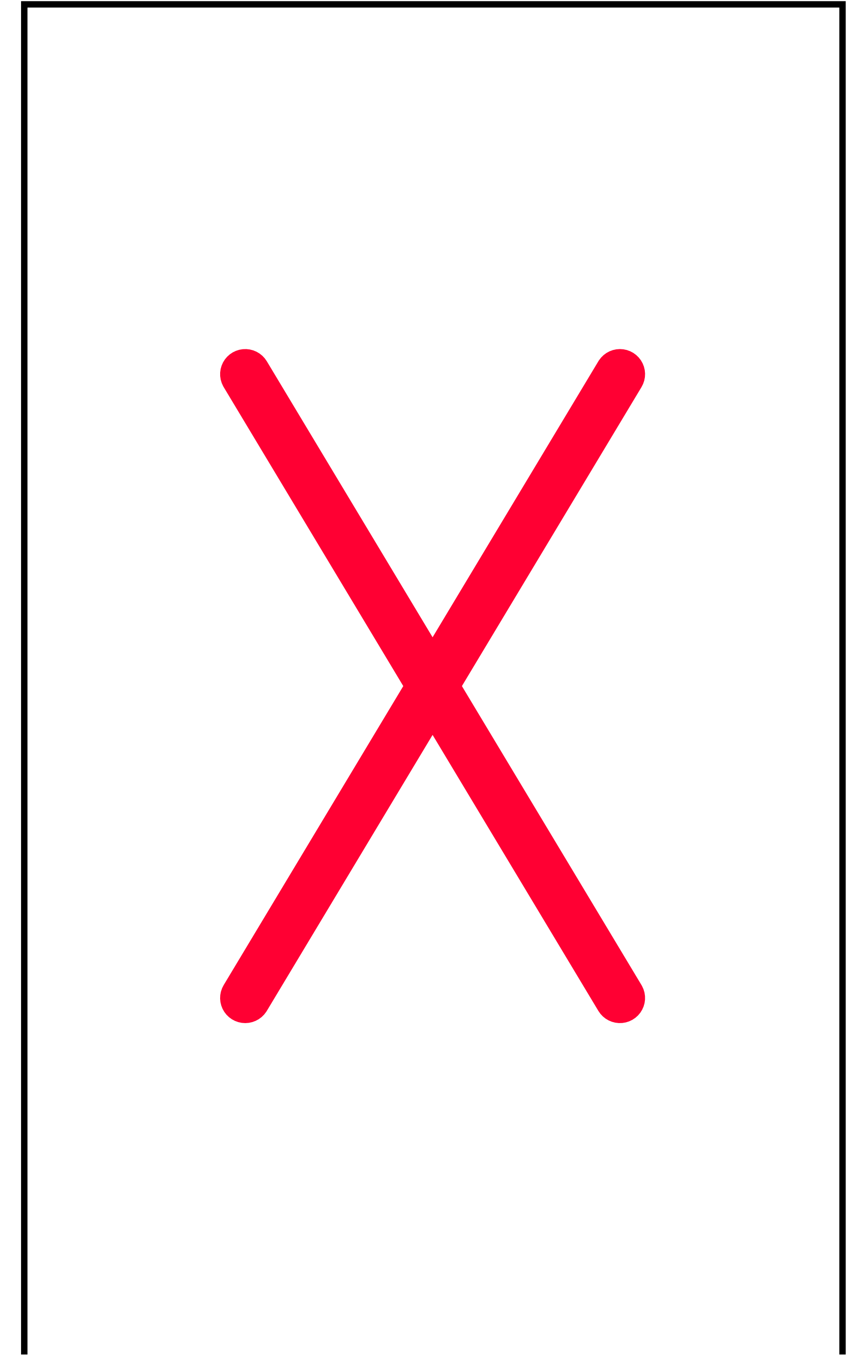
如前所述，通过下列命令编译和连接程序：

icc430 tutor3 -r -L -q （回车键）

3.6.2 察看列表

从列表中用户可以看到编译器为中断函数产生的代码：

```
\   0000                NAME    tutor3(16)
\   0000                RSEG    CODE(1)
\   0000                COMMON  INTVEC(1)
\   0000                RSEG    UDATA0(1)
\   0000                RSEG    IDATA0(1)
```



```
  30          {
\   0000  0C12              PUSH    R12
  31              if (++clock == 10)
\   0002  92530000          ADD     #1,&clock
\   0006  B2900A00          CMP     #10,&clock
\   000A  0000
\   000C  0220              JNE     (?0001)
  32                      clock = 0;
\   000E  82430000          MOV     #0,&clock
\   0012            ?0001:
  33             /* Display 1,2,3,...,9,0,1,2,... */
  34          LCDMEM[0] = digit[clock];
\   0012  1C420000          MOV     &clock,R12
\   0016  D24C0200          MOV.B   digit(R12),&49
\   001A  3100
  35          }
\   001C  3C41              POP     R12
\   001E  0013              RETI
\   0020            main:
  36
  37
  38          void main(void)
  39          {
  40          /* Initialize LCD driver (4Mux mode) */
  41          LCDCTL = 0xFF;
\   0020  F2433000          MOV.B   #255,&48
  42          /* Initialize Basic Timer
  43          Interrupt FQ is ACLK/256/128 = 1 Hz */
  44
  45
  46          BTCTL =  0xF6;
\   0024  F240F600          MOV.B   #246,&64
\   0028  4000
  47          ME2   |= 0x80; /* Set Basic Timer Module Enable */
\   002A  F2D08000          BIS.B   #128,&5
\   002E  0500
  48          BTCTL &=~0x40; /* Disable Basic Timer Reset */
\   0030  F2F0BF00          AND.B   #191,&64
\   0034  4000
  49          IE2   |= 0x80; /* Set Basic Timer Interrupt Enable */
\   0036  F2D08000          BIS.B   #128,&1
\   003A  0100


  50          clock = 0;
\   003C  82430000          MOV     #0,&clock
  51          /* Enable interrupts */
  52          _EINT();
\   0040  32D2              EINT
\   0042            ?0003:
  53          /* wait for interrupt */
```

第4章 C编译器选项摘要

本章给出 C 编译器选项的概略摘要，并说明怎样从嵌入 Workbench 或命令行设置选项。

选项可划分到下列各段（sections）中，在嵌入式 workbench 版本中对应于 ICC430 选项页（pages）:

Code generation（代码产生）　#undef
Debug（调试）　include（包含）
#define　Target（目标）
List（列表）

Command Line（命令行）段提供有关这些选项的信息，它仅在命令行版本中可用。

有关每一个选项的完整参考请参见下章:

“C 编译器选项参考”

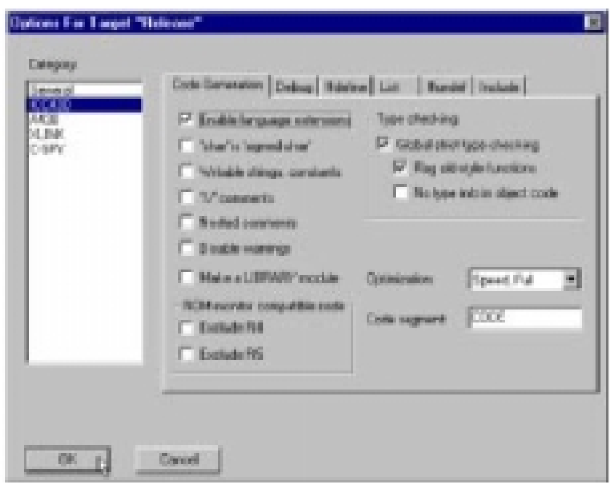
这些章使用下列符号

样式	用于
	标识专用于 IAR 系统工具嵌入式 workbench 接口版本的指令
	标识专用于 IAR 系统工具命令行版本的指令

4.1 设置 C 编译器选项

4.1.1 在嵌入式 Workbench 中设置 C 编译器选项

为了在嵌入式 Workbench 中设置 C 编译器选项，可从 Project（工程）菜单中选择 Options...（选项...），并在 Category 列表中选择 ICC430，屏幕上显示编译器选项页：



然后单击与用户想要察看或改变的选项类别相对应的页(tab)。

4.1.2 从命令行设置 C 编译器选项

为了设置 C 编译器选项，用户可以把选项包含在命令行中，其位置是在 icc430 命令之后，在源文件名之前或之后。例如，当编译源文件 prog 时，如果要使列表产生在缺省列表文件名（prog.lst）之中，那么可键入：

icc430 prog –L　（回车）

某些选项接受文件名，它在选项字母之后，带一个隔开的空格。例如要使列表产生到文件

list.lst：

icc430 prog –l list.lst （回车）

某些选项接受不是文件名的字符串。它被包括在选项字母之后，但没有空格。

例如，要把列表产生缺省文件名中，但它在子目录 list 下：

icc430 prog –L list （回车）

一般来说，选项在命令行中的次序，不管是互相间或与源文件名的相对位置，均不重要。其例外是使用两个或多个-l 选项时，次序是重要的。

也可以在 QCC430 环境变量中规定选项。编译器自动把此变量的值附加到每一个命令行，所以它提供了规定选项的一种方便的方法，这些变量是每一次编译都需要的。

4.2 选项摘要

下面是所有编译器选项的一览表。关于任何选项的完整说明，见下一章“C 编译器选项参考”中选项类别名下各节的叙述。

选项	说明	节(Section)
-A prefix	汇编输出到预定的文件名	列表
-a filename	汇编输出到已命名的文件	列表
-b	生成目标库模块	命令行
-C	嵌套注释	代码产生
-c	字符是有符号的字符	代码产生
-Dsymb[=xx]	定义符号	#define
-e	使能语言扩展	代码产生
-F	在函数之后换页（Form-feed）	列表
-f filename	扩展命令行	命令行
-G	打开标准输入作为源	命令行
-g	全局严格类型检查	代码产生
-gA	标志老式函数	代码产生
-go	目标代码中无类型信息	代码产生
-Hname	设置目标模块名	代码产生
-lprefix	包含路径	命令行
-i	加入#include 文件文本	列表
-k	//注释	代码产生
-L[prefix]	列表到预定的源文件名	列表
-L filename	列表到已命名的文件	列表
-N prefix	预处理器输出预定的文件名	列表
-n filename	预处理器到已命名的文件	列表
-O prefix	设置目标文件名前缀	命令行
-o filename	设置目标文件名	命令行
-p nn	行/页	列表
-q	插入助记符	列表
-p	产生适用于 PROM 的代码	命令行
-R name	设置代码段名	代码产生
-r[012][i][n][r]	产生调试信息	调试
-S	设置静态操作	命令行
-s[0-9]	速度最佳化	代码产生
-T	仅列出有效行	列表
-tn	制表空格	列表
-Usymb	未定义符号	#undef

-ur[4][5]	ROM 监控兼空代码	代码产生
-w	禁止警告	代码产生
-X	解释 C 声明	列表
-x[DFT2]	交叉引用（交叉参考）	列表
-z[0-9]	大小（大小）最佳化	代码产生

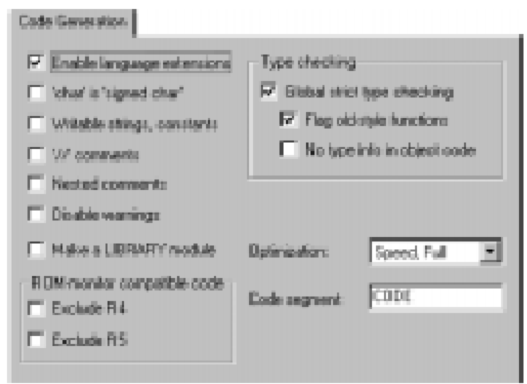
第5章 C 编译器选项参考

本章给出 MSP430C 编译器每一个选项的详细资料，这些选项按其功能被划分为各种类别。

5.1 代码产生

代码产生选项决定源程序的译码和目标代码的产生。

5.1.1 嵌入式 Workbench



5.1.2 命令行

-e	使能语言扩展
-c	‘char（字符）’为‘signed char（有符号字符）’
-y	可写字符串，常数
-k	‘//’注释
-C	嵌套注释
-w	禁止警告
-b	生成 LIBRARY（库）模块
-g	全局严格类型检查
-gA	标志老式（old-style）函数
-go	目标代码中无类型信息
-z[0-9]	大小（尺寸）最佳化
-s[0-9]	速度最佳化
-Rname	代码段
Ur[4][5]	ROM 监控兼容代码

1、使能语言扩展（ENABLE LANGUAGE EXTENSIONS）（-e）

句法：-e

使能对 C 语言的目标有关(target dependent)的扩展。

通常，为了保持兼容性，禁止语言扩展。如果用户要在源（程序）中使用语言扩展，那么用户必须通过包含此选项来使能它。

有关语言扩展的详细资料，可参见“语言扩展”一章。

2、‘CHAR（字符）是 SIGNED CHAR（有符号字符）’（‘CHAR’ IS ‘SIGNED CHAR’）(-c）

句法：-C

生成等价于 Signed char（有符号字符的 char（字符）类型。例如，为了与不同的编译器兼容，使编译器把 char 型解释为 signed char，要使用此项。

注释：运行时间（run-time）库是在没有选择 char is signed char（字符为有符号字符）(-c）的情况下被编译的，所以如果用户对程序使用了此选项并用 Global strict type check （全局严格类型检查）(-g）或 Generate debug information（产生调试信息）(-r）选项使能类型检查，那么用户可能从连接器得到类型不匹配警告。

3、可写的字符串，常数（WRITABLE STRINGS，CONSTANTS）(-y）

句法：-y

使编译器把字符串文字编译为可写的变量（Writable variables）。

通常，字符串文字被编译为只读（read-only）。如果用户想要能写字符串文字，用户可使用 Writable strings, constants（可写的字符串，常数）(-y）选项，使字符串被编译为可写的变量。

注意，用字符串初始化的数组（如 char c[]=“string”）总是被编译为初始化变量，不受 Writable strings, constants（可写的字符串，常数）(-y）选项的影响。

4、‘//’注释（‘//’COMMENTS）(-k）

句法：-K

以 C++样式表示注释，即，注释由‘//’引导，并延展到行的末尾。

通常，为了兼容起见，编译器不接受 C++样式的注释。如果用户的源程序要包含 C++样式的注释，那么为了使之能被接受，必须使用‘//’注释（-K）选项。

5、嵌套注释（NESTED COMMENTS）(-C）

句法：-C

使能嵌套注释。

通常，编译器把嵌套的注释当作错误来处理并在遇到它是发出警告，例如在注释结束不正确时就会出现这种情况。如果用户想要使用嵌套的注释，例如要注释包含注释的代码段时，应当使用 Nested Comments（嵌套注释）(-C）选项以禁止这种警告。

6、禁止警告（DISABLE WARNINGS）(-w）

句法：-w

禁止编译器警告信息。

通常，编译器发出标准警告信息，用 Global strict type check（全局严格类型检查）(-g）选项可以使能任何附加的警告信息。要禁止所有警告信息，可使用 Disable warnings（禁止警告）(-w）选项。

7、生成库模块（MAKE A LIBRARY MODULE）(-b）

句法：-b

产生是模块而不是程序模块的目标文件。

编译器通常产生准备用 XLINK 连接的程序模块，如果用户代之以想要用 XLIB 包含在库中的库模块，那么应当使用 MAKE a LIBRARY module（生成库模块）（-b）选项。

8、全局严格类型检查（GLOBAL STRICT TYPE CHECKING）（-g）

句法：-g[A][0]

在整个源（程序）中使能类型信息的检查。

在源（程序）中有一类情况，它们可以指示可能的编程错误，但是为了兼容起见，编译器和连接器通常忽略它们。为了使编译器和连接器每次遇到这种情况时发出警告，或使用 Global strict type checking（全局严格类型检查）（-g）选项。

9、标志老式函数（FLAG OLD-STYLE FUNCTIONS）（-gA)

句法：-gA

通常，Global strict type checking（全局严格类型检查）（-g）选项不对老式 K&R 函数发出警告。为了使能这种警告，可使用 Flag old-style functions（标志老式函数）（-gA）选项。

10、在目标代码中无类型信息（NO TYPE INFO IN OBJECT CODE）（-g0)

句法：-g0

通常，Global strict type checking（全局严格类型检查）（-g）选项在目标模块中包含类型信息，这增加了它的大小和连接时间，允许连接器发出类型检查警告。为了去掉此信息，避免这种大小上和时间上的增加但禁止连接器类型检查警告，可使用 No type info in object code（目标代码中无类型信息）（-g0）选项。

当连接多个模块时，应注意：无类型信息编译的模块中的目标被认为是无类型的（typeless），无类型信息编译是指没有-g 选项的编译或编译时采用了带修正符的-g 选项。因此决不会有任何类型不匹配的警告来自不带类型信息而编译的模块内的声明，甚至带相应声明的模块已完成带类型信息的编译也是如此。

由 Global strict type checking（全局严格类型检查）（-g）选项检查的情况是：

- 对未声明函数的调用。
- 未声明的 K&R 形式参数。
- 在非空(non-void)函数中丢失返回值。
- 未引用的局部或形式参数。
- 未引用的 goto 标号。
- 未到达的代码。
- 与 K&R 函数不匹配或改变了的参数。
- 未知符号的#undef。
- 合法但含糊的初始化。
- 超范围的常量数组索引。

举例

下列例子分别说明了这些错误类型中的每一种。

（1）对未声明函数的调用

程序：

```
void my_fun(void) { }
int main(void)
{
  my_func();    /* mis-spelt my_fun gives undeclared
                   function warning */
  return 0;
}
```

错误：

```
my_func();      /* mis-spelt my_fun gives undeclared
                function warning */
--------^
"undecfn.c",5  Warning[23]: Undeclared function
'my_func'; assumed "extern" "int"
```

（2）未声明的 K&R 形式参数

程序：

```
int my_fun(parameter)   /* type of parameter not declared
                        */
{
  return parameter+1;
}
```

错误：

```
int my_fun(parameter)   /* type of parameter not declared
                        */

---------------------^
"undecfp.c",1  Warning[9]: Undeclared function parameter
'parameter'; assumed "int"
```

（3）在非空（non-void）函数中丢失返回值

程序：

```
int my_fun(void)
{
  /* ... function body ... */
}
```

错误：

```
}
^
"noreturn.c",4  Warning[22]: Non-void function: explicit
"return" <expression>; expected
```

（4）未引用的局部或形式参数

程序：

```
void my_fun(int parameter)      /* unreferenced formal
                                parameter */
{
  int localvar;                 /* unreferenced local
                                variable */
  /* exit without reference to either variable */
}
```

错误：

```
}
^
"unrefpar.c",6  Warning[33]: Local or formal 'localvar'
was never referenced
"unrefpar.c",6  Warning[33]: Local or formal 'parameter'
was never referenced
```

（5）未引用的 goto 标号

程序：

```
int main(void)
{
  /* ... function body ... */
  exit:                         /* unreferenced label */

  return 0;
}
```

错误：

```
}
^
"unreflab.c",7  Warning[13]: Unreferenced label 'exit'
```

（6）未到达的代码

程序：

```
#include <stdio.h>
int main(void)
{
  goto exit;
  puts("This code is unreachable");
  exit:
  return 0;
}
```

错误：

```
  puts("This code is unreachable");
------^
"unreach.c",7  Warning[20]: Unreachable statement(s)
```

（7）与 K&R 函数不匹配或改变了的参数

程序：

```
int my_fun(len,str)
int len;
char *str;
{
  str[0]='a' ;
  return len;
}
char buffer[99] ;
int main(void)
{
  my_fun(buffer,99) ;   /* wrong order of parameters */
  my_fun(99) ;          /* missing parameter */
  return 0 ;
}
```

错误：

```
my_fun(buffer,99) ;     /* wrong order of parameters */
--------------^
"varyparm.c",14  Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(buffer,99) ;     /* wrong order of parameters */
-----------------^
"varyparm.c",14  Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(99) ;            /* missing parameter */
----------^
```

（8）未知符号的#undef

程序：

```
#define my_macro 99
/* Misspelt name gives a warning that the symbol is
unknown */
#undef my_macor
int main(void)
{
  return 0;
}
```

错误：

```
#undef my_macor
--------------^
"hundef.c",4  Warning[2]: Macro 'my_macor' is already
#undef
```

（9）合法但含糊的初始化

程序：

```
typedef struct t1 {int f1; int f2;} type1;
typedef struct t2 {int f3; type1 f4; type1 f5;} type2;
typedef struct t3 {int f6; type2 f7; int f8;} type3;
type3 example = {99, {42,1,2}, 37};
```

错误：

```
type3 example = {99, {42,1,2}, 37} ;
-----------------------------------^
"ambigini.c",4  Warning[12]: Incompletely bracketed
initializer
```

（10） 超范围的常量数组索引

程序：

```
char buffer[99] ;
int main(void)
{
  buffer[500] = 'a' ;   /* Constant index out of range */
  return 0;
}
```

错误：

```
buffer[500] = 'a' ;     /* Constant index out of range */
-----------^
"arrindex.c",5  Warning[28]: Constant [index] outside
array bounds
```

11、大小最佳化（OPTIMIZE FOR SIZE）(-z）
句法：-z[0-9]
使编译器产生大小为最小的最佳代码。
在通常情况下，编译器按级别 3 的最小大小（见下述）实现最佳化。用户也可以使用。Z 选项改变实现最小大小的最佳化的级别，如下述：

修改符	级别
0	不最佳化
1-3	完全可调试
4-6	某此结构不能调试
7-9	完全最佳化。某此结构不能调试。

12、速度最佳化（OPTIMIZE FOR SPEED）(-s)
句法：-s[0-9]
使编译器以达到最高执行速度对代码实现最佳化。
通常，编译器按级别 3（见下述）的最高执行速度对代码实现最佳化。如下所示，用户可以使用-s 选项改变实现最高执行速度的最佳化级别：

修改符	级别
0	不最佳化
1-3	完全可调试
4-6	某此结构不能调试
7-9	完全最佳化。某此结构不能调试。

13、代码段（CODE SEGMENT）(-R）
句法：-Rname
设置代码段的名字。
通常，编译器把可执行代码放在名为 CODE 的段中，缺省情况下，连拉器把该段放在可变的地址。如果用户想要能够为代码指定一个明确的地址，那么用户可以使用-R 选项指定一个特定的代码段的名字，然后可以在连接器命令文件中把它分配到一个固定的地址。

14、ROM 监控兼容代码（ROM-MONITOR COMPATIBLE CODE）(-ur）
句法：-ur[4][5]
使编译器不使用寄存器 R4 和/或 R5 产生 ROM 兼容代码。

5.2 Debug（调试）

Debug（调试）选项决定包含在目标代码中调试信息的等级。

5.2.1 嵌入式 Workbench

5.2.2 命令行

-r[012][i][n][r] 产生调试信息

产生调试信息（GENERATE DEBUG INFORMATION）(-r)

句法：-r[012] [i][n][r]

使编译器在目标模块中包含 C-SPY 和其他符号化调试器所需的附加信息。

通常为了提高代码的效率，编译器不包含调试信息。为了生成可用 C-SPY 调试的代码，用户只需简单不带修改符地包含此选项。

要生成可用其他调试器调试的代码，可选择一个或多个选项，如下所示：

选项	命令行
加#include 文件信息	i
在目标代码中隐藏源（代码）	n
无寄存器变量	r
代码加至语句	0,1,2

通常 Generate debug information（产生调试信息）(-r）选项不包含加#include 文件调试信息，这是因为一般来说这没有什么意义，而且大多数非 C-SPY 的调试器也不支持在#include 文件内进行调试。如果用户要在#include 文件内调试，例如如果#include 文件包含不同于常用函数的声明，那么用户可以使用 Add #include file information（加#include file 信息）修改符（-ri)。在某些非 C-SPY 调试器中，附带的影响是源行记录包含了全局（=总）行的计数，这可能影响源行的显示。

Generate debug information（产生调试信息）(-r）选项通常在目标文件中包含 C 源（代码）行，所以在调试期间内可显示它们。如果用户想要隐藏它们以便减少目标文件的大小，那么用户可以使用 Suppress source in object code（在目标代码中隐藏源代码）(-rn）修改符。对于大多数其他不包含有关如何使用 IAR 系统 C 编译器专用信息的调试器，使用此选项。

通常，编译器试图把局部变量（Locals）作为寄存器变量。但是某些调试器不能处理寄存器变量；为了抑制寄存器变量的使用，可以使用 No registar variables(无寄存器变量)（-rr）修改符。

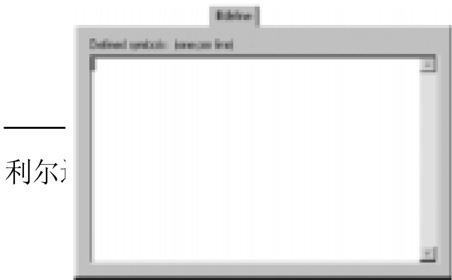
Code added to statements（代码加至语句）选项把一个（-r1）或两个（-r2）NOP（空操作）

加至每一语句所产生的代码。如果用户的调试工具专门要求用户这样做，那么只有使用这些选项之一。

5.3 #define

#define 选项允许用户定义 C 编译器所使用的符号。

5.3.1 嵌入式 Workbench



5.3.2　命令行

-D　　定义符号

定义符号（DEFINED SYMBOLS）（-D）

句法：-Dsymb[=xx]

定义名字为 symb、值为是 xx 的符号。如要未指定值，那么使用 1。

Defined symbols（定义符号）（-D）具有与源文件顶部#define 语句同样的效果。

-Dsymb 等效于# define symb

Defined symbols（定义符号）（-D）选项可用于在命令行上更为方便地规定（符号的）值或选择，否则它们将在源文件中规定。例如，用户可以根据符号 testver 是否被定义来把源（代码）安排到程序的测试或生产版本。为了做到这一点，用户可以使用包含段（include sections），例如：

```
# ifdef testver
…      ;additional code line
for test version only
#endif
```

然后，如下所述，用户将在命令行中选择所需的版本：

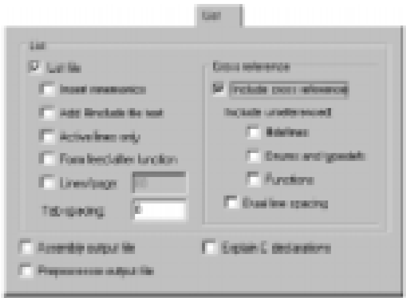
生产版本：icc 430 prog

测试版本：icc430 prog-Dtestver

5.4 列表（LIST）

List(列表)选项决定是否产生列表，以及包含在列表中的信息。

5.4.1 嵌入式 Workbench



5.4.2 命令行

选项	说明
-L [prefix]	列表到预定的源文件名
-l filename	列表到已命名的文件
-q	插入助记符
-I	加入# include 文件文本
-T	列出有效行
-F	函数之后换页
-Pnn	行/页
-tn	制表空格
-x[DFT2]	交叉引用（交叉参考）
-Aprefix	汇编输出到预定的文件名

-a filename	汇编输出到已命名的文件
-N prefix	预处理器输出到预定的文件名
-n filename	预处理器输出到已命名的文件
-X	解释 C 声明

1、列表文件

（1）列表到预定的源文件名（List to prefixed source name）(-L)

产生列表并送到文件名与源文件相同，但扩展名为.lst 的文件。如果需要的话，可由参数给出前缀。

通常，编译器不产生列表。为了简化产生列表，用户可以使用不带前缀的-L 选项。例如，要在文件 prog.lst 中产生列表，用户可使用：

icc430prog –L （回车）

为了产生列表并送到不同的目录中，用户可使用后随目录名的-L 选项。例如，要产生列表并送到目录\list 内相应的文件名中：

icc430prog –L list\ （回车）

这将使文件送到 list\prog.lst 而不是缺省的 prog.lst。

-L 不能与-l 同时使用。

（2）列表到已命名的文件（List to named file）(-l)

句法：-l filename

产生列表并送到已命名的文件，该文件号具有缺省的扩展名.lst。

通常，编译器不产生列表。为了把产生列表并送到已命名的文件，用户可以使用-l 选项。例如，为了产生列表并送到文件列表并送到文件 list.lst 中，可使用：

icc 430 prog –l list （回车）

更为经常的是，用户不需要指定特定的文件名，在这种情况下用户可代之以使用-L 选项。

此选项不能与-L 选择同时使用。

2、插入助记符（INSERT MNEMONICS）(-q)

句法：-q

在列表中包含所产生的汇编行。

通常，编译器在列表中不包含所产生的汇编行。如果用户想要包含它们，例如能检查由特定语句产生的代码的效率，用户可使用 Insert mnemonics（插入助记符号）(-q) 选项。

注意，只有在指定列表时才能使用此选项。

也可参见选项-a,-A –l 和-L。

3、加入#Include 文件文本（ADD#INCLUDE FILE TEXT）(-i)

句法：-I

使得列表包含#include 文件。

通常列表不包含#include 文件，这是因为它们通常只包含头部信息，会在列表中浪费空间。为了包含#include file text（加入#include 文件文本）(-i) 选项。

4、仅列出有效行（ACTIVE LINE ONLY）(-T)

句法：-T

使编译器仅列出有效的源代码行。

通常，编译器列出所有的源代码行。为了通过删除无效行（例如在错误的#if 结构中的行）来

节省列表空间，用户可使用 Active lines only（仅列出有效行）（-T）选项。

5、函数之后换页（FORM-FEED AFTER FUNCTION）（-F）

句法：-F

在汇编列表中每一个列出的函数之后产生换页（form-feed）

通常，列表简单地在下一行开始每一个函数。为了使每一个函数出现在新的页的顶部，用户可以包含此选项。

对于未列出的函数，从不产生换页，例如，像在#include 文件中那样。

6、行/页（LINES/PAGE）（-p）

句法-pnn

把列表格式化到页，并规定每页的行数（其范围为 10 至 150）。

通常，列表没有格式化到页。为了把它格式化到页并在每页换纸，用户可以使用 Lines/page（行/页）（-p）选项。例如，对于每页 50 行的打印机：

icc 430 prog –p50 (回车)

7、制表空格（TAB STACING）(-t)

句法：-tn

把每个制表间隔的字符位置数目设置为 n，它必须在 2 至 9 范围内。

通常，列表按 8 个字符的制表空格格式化。如果用户想要不同的制表空格，那么用户可以用 Tab spacing（制表空格）（-t）选项对它进行设置。

8、交叉引用（CROSS REFERENCE）（-x）

句法：-x[DFT2]

在列表中包含交叉引用（交叉参考，cross-reference）表。

通常，编译器在列表中不包含全局符号。为了在列表的末尾包括所有变量目标以及所有被引用的函数、#deffine 语句、enum 语句和 typedef 语句的列表，用户可以使用不带修正符的 Cross-reference（交叉引用）（-x）选项。

当用户选择 Cross-reference（交叉引用时）时，下列选项可供使用：

命令行	选项
D	显示未引用的#defines
F	显示未引用的函数
T	显示未引用的 typedefs 和 enum 常数

9、汇编输出文件

（1）汇编输出到预定的文件名（Assembly output to prefixed filename）（-A）

句法：-A prefix

产生汇编源代码并送到 perfix source.s43（预定源文件名.s43）

缺省情况下，编译器不产生汇编源代码。为了把汇编代码送到文件名与源文件名相同，但扩展名为.s43 的文件中，使用不带参数的-A。例如：

icc430 prog –A (回车)

它产生汇编源代码并送到文件 prog.s43 中。

为了把汇编源代码送到不同目录内同样文件名的文件中，可使用把目录作为参数的-A 选项。例如：

icc430 prog –Aasm\ （回车）

将在文件 asm\prog.s43 中产生汇编源代码。

汇编源代码可以用适当的 IAR 汇编器汇编。

如果也使用-l 或-L 选项，那么 C 源代码行将包含在汇编源代码文件中作为注释。

-A 选项不能与-a 选项同时使用。

（2）汇编输出到已命名的文件（Assembly output to named file）(-a)

句法：-a filename

产生汇编源代码并送到 filename.s43

缺省情况下，编译器不产生汇编源代码。此选项产生汇编源代码并送到已命名的文件。

文件全名由文件名和其前面的路径名（可选）以及其后的扩展名（可选）组成。如果不给出扩展名，那么使用目标指定（target-specific）的汇编源代码扩展名。

汇编源代码可由适当的 IAR 汇编器汇编。

如果也使用-l 或-L 选项，那么 C 源代码行将包含在汇编源代码文件中作为注释。

此选项不能与-A 选项同时使用。

10、 预处理器输出到预定的文件（PREPROCESSOR TO PREFIXED FILENAME）(-N)

句法：-N prefix

产生预处理器的输出并送到 prefixsource.i

缺省情况下，编译器不产生预处理器输出。为了把预处理器的输出送到文件名与源文件名相同但扩展名为.i 文件，可使用不带参数的-N。

例如：icc430 prog –N （回车）

产生预处理器输出并送到文件 prog.i 中。

为了把预处理器输出送到相同文件名但在不同目录下的文件中，可使用以目录作为参数的-N 选项。例如：

icc430 prog –Npreproc\ （回车）

将在文件 preproc\prog.i 中产生源代码。

11、 预处理器输出到已命名的文件（PREPROCESSOR TO NAMEED FILE）(-n)

句法：-n filename

产生预处理器输出并送到 filename.i。

缺省情况下，编译不产生预处理器输出。此选项将产生预处理器输出并送到已命名的文件。

文件全名由文件名和其前面的路径（可选）以及其后的扩展名（可选）组成。如果不给出扩展名，那么使用扩展名.i。

此选项不能与-N 同时使用。

12、 解释声明（EXPLAIN C DECLARATIONS）(-X)

句法：-X

显示文件中每一个 C 声明的英文说明。

为了获得 C 声明的英文说明，例如为了有助于错误消息的研究，可以使用 Explain Cdeclarations（解释 C 声明）(-X）选项。

例如，声明

```
Void (*signal (int-sig, void (* func) ())) (int);
```

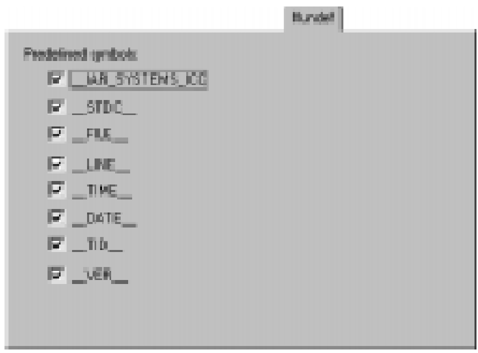
给出说明如下：

```
storage class: extern
  [func_attr:0210] prototyped ?cptr0 function returning
    [attribute:0110] ?dptr0 - ?cptr0 code pointer to
      [func_attr:0210] prototyped ?cptr0 function
returning
        [attribute:0110] ?dptr0 - void
      and having following parameter(s):
        storage class : auto
        [attribute:0110] ?dptr0 - int
      and having following parameter(s):
        storage class: auto
        [attribute:0110] ?dptr0 - int
        storage class : auto
        [attribute:0110] ?dptr0 - ?cptr0 code pointer to
          [func_attr:0210] ?cptr0 function returning
            [attribute:0110] ?dptr0 - void
```

5.5 #undef

5.5.1 #undef 选项使用户能取消对预定义符号的定义

嵌入式 Workbench



5.5.2 命令行

-Usymb　预定义符号

预定义符号（PREDEFINED SYMBOLS）(-U)

句法：-Usymb

取消已命名符号的定义。

通常，编译器提供各种预定义的符号。如果用户想要去除其中之一，例如要避免与用户自己的具有相同名字的符号相冲突，用户可使用 Predefined symbols（预定义符号）(-U) 选项。

关于预定义符号的列表，请参见“预定义符号参考”一章。

为了去除符号的定义，在 Predefined symbols（预定义符号）表中去除选择（deselect）它。

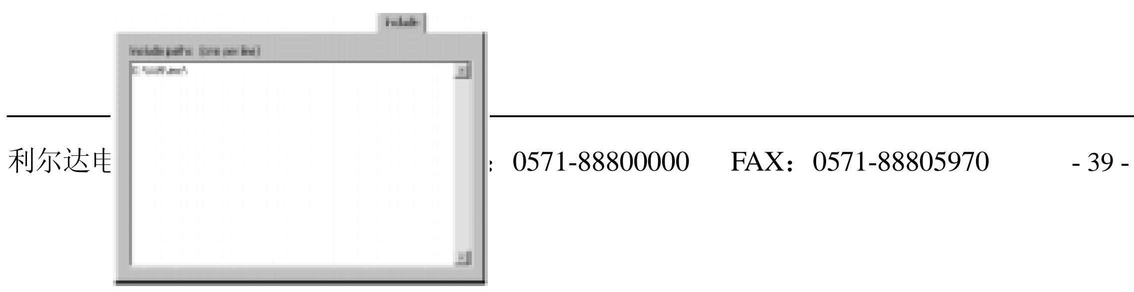
例如，去除符号_VER_，使用：

icc430 prog　_U_VER_　（回车）

5.6 INCLUDE

5.6.1 Include（包含）选项使用户能定义 C 编译器的路径。

嵌入式 Workbench



5.6.2 命令行

-I prefix 包含路径

包含路径（INCLUDE PATHS）(-I)

句法：-I prefix

把前缀（prefix）加到#include 文件前缀表中。

通常，编译器仅在源目录（假如文件名含在引号而不是在尖括号中），C-INCLUDE 路径，以及最终在当前目录中搜索包含文件。如果用户已把#include 文件放在某些其他目录中，那么用户必须使用 Include paths（包含路径）(-I）选项以便把该目标通知编译器。

例如：icc430 prog -I\mylib\ (回车)

注意，编译器简单地把-I prefix 加到包含文件名的开始处，所以重要的是：如果有必要，应包含最终的反斜杠（backslash \)。

在单个命令行上，对允许的-I 选项的数目没有限制。当使用多个-I 选项时，为了避免命令行超出操作系统的限制，用户可以使用命令文件；见-f 选项。

注：编译器的#include 文件搜索步骤的完整说明如下：

当编译器遇到在尖括号内的包含文件名

例如：#include<stdio.h>时它执行下列搜索序列：

- 文件名，它前面依次冠以每一个-I prefix。
- 文件名，它前面依次冠以 C-INCLUDE 环境变量中每一个路径（如果有的话）
- 单独文件名

当编译器遇到在双引号内的包含文件名，例如：

#include“vars.h”时，它搜索前面冠以源文件路径的文件名，然后执行与带尖括号的文件名相同的序列。

5.7 命令行

命令行中有下列附加选项可供使用。

-f filename	扩展命令行
-G	打开标准输入作为源
-H name	设置目标模块名
-O oprefix	设置目标文件名前缀
-o filename	设置目标文件名
-P	产生适用于 PROM 的代码
-S	设置静态操作

5.7.1 扩展命令行（EXTEND THE COMMAND LINE）(-f)

句法：-ffilename

从已命名的，具有缺省扩展名.xcl 的文件中读命令行选项。

通常，编译器只从命令行本身以及 QCC430 环境变量中接受命令参数。为了生成更加易于管理的长命令行，并避免任何操作系统命令行长度的限制，用户可以使用-f 选项指定一个命

令文件，编译器可从此文件中读命令行项（items)，就像它们已被健入到选项位置上一样。因为换行（newline）字符起空格（space）或制表（tab）字符的作用，所以，在命令文件中，除了可能使用多行之外，用户必须严格规定各项的格式，就像它们是在命令行本身之上一样。

例如，命令行：

icc430 prog -r-l-D testver “-D username=John Smith”

-D userid=463760 (回车)

可以用

icc430 prog -r-l-D testver –f userinfo (回车)

和包含下列内容的文件 userinfo.xcl：

“-D username=John Smith”

-D userid=463760

来代替。

5.7.2 打开标准输入作为源（OPEN STANDRD INPUT AS SOURCE）(-G)

句法：-G

打开标准输入作为源，代替从文件中读出源（代码)。

通常，编译器从命令行上命名的文件中读出源代码。如果用户想要代之以使它从标准输入（通常是键盘）中读出源代码，那么用户可以使用-G 选项并省略源文件名。

源文件名设置为 stdin.c。

5.7.3 设置目标模块名（-H）

句法：-Hname

通常，目标模块的内部名是源文件名，不带目录名或扩展名。为了显示地设置目标模块名，用户可以使用-H 选项，例如：

icc430 prog –Hmain (回车)

因为产生双重模块一般将导致连接器错误，所以当几个模块具有相同文件名时，此选项特别有用。例子是在源文件是预处理器产生的暂时文件时。下列（其中%1 是包含在源文件名中的操作系统变量）将给出连接器双重名字（duplicate name）

错误：

```
preproc %1.c temp.c      ; preprocess source, generating
                           temp.c
icc430 temp.c            ; module name is always 'temp'
```

为了避免这种错误，使用-H 以保持持原来的名字：

```
preproc %1.c temp.c      ; preprocess source, generating
                           temp.c
icc430 temp.c -H%1       ; use original source name as
                           module name
```

5.7.4 设置目标文件名前缀（SET OBJECT FILENAME PREFIX）(-O)

句法：-O prefix

设置用于目标文件的前缀。

通常（除非使用-O 选项）目标储存在文件名与源文件名相对应，但扩展名为 r.43 的文件中。为了把目标储存在不同的目标中，用户可以使用-O 选项。

例如，为了把目标储存在\obj 目录中，可使用：

icc430 prog –O\obj\ （回车）

-O 选项不能和-o 选项同时使用。

5.7.5 设置目标文件名（SET OBJECT FILENAME）（-o）

句法：-o filename

设置目标模块将储存在其中的文件名。文件全名由可选的路径名，必需的文件名，以及可选的扩展名（缺省为.r43）组成。

通常编译器把目标代码储存在文件中，该文件的全名为：

- 由-o 指定的前缀，加上
- 源文件的文件名，加上
- 扩展名.r43

为了使目标（模块）储存在不同的文件中，用户可使用-o 选项。例如，要把它储存在文件 obj.r43 中，用户可以使用：

icc430 prog –o obj （回车）

如果用户想要用具有相应的文件名但在不同目录内的文件储存目标，那么使用-O 选项。

-o 选项与-O 选项不同时使用。

5.7.6 产生适用于 PROM 的代码（GENERATE PROMABLE CODE）（-P）

句法：-P

使编译器产生适合于在只读存储器（PROM）中运行代码。

为了与其它的 IAR 编译器兼容，要包含此选项，但是在 MSP430C 编译器中总是有效的。

5.7.7 设置静态操作（-S）

句法：-S

使编译器工作时不把不必要的信息送到标准输出（通常为屏幕）。

通常，编译器发出介绍性的信息和最终的统计报告。为了禁止这些输出，用户可以使用-S 选项。这不影响错误和警告信息的显示。

第6章 配 置

本章说明怎样为不同的要求配置 C 编译器。

6.1 引言

基于 MSP430 微处理器的系统在其需求上可以有很大的变化。

如下所示，环境的每一特性或使用可以由编译器的一个或多个配置元素来控制。

特性	可配置的元素
存储器映射	XLINK 命令文件
非易失性 RAM	XLINK 命令文件
堆栈大小	XLINK 命令文件
Putcher/getchar 函数	运行时间库模块
Printf/scanf 工具	XLINK 命令文件

堆大小	堆库模块
硬件/存储器初始化	_low_level_inlt 模块

以下各节叙述每一个上述特性。注意，许多配置过程包含了编辑标准文件，在开始之前，用户可能想要制作原始文件的副本。

6.2 XLINK 命令文件

为了创建适用于特定工程的 XLINK 命令文件，用户首先应当从 C：\iar\icc430 中复制文件 lnk430.xcl。然后用户应按昭文件中所述修改此文件，以便规定目标系统存储器映射的细节。

6.3 运行时间库（Run-Time Library）

XLINK 命令文件与库文件 cl430.r43 有关。这通常不应被修改。

6.4 存储器映射

用户需要对 XLINK 规定 ROM 和 RAM 的硬件环境地址。通常这应在 XLINK 命令文件模板的副本上进行。

连接选项规定：

- ROM 区域：用于函数，常数，以及初始值。
- RAM 区域：用于堆栈和变量。

有关规定存储器地址范围的详细资料，参见 XLINK 命令文件模板的内容以及《MSP430 汇编器、连接器和库编程指南》中“XLINK”一节。

非易失性 RAM

编译器通过 no-init 类型修正符和存储器#pragma 支持变量的声明，这些变量将驻留在非易失性 RAM 中。编译器把这种变量放置在单独的段 NO-INIT 中，用户可以把它分配到硬件环境中非易失性 RAM 的地址范围内。运行时间（run-time）系统不初始化这些变量。

为了把 NO-INIT 段分配到非易失性 RAM 地址，用户需要修改 XLINK 命令文件。有关把段分配到给定地址的详细资料，请参阅《MSP430 汇编器，连接器和库编程指南》中“XLINK”一节。

6.5 堆栈大小

编译器把堆栈用于各种用户程序操作，所需的堆栈大小（大小）与这些操作的详细情况有密切的关系。如果给出的堆栈太小，那么堆栈通常会使储存的变量被重写，从而很可能导致程序失败。如果给出的堆栈太大，那么 RAM 将被浪费。

6.5.1 估计所需的堆栈大小

堆栈用于下列各项：

- 在函数调用之间保存寄存器变最。
- 储存局部变量和参数。
- 储存表达式的中间结果。
- 储存运行时间库子程序的中间值。
- 保存函数调用的返回地址。
- 保存中断期间内处理器的状态。

总的所需的堆栈大小是最坏情况下上述每一项所需大小的总和。

6.5.2 改变堆栈大小

若连接器命令中使用表达式 CSTACK+200，则在连接器命令文件中缺省堆栈大小被设置为 512（200h）个字节：

-Z（DATA）CSTACK+200

若要改变堆栈大小，可编辑连拉器命令文件并用想要使用的堆栈大小代替 200。

6.6 输入和输出

6.6.1 PUTCHAR 和 GETCHAR

函数 PUTCHAR 和 GETCHAR 是通过它 C 通实现所有基于字符的 I/O 的基本函数。为了能使用任何基于字符的 I/O，用户必须利用硬件环境提供的工具，提供这两个函数的定义。

创建新 I/O 子程序的起点是文件

C:\iar\icc430\putchar.c 和 C:\iar\icc430\getchar.c

1、定制 putchar

创建定制 putchar 版本的步骤如下：

对源文件 putchar.c 作所需的补充并把它保存回到相同名字下（或者把 putchar.c 用作模板，创建用户自己的子程序）

例如，下面的代码使用存储器映射 I/O 写到 LCD 显示：

```
int putchar (int outchar)
/* a very basic LCD putchar routine */
/* pos must be initialized to 15, the */
/* LCD must be initialized and character-map */
/* must be set up with the lookup table. */
/* It will then display the first is characters */
/* supplied */
{if (pos>0)
LCDMEM[-pos]=character_map(outchar):
}
```

- 利用合适的处理器选项编译修改后的 putchar。

Icc430 putchar –b

这将创建名为 putchar.r43 的最佳化的替代目标模块。

- 把新的 putchar 模块加至适当的运行时间库模块，以代替原来的模块。例如，为了把新的 putchar 模块加至标准库，可使用命令：

```
xlib
def-cpu MSP430
rep-mod putchar cl4308ss
exit
```

现在库模块 CL430 将具有修改后的 putchar 以代替原来的 putchar（在用户重写 putchar 模块之前，要确保已保存了原来的 CL430.r43 文件）。

注意：XLINK 允许用户利用-A 选项在把修改后的模块安装到库中之前对它进行测试。把下行加到用户的.xcl 连接文件中：

```
-A putchar
cl430
```

这将装载用户的 putchar.r43 版本，代替 CL430 库中的版本。见《MSP430 汇编器、连接器和库编指南》。注意，putchar 当作 printf 函数的低级部分使用。

2、定制 getchar

低级 I/O 函数 getchar 用两个 C 文件 getchar.c 和 llget.c 来连接。

6.6.2 PRINTF 和 SPRINTF

Printf 和 sprintf 函数使用名为-formatted-write 的通用格式（Common formatter）。ANSI 标准版本的-formatted-write 非常大，提供了许多应用并不需要的工具。为了减少存储器的消耗，在标准 C 库中也提供了下列两种替代的较小的版本：

1、-medinm-write

除了不支持浮点数之外，与-formatted-write 一样。任何使用%f,%g,%G,%e 和%E 分类符的企图将产生错误：

FLOATS? wrong formatter installed!

-medinm-write 比-formatted-write 小得多。

2、-small-write

除了对 int 目标只支持%%，%d，%o,%c，%s 和%x 分类符，且不支持 field width（域密度）和 precision（精度）参数外，-small-write 和-medinm-write 一样。-small-write 的大小是-formatted-write 大小的 10-15%。缺省的版本是-small-write。

6.6.3 选择写格式版本

写格式的选择在 XLINK 控制文件中实现。

缺省选择-small-write 由下行实现：

-e-small-write=-formatted-write

要选择完整的 ANSI 版本，删除这行。

要选择-medinm-write，用下行代替此行：

-e –medinm-write=-formatted-write

6.6.4 简化 PRINTF（REDUCED PRINTF）

对于许多应用来说，并不需要 sprintf，即使是具有-small-write 格式的 printf 也已提供了较多的功能。另一方面，为了支持特殊格式的需求和/或非标准输出设备，可能需要定制的输出子程序。

对于这些应用，在文件 intwri.c 中提供了完整 printf 函数（无 sprintf）的高度简化的版本。可以修改此文件以符合用户的需求并利用上面关于 putchar 所述的步骤把编译后的模块插入到库中以代替原有的模块。

6.6.5 SCANF 和 SSCANF

与 prinff 和 sprintf 函数相类似，sprintf 和 sscanf 使用名为-formatted-read 的通用格式（Common formatter）。-formatter-read 的 ANSI 标准版本非常大，提供了许多应用不需要的工具。为了减少存储器的消耗，在标准 C 库中也提供了另一种较小的版本：

-medinm-read

除了不支持浮点数之外，它与-formatted-red 一样。-medinm-read 比-formatted-read 小得多。

缺省版本是-medinm-read。

6.6.6 选择读格式（READ FORMATTER）版本

读格式（read formatter）的选择在 XLINK 控制文本中实施工。缺省选择-medinm-read,由下行实现：-e-medinm-read=-formatted-read

为了选择完全 ANSI 版本，删除此行。

6.7 寄存器 I/O

利用存储器映射的内部特殊功能寄存器（SFRs），程序可以访问 MSP430I/O 系统。

除了一元&（地址）运算符之外，所有应用于整型的运算符均可用于 SFR 寄存器。提供了适用于 MSP430 系列的预定义的 sfrb/sfrw 声明；参见“运行时间库”和“扩展关键字参考”一章。

预定义的特殊功能寄存器（SFRs）和中断子程序在下列头文件中给出：

处理器	头文件
MSP430X31X	Io310.h
MSP430X32X	Io320.h
MSP430X33X	Io320.h

这些文件在 icc430 子目录中提供。

6.8 堆大小（HEAP SIZE）

如果在程序中使用库函数 malloc 或 calloc，那么 C 编译器将创建堆（heap）或进行分配的存储器。缺省的堆大小是 2000 个字节。

下列文件中说明了改变堆大小的步骤：

C:\iar\etc\heap.c

用户可以通过在.xcl 连接文件中包含下列行来测试修改后的堆模块：

-A heap

cl 430l

这将装载用户的 heap.r43 版本代替 cl4301 库中的版本

6.9 初始化

处理器复位时，执行名为 CSTARTUP 的运行时间系统子程序，它通常实现下列功能：

- 初始化堆栈指针。
- 初始化 C 文件级（file-level）和静态变量。
- 调用用户程序函数 main。

如果用户程序退出（无论是通过 exit 或 abort），那么 CSTARTUP 也负责接收和保持控制。

6.9.1 变量和 I/O 初始化

在某些应用中，用户可能想要初始化 I/O 寄存器，或省略由 CSTARTUP 执行的数据段的缺省初始化。

用户可以通过提供子程序_low_level_init 的定制版来做到这一点，该子程序在数据段被初始化之前从 CSTARTUP 调用。

_low_level_init 的返回值决定数据段是否被初始化。运行时间库包含了_low_level_init 的伪版本（dummy version），它简单地返回 1，使 CSTARTUP 初始化数据段。

文件 lowinit.c 中提供了_low_level_init 的源代码，缺省情况下此文件位于 icc430 目录中。为了实现用户自己的 I/O 初始化，要创建此子程序包含进行初始化所必需代码的版本。

如果用户也想禁止数据段的初始化，使子程序返回 0。编译定制的子程序并把它与用户的代码的其余部分连接。

6.9.2 修改 CSTARTUP

如果用户想要修改 CST 本身，那么用户需要与用户选择的编译选项相一致的选项重新汇编 CSTARTUP。

汇编适当的 CSTARTUP 副本的总的步骤如下所述：

- 对 CSTARTUP 的汇编源代码作任何必需的修改，缺省情况下该源代码在下列文件中提供：C:\iar\icc430\cstartup.s43，并把它保存在同样的文件名下。
- 汇编 CSTARTUP。

这将创建名为 cstartup.r43 的目标模块文件。

然后，用户应当在连接器命令文件中使用下列命令以便使 XLINK 使用用户已定义的 CSTARTUP 模块来代替 library 中的模块：

```
-A  cstartup
-C  library
```

在嵌入式 Workbench 中，把修改后的 cstartup 文件加到用户的工程中，并在连接器命令文件中在 library 之前加上-C。

第7章 数据表示法

本章叙述 MSP430C 编译器怎样表示每一种 C 数据类型，并给出提高代码效率的建议。

7.1 数据类型

MSP430C 编译器支持所有 ANSI C 基本元素。用位于低存储器地址的最低有效部分储存变量。下表给出每一种 C 数据类型的大小和范围：

数据类型	字节数	范围	注释
Sfrb,sfrw	1		见“扩展关键字”一章
Char（缺省）	1	0 至 255	等效于 unsigned char
Char（使用-c 选项）	1	-128 至 127	等效于 signed char
Signed char	1	-128 至 127	
Unsigned char	1	0 至 255	
Short, int	2	-2^{15} 至 $2^{15}-1$	-32768 至 32767
Unsigned short,			
Unsigned int	2	2 至 $2^{16}-1$	0 至 65535
Long	4	-2^{31} 至 $2^{31}-1$	-2147483648 至 2147483647
Unsigned long	4	0 至 $2^{32}-1$	0 至 4294967295
Pointer	2		见“指针”一节
Enum	1 至 4		见下述
Float	4	±1.18E-38 至±3.39E+38	
Double,long double	4	±1.18E-38 至±3.39E+38	
		(与 float 相同)	

7.1.1 枚举类型（ENUM TYPE）

enum 关键词所创建的每一个目标(对象)具有包含其值所需的确良最短整型(char,short,int 或 long)。

7.1.2 字符类型（CHAR TYPE）

在编译器中缺省情况下 char 类型是无符号的，但是 char is signed char（字符是有符号的字符）(-C）选项使用户可把它变为有符号的。但是要注意：库是以 char 类型为无符号的方式被编译。

7.1.3 浮点（FLOATING POINT）

在标准 IEEE 格式中，浮点使用 4 字节数来表示。低于最小极限值的浮点数被看作零而溢出给出不定的结果。

4 字节浮点格式

4 字节浮点数的存储器布局为：

31	30　　23	22　　0
S（符号）	Exponent（指数）	Mantissa（尾数）

数的值为：

(-1）s*2(Exponent-127)*1.Mantissa

零由 4 个字节的零表示。

浮点运算符（+，-，*，/）的精度近似为 7 个十进小数位。

7.1.4 特殊功能寄存器变量（SPECIAL FUNCTION REGISTER VERIABLES）

特殊功能寄存器（sfr）变量直接位于内部 RAM 单元。Sfrb 范围为 0×00 至 0×FF，sfrw 范围为 0×100 至 0×1FF。Sfrb 类型使符号名（symbolic name）与此范围内的字节相联系。该地址处寄存器可以符号化地被寻址，但没有分配存储器空间。

7.1.5 位域（BITFIELDS）

表达式中的位域具有与基本类型相同的数据类型（signed 或 unsigned char, short,int 或 long)。

具有基本类型 char,short 和 long 的位域是对 ANSIC 整数位域的扩展。

位域变量封装在从 LSB 位置开始的特定类型的元素中。

7.2 指针

这一节叙述 MSP430C 编译器中代码指针和数据指针的使用。

7.2.1 代码指针

代码指针的大小是 2 个字节并可指向 0×0000 至 0×FFFF 范围内的存储器。

7.2.2 数据指针

数据指针的大小是 2 个字节，并能指向 0×0000 至 0×FFFF 范围内的存储器。

7.3 指高代码的效率

为了避免使用低效的语言结构，理解 MSP430 结构的限制是重要的。下面是有关怎样最好地使用 MSP430 编译器的建议。

因为在 MSP430 上位域类型执行缓慢，所以应当仅在为了保存数据存储器空间时才使用它。用 unsigned char 或 unsigned int 的位屏蔽代替位域。如果用户必须使用位域，那么为了提高效率，使用 unsigned。

在其模块之外不使用的变量应当被声明为静态的（static），这是因为这样将改善把它们暂时保存在寄存器中的可能性。

- 在可能时，使用无符号(unsigned)数据类型。某些时候 unsigned(无符号)操作比 signed（有符号）操作执行效率更高。这特别适用于除法和求模运算。
- 使用 ANSI 原型，对 ANSI 函数的调用比对 K&S 样式（K&S-style）函数的调用能效率更高地被执行；参见“K&S 和 ANSI C 语言定义”一节。
- 对 16 位数据类型（例如 short 和 unsigned short）MSP430 的工作效率更高。通常 8 位数据类型的使用节省数据空间，但不减少代码大小。MSP430 的结构不直接支持 32 位数据类型，因而其效率较低。
- 自动比例(scalar auto)变量常被分配在寄存器中。所以只要有可能，就应使用自动而不是静态（变量）。
- 函数的头两个参数（parameters）在寄存器中传送；参见“调用约定”一节能。因此以参数（parameters）而不是静态变量方式把自变量(arguments)传送给函数是更为有装的。
- 复制结构（structs）和联合（unions）是耗费较大的操作。应避免运行时结构/联合的赋值，带结构/联合参数的函数以及返回结构/联合的函数。只要有可能，应采取指向结构/联合的指针操作。
- 每当调用了函数时，声明在这些函数中具有初始值的非自动比例（Non-scalar auto）变量（结构、联合和数组）将引起运行时（run-time）复制。对于常数变量，使用储存类“静态常量”（the storage class“static const”）可避免这种情况。

第8章 通用 C 库定义

本章给出对 C 库函数的介绍并按照关文件对它们作了概括。

8.1 引言

IARC 编译器包提供了大多数重要的 C 库定义，它们应用于基于 PROM 的嵌入式系统。这些函数具有三种类型：

- 标准 C 库定义，适用于用户程序。这些在本章中草药说明。
- CSTARTUP，包含启动(Start-up)代码产单个程序模块。
- 内在函数(Intrinsic functions)，提供 MSP430 特性的低级使用。

8.2 库模块文件

大多数库定义不加修改即可使用，即：直接由库目标文件提供。用户可能需要为用户的目标应用定制某些面向 I/O（I/O-oriernted）子程序（例如 putchar 和 getchar）。

所提供的库目标文件已经使用 Flag old-style functions（标志老式函数）（-gA）选项进行编译。

8.3 头部文件

用户程序通过头部文件访问库定义，头部文件使用#include 伪指令与之相结合。为了避免浪费编译时间，定义被划分到许多不同的头部文件，每一个文件覆盖特定的函数区域，使用户能只包含所需的部分。

在对函数的定义作任何引用之前，包含合适的头部文件旧紧要的。不做到这一点将导致执行期间内调用失败，或者在编译或连接时产生错误或警告信息。

8.4 库定义小结

本节列出了头部文件并概括了包含在每一个文件中的函数。头部文件另外可包含目标特定(target-specific)的定义——这些将在“语言扩展”一章中说明。

1、字符处理（CHARACTER HANDLING）——ctype.h

Isalnum	int	Isalnum(int c)	是否为字母或数字
Isalpha	Int	Isalpha(int c)	是否为字母
Iscntrl	int	Iscntrl(int c)	是否为控制码
Isdigit	Int	Isdigit(int c)	是否为数字
Isgraph	int	Isgraph(int c)	是否为可打印的非空白字符
Islower	Int	Islower(int c)	是否为小写字母
Isprint	int	Isprint(int c)	是否为可打印字符
Ispunct	Int	Ispunct(int c)	是否为标点符号字符
Isspace	int	Isspace(int c)	是否为空白字符
isupper	Int	isupper(int c)	是否为大写字母
Isxdigit	Int	Isxdigit(int c)	是否为 16 进制数字
Tolower	int	Tolower(int c)	转换为小写
toupper	Int	toupper(int c)	转换为大写

2、低级子程序（LOW-LEVEL ROUTINES）-icclbutl.h

_formatted_read	Int _formatted _read (const char **line, const char **format, va-list ap)	读格式化数据
_formatted_write	int _formatted _write (const char **format, void outputf(char, void *)void*sp, va_list ap)	格式化并写数据
_medium_read	Int _formatted _read (const char **line, const char **format, va-list ap)	读除浮点数之外的格式化数据
medium write	int _formatted _write (const char **format, void outputf(char, void *)void*sp, va_list ap)	写除浮点数之外的格式化数据
medium write	int _formatted _write (const char **format, void outputf(char, void *)void*sp, va_list ap)	小型格式化数据写子程序

3、数学函数

Acos	Double	Acos(double arg)	反余弦
Asin	Double	Asin(double arg)	反正弦
Atan	Double	Atan(double arg)	反正切
Atan2	Double	Atan2(double arg1,double arg2)	自变量比的反正切
Ceil	Double	Ceil(double arg)	大于或等于 arg 的最小整数
Cos	Double	Cos(double arg)	余弦
Cosh	Double	Cosh(double arg)	双曲余弦
Exp	Double	Exp(double arg)	指数
Fabs	Double	Fabs(double arg)	双精度的浮点绝对值
Floor	Double	Floor(double arg)	小于或等于 arg 的最大整数
fomod	Double	fomod(double arg1,double arg2)	浮点余数
Fxexp	Double	Fxexp(double arg1,int *arg2)	把浮点数分为两部分
ldexp	Double	ldexp(double arg1,int *arg2)	乘以 2 的幂
Log	Double	Log(double arg)	自然对数
Log10	Double	Log10(double arg)	以 10 为底的对数
Modf	Double	Modf(double value, double*iptr)	分为整数和小数部分
Pow	Double	Pow(double arg1,double arg2)	求幂
Sin	Double	Sin(double arg)	正弦
Sinh	Double	Sinh(double arg)	双曲正弦
Sqrt	Double	Sqrt(double arg)	平方根
Tan	Double	Tan(double arg)	正切
tanh	Double	tanh(double arg)	双曲正切

4、非局部跳转（NON-LOCAL JUMPS）-setjump.h

Longjmp	void	Longjmp(jmp_but env, int val)	长跳转
setjmp	Int	Setjmp(jmp_but env)	设置跳转返回点

5、变量参数（VARIABLE ARGUMENTS）-stdarg.h

Va_arg	type	Va_arg(Va_list ap, mode)	函数调用中下一个参数
Va_end	Void	Va_end (va_list ap)	结束读函数调用参数
Va_list	Char*	Va_list[1]	参数列表类型
Va_start	Void	Va_start(va_list ap,parmN)	开始读函数调用参数

6、输入/输出（INPUT/OUTPUT）-stdio.h

Getchar	Int	Getchar (void)	读字符
Gets	Char*	Gets (char*s)	读字符串
Printf	Int	Printf(const char*s,const char*format,...)	写格式化数据
Putchar	Int	Putchar(int value)	写字符
Puts	Int	Puts(const char*s)	写字符串
scanf	Int	Csanf(const char*s,const	读格式化数据

		char*format,...)	
Sprintf	Int	Sprintf (char*s,const char*format,...)	把格式化数据写入字符串
sscanf	Int	Sscanf (const char*s,const char*format,...)	从字符串读格式化数据

7、通用程序（GENERAL UTILITIES）-stdlib.h

Abort	Void abort (void)	非正常结束程序
Abs	Int abs (int j)	绝对值
Atof	Double atof (const char*nptr)	把 ASCII 转换成 double
Atoi	Int atoi (const char*nptr)	把 ASCII 转换成 int
Atol	Long atol (const char*nptr)	把 ASCII 转换成 long int
Bsearch	Void *bsearch (const void * key,const void *base,size-t nmemb, size_t size,int (*compare) (const void * _key, const void * _base)	在数组中进行搜索
Calloc	Void * calloc (size_t nelem, size_t elsize)	为目标（对象）数组分配存储器
Div	Div_t div (int numer, int denom)	除
Exit	Void exit (int status)	结束程序
Free	Void free (void * ptr)	释放存储器
Labs	Long int labs (long int j)	长整型绝对值
Ldiv	Lidv_t ldiv (long int numer, long int denom)	长整型除
Malloc	Void * malloc (size_t size)	分配存储器
Qsort	Void qsort (const void * base,size_t nmemb, size_t size) int (*compare)(const void *_key,const void * _base));	进行数组的排序
Rand	Int rand (void)	随机数
Realloc	Void * realloc (void *ptr, size_t size)	重新分配存储器
Srand	Void srand (unsigned int seed)	设置随机数序列（的种子）
Strtod	Double strtood (const char * nptr, char * *endptr) double	把字符串转换为（双精度）数
Strtol	Long int strtod (const char * nptr, char * * endptr,int base)	把字符串转换为 long integer (长整型)数
strtoul	Unsigned long int strtoul (const char* nptr, char * *endptr, int base)	把字符串转换为 unsigned long integer（无符号长整型）数

8、字符串处理（STRING HANDLING）-string.h

Memchr	Void * memchr (const void * s, int c, size_t n)	在存储器中搜索字符
Memcmp	Int memcmp (const void * s1,const void *s2, size_t n)	比较存储器(内容)
Memcpy	Void * memcpy (void * s1, const void * s2, size _ t n)	复制存储器(内容)
Memmove	Void *memmove (void * s1, const void * s2, size _ t n)	移动存储器(内容)
Memset	Void * Memset(void * s, int c, size_t n)	设置存储器

Strcat	Char *strcat (char *s1, const char *s2)	逻辑字符串
Strchr	Char * strchr (const char * s, int c)	在字符串中搜索字符
Strcmp	Int strcmp (const char*s1, const char *s2)	比较两个字符串
Strcoll	Int strcoll (const char*s1, const char *s2)	比较字符串
Strcpy	Char * strcpy (char * s1, const char *s2)	复制字符串
Strcspn	Size-t * strcspn (const char * s1, const char *s2)	在字符串中跨过被排除的字符
Strerror	Char * strerror (int errnum)	给出错误信息字符串
Strlen	Size-t * strlen (const char * s)	字符串长度
Strncat	Char * Strncat (char * s1, const char * s2, size_t n)	把指定数量的字符与字符串连接
Strncmp	Int * strncmp(const char * s1, const char * s2, size_t n)	把指定数量的字符与字符串比较
Strncpy	Char * Strncpy(char * s1, const char * s2, size_t n)	从字符串复制指定的字符
Strpbrk	Char * Strpbrk (const char * s1, const char * s2, size_t n)	在字符串寻找任何一个指定字符
Strrchr	Char * Strrchr (const char * s, int c)	从字符串右端寻找字符
strspn	Size-t strspn (co nst char * s1, const char * s2)	在字符串中跨过字符
Strstr	Char *strstr (const char * s1, const char * s2)	搜索子字符串
strtok	Char * strtok (const char * s1, const char * s2)	断开字符串至标记
strxfrm	Size-t strxfrm (const char * s1, const char * s2, size_t n)	转换字符串并返回长度

9、通用定义（COMMON DEFINTIONS）——stddef.h

无函数（各种定义，包括 size-t,NVLL,ptrdiff_t, offsetof,等等）

10、 整型（INTEGRAL TYPES）——limits.h

无函数（整型的各种极限和大小）。

11、 浮点型（FLOATING-POINT TYPES）-float.h

无函数（浮点型的各种极限和大小）。

12、 错误（ERRORS）-errno.h

无函数（各种错误返回值）。

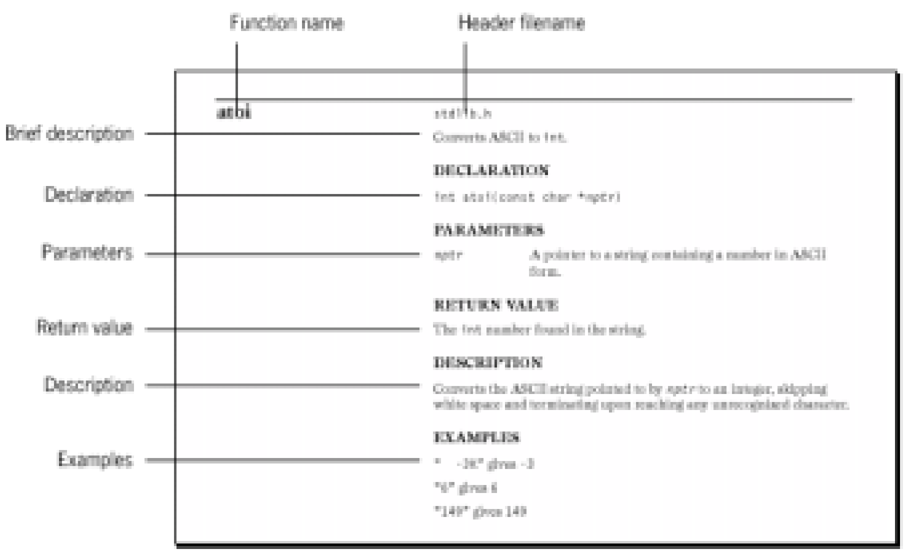
13、 确定（ASSERT）-assert.h

assert void assert (int expression)检查表达式

第9章 C 库函数参考

本章给出 C 库函数按字母排列的表，它们操作的完整说明，以及每一个函数可以使用的选项。

每个函数说明的格式如下：



FUNCTION NAME（函数名）

C 库函数的名称。

HEADER FILENAME（头部文件名）

函数的头部文件名。

BRIEF DESCRIPTION（简要说明）

函数的简明概括。

DECLARATION（声明）

C 库的声明

PARAMETERS（参数）

声明中每一个参数的细节

RETURN VALUE

如果有的话，由函数返回的值。

DESCRIPTION（说明）

包含函数最一般使用的详细说明。这包括关于函数用于什么的信息，以及任何特殊情况和通用 pitfalls 的讨论

EXAMPLES（例子）

说明函数使用的一个或多个例子

abort Stdlib.h

非正常结束程序

声明

Void abort (void)

参数

无

	返回值 无 说明 非正常结束程序并不返回调用者。此函数调用 exit 函数，缺省情况下其入口驻留 CSTARTUP 中。
abs	Stdlib.h 绝对值 声明 int abs (intj) 参数 j 一个 int (整型)值 返回值 具有 j 的绝对值的 int（整型） 说明 计算 j 的绝对值
acos	Math.h 反余弦 声明 double acos (double arg) 参数 arg 在[-1，+1]范围内的 double（双精度） 返回值 arg 的 double(双精度)反余弦值，在[1，Pi]范围内。 说明 计算 arg 的反余弦主值，以弧度计。
asin	Math.h 反正弦 声明 double asin (doble arg) 参数 arg 在[-1，+1]范围内的 double（双精度） 返回值 arg 的 double(双精度)反余弦值，在[-Pi/2,+Pi/2]范围内。 说明 计算 arg 的反余弦主值，以弧度计。
assert	Assert.h 检查表达式 声明 void assert (int expression) 参数 expression 被检查的表达式 返回值 无 说明 这是检查表达式的宏。如果它为假，那么它输出信息至 stderr 并调用 abort。信息具有下列格式： File name; Line num # Assertion failure “expression” 要忽略 assert 调用，在#include<assert.h>语句之前，放置#define NDEBUG 语句。

atan	Math.h 反正切 声明 double atan (double arg) 参数 arg double（双精度）值 返回值 arg 的 double（双精度）反正切值，在[-Pi/2,+Pi/2]范围内。 说明 计算 arg 的反正切值。
Atan2	Math.h 自变量比的反正切 声明 double atan2 (double arg1, double arg2) 参数 arg1 double (双精度)值 arg2 double (双精度)值 返回值 arg1/arg2 的 double（双精度）反正切值，在[-Pi/2, +Pi/2]范围内。 说明 计算 arg1/arg2 的反正切值，使用两个参数的符号以确定返回值所在的象限。
atof	Stdib.h 把 ASCII 转换为 double （双精度）数 声明 double atof (const char*nptr) 参数 nptr 指向包含 ASCII 形式数字字符串的指针。 返回值 字符串中找到的 double（双精度）数。 说明 把 nptr 所指向的字符串转换为双精度浮点数，跳过空白字符并在到达任何不可识符字符时结束 例子 “-3K”给出-3.00 “.0006”给出 0.0006 “1e-4”给出 0.0001
atoi	Stdlib.h 把 ASCII 转换为 int (整型)数 声明 int atoi (const char * nptr) 参数 nptr 指向包含 ASCII 形式数字字符串的指针。 返回值 字符串中找到的 int (整型)数。 说明 把 nptr 所指向的 ASCII 字符串转换为整型数，跳过空白字符并在到达任何不可识别字符时结束 例子 “-3K”给出-3

	“6”给出 6 “149”给出 149
atol	Stdlib.h 把 ASCII 转换为 long int (长整型)数 声明 long atol (const char * nptr) 参数 nptr 指向包含 ASCII 形式数字字符串的指针 返回值 字符串中找到的 long（长整型）数。 说明 把 nptr 所指向的 ASCII 字符串中找到的数转换为长整型值，跳过空白字符并在达到不可识别字符时结束。 例子 “-3K”给出-3 “6”给出 6 “149”给出 149
bsearch	Stdlibb.h 在数组中进行搜索 声明 void * bserach (const void * key, const void * base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_key, const void * _base)); 参数 key 指向要查找对象的指针 base 指向被搜索数组的指针 nmemb base 所指向的数组的元素数 size 数组元素的大小 compare 比较函数，它有两个参数且返回值为： <0(负值) 若_key 小于_base 0 若_key 等于_base >0 (正值) 若_key 大于_base 返回值 结果 值 成功 指向与 key 相符合的数组元素的指针 不成功 null 说明 在一个数组中（该数组由指针 base 指向，元素数为 nmemb）搜索与指针 key 指向的对象相符合的元素。
calloc	Stdlib.h 为数组分配内存 声明 void * calloc (size_t nelem, size_t elsize) 参数 nelem 数组元素数 elsize size-t 类型的值，它指定每一个元素的大小 返回值 结果 值 成功 指针，指向存储器块的开始处（最代地址）

	不成功　零，如果没有所需大小或更大的存储器块可供使用 说明 为给定大小的目标数组分配存储器块。为了确保可移植性，大小不是以诸如字节这样的绝对存储单元给出，而是由 size of 函数返回的 size 或 sizes 给出。存储器的可用性取决于缺省的堆大小（heap size），参见“heap size(堆大小)。一节
ceil	Math.h 大于或等于 arg 的最小整数 声明 double ceil (double arg) 参数 arg double (双精度)值 返回值 具有大于或等于 arg 的最小整数值的 double (双精度)值 说明 计算大于或等于 arg 的最小整数值。
cos	Math.h 余弦 声明 double cos (double arg) 参数 arg 以弧度计的 double（双精度）值 返回值 arg 的 double（双精度）余弦值 说明 计算 arg 弧度的余弦
cosh	Math.h 双曲余弦 声明 double cosh (double arg) 参数 arg 以弧度计的 double（双精度）值 返回值 arg 的 double（双精度）双曲余弦值 说明 计算 arg 弧度的双曲余弦
div	Stdlib.h 除法 声明 div_t div (int numer, int denom) 参数 mumer int (整型)被除数（分子） demon int (整型)除数（分子） 返回值 div_t 类型的结构，它保存相除得到的商和余数 说明 被除数 numet 除以除数 denom。类型 div_t 定义在 stdlib.h 中。 如果除不尽，则商是最接近于代数商的大小较小的整数。 结果定义如下：

quot（商）* denom(除数) +rem(余数—)==numer（被余数）

exit	Stdlib.h 结束程序 声明 void exit (int status) 参数 status int(整型)状态值 返回值 无 说明 正常结束程序。此函数不返回到调用者。缺省情况下，此函数入口驻留在 CSTARTUP
exp	Math.h 指数 声明 double exp (double arg) 参数 arg double (双精度)值 返回值 double(双精度)的 arg 指数函数值 说明 计算 arg 的指数函数
fabs	Math.h 双精度浮点绝对值 声明 double fabs (double arg) 参数 arg double (双精度)值 返回值 arg 的 double(双精度)绝对值 说明 计算浮点数 arg 的绝对值
floor	Math.h 小于或等于（参数）的最大整数 声明 double floor (double arg) 参数 arg double (双精度)绝对值 返回值 小于或等于 arg 的最大整数的 double（双精度）值 说明 计算小于或等于 arg 的最大整数的值。
fmod	Math.h 浮点余数 声明 double fmod (double arg1, double arg2)

	参数 arg1 double (双精度)被除数 arg2 double (双精度)除数 返回值 arg1/arg2 相除得到的 double（双精度）余数 说明 计算 arg1/arg2 的余数，即 arg1-I* arg2 的值，其中 I 是某个整数，若 arg2 非零，则结果的符号与 arg1 的符号相同，大小小于 arg2 的大小。
free	Stdlib.h 释放存储器 声明 viod free (void * ptr) 参数 ptr 指针，它指向以前由 malloc, calloc,或 realloc 分配的存储器块。 返回值 无 说明 释放被 ptr 指向的对象所使用的存储器。Ptr 必须在以前已由 malloc, calloc,或 realloc 赋值。
frexp	Math.h 把浮点数分为两部分 声明 arg1 被划分的浮点数 arg2 指针，指向包含 arg1 指数的整数 返回值 arg1 double（双精度）尾数，在 0.5 至 1.0 的范围内。 说明 把浮点数 arg1 分为两部分：储存在*arg2 中的指数，以及作为函数值返回的尾数 其值如下： mantissa（尾数）$\times 2^{exponent(指数)}$=value（值）
getchar	Stdlib.h 读字符 声明 int getchar (void) 参数 无 返回值 来自标准输入的下一个字符的 int（整型）ASCII 值 说明 从标准输入流读下一个字符。 用户应当为特定的目标硬件配置定制此函数。在文件 getchar.c 中以源格式提供此函数。
gets	Stdlib.h 读字符串 声明 char * gets (char *s) 参数 s 指向将功赎罪接收输入的字符串的指针

	返回值 结果　　　值 成功　　　等于 S 的指针 不成功　　Null（空） 说明 从标准输入读下一个字符串并把它放入所指的字符串。字符串由行的确良结束和文件结束而中止。用零代替 end-of-line（行结束）字符。 此函数调用 getchar，它必须适应特定的目标硬件配置。
isalnum	Ctype.h 是否为字母或数字 声明 int isalnum(int c) 参数 C 代表字符的 int（整型）数 返回值 int （整型）数，若 c 是字母或数字，则返回值非零；否则返回值为零。 说明 测试字符是否是字母或数字
isalpha	Ctype.h 是否为字母 声明 int isalpha (int c) 参数 c 代表字符的 int（整型）数 返回值 int （整型）数，若 c 是字母或数字，则返回值非零；否则返回值为零 说明 测试字符是否是字母或数字
iscntrl	Ctype.h 是否为控制码 声明 int iscntrl (int c) 参数 c 代表字符的 int （整型）数 返回值 int (整型)数，若 c 是控制码，则返回值非零；否则返回值为零 说明 测试字符是否是控制字符
isdigit	Ctype.h 是否为数字 声明 int isdigit (int c) 参数 c 代表字符的 int（整型）数 返回值 int (整型)数，若 c 是数字，则返回值非零；否则返回值为零 说明 测试字符是否是十进制数字。

isgraph	Ctype.h 是否为可打印的非空白字符 声明 int isgraph (int t) 参数 c 代表字符的 int（整型）数 返回值 int (整型)数，若 c 是非空白的可打印字符，则返回值非零；否则返回值为零 说明 测试字符是否是非空白的可打印染字符
islower	Ctype.h 是否为小写字母 声明 int islower (int c) 参数 c 代表字符的 int （整型）数 返回值 int (整型)数，若 c 为小写字母，则返回值为零 说明 测试字符号是否是小写字母
isprint	Ctype.h 是否为可打印字符 声明 int ispunct (int c) 参数 c 代表字符的 int（整型）数 返回值 int (整型)数，若 c 是除了空白、数字或字母之外的可打印字符，则返回值非零；否则返回值为零 说明 测试字符是否是除了空白、数字、或字母之外的可打印字符
isspace	Ctype.h 是否为空白字符 声明 int isspace (int c) 参数 c 代表字符的 int（整型）数 返回值 int （整型）数，若 c 是空白字符，则返回值非零；否则返回值为零 说明 测试字符是否是空白字符，即下列字符之一： 字符　　符号 空格　　‘ ‘ 换页　　\f 换行　　\n 回车　　\r 水平制表　\t 垂直制表　\v

isupper	Ctype.h 是否为大写字母 声明 int isupper (int c) 参数 c 代表字符的 int （整型）数 返回值 int (整型)数，若 c 是大写字母，则返回值非零；否则返回值为零 说明 测试字符是否是大写字母
isxdigit	Ctype.h 是否为十六制数字 声明 int isxdigit (int c) 参数 c 代表字符的 int（整型）数 返回值 int (整型)数，若 c 是大写或小写十六进制数字，则返回值非零；否则返回值为零 说明 测试字符是否是大写或小写的十六进制数字，即 0-9，a-f 或 A-F 之一
labs	Stdlib.h 长整型绝对值 声明 long int labs (long int j) 参数 j long int (长整型)值 返回值 j 的 long int (长整型)绝对值 说明 计算长整形数 j 的绝对值
ldexp	Math.h 乘以 2 的幂 声明 double ldexp (double arg1, double arg2) 参数 arg1 double（双精度）的乘数值 arg2 int（整型）的幂值 返回值 arg1 乘以 2 的 arg2 次幂的 double（双精度）值 说明 计算浮点数乘以 2 的幂的值
ldiv	Stdlib.h 长整型除 声明 ldiv_t ldiv (long int numer, long int denom) 参数 numer long int (长整型)被除数（分子） denom long int (长整型)除数（分母）

	返回值 ldiv_t 类型的结构，它保存相除得到的商和余数 说明 被除数 numer 除以除数 denom。类型 ldiv_t 定义在 stdlib.n 中。 如果除不尽，则商是最接近于代数商的大小较小的整数。结果定义如下： quot（商）*denom（除数）+rem（余数）==numer（被除数）
log	Math.h 自然对数 声明 double log (double arg) 参数 arg double (双精度)值 返回值 arg1r double (双精度)自然对数值 说明 计算一个数的自然对数
Log10	Math.h 以 10 为底数的对数 声明 double log10 (double arg) 参数 arg double（双精度）数 返回值 arg 的 double (双精度)以 10 为底的对数值 说明 计算一个数的以 10 为底的对数值
longjmp	Setjmp.h 长跳转 声明 viod longjmp (jmp_buf env, int val) 参数 env 保存环境的 jmp_buf 类型结构，由 setjmp 设置 val 由相应的 setjmp 返回的 int（整型）值。 返回值 无 说明 恢复先前由 setjmp 保存的环境。这使得程序如从相应的 setjmp 返回一样维续执行，返回数值 val。
malloc	Stdlib.h 分配存储器 声明 void * malloc (size_t size) 参数 size 对象大小的 size_t 类型值 返回值 结果 值 成功 指针，指向存储器块的开始处（最低字节地址） 不成功 零，若无所需大小或更大的存储器块可供使用时 说明

为规定大小的对象分配存储器块。见“堆大小”

---

memchr

String.h

在存储器中搜索字符

声明

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

参数

s　指向目标的指针

c　代表字符的 int（整型）数

n　规定每一个对象大小的 size_t 类型值

返回值

结果	值
成功	指针，指向 s 所指的 n 个字符中 c 首次出现处
不成功	Null(空)

说明

在被指出的给定大小的存储器范围内搜索字符首次出现处

单个字符和对象中的字符均当作 unsigned（无符号）处理

---

mememp

String.h

比较存储器

声明

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

参数

s1　指向第一个对象的指针

s2　指向第二个对象的指针

n　规定每一个对象大小的 size_t 类型值

返回值

整数，它指示由 s1 所指向的对象的首几个字符与由 s2 所指向的对象的首 n 个字符的比较结果

返回值	含义是
>0	s1>s2
=0	s1=s2
<0	s1<s2

说明

比较两个对象的首 n 个字符

---

memcpy

String.h

复制存储器

声明

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

参数

s1　指向目的对象的指针

s2　指向源对象的指针

n　被复制的字符数

返回值

s1

说明

把指定数目的字符从源对象复制到目的对象。

若对象重叠，则结果不确定，所以应代之以使用 memmove。

memmove	String.h 移动存储器 声明 void * memmove (void *s1, , const void *s2, size_t n) 参数 s1　指向目的对象的指针 s2　指向源对象的指针 n　　被复制的字符数 返回值 s1 说明 把指定数目的字符从源对象复制到目的对象。 复制按下述方式发生：好像是首先把源字符复制到一个中间数组，该数组与任一个对象均不重叠；然后把字符从中间数组复制到目的对象。
memset	String.h 设置存储器 声明 void * memset (void *s, int c ,size_t n) 参数 s　指向目的对象的指针 c　代表字符的 int（整型）数 n　对象的大小 返回值 S 说明 把字符（转换为 unsigned char）复制到目的对象开始处规定数目字符中的每一个。
modf	Math.h 分为小数和整数部分 声明 double modf (double value, double * iptr) 参数 value double (双精度)值 iptr　指向 double（双精度）数的指针，它接收 value 的整数部分 返回值 value 的小数部份 说明 计算 value 的小数和整数部分。两部分的符号与 value 的符号相同
pow	Math.h 求幂 声明 double pow (double　arg1, double arg2) 参数 arg1 double (双精度)数 arg2 double （双精度）幂 返回值 arg1 的 arg2 次幂 说明 计算一个数的幂。

printf

Stdio.h
写格式化数据
声明
int printf (const char * format, …)
参数
format 指向格式字符串的指针
…… 在 format 控制下将被打印的可选值
返回值

结果	值
成功	所写的字符数
不成功	若发生错误，则返回负值

说明
把格式化的数据写至标准输出流，返回所写的字符数，若发生错误，则返回负值。因为完整的格式需要大量的空间，所以提供了几种不同的格式以供选择。详见“配置”一章。
Format 是一个字符串，它包括被打印的字符序列和转换规格。每一个转换规格使跟随在 format 字符串之后下一个接连的参数被求值，转换和写。
转换规格的形式如下：
%[flags][field-width][.precision]
[length-modifier] conversion
[ ]内的项是可选的

Flags (标志)
Flags 如下：

Flag	功能
-	左对齐域
+	有符号值总是由正或负号开始
space	数值总是由负号或空格开始
#	替换形式
指定符	功能
octal	第一个数字总是零
G g	打印小数点并保持尾部零
E e f	打印小数点
X	前缀 0X 非零值
x	
o	零填充到域宽度(对于 d, i , o, u, x, X, e, E, f, g 和 G 指定符)。

Fiele-width（域宽度）
Field-width 是域中被打印的字符数。如果需要，用空格填充域。负值表示左对齐域。*的 field width（域宽）代表下一个接连参数的值，它应当是整数。
Precision (精度)
对于整数(d, i, o, u, x, 和 X), precision 是数字个数；对于浮点值（e, E 和 f),是打印的小数位数;对于 g 和 G,是有效数字个数。*的 field width Length modifier（长度修正符）
每一种 length-modifier（长度修正符）的功能如下：

修正符	用途
h	在 d, l, u, x, X 或 o 指定符前表示短整型或无符号短整型值。
l	在 d, l, u, x, X 或 o 指定符前表示长整型或无符号长整型值。
L	在 e, E, f, g, 或 G 指定符前表示长双精度值

Conversion

Conversion (转换)每一个值的结果如下：

Conversion（转换）	结果
d	有符号十进制值
i	有符号十进制值
o	无符号八进制值
u	无符号十进制值
x	无符号十六进制值，使用小写（0-9，a-f）
X	无符号十六进制值，使用大写（0-9，A-F）
E	形式为[-]d.ddd e+dd 的双精度值
f	形式为[-]d.ddd E+dd 的双精度值
g	双精度值，在 f 或 e 两种形式中用较为合适的一种形式
G	双精度值，在 F 或 E 两种形式中用较为合适的一种形式
c	单字符常数
s	字符串常数
p	指针值（地址）
n	无输出，但在下一个参数所指向的整数中存储器写的字符数
%	%字符

注意：提升规则(promotion rules)把所有 char（字符）和 short int(短整型)参数转换为 int（整型），同时把 floats（浮点）转换为 double（双精度）。

Printf 调用库函数 putchar，它必须与目标硬件配置相适应。

Printf 的源代码在文件 printf.c 中提供。使用较少程序空间和堆栈的简化版本的源代码在文件 intwri.c 中提供。

例子

在下列 C 语句之后：

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

不同的 printf 函数调用的效果于下表；△代表空格：

语句	输出	输出字数
printf("%c",p[1])	B	1
printf("%d",i)	6	1
printf("%3d",i)	ΔΔ6	3
printf("%.3d",i)	006	3
printf("%-10.3d",i)	006ΔΔΔΔΔΔΔ	10

printf("Value=%+3d",i)	Value=Δ+6	9
printf("%10.*d",i,j)	ΔΔΔ-000006	10
printf("String=[%s]",p)	String=[ABC]	12
printf("Value=%lX",l)	Value=186A0	11
printf("%f",f1)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",f1)	1.000000e-07	12
printf("%16e",d)	ΔΔΔΔ2.200000e+00	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",f1)	1e-07	5
printf("%g",f2)	750000	6
printf("%g",d)	2.2	3

putchar

Stdio.h
写字符
声明
int putchar (int value)
参数
val 代表要写的字符的 int（整型）值。
返回值

结果	值
成功	value
不成功	EOF 宏

说明
把字符写至标准输出。
用户应当为特定的目标硬件配置定制此函数。在文件 putchar.c 中以源代码格式提供此函数。
此函数由 printf 调用。

puts

Stdio.h
写字符串
声明
int puts (const char*s)
参数
s 指向被写字符串的指针。
返回值

结果	值
成功	非负值
不成功	如果出现错误则返回-1

说明
把后面跟随换行(new-line)字符的字符串写至标准输出流

qsort

Stdlib.h
对数组排序
声明
void qsort (const void * base, size_t nmemb, size_t size, int (*compare)(const void * _key, const viod *_base));
参数
base 指向要排序数组的指针
nmemb base 所指向的数组的元素数。
Size 数组无素的大小
Compare 比较函数，它比较两个参数（自变量）并返回：
<0（负值） 若_key 小于_base
0 若_key 等于_base

	>0（正值） 若_key 大于_base 返回值 无 说明 对 base 所指向的 nmemb 个元素的数组排序。
rand	Stdlib.h 随机数。 声明 int rand (void) 参数 无 返回值 随机数序列下一个 int (整型)值。 说明 计算当前伪随机整数序列中下一个值，转换到在[0，RAND-MAX]范围内。 有关怎样为伪随机序列“seed”的说明。请参见 srand。
realloc	Stdlib.h 重新分配存储器 声明 void * realloc (void * ptr, size_t size) 参数 ptr 指向存储器块起点的指针。 Size 规定目标（对象）大小的 size_t 类型值 返回值 结果 值 成功 指向存储器块起点（最低地址）的指针 不成功 如果没有所需大小或更大的存储器块可供使用，则返回 Null 说明 改变存储器块的大小（它必须由 malloc, calloc,或 realloc 分配）
scanf	Stdio.h 读格式化数据 声明 int scanf (const char * format,…) 参数 format 指向格式字符串的指针 指向将要接收数值的变量的指针 返回值 结果 值 成功 成功转换数 不成功 若输入读完则返回-1 说明 从标准输入读格式化数据。 因为完整的格式符需要许多空间，所以有几种不同的格式符可供选择。详情见“输入和输出”一节。 Format 是包含普通字符（ordinary characters）和转换规格（conversion specifications ）的序列。每一个普通字符从输入读匹配的字符。每一个转换规格接受符合规格的输入，对它进行转换，并把它赋给 format 之后下一个接连的参数所指向的对象。 如果格式字符串包含空白（white-space）字符，那么输入一直扫描到非空白

	字符被找到为止。 转换规格的形式如下： %[assign-suppress][field-width][length-modifier]conversion [ ]内的项是可选的。 Assign suppress (赋值抑制) 如果在此位置包含*，那么域被扫描但不进行赋值。 Field-width（域宽度） Field-width 是被扫描的最大域。缺省值为直到发生不匹配为止。 Length-modifier (长度修正符) 每一个 length-modifier（长度修正符）的功能如下 length modifier 在…之前 意义
l	d, i, 或 n long int （长度型）而不是 int（整型） o, u 或 x unsigned long int (无符号长整型)而不是 unsigned int（无符号整型） e, E, g, G 或 f double（双精度）操作数而不是 float（浮点）
h	d, i, 或 n short int (短整型)而不是 int（整型） o, u 或 x unsigned short int (无符号短整型)而不是 unsigned int（无符号整型） e, E, g, G 或 f long double（长双精度）操作数而不是 float（浮点）
	Conversion（转换） 每一个转换的意义如下： 转换 意义 d 可选有符号十进制整数值。 i 用标准 C 表示法表示的可选有符号整型值，即是十进制、八进制或十六进制。 o 可选有符号八进制整数。 u 无符号十进制整数。 x 可选有符号十六进制整数。 X 可选有符号十六进制整数（等价于 x)。 f 浮点常数。
	转换 意义 eEgG 浮点常数（等价于 f） s 字符串 c 一个或 field-width（域宽）字符。 n 不读，但在下一个参数所指向的整数中存储迄今所读的字符数。 p 指针值（地址)。 [任何数目的字符，它与结尾]之前的任何字符相匹配。例如，[abc]意味着 a, b, 或 c. [ ] 任何数目的字符，它与]或下一个结尾]之前的任何字符相匹配。例如，[ ]a, b ,或 c。 [ˆ 任意数目的字符，它与结尾]之前的任何字符不一致。例如，[ˆabc]意味着不是 a, b, 或 c。 [ˆ] 任何数目的字符，它与]或下一个结尾]之前的任何字符不一至。例如，[ˆ]abc]意味着不是]，a, b, 或 c。 % %字符。
	在除了 c, n 之外的所有转换(conversions)以及所有类型的[中，前导的空白字符被跳过。 Scanf 不直接调用 getchar，它必须适应于实际目标硬件配置。 例子

	在下列程序： int n, i; char name[50]; float x; n=scanf (“%d%f%s”, &i, &x, name) 之后，输入行 25 54.32E-1 Hello world” 将设置变量如下： n=3, i=25, x=5.432, name= “Hello world” 而带有输入行： 56789 0123 56a72 的函数： scanf (“%2d%f%* d%[0123456789]”,&i, &x, name) 将设置变量如下： i=56, x=789.0, name=“56”(0123 未赋)
setjmp	Setjmp.h 设置跳转返回点。 声明 int setjmp (jmp_but env) 参数 env jmp_but 类型的对象，setjmmp 将把环境（environment）存放到它之中， 返回值 零 执行相应的 longjmp 将使程序就好像它是从 setjmp 返回一样继续执行，在此情况下返回在 longjmp 中给出的 int（整型）值。 说明 在 env 中保存环境以供 longjmp 今后使用。 注意：setjmp 必须用在相同的函数或比相应的对 longjmp 调用更高嵌套级别中。
sin	Math.h 正弦 声明 double sin (double arg) 参数 arg 以弧度计的 double（双精度）值。 返回值 arg 的 double（双精度）正弦值。 说明 计算一个数的正弦。
sinh	Math.h 双曲正弦 声明 double sinh (double arg) 参数 arg 以弦度计的 double（双精度）值。 返回值 arg 的 double（双精度）双曲正弦值。 说明 计算 arg 弧度的双曲正弦值。

sprintf	Stdio.h 把格式化数据写到字符串中。 声明 int sprintf (char *s, const char * format,…) 参数 s 指针，指向接收格式化数据的字符串。 Format 指向格式字符串的指针。 … 将在 format 控制下被打印的可选值。 返回值 结果 值 成功 写的字符数 不成功 如果出现错误，则返回负值。 说明 除了直接输出到字符串外，操作完全和 printf 一样。详见 printf。 Sprintf 不使用 putchar 函数，因此对目标配置即使不能使用 putchar,sprintf 也可被使用。 因为完整的格式函数要求大量空间，所以有几种不同的格式可供选择。详细资料请参见“输入和输出”一节。
sqrt	Math.h 平方根 声明 double sqrt (double arg) 参数 arg double（双精度）值。 返回值 arg 的 double（双精度）平方根。 说明 计算一个数的平方根。
srand	Stdlib.h 设置随机数序列。 声明 void srand (unsigned int seed) 参数 seed 识别特定随机数序列的 unsigned int（无符号整型）值。 返回值 无。 说明 选择可重复的伪随机数序列。 函数 rand 用于从序列中取得接连的随机数。如果在调用 srand 之前调用 rand 那么所产生的序列是在调用 srand（1）之后产生的序列。
Sscanf	Stdio.h 从字符串中读格式化数据。 声明 int sscanf (const char*s, const char * format,…) 参数 s 指针，指向包含数据的字符串。 Format 指向格式字符串的指针 … 可选的指针，指向将接收数值的变量。 返回值

结果	值
成功	成功转换数
不成功	若输入取完则返回-1

说明

除了从字符串 s 取输入外，操作与 scanf 完全一样。详见 scantf。

Sscantf 不使用 getchar，所以即使对目标配置不能使用 getchar，sscanf 也可使用。

因为完整的格式函数要求大量空间，所以有几种不同的格式可供选择。更为详细的资料请参见“配置”一章。

strcat

String.h

连接字符串。

声明

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

参数

s1　指向第一个字符串的指针。

s2　指向第二个字符串的指针。

返回值

s1

说明

把第二个字符串的副本附加到第一个字符串的末尾。第二个字符串的初始字符覆盖第一个字符串结尾的空(null)字符。

strchr

String.h

在字符串中搜索字符。

声明

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

参数

c　　代表字符的 int（整数）值

s　　指向字符串的指针。

返回值

如果成功，返回指针，它指向 s 指字符串中首次出现 c（被转换为字符）处。如果由于找不到 c 而失败，则返回 null。

说明

在字符串中寻找字符（被转换为 char）的首次出现。结尾的 null（空）字符被当作字符串的一部分。

Strcmp

String.h

比较两个字符串。

声明

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

参数

s1　指向第一个字符串的指针。

s2　指向第二个字符串的指针。

返回值

比较两个字符串的 int（整型）结果：

返回值	含义
>0	s1>s2
=0	s1=s2
<0	s1<s2

说明

比较两个字符串。

strcoll

String.h

比较字符串。

声明

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

参数

s1　指向第一个字符串的指针。

s2　指向第二个字符串的指针。

返回值

比较两个字符串的 int（整型）结果：

返回值	含义
>0	s1>s2
=0	s1=s2
<1	s1<s2

说明

比较两个字符串。此函数的操作与 strcmp 相同，仅为满足兼容性而提供。

strcpy

String.h

复制字符串。

声明

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

参数

s1　指向目的对象的指针。

s2　指向源字符串的指针。

返回值

s1

说明

把字符串复制到目的对象。

strcspn

String.h

在字符串中跨过被排除的字符。

声明

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

参数

s1　指向从属（subject）字符串的指针。

s2　指向对象（object）字符串的指针。

返回值

从属字符串（s1）的最大起始段的长度，该段完全不包含来自对象字符串(s2)的字符。

说明

查找从属字符串（s1）的最大起始段完全不包含来自对象字符串的字符。

strerror

String.h

给出错误信息字符串。

声明

```
char * strerror (int errnrm)
```

参数

errnum 要返回的错误信息(对应的整型数)

返回值

strerror 是执行定义（implementation-defined）函数。在 MSP430C 编译器中它返回下列字符串。

Errnum	返回的字符串
EZERO	“no error”

EDOM　　"domain error"
ERANGE　　"range error"
Errnum<0 || errnum>max_err_num　　"unknown error"
所有其他数　　"error No.errnum"
说明
返回错误信息字符串

---

strlen

String.h
字符串长度。
声明
size_t strlen (const char*s)
参数
5　　指向字符串的指针
返回值
表示字符串长度的 size_t 类型的对象
说明
寻找字符串中的字符数，不包括结尾的空（null）字符。

---

strncat

String.h
把指定数目的字符与字符串连接。
声明
char*strncat (char *s1,const char *s2, size_t n)
参数
s1　　指向目的字符串的指针。
s2　　指向源字符串的指针。
n　　所用的源字符串的字符数。
返回值
s1
说明
把来自源字符不大于 n 个的起始字符连接到目的字符串的末尾。

---

strncmp

String.h
把指定数目的字符与符串相比较。
声明
int strncmp (const char * s1,const char *s2, size_t n)
参数
s1　　指向第一个字符串的指针。
s2　　指向第二个字符串的指针。
n　　要比较的源字符串的字符数。
返回值
两个定符串不大于 n 个的起始字符的 int（整型）比较结果。

返回值	含义
>0	s1>s2
=0	s2=s2
<0	s1<s2

说明
比较两个字符串不多于 n 个的起始字符。

---

strncpy

String.h
从字符串中复制指定数目的字符。
声明
char * strncpy (char *s1, const char *s2, size_t n)
参数

	s1　　指向目的对象的指针。 s2　　指向源字符串的指针。 n　　要复制的源字符串的字符数。 返回值 s1 说明 把来自源字符不多于 n 个的起始字符复制到目的对象。
strpbrk	String.h 在字符串中寻找任何一个指定字符。 声明 char * strpbrk (const char * s1, const char *s2) 参数 s1　　指向从属字符串的指针。 s2　　指向对象字符串的指针。 返回值 结果　　　值 成功　　指针，指向来自对象字符串的任何字符在从属字符串中首次出现处 不成功　若未找到则返回 null（空） 说明 在一个字符串中搜索来自第二个字符串中的任何字符的任何出现（位置）。
strrchr	String.h 从字符串右边开始寻找字符。 声明 char * strrchr (const char *s, int c) 参数 s　　指向字符串的指针。 c　　代表字符串的 int（整型）数。 返回值 如果成功，则返回一个指针，它指向 s 所指示字符串 c 最后出现处。 说明 在字符串中搜索字符（被转换为 char）最后出现（的位置）。结尾的空（null）被看作字符串的一部分。
strspn	String.h 在字符串中跨过字符 声明 size_t strspn (const char * s1, const char *s2) 参数 s1　　指向从属字符串的指针。 s2　　指向对象字符串的指针。 返回值 从属字符串(s1)的最大起始段的长度，该段完全包含来自对象字符串（s2）的字符。 说明 查找从属字符串（s1）的最大起始段，该段完全包含来自对象字符串（s2）的字符。
strstr	String.h 搜索字符串。 声明 char * strstr (const char*s1,const char *s2)

参数
s1　　指向从属字符串的指针。
s2　　指向对象字符串的指针。
返回值

结果	值
成功	指针，指向从属字符串（s1）中对象字符串(s2)的字符序列（除了结尾的空字符外）首次出现的位置
不成功	如果字符串未找到，则返回 null（空）；如果 s2 指向具有零长度的字符串，则返回 s1。

说明
在一个字符串中搜索来自第二个字符串中的出现。

strtod

Stdlib.h
把字符串转换为 doule（双精度）值。
声明
```
double strtod(const char * nptr, char * *endptr)
```
参数
nptr　　指向字符串的指针。
Endptr　指向字符串指针的指针。
返回值

结果	值
成功	nptr 指向的字符串中浮点常数 ASCII 表示的转换结果（为 double（ 双精度）数），endptr 指向常数之后的第一个字符。
不成功	零，endptr 指示第一个非空白字符。

说明
把一个数 ASCII 表示转换为 double（双精度）值，去掉任何前导空白。

strtok

String.h
断开字符串至标记。
声明
```
char * strtok (char*s1, const char *s2)
```
参数
s1　　指针，指向被断开至标记的字符。
s2　　指向定界符字符串的指针。
返回值

结果	值
成功	指向标记的指针
不成功	零

说明
在字符串 s1 中寻找下一个标记，它被来自定界符字符串 s2 的一个或多个字符隔开。
当用户第一次调用 strtok 时，s1 应当是用户想要断开到标记的字符串。Strtok 保存此字符串。在第一次后续的调用中，s1 应当为 null。Strtok 在它保存的字符串中搜索下一个标记。从一次调用到另一次调用，s2 可以是不同的。
如果 strtok 寻找到标记，那么它返回指向其中第一个字符的指针。否则它返回 null(空)。如果标记不在字符串的末尾，那么 strtok 用空字符（\0）代替定界符。

strtol

String.h
把字符串转换为长整型值。
声明
```
long int strtol (const char * nptr, char * *endptr, int base)
```

参数
nptr　指向字符串的指针。
Endptr　指向字符串指针的指针。
Base　规定基数的 int（整型）值。
返回值

结果	值
成功	对 nptr 所指向的字符串中整型常数的 ASCII 表示进行转换所得的 long int（长整型）结果，endptr 指向常数之后的第一个字符。
不成功	零，endptr 指向第一个非空白字符。

说明
把一个数的 ASCII 表示转换为使用指定基数的 long int（长整型）值，并去除任何前导的空白。
如果基数为零，那么预期的序列是普通的整数。否则预期的序列包含代表整数的数字和字母，其基数由 base 规定（必须在 2 和 36 之间）。字母[a,z]和[A,Z]归于值 10 至 35。若基数为 16，那么十六进制整数的 ox 部分被允许当作起始序列。

strtoul

Stdlib.h
把字符串转换为无符号长整型值。
声明
unsigned long int strtoul (const char * nptr, char * *endptr, base int)
参数
nptr　指向字符串的指针。
Endptr　指向字符串指针的指针。
Base　规定基数的 int（整型）值。
返回值

结果	值
成功	对 nptr 所指向的字符串中整型常数的 ASCII 表示进行转换所得的 unsigned long int（无符号长整型）结果，endptr 指向常数之后的第一个字符。
不成功	零，endptr 指向第一个非空白字符。

说明
把一个数的 ASCII 表示转换为使用指定基数的 unsigned long int（长整型）值，并去除任何前导的空白。
如果基数为零，那么预期的序列是普通的整数。否则预期的序列包含代表整数的数字和字母，其基数由 base 规定（必须在 2 和 36 之间）。字母[a,z]和[A,Z]归于值 10 至 35。若基数为 16，那么十六进制整数的 ox 部分被允许当作起始序列。

strxfrm

String.h
转换字符串并返回长度
声明
size_t strxfrm (char*s1, const chaar *s2, size_t n)
参数
s1　被转换字符串的返回位置。
s2　转换的字符串。
n　放在 s1 中的最大字符数。
返回值
被转换字符串的长度，不包括结尾的空（null）字符。
说明
按以下所述进行转换：若 strcmp 函数用于两个被转换的字符串，则它返回与把 strcoll 函数应用于同样两个原函数所得结果相对应的值。

tan	Math.h 正切 声明 double tan (double arg) 参数 arg 以弧度计的 double（双精度）值 返回值 arg 的 double（双精度）双曲正切值。 说明 计算 arg 弧度的正切值。
tanh	Math.h 双曲正切 声明 double tanh (double arg) 参数 arg 以弧度计的 double（双精度）值 返回值 arg 的 double（双精度）正切值。 说明 计算 arg 弧度的双曲正切值。
tolower	Ctype.h 转换为小写字符 声明 int tolwer（int c） 参数 c 字符的 int（整型）表示。 返回值 对应于 c 的小写字符的 int（整型）表示。 说明 把字符转换为小写。
toupper	Ctype.h 转换为大写字符 声明 int toupper (int c) 参数 c 字符的 int (整型)表示。 返回值 对应于 c 的大写字符的 int（整型）表示。 说明 把字符转换为大写。
Va_arg	Stdarg.h 函数调用中的下一个变量 声明 tpye va_arg (va_list ap, mode) 参数 ap va_list 类型中的一个值 mode 类型名，在类型后加上一个后缀*即可获得指向一个具有特定类型对象的指针的类型名。 返回值

	见下述 说明 宏，它在函数调用中用下一个参数的类型 和值扩展表达式。在由 va-start 初始化之后，这是由 parmn 所规定的之后的参数。Va-arg 使 ap 依次传送接连的参数。 关于 va-arg 和相关宏的例子，请参见文件 printf.c 和 intwri.c。
Va_end	Stdarg.h 结束读函数调用参数 声明 void va_end (va_list ap) 参数 ap 指向可变参数(variable_argument)表的 va-list 类型的指针。 返回值 见下述 说明 宏，它便于从函数的正常返回，该函数的可变参数表被对 va_list ap 进行初始化的扩展 va start 所引用。
Va_list	Stdarg.h 变量表类型 声明 char*va_list[1] 参数 无 返回值 见下述 说明 适合于保存 va_arg 和 va_end 所需信息的数组类型。
Va_start	Stdarg.h 开始读函数调用参数 声明 void va_start (va_list ap, parmN) 参数 ap　指向可变参数表（variable_argument list）的 va-list 类型的指针。 ParmN　函数定义中可变参数表内最右参数的识别符。 返回值 见下述 说明 宏，它初始化 va_arg 和 va_end 所使用的 ap。
_formatted_read	Icclbutl.h 读格式化数据 声明 int _formatted_read(const char **line,const char **format,va_list ap) 参数 line　指向扫描数据指针的指针。 Format　指向标准 scanf 格式规格字符串指针的指针。 Ap　指向可变参数表的 va_list 类型的指针。 返回值 成功转换数 说明

	读格式化数据。此函数是 scanf 的基本格式化函数（basic formatter）。 _formatted_read 是同时可再用(reusable)(可用入（reentrant）)的。 注意：如上所述，_formatted_read 的使用需要文件 stdarg.h 中专用的 ANSI 定义的宏。 特别是： ● 必须有 va_list 类型的变量 ap。 ● 在调用_formatted_read 之前，必须有对 va_start 的调用。 ● 在离开当前上下关系之前，必须有对 va_end 的调用。 va_start 的参数必须是可变参数表最左边的形式参数。
_formatted_write	Icclbutl.h 格式化和写数据。 声明 int _formatted_write (const char * *format,void outputf(char,void *),void*sp, va_list ap) 参数 format　指向标准 printr/sprintf 格式规格字符串的指针。 Outputf　指向子程序的函数指针，该子程序实际上写由_formatted_write 所创建的单个字符。 此函数的第一个参数包含实际字符值，第二个是指针，其值总是等价于_formatted_write 的第三个参数。 Sp　指向某些类型数据结构的指针，低级输出函数可能需要这些结构。如果不需要多于字符值的任何东西，仍然要用（void *）0 规定此参数并在输出函数中加以声明。 Ap　指向可变参数表的 va_list 类型的指针。 返回值 所写的字符数 说明 格式化写数据。此函数是 printf 和 sprintf 的基本格式化函数，但是通过它的通用接口可以容易地适应写非标准显示设备。 因为完整的格式需要大量的空间，所以有几种不同的格式可供选择。详细资料请参见“配置”一章。 _formatted_write 是同时可再用（reusable）（可再入（reentrant））的。 注意：如上所述，_formatted_write 的使用需要文件 stdarg.h 中专用的 ANSI 定义的宏。 特别是： ● 必须有 va_list 类型的变量 ap。 ● 在调用_formatted_write 之前，必须有对 va_start 的调用。 ● 在离开当前上下关系之前，必须有对 va_end 的调用。 ● va_start 的参数必须是可变参数表最左边的形式参数。 有关怎样使用_formatted_write 的例子，请参见文件 printf.c。
_medium_read	Icclbutl.h 读除了浮点数之外的格式化数据。 声明 int _medium_read (const char * *line,const char* *format, va_lisst ap) 参数 line　指向扫描数据指针的指针。 Format　指向标准 scanf 格式规格字符串指针的指针。 Ap　指向可变参数表的 va-list 类型的指针。 返回值 成功转换数

	说明 _formatted_read 的简化版本，它只有一半大小，但不支持浮数。更为详细的资料见_formatted_read。
_medium_write	Icclbutl.h 读除了浮点数之外的格式化数据。 声明 int _medium_write (const char *format,void outputf (char,void *),void *sp, va_lisst ap) 参数 Format 指向标准 printf/sprintf 格式规格字符串指针的指针。 Outputf 指向子程序的函数指针，该子程序实际上写由_formatted_write 所创建的单个字符。 此函数的第一个参数包含实际字符值，第二个是指针，其值总是等价于_formatted_write 的第三个参数。 Sp 指向某些类型数据结构的指针，低级输出函数可能需要这些结构。如果不需要多于字符值的任何东西，仍然要用（void *）0 规定此参数并在输出函数中加以声明。 Ap 指向可变参数表的 va_list 类型的指针。 返回值 所写的字符数 说明 _formatted_wrie 的小型版本，它大约只有四分之一大小，且仅使用权用约 15 个字节的 RAM。 _small_write 格式函数只支持下列 int（整型）对象的指定符： % %， %d, %o, %c, %s 和%x 它不支持域宽或精度参数，且若使用了不支持的指定符或修正符也不产生诊断信息。 更进一步的资料见_formatted_write。

第10章　语言扩展

本章概括了 MSP430C 编译器提供的扩展，它们支持 MSP430 微处理器的特殊性能。

10. 1 引言

用三种方法提供扩展：

- 作为扩展关键字。缺省情况下，编译器遵守 ANSI 规格，MSP430 扩展不能使用。命令行选项-e 使得扩展关键字可供使用，因此要保留它们，使其不能被用作变量名。
- 作为#pragma 关键字。它们提供#pragma 伪指令，它们控制编译器分配存储器，编译器是否允许扩展关键字以及编译器是否输出警告消息。
- 作为内在函数(intrinsic functions)。它们提供对非常低级的处理器细节的直接访问。使能内在函数包括在文件 in430.h 中。

10.2 扩 展 关 键 字 摘 要

扩展关键字提供下列工具：

10.2.1 I/O 访问（I/O ACCESS）程序可利用下列数据类型访问 MSP430 I/O 系统：

程序可利用下列数据类型访问 MSP430 I/O 系统：

sfrb, sfrw

10.2.2 非易失性 RAM（NON-VOLATILE RAM）

通过使用下列数据类型修正符，可把变量放在非易失性 RAM 中：no-init。

10.2.3 函数（FUNCTIONS）

要抑制(override)编译器用以调用函数的缺省机构,可使用下列函数修正符之一:interrupt, monitor。

10.3 #PRAGMA 伪指令摘要

#pragma 伪指令提供对扩展特性的控制，同时保留在标准语言句法之内。

注意：不管-e 选项如何，#pragma 伪指令是可供使用的。

以下种类的#pragma 函数是可供使用的：

10.3.1 位域取向(BITFIELD ORIENTATION)

```
#pragma bitfield=default
#pragma bitfield=reversed
```

10.3.2 代码段（CODE SEGMENT）

```
#pragma codeseg (seg-name)
```

10.3.3 扩展控制（EXTENSION CONTROL）

```
#pragma language=default
#pragma language=extended
```

10.3.4 函数属性（FUNATION ATTRIBUTE）

```
#pragma function=default
#pragma function=interrupt
#pragma function=monitor
```

10.3.5 存储器使用（MEMORY USAGE）

```
#pragma memory=constseg (seg-name) [:type]
#pragma memory=dataseg (seg-name) [:type]
#pragma memory=default
#pragma memory=no-init
```

10.3.6 警告消息控制（WARNING MESSAGE CONTROL）

```
#pragma warnings=default
```

```
#pragma warnings=off
#pragma warnings=on
```

10.4 预 定 义 符 号 摘要

预定义符号允许检查编译时（compile-time）环境。

函数	说明
--DATE--	Nmm dd yyyy(月 日 年)格式的当前日期
--FILE--	当前源文件名
--IAR-SYSTEMS-ICC	LAR C 编译器识别符
--LINE--	当前源行号
--STDC--	ANSI C 编译器识别符
--TLD--	目标识别符
--TIME--	hh:mm:ss(时：分：秒)格式的当前时间
--VER--	返回 int（整型）版本号

10.5 内在函数摘要（INTRINSIC FUNCTION SUMMARY）

内在函数允许 MSP430 微处理器非常低级的控制。为了在 C 应用程序中使用它们，包含头部文件 in430.h。内在函数编译到在线（in-line）代码，单个指令或短指令序列均如些。

关于内在函数功能的详细资料，请参见 MSP430 处理器的制造厂文档。

内在（函数）	说明
-args$	返回函数参数数组。
-argt$	返回参数类型
-NOP	Nop（空操作）指令
-EINT	使能中断
-DINT	禁止中断
-BIS-SR	在状态寄存器中设置位
-BIS-SR	在状态寄存器中清除位
-OPC	插入 DW const 伪提令

10.6 其 他 扩展

10.6.1 $字符（$CHARACTER）

为了与 DEC/VMS C 兼容，字符$已被加入识别符有效字符集中。

10.6.2 在编译时使用 SIZEOF

“size of 操作符不能用在#if 和#elif 表达式内”这种 ANSI 特定的限制已被取消。

第11章 扩展关键字参考

本章按字母顺序叙述扩展关键字。

下列通用参数用在几种定义中：

参数

Storage-class	表示可选关键字 extern 或 static
declarator	表示标准 C 变量或函数声明

interrupt

声明中断函数

句法

storage-class interrupt function-declarator

storage-class interrupt [vector] function-declarator

参数

function-declarator　无参数的空（void）函数声明符。

[vector]　无参矢量地址的方框号常数表达式。

说明

interrupt 关键字声明在处理器中断时被调用的函数。

函数必须为空（void）且无参数。

如果规定了矢量，那么函数的地址将插入该矢量。如果没有规定矢量，那么用户必须在矢量表（最好位于 cstartup 模块）中为中断函数提供适当的入口。

例子

提供了某些包含文件,它们定义了特定的中断函数;参见“运行时间库(run-time library)”一节。用下列语句，使用预定义的中断定义它：

```
interrupt void name (void)
{ }
```

其中 name 是中断函数的名字。

```
Interrupt [0×18] void UART-handler (void)
{
if (TCCTL&4)
  receive ( );
else
  transmit ( );
}
```

矢量地址（在本例中为 0×18）是对于 INTVEC 段（0×FFE0）的偏移。此例将把矢量置于 0×FFF8 单元。

monitor

函数执行期间禁止中断

句法

storage-class monitor function-declarator

说明

monitor 关键字使得中断在函数执行期间内被禁止。这允许执行禁止中断(atomic)的操作，例如根据信号（semaphores）的操作，该信号控制多进程资源的访问。在所有其他方面，用 monitor 声明的函数与普通函数等价。

例子

下面 got-flag 例子在测试标志时禁止中断。如果标志未被设置，那么函数将设置它。当退出函数时，中断被置为其原先的状态。

```
char printer_free;                /* printer-free
                                  semaphore */
monitor int got_flag(char *flag)  /* With no interruption */
{
  if (!*flag)                     /* test if available */
  {
    return (*flag = 1);           /* yes - take */
  }
  return (0);                     /* no - do not take */
}

void f(void)
{
  if (got_flag(&printer_free))    /* act only if
                                  printer is free */
  .... action code ....
```

No-init

非易失性（non-volatile）变量修正符。

句法

storage-class no-init declarator

说明

缺省情况下，编译器把变量置于主易失性 RAM 中并在启动时对其进行初始化。No-init 类型修正符使编译器把变量放在非易失性 RAM 区域中要紧的，程序必须被规定在 0×0000 至 0×FFFF 地址范围内。更为详细的资料，请见“非易失性 RAM”一节。

然而，实际使用范围稍小一些（0×0200 至 0×0FFDF）。

例子

下例表示 no-init 修正符有效和无效的用法。

```
no_init int settings[50];   /* array of non-volatile
                               settings */
((no_init far i ;))         /* conflicting type
                               modifiers - invalid */
no_init int i = 1 ;         /* initializer included -
                               invalid */
```

Sfrb

声明单字节 I/O 数据类型的对象。

句法

sfrb identifier=constant-expression

说明

sfrb 表示 I/O 寄存器，它：

- 等价于 unsigned char（无符号字符）
- 只能被直接寻址；即不能使用&操作符。
- 驻留在地址范围 0×00 至 0×FF 内固定位置的某处。

Sfrb 变量的值是地址位于 constant-expression 的 SFR 寄存器的内容。除了一元&（地址）操作符之处，所有用于整型的操作符均可用于 sfrb 变量。

例子

```
sfrb P0OUT = 0x11;             /* Defines P0OUT */
sfrb P0IN = 0x10;
void func()
{
  P0OUT = 4;                   /* Sets entire variable
                               P0OUT = 00000100 */
  P0OUT |= 4;                  /* Sets one bit
                               P0OUT = XXXXX1XX */
  P0OUT &= ~8                  /* Clears one bit
                               P0OUT = XXXX0XXX */
  if (P0IN & 2) printf("ON");  /* Read entire P0IN and
                               mask bit 2 */
}
```

sfrw

声明两字节 I/O 数据类型的对象。

句法

sfrw identifier=constant-expression

说明

sfrw 表示 I/O 寄存器，它：

- 等价于 unsigned char（无符号短整型）
- 只能被直接寻址；即不能使用&操作符。
- 驻留在地址范围 0×100 至 0×1FF 内固定位置的某处。

Sfrw 变量的值是地址位于 constant-expression 的 SFR 寄存器的内容。除了一元&（地址）操作符之处，所有用于整型的操作符均可用于 sfrw 变量。

例子

```
sfrw    WDTCTL = 0x120;        /* Defines watchdog time
                               control register */
void func(void)
{
  WDTCTL = 0x5A08;             /* Set watchdog time
                               control register */
}
```

第12章　#PRAGMA 伪指令参考

本章按照字母顺序叙述#pragma 伪指令。

Bitfields=default

恢复缺省的位域（bitfields）储存次序。

句法

#pragma bitfields=default

说明

使编译器按其正常次序分配位域。见 bitfields=reversed。

Bitfields=reversed

倒转位域的储存次序。

句法

#pragma bitfields=reversed

说明

使编译器从域的最高有效位，而不是从域的最低有效位开始分配位域。

ANSI 标准允许储存顺序是与执行有关的，所以用户可以用这个关键字以避免可移植性问题。

例子

存储器中下列结构的缺省布局如下图所示：

```
struct
{
 short a:3;      /* a is 3 bits */
 short :5;       /* this reserves a hole of 5 bits */
 short b:4;      /* b is 4 bits */
} bits;          /* bits is 16 bits */
```

15 – 12	11 – 8	7 – 3	2 – 0
hole (4)	b: 4	hole (5)	a: 3

与此相比较，下列结构具有下图所示的布局：

codeseg	设置代码段名字 句法 #pragma codeseg (seg-name) 其中 seg-name 规定段的名字，它必须不与数据段发生冲突。 说明 此伪指令把后续的代码放在所命名的段内，它等价于使用-R 选项。此 pragma 只能被编译器执行一次。 例子 下例把代码段定义为 ROM： #pragma codeseg (ROM)
Function=default	把函数定义恢复到缺省类型。 句法 #pragma function=default 说明 取消 function=interrupt 和 function=monitor 伪指令 例子 下面的例子规定外部函数 f1 是中断函数,而 f3 是正常函数. #pragma function=interrupt extern void f1(); #pragma function=default extern int f3(); /* Default function type */
Function=interrupt	使函数定义为 interrupt 句法 #pragma function=interrupt 说明 此伪指令使后续函数定义为 interrupt 类型。这是函数属性 interrupt 的另一种（形式）。 注意：#pragma function=interrupt 不是提供矢量选项。 例子下例表示中断函数 process-int(此函数的地址必须放入 INTVEC 表中)。

```
#pragma function=interrupt
void process_int()       /* an interrupt function */
{
  ...

}
#pragma function=default
```

Function=monitor

使函数定义为不可中断（atomic）。

句法

#pragma function=monitor

说明

使后续的函数定义为 montior 类型。这是函数属性 montior 的另一种（形式）。

例子

下述函数 f2 执行时将暂时禁止中断。

```
#pragma function=monitor
void f2()          /* Will make f2 a monitor function */
{
"...
}
#pragma function=default
```

Language=default

把扩展字的可用性恢复为缺省情况。

句法

#pragma language=default

说明

使扩展字可用性返回到由 C 编译器-e 选项所设置的缺省情况见 language=extended

例子

见下面例子 language=extended

Language=extended

使扩展字可供使用。

句法

#pragma language=extended

说明

不管 C 编译器-e 选项的状态如何，使扩展关键字可供使用；见“使能语言扩展（-e）”一节。

例子

在下面的例子中，为了禁止中断，规定了 extended 语言修正符。

```
#pragma language=extended
no_init int sys_para;       /* put in non-volatile RAM */
#pragma language=default
int mycount;
```

Memory=constseg

缺省情况下把常量引至所命名的段。

句法

#pragma memory=constseg(seg-name)

说明

缺省情况下把常量引至所命名的段。

后续声明隐含地取得存储类 const。


```
#pragma memory=constseg(TABLE)
char arr[] = {6, 9, 2, -5, 0};
#pragma memory = default
```

关键字 no-init 和 const 可以覆盖设置(override placement)。
段必须不是编译器保留段名之一。
例子
下例把常量数组 arr 放入 ROM 段 TABLE 中。

Memory=dataseg

缺省情况下把变量引至所命名的段。
句法
#pragma memory=dataseg(seg-name)
说明
缺省情况下把变量引至所命名的段。
缺省情况可被关键字 no-init 和 consst 所抑制。
如果省略，那么变量将被置于 UDATA0（未初始化变量）或 IDATA0（初始化变量）。
变量定义中不会提供初始化值。在任何给定模块中可以定义多达 10 个不同的替换数据段。用户可以在程序的任何点切换到任何一个以前定义的数据段。
例子
下面的例子把三个变量放入名为 USART 的读/写区域。

```
#pragma memory=dataseg(USART)
char USART_data;              /* offset 0 */
char USART_control;           /* offset 1 */
int USART_rate;               /* offset 2, 3 */
#pragma memory = default
```

Memory=default

把对象的存储器分配恢复到缺省区域。
句法
#pragma memory=default
说明
把对象的存储器分配恢复到缺省区域。
后续未初始化数据分配到 UDATA0，初始化数据在 IDATA0 中。

Memory=no-init

缺省情况下把变量引入 NO-INIT 段。
句法
#pragma memory=no-init
说明
把变量引入 NO-INIT 段，所以它们将不被初始化并将驻留在非易失性 RAM 中。这是存储器属性 no-init 的另一种（形式）。缺省情况可被关键字 const 所抑制。
NO-init 段必须被连接以符合非易失性 RAM 的物理地址；详细情况请见“配置”一章。
例子
下面的例子把变量 buffer 放入未初始化存储器。变量 i 和 j 放入 DATA 区域。

```
#pragma memory=no_init
char buffer[1000];        /* in uninitialized memory */
#pragma memory=default
int i,j;                  /* default memory type */
```

	注意：如果遇到函数声明，那么非缺省(mon-default)存储器#pragma 将产生错误信息。局部变量和参数不能驻留在不同于其缺省段，堆栈的任何其他段中。
Warnings=default	把编译器警告输出恢复到缺省状态。 句法 #pragma warnings=default 说明 使编译器警告消息的输出返回到由 C 编译器-w 选项设置的缺省状态。参见#pragma warnings=on 和#pragma warnings=off。
Warnings=off	关闭编译器警告输出。 句法 #pragma warnings=off 说明 不管 C 编译器-w 选项的状态如何，禁止编译器警告消息的输出；见“禁止警告（-w）”一节。
Warnings=on	接通编译器警告输出。 句法 #pragma warnings=on 说明 不管 C 编译器-w 选项的状态如何，使能编译器警告消息的输出；见“禁止警告(-w)”一节。

第13章　预定义符参考

本章给出有关编译器预定义符号的参考资料。

--DATE--	当前日期 句法 --DATE-- 说明 以 mmm dd yyyy 形式返回编译的日期。
--FILE--	当前源文件名 句法 --FILE-- 说明 返回当前正在被编译的文件名。
--IAR-SYSTEMS-ICC	IAR C 编译器识别符 句法 --IAR-SYSTEMS-ICC 说明 返回数字 1。可以用#ifdef 测试此符号以检测是否正在由 IAR 系统 C 编译进行编译。
--LINE--	当前源文件行号

句法

--LINE--

说明

返回作为 int（整型数）的当前正在被编译的源文件的当前行号。

--STDC--

IAR C 编译器识别符

句法

--STDC--

说明

返回数字 1，可以用#ifdef 测试此符号以便检测是否正由 ANSI C 编译器编译。

--TID--

目标识别符

句法

--TID--

说明

目标识标符包含以下几部分：对每一个 IAR 系统 C 编译器为唯一的（即对每一个目标为唯一的）数，内在标志(intrinsic flag)，-V 选项的值，以及对应于-m 选项的值：

31　16　15　14　8 7　4　3　0

（未用）	内在支持	目标识别符，对每一个目标处理器为唯一	-v 选项值（若支持）	-m 选项值（若支持）

对于 MSP430，目标识别符（target-IDENT）为 43。

--TID--的值结构如下：

（0×8000|（t<<8）|(v<<4)|m

用户可以提取值如下：

```
f = (__TID__) & 0x8000;
t = (__TID__ >> 8) & 0x7F;
v = (__TID__ >> 4) & 0xF;
m = __TID__ & 0x0F;
```

注意：在宏名的每一端有两条下划线。

要寻找当前编译器的 target-IDENT(目标识别符)的值，执行：

printf(“%ld”, (--TLD-->>8) &0×7F)

关于--TLD--使用的例子，参见文件 stdarg.h。

为了表示编译器识别内在函数（intrinsic functions），在 MSP430 C 编译器中，最高位 0×8000 被设置。这可能对用户怎样写头文件有影响。

--TIME--

当前时间

句法

--TIME--

说明

以 hh:mm:ss(时：分：秒)形式返回编译时间。

--VER--

返回编译器版本号

句法

--VER--

说明

返回作为 int（整型数）的编译器版本号。

例子

下列输出版本为 3.34 的消息。

```
#if--VER--= =334
#message “compiler version 3.34”
#endif
```

第14章　内在函数参考

本章给出关于内在函数（intrinsic functions）的参考资料。

要使用内在函数，需包含头文件 in430.h。

-args $

返回函数数组。

句法

-args $

说明

-args $是保留字，它返回字符数组（char *），该数组包含当前函数的形式参数的说明表：

偏移量	内容
0	-argt $形式的参数 1 的类型
1	以字节计的参数 1 的大小
2	-argt $格式的参数 2 的类型
3	以字节计的参数 2 的大小
2n-2	-argt $格式的参数 n 的类型
2n-1	以字节计的参数 n 的大小
2n	\0

大于 127 的大小被报告为 127。

-args $只能在函数定义中使用。关于使用-args$的例子，请参见文件 stdarg.h。如果规定了可变长度（varargs）参数表，那么认为参数表结束于最后一个显式（explicit）参数；用户不能简单地决定可选参数的类型或大小。

-args $

返回参数的类型

句法

-argt $ (V)

说明

返回值及其相应的含义示于下表。

值	类型
26	Unsigned char (无符号字符)
27	Char（字符）
28	Unsigned short（无符号短整型）
29	Short（短整型）
30	Unsigned int（无符号整型）
31	Int（整型）

下面例子使用-argt $并测试 integer（整型）或 long（长整型）参数类型。

-BIC-SR	清除状态寄存器位。
	句法
	unsigned short –BIC-SR(unsigned short mask)
	说明
	使用 BIC mask, SR 指令清除位。返回更新前 SR 的内容。
	例子
	/* disable interrupts */
	old_SR=_BIC_SR(0x08);
	/* restore interrupts */
	_BIS_SR(old_SR);

-BIS-SR	设置状态寄存器位。 句法 unsigned short-BIS-SR(unsigned short mask) 说明 使用 BIS mask，RS 指令设置位。返回更新前 SR 的内容。 例子 /*enter low power mode LPM3 */ -BIS-SR（0Xc0); 说明 使用 DINT 指令禁止中断。
-EINT	使能中断

	句法 -EINT（void） 说明 使用 EINT 指令使能中断。
-NOP	执行 NOP（空操作）指令 句法 -NOP（void） 说明 执行 NOP（空操作）指令
-OPC	执行 DW 常数伪指令 句法 -OPC(const unstigned char) 说明 插入 DW 常数。

第15章　汇编语言接口

MSP430 C 编译器允许汇编语言模块与经编译的 C 模块相结合。这特别适用于小型对时间要求严格的子程序，这种子程序要求用汇编语言编写，然后从 C 主程序中调用。本章叙述 C 主程序和汇编语言子程序之间的接口。

15.1 创建 shell

推荐的创建具有正确接口的汇编语言子程序的方法是将从用 C 编译器创建汇编语言源程序开始。用户可以容易地把子程序的函数体加到这个 shell。

Shell 源程序仅需要声明所需的变量并完成对它们的简单访问，例如：

```
int k;
int foo(int i, int j)
{
  char c;
  i++;          /* Access to i */
  j++;          /* Access to j */
  c++;          /* Access to c */
  k++;          /* Access to k */
  return i;
}
void f(void)
{
  foo(4,5);     /* Call to foo */
}
```

然后此程序编译如下：

icc430 shell –A-q-L

-A 选项创建汇编语言输出，-q 选项包含 C 源代码行作为汇编程序注释，-L 选项创建列表.

结果是汇编源文件 shell.s43，它包含声明、函数调用、函数返回、变量返回，以及列表文

件 shell.lst。

下节详细叙述接口。

15.2 调用约定

15.2.1 寄存器使用

编译器使用两组处理器寄存器。

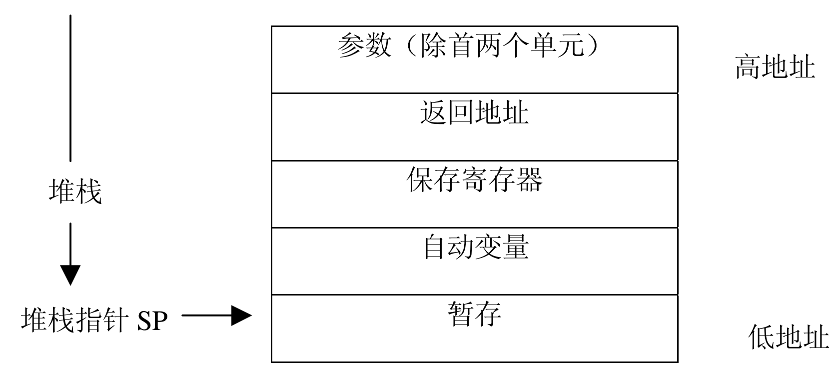
暂存寄存器（scratch registeers）R12 至 R15 用于参数传送，因此在调用之间通常不被保留。

其他通用寄存器 R4 至 R11 主要用于寄存变量和中间结果，且在调用之间必须加以保留。

注意：-ur 选项阻止编译器使用寄存器 R4 和/或 R5。

15.2.2 堆栈结构和参数传送

每次函数调用创建堆栈结构如下：



依照右至左的次序把参数传送到汇编子程序。最左两具参数被传送到寄存器，除非它们是 struct（结构）或 union（联合），在这种（即最左两个参数是结构或联合）情况下，它们也被传送到堆栈上。其余参数总是传送到堆栈上。以下面的调用为例：

f(w, x, y, z)

因为参数按照从右至左的次序被处理，所以 z 将首先被装到堆栈上；后随 y, x, 根据它们类型，或是在 R14，R15，R14 或是在堆栈；w 也是如此。结果在 R12（对于 32 位类型在 R13：R12）中返回，如果是 struct/union（结构/联合）将在 R12 所指向的特定区域中返回。

以上所述可概括于下表中：

参数	<32 位类型	32 位类型	结构/联合
第 4 个（z）	在堆栈上	在堆栈上	在堆栈上
第 3 个(y)	在堆栈上	在堆栈上	在堆栈上
第 2 个(x)	R14	R15：R14	在堆栈上
第 1 个(w)	R12	R13：R12	在堆栈上
结果	R12	R13：R12	特定区域

15.2.3 INTERRUPT（中断）函数

中断函数保存暂存储器和 SR（状态寄存器）以及寄存器 R4 至 R11。

状态寄存器被保存作为中断调用过程的一部分。然后使用 PUSH Rxx 指令保存子程序所用的

任何寄存器。退出时，用 POP Rxx 指令恢复这些寄存器并利用 RTI 指令重新装载状态寄存器并从中断返回。

15.2.4 MONITOR（监控）函数

在（使用）monitor 指令的情况下，进入时编译器用 PUSH SR 保存处理器状态，并用 DINT 禁止中断。退出时，它用 RTI 重新装载状态寄存器（和原先的中断标志）并返回。

15.3 从 C 调用汇编子程序

要从 C 中调用的汇编子程序必须：

- 遵守上述调用约定。
- 具有 PUBLIC 入口。(entry-point)标记。
- 如在 extern int foo（）或 extern int foo (int, int j)中那样，在任何调用前被声明为 external，允许参数的类型检查和可选择的提升。

15.3.1 局部储存分配

如果子程序需要局部储存，可以用下列方式中的一个或多个进行分配：

- 在硬件堆栈上。
- 在静态工作空间，当然这是假设不要求子程序是同时可重用(re-usable)或“可重入”（“re-entrant”)。

函数总是可使用 R12 至 R15 且不需要保存它们，如果在使用之前入栈也可使用 R6 至 R11。对于 ROM 监控兼容代码，不应使用 R4 和 R5。

如果用-ur45 编译 C 代码，但应用程序不运行在 ROM 监控之中，那么在汇编子程序中使用 R4 和 R5 而不保存它们是可能的，这时因为 C 代码将不使用它们。

15.3.2 中断函数

因为在前台（foreground）函数调用期间内可能发生中断，所以调用约定不能用于中断函数。因此，如下所述，对中断函数子程序的要求不同于对正常函数子程序的要求：

- 子程序必须保留所有已使用的寄存器，包括暂存（scratch）寄存器 R12——R15。
- 子程序必须用 RTI 退出。
- 子程序必须把所有标志作为未定义的处理。

15.3.3 定义中断矢量

如上所述，作为以汇编语言定义 C 中断函数这种代替方法，用户可以免于汇编中断子程序并把它直接安装在中断矢量中。

中断矢量定位于 INTVEC 段中。

第16章 段参考

MSP430 C 编译器把代码和数码放入已命名的段中，它们由 XLINK 所指出。为了编程汇编语言

模式，需要段的详情，当说明编译器的汇编语言输出时，它也是有用的。

本章提供按字母顺序排列的段的列表。

对于每个段，它显示：

- 段的名字。
- 内容的简要说明。
- 段是读/写还是只读的。
- 段是可以从汇编语言访问(assembly-accessible,汇编可访问的)还是只能从编译器访问。
- 段的内容和使用的较完整的说明。

存储器映象图

下图表示 MSP430 存储器映象以及在每一个存储器区域内段的分配。

地址	段	说明
FFFF FFE0	INTVEC	矢量
FFDF	CCSTR	当使用-y 选项时，字符串文字初始化程序
	CSTR	C 字符串文字
	CONST	用于储存常数对象
	CDATA0	用于 IDATA0 中变量的变量初始化程序
8000	CODE	保存用户程序

地址	段	说明
7FFF	CSTACK	堆栈
	NO-INIT	保存未初始化(no-init)变量
	ECSTR	字符串文字的可写副本
	UDATAO	未专门初始化的变量
0000	IDATAO	保存从 CDATAO 初始化了的变量

DATA（数据）

CCSTR	字符串文字 类型 只读 说明 汇编可访问的（assembly-accessible)。 保存 C 字符串文字。启动时此段被复制到 ECSTR。更为详细的资料，请参见 C 编译器可写字符串(writable strings)(-y)选项，见“代码产生”一节。也可参见“CSTR” “ECSTR”。
CDATAO	适用于位于 IDATAO 变量的初始化常数，由 CSTARTUP 实现。 类型 只读 说明 汇编可访问的（Assembly-accessible)。 CSTARTUP 把初始化值从引段复制至 IDATAO 段。
CODE	代码 类型 只读 说明 汇编可访问的（Assembly-accessible)。 保存用户程序代码和各种库子程序，注意：任何从 C 调用汇编语言子程序必须符合使用的存储器模块的调用约定。更为详细的资料，请参见“从 C 调用汇编

	子程序”一节。
CONST	常量 类型 只读 说明 汇编可访问的（Assembly-accessible）。 用于储存 const 对象，可在汇编语言子程序中用于声明常量数据。
CSTACK	堆栈 类型 读/写 说明 汇编可访问的（Assembly-accessible）。 保存内部堆栈。 通常在 XLINK 文件中由命令定义此段和长度。 -Z（DATA）CSTACK+nn=start 其中 nn 是长度，start 是地址
CSTR	字符串文字 类型 只读 说明 汇编可访问的（Assembly-accessible）。 当 C 编译器可写字符串（writable strings）(-y)选项未选时（缺省情况），保存 C 字符串文字。更为详细的资料，可参见“可写字符串，常数（-y）”一节。也可参见“CCSTR”“ECSTR”。
ECSTR	字符串文字可写副本。 类型 读/写 说明 汇编可访问的（Assembly-accessible）。 保存 C 字节串文字。更为详细的资料，请参见“可写字符串，常数（-y）一节。还可参见“CCSTR”和“CSTR”。
IDATA0	变量的初始化静态数据。 类型 读/写 说明 汇编可访问的（Assembly-accessible）。 保存内部数据存储器中静态数据，在 cstartup.s43 中它自动被 CDATA0 初始化。参见上述“CDATA0”。
INTVEC	中断矢量 类型 只读 说明 汇编可访问的（Assembly-accessible）。 保存通过使用 interrupt 扩展字产生的中断矢量表（它也可用于用户编写的中断矢量表入口）。它必须位于地址 0×FFE0。
NO-INIT	非易失性变量 类型 读/写 说明

	汇编可访问的（Assembly-accessible)。 保存放在非易失性存储器中的变量。这些变量由编译器分配，声明 no-init 或利用存储器#pragma 创建 no-init，或从汇编语言源程序人工创建
UDATA0	未初始化静态数据 类型 读/写 说明 汇编可访问的（Assembly-accessible)。 保存存储器中变量，它未初显示地初始化；这些变量被隐含地初始化至全零，这是由 CSTARTUP 实现的。

第17章 K&R 和 ANSIC 语言定义

本章叙述 C 语言的 K&R 描述和 ANSI 标准之间的差别。

17.1 引言

有两种主要的标准 C 语言定义：

- kernighan&richie，通常被缩写为 K&R，这是由 C 语言作者给出的原始定义，叙述在它们的著作《The C Programming Language(C 编程语言)》中。
- ANSI

ANSI 定义是原始的 K&R 定义的发展。它增加了可移植性和参数检查的工具，删除了少量多余的关键字。IAR 系统 C 编译器遵守 ANSI 批准的标准 x3.159-1989。

在 Kernighan 和 Richie 所著的《The C Programming Language （C 编程语言)》的量新版本中深入叙述了这两种标准。本章概括了这两种标准之间的差别，对于熟悉 K&R C 但想要使用新的 ANSI 工具的程序员特别有用。

17.2 定义

17.2.1 ENTRY 关键字

在 ANSIC 中 entry 关键字被删除，因而允许 entry 为用户定义符号。

17.2.3 CONST 关键字

ANSI C 增加了 const，它是表示被声明的对象是不可修改的属性，因而可以被编译入只读存储器段。例如：

```
const int i;            /* constant int */
const int *ip;          /* variable pointer to
                        constant int */

int *const ip;              /* constant pointer to
                            variable int */
typedef struct              /* define the struct
                            'cmd_entry' */
{
  char *command;
  void (*function)(void);
}
cmd_entry
const cmd_entry table[]=    /* declare a constant object
                            of type 'cmd_entry' /*
```

17.2.3 VOLATIL 关键字

ANSI C 增加了 volatile，它是指示对象可以被硬件修改，因而任何访问不应被最佳化去除。

17.2.4 SIGNED 关键字

ANSI C 增加了 signed，它是指示整型是有符号的属性。它与 unsigned 相反，可用在任何整型指定符(integer type-specifier)之前。

17.2.5 VOID 关键字

ANSI C 增加了 void，它是可用于声明函数返回值，函数参数，以及通用指针的类型指定符(type-specifier)。例如：

```
void f();               /* a function without return
                        value */
type_spec f(void);      /* a function with no parameters
                        */
void *p;                /* a generic pointer which can be
                        /* cast to any other pointer and
                        is assignment-compatible with any
                        pointer type */
```

17.2.6 ENUM 关键字

ANSI C 增加了 enum，它是能放便地定义具有连续数值的接连的已命名的整型常数。

例始：

```
enum{zero, one, two, step=6, seven, eight};
```

17.2.7 数据类型

在 ANSI C 中完整的基本数据类型集是：

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*               /* Pointer */
```

17.2.8 函数定义参数

在 K&R C 中，函数参数由函数体之前常规的声明语句声明。在 ANSI C 中，参数表中的每一个参数由它的类型 识别符为先导。

例子：

K&R	ANSI
Long int g (s) Char *s; {	Long int g (char*s) {

ANSI 型函数的参数总是进行类型检查(type-checked)。IAR 系统 C 编译器仅在使用 Global strict type check（全局严格类型检查）(-g）选项时才检查 K&R 型函数的参数。

17.2.9 函数声明

在 K&R C 中，函数声明不包括参数，在 ANSI C 中，函数声明包括参数。例如：

类型	例子
K&R	Extern int f ( );
ANSI（命名形式）	Extern int f (long int val);
ANSI（未命名形式）	Extern int f(long int);

在 K&R 情况下，通过声明对函数的调用不能进行参数型检查，假如有参数类型不匹配，调用将失败。

在 ANSI C 情况下，函数参数类型将对照声明中参数的类型进行检查。如果有必要，函数调用的参数将倾向于声明中参数的类型，就像对赋值操作符参数一样。

ANSI 还规定为了表示可变的参数数目，将包括省略号（三个点）作为最终的形式参数。

如果使用对 ANSI 类型函数的外部或前向(forward)引用，那么函数声明应当出现在调用之前。因为对于提升参数（char 或 float）ANSI 和 K&R 类型声明是不兼容的，所以把它们混合是不安全的。

具有可变数目参数的函数必须在调用之前被声明。调用这样的函数是不安全的，除非它已经用包含省略号的原型进行声明。头部文件 stdio.h 包含了标准函数 printf, scanf, 等等的声明。

注意：在 IAR 系统 C 编译器中，-g 选项将找出函数调用和声明之间，包括模块之间所有不兼容问题。

17.2.10 十六进制字符串常数

ANSI 允许 由反斜杠后随时×以及任何数目十六进制数字所表示威者的十六进制常数。例如：

#define Escape-c“\×1b\×43”/* Escape‘C’\0*/

\×43 代表 ASCII C，如果直接被包含，那么它将被认为是十六进制常数的一部分。

17.2.11 结构和联合赋值

在 K&R C 中，函数和赋值运算符的参数可以是指向 struct（结构）或 union（联合）对象的指针，而不是 struct（结构）或 union（联合）对象本身。

ANSI C 允许函数和赋值操作符的参数是 struct（结构）或 union（联合）对象，或者是指向它们的指针。函数也可以返回结构或联合：

```
struct s a,b;                /* struct s declared earlier
                             */
struct s f(struct s parm);   /* declare function
                             accepting and returning
                             structs */
a = f(b);                    /* call it */
```

为了进一步增加结构的可用性，ANSI 允许自动变量被初始化。

17.2.12 共享变量对象

各种 C 编译器在其对各模块之一间共享变量对象的处理上有所不同。IAR 编译 C 编译器使用称之为“strict REF/DEF”的方案，它在 ANSI 辅助文档 Rationale For C 中被介绍。它要求除了 1（号）以外所有模块在变量声明之前使用关键字 extern。例如：

Module #1	Module #2	Module #3
Int I;	Extern int I;	Extern int I;
Int j=4;	Extern int j;	Extern int j;

17.2.13 #elif

ANSI C 的新型#elif 伪指令允许较多紧凑嵌套 else-if 结构。

```
#elif expression
……
```

等价于：

```
#else
#if expression
……
#endif
```

17.2.14 #error

提供#error 伪指令以便和条件编译结合在一起使用。当找到#error 伪指令时，编译器将发出错误消息并结束（运行）。

第18章　诊　断

诊断错误和警告消息可分为六类：

- 命令行错误消息。
- 编译错误消息。
- 编译警告消息。
- 编译致命错误码消息。
- 编译存储器溢出消息。
- 编译内部错误消息。

命令行错误消息

当编译器发现命令行上给出的参数错误时出现命令行错误。在此情况下，编译器发出自说明（self-explanatory）消息。

编译器错误消息

当编译器发现明显违反 C 语言规则的结构，以致不能产生代码时，将产生编译错误消息。

IAR C 编译器在兼容性问题上比许多其它编译器更为严格。特别是当没有显式提升指针和整型（数据）时，它们被看作是不兼容的。

编译警告消息

当编译器发现对编译有影响但没有严重到妨碍编译完成的编程错误或省略时，产生编译警告消息。

编译致命错误消息

当编译器已发现不仅妨碍代码产生，而且使得源代码的进一步处理已经没有意义的状况时，

产生编译致命错误消息。在发出此消息后，编译中止。编译致命错误在本章“编译错误消息（compilation error messages）”一节中叙述，并标记为 fatal（致命的）。

编译存储器溢出消息

当编译器运行至超出存储器范围之外，它发出特定的消息：

COMPILER OUT OF MEMORY

Dynamic memory used:nnnnnn bytes

如果出现这种错误，解决办法是增加系统存储器或把源文件分为较小的模块。还要注意：下列选项使编译器使用较多的存储器：

选项	命令行
Insert mnemonics(插入助记符)	-q
Cross-reference（交叉引用）	-x
Assemby output to prefixed filename(汇编输出到预定义的文件名)	-A
Generate PROM able code（产生可存入 PROM 的代码）	-p
Generate debug information（产生调试信息）	-r

更为详细的资料，请参见《MSP430 Command line interface Guide(MSP430 命令行接口指南)》。

编译内部错误信息

编译器内部错误信息表示由于编译器本身的错误，例如内部一致性检查的错误而出现严重的非预料的错误。在发出自说明（self-explanatory）消息后，编译器中止 C 运行。内部错误通常不应发生且应向 IAR 系统技术支持小组报告。用户的报告应当包括有关问题的所有可能的信息，并最好还有一个磁盘，它带有产生内部错误的程序。

18. 1 编译错误信息

下表列出了编译错误信息：

编号	错误信息	建议
0	Invalid syntax(无效句法)	编译器不能对语句或声明译码
1	Too deep #include nesting(max is 10)(#include 嵌套太深)（最大为 10）	致命错误。编译器对#include 文件的嵌套极限被超过。一个可能的原因是由于疏忽而使用了递归的#include 文件。
2	Failed to open #include file ‘name’(打开#include 文件 ‘name’ 失败)	致命错误。编译器不能打开#include 文件。可能的原因是文件不存在于规定的目录（可能由于错误的-I 前缀或（-INCLUDE 路径）或者被禁止读。
3	Invalid #include filename(无效的#include 文件名)	致命错误。#include 文件名无效。注意：#include 文件名必须被写为<file>或“file“。
4	Unexpected end of file encauntered(遇到未预期的文件结束)	致命错误。在声明，函数定义，或在宏扩展期间内遇到文件的结束。可能的原因是错误的（）或{}嵌套。
5	Too long source line(max is 512 chars); truncated(源代码行太长（最大为 512 个字符），被截断)	源代码行长度超过编译器极限。
6	Hexadecimal constant without digits(十六进制常数无数字)	发现十六进制常数的前缀 0x 或 0X，但没有后随的十六进制数字。
7	Character constant larger than“long”（字符常数大于“long”）	字符常数包含太多的字符，以致不能在长整型的空间中安置。
8	Invalid character encountered: ‘\xhh’;ignored(遇到无效字符:	发现未包含在 C 字符集中的字符。

	‘\xhh’;被忽略	
9	Invalid digits in octal constant(无效浮点常数)	发现浮点心常数太大或具有无效句法。关于合法形式，见 ANSI 标准。
10	Invalid digits in octal constant 入进制常数中有无效数字	编译器在八进制常数中发现非八进制数字。有效的八进制数字为 0-7
11	Missing delimiter in literal or character constant(在文字或字符常数中遗漏定界符)	在字符或文字常数中未发现结束定界符。
12	String too long(max iss 509)(字符串太长（最大为 509）)	超过单个或连接字符串的长度极限。
13	Argument to #define too long(max is 512)(#define 的参数太长（最大为 512）	结束的行使#define 行太长
14	Too many formal paramenters for #define (max is 127)(#define 形式参数太多（量大为 127）)	致命错误。在宏定义（#define 指示符中发现太多的形式参数。
15	‘,’or’)’expected(预待‘,’或’)’)	编译器发现函数定义头部或宏定义的句法无效。
16	Identifier expected(期待识别符)	在声明、goto 语句或预处理器行中丢失识别符
17	Space or tab expected(期待空格或制表符)	必须用制表或空格字符把预处理器参数与伪指令隔开。
18	Macro parameter ‘name’ redefined(宏参数‘name’被重定义)	#define 语句中符号的形式参数被重复。
19	Unmatched #else, #endif 或#elif（不匹配的#else, #endif 或#elif）	致命错误。丢失了#if,#ifdef 或#ifdef
20	No such pre-processor command: ‘name’(无这样的预处理器命令)	#后随未知的识别符
21	Unexpected token found in pre-processor line（在预处理器行中发现未预期的记号）	在读参数部分之后预处理器行不空。
22	Too many nested parameterized macros(max is 50)(嵌套参数化宏太多	致命错误。预处理器极限被超过。
23	Too many active macro parameters(max is 250)(有效宏参数太多（最大为 250）)	致命错误。预处理器极限被超过。
24	Too deep macro nesting(max is 100)(宏嵌套太深（最大为 100）)	致命错误。预处理器极限被超过。
25	Macro ‘name’ called with too many parameters（用太多的参数调用宏‘name’）	致命错误。用比声明更多的参数调用参数化的#define 宏。
26	Actual macro parameter too long (max is 512)(实际宏参数太长（最大为 512）	单个宏参数不能超过源代码行的长度
27	macro ‘name’called with too few parameters(用太少的参数调用宏‘name’)	用比声明更少的参数调用参数化#define 宏
28	Missing #endif (遗漏#endif)	致命错误。在虚假条件之后跳跃文本期间内遇到文件结束。
29	Type specifier expected(期待类型指定符)	丢失类型说明。这可能发生在 struct(结构)，union（联合)，原型函数定义/声明，或在 K&R 函数形式参数声明中。
30	Identifier unexpected(未预期的识别符)	有无效的识别符。这可能是在如下类型名定义：sizeof(int * ident);中的识

		别符或两个接连的识别符。
31	Identifier ‘name’ redeclared(识别符‘name’被重新声明)	存在重新声明识别符。
32	Invalid declaration synatan(声明句法无效)	存在不能译码的声明
33	Unbalanced‘(’or‘)’in declarator(在声明中‘(’或‘)’不平衡)	在声明中存在括号错误
34	C statement or fune-def in #include file, add“I”to the “-r” switch(在#include 文件中 C 语句或函数定义，把“I”加至 “-r”开关)	当使用 C-SPY 调试器时,为了在#include 代码中得到合适的 C 源代码行，必须规定-ri 选项。其他源代码调试器（不使用 UBROF 输出格式）不能用#include 文件中的代码工作。
35	Invalid declaration of “struct”, “union”or “enum”type（无效的“struct”, “union”or “enum”类型声明）	结构，联合或枚举后随无效的标记
36	Tag identifier ‘name’redeclared(重新声明的标签识别符‘name’)	结构，联合或枚举。标签在当前范围内已被定义。
37	Function ‘name’declared with –in “struct” or “union”（在“struct” 或“union”中函数‘name’被声明）	函数被声明为 struct（结构）或 union（联合）的成员
38	Invalid width of field(max is nn)(无效的域宽度（最大为 nn）)	声明的域宽度超过整数的大小（根据目标处理器，nn 是 16 或 32）
39	‘,’ or ‘;’ expected(期待 ‘,’ 或 ‘;’)	在声明的末尾遗漏了 ‘,’ 或 ‘;’
40	Array dimension outside of “unsigned int”bounds（数组维数超过无符号整型数的边界）	数组大小为负或大于无符号整数所能表达的范围
41	member‘name’of “struct” or “union”redeclared（“struct”or“union”的成员‘name’被重新声明）	结构或联合的成员被重新声明。
42	Empty “struct” or “union”(空“struct”或 “union”)	存在不包含成员的结构或联合的声明
43	Object cannot be initialized(对象不能被初始化)	试图对 type 声明或 struct（结构）或 union（联合）成员进行初始化
44	‘;’ expected(期待 ‘;’)	语句或声明需要结束的分号
45	‘]’ expected(期待 ‘]’)	数组声明或数组表达式错
46	‘:’ expected(期待 ‘:’)	在 default, case 标号后或在? -运算了符中遗漏了冒号
47	‘(’ expected(期待 ‘(’)	可能原因是形式有误的 for, if 或 while 语句
48	‘)’ expected(期待 ‘)’)	可能原因是形式有误的 for, if 或 while 语句或表达式
49	‘,’ expected(期待 ‘,’)	存在无效的声明
50	‘{’ expected(期待 ‘{’)	存在无效的声明或初始化
51	‘}’ expected(期待 ‘}’)	存在无效的声明或初始化
52	Too many local variables and formal paramaters(max is 1024)（内部变量和形式参数太多（最大为 1024)）	致命错误。超过编译器极限。
53	Declatator too complex (max is 128 ‘(’ and/or ‘*’)）（声明太复杂（最大为 128	声明包含太多的（,)，或*

	个‘(’ 和/或‘*’))	
54	Invalid storage class(无效的存储类)	规定了无效的对象存储类
55	Too deep block nesting(max is 50)(块嵌套太深（最大为 50）	致命错误。函数定义中{}嵌套太深
56	Array of functions(函数数组)有效形式是：指向函数的指针数组 int aray [5](); int (*array [5])( );	企求声明函数数组 /*无效*/ /*有效*/
57	Missing array dimension specifier(遗漏了数组大小指定符)	存在遗漏指定大小的多维数组声明只有第一个大小可以被去除（在 exyern 外部）数组的声明和函数形式参数中）
58	Identifier ‘name’ redefined(识别符‘name’被重新定义)	存在声明识别符的重定义
59	Function returning array(函数返回数组)	函数不能返回数组
60	Function definition expected(期待函数定义)	发现没有后续函数定义的 K&R 函数头部，例如：int f (I);/*invalid*/
61	Missing identifier in declartion(在声明中遗漏了识别符)	声明缺少识别符
62	Simple variable or array of a “void” type(“void”类型的简单变量或数组)	只有指针，函数，以及形式参数可以是 viod 类型
63	Function returning function（函数返回函数）	函数不能返回函数，如 int f ( ) ( );/* invalid */
64	Unknown size of variable object ‘name’(变量对象‘name’大小未知)	所定义的对象具有未知的大小，这可能是没有给定大小的外部数组或只部分声明的 struct（结构）或 union（联合）对象。
65	Too many errors encountered(>100)(遇到太多的错误（>100))	致命错误，在一定数目的诊断消息后，编译器宣告失败。
66	Function ‘name’redefined(函数‘name’被重新定义)	遇到函数的多重定义
67	Tag ‘name’ urodefined（标签‘name’未定义）	存在类型未定义的 enum 变量定义或在函数原型中对未定义的 struct（结构）或 union（联合）的引用
68	“case” outside “switch”(在“switch”之外的“case”)	存在没有任何有效 switch 语句的 case。
69	“interrupt” function may not be referred or called（“interrupt”(中断)函数不能被引用或调用）	Interrupt（中断）函数调用被包含在程序内。中断函数只能被实时系统调用。
70	Duplicated “case” label:nn（双重“case”标号：nn）	多于一次地把同一个常数值用作 case 的标号
71	“default” outside “switch”（ “default” 在“switch”之外）	存在没有任何有效 switch 语句的 default
72	Multiple “default” within “switch”(在“switch”语句中有多个“default”)	在一条 switch 语句中有多于 1 个的 default
73	Missing “while” in “do”-“white” statement（在“do”-“white” 语句中遗漏了解“while”）	可能原因是在多条语句周围遗漏了{ }。
74	Label ‘name’ redefined（标号 ‘name’被	在同样函数中标号多于一次被定义

	重新定义）	
75	“continue”outside iteration sratement(“continue”在迭代语句之外)	在任何有效的 while，do...while 或 for 语句之外存在 continue
76	“break” outside “switch”or iteration sratement （“break”在 “switch”或迭代语句之外)。	在任何有效的 switch，while，do...while 或 for 语句之外存在 break
77	Undefined label ‘name’（未定义的标号‘name’）	在函数体内存在没有 label:定义的 goto label

78	Pointer to a field not allowed（不允许指向域的指针）
	Struct
	{
	int * f: b /*invalid */
	}
		存在指向 struct（结构）或 union（联合）的域成员的指针

79	Argument of binary operator missing（遗漏了二元运算符的参数）	遗漏了二元运算符的第一个或第二个参数。
80	Statement expected（期待语句）	在期待语句的地方发现了？:，]或}之一
81	Declaration after statement（声明在语句之后）这可能是由于不想要的；而引起，例如： Int I;; Char c; /*invalid*/ 由于第 2 个；是语句，它导致了在语句之后的声明	在语句之后发现了声明
82	“else” without preceding “if”（“else”没有前导的 “if”）	可能的原因是错误的{ }嵌套
83	“enum”constant (s) outside “int”or “unsigned int” range (“enum”常数在“int”或 “unsigned int”范围之外)	把枚举常数创建得太小或太大
84	Function name not allowed in this context（在此范围中不允许函数名）	企图把函数名用作间接地址
85	Empty “struct”, “union”or “enum”（空的“struct”, “union”或 “enum”）	存在不包含成员的 struct（结构）或 union（联合）的定义或不包含枚举常数的 enum（枚举）定义。
86	Invalid formal parameter(无效的形式参数)可能的原因是： Int f (); /* valid K&R declaration */ Int f (i); /* invalid K&R declaration */ Int f (inti); /* valid ANSI declaration */ Int f (i); /* invalid ANSI declaration */	在函数声明中存在形式参数错误
87	Redeclared formal parameter: ‘name’(重新声明的形式参数：‘name’)	在 K&R 函数定义中形式参数多于一次被定义。
88	Contradictory function declaration（矛盾的函数声明）	Void 和其他类型的指定符一起出现在函数参数类型列表中
89	“...” without previous parameter（s）	...不能是唯一规定的参数描述

	（“…”没有前面的参数）例如： int f (…); /*lnvalid */ int f (int, …); /*lnvalid */	

90	Formal parameter identifier missing（遗漏形式参数识别符）例如：
	
	int f(int *p, char, float ff) /* Invalid - second parameter has no name */
	{
	/* function body */
	}
		在原型函数定义的头部遗漏了参数识别符

91	Redeclared number of formal parameter（重新声明形式参数的个数） 例如： int f(int,char); /* first declaration -valid */ int f(int); /* fewer parameters -invalid */ int f(int,char,float);/* more parameters -invalid */	以不同于第一次声明的参数个数声明了原型函数。
92	Prototype appeared after reference（原型出现在引用之后）	函数的原型声明出现在它被定义或作为K&R 函数被引用之后。
93	initializer to field of width nn(bits) out of range（对宽度为 nn（位）的域的初始化超出了范围）	用太大的常数对位域进行初始化，以致不能安放大镜在域空间之内
94	Fields of width 0 must not be named（不能命名宽度为 0 的域）	零长度的域只能用于把域与下一个 int（整型）边界对齐，不能通过标别符访问
95	Second operand for division or modulo is zero（除法的第二个操作数或模数为零）	企图除以零
96	Unknown size of object pointed to（指针所指向的对象大小未知）	在必须知道大小的表达式中使用了不完整的指针类型。
97	Undefined “static” function ‘name’（未定义的“静态”函数‘name’）	函数用 static（静态）存储类型声明，但从未定义。
98	Primary expression expected（预期基本表达式）	遗漏了表达式
99	Extended keyword not allowed in this context（在此范围内不允许扩展的关键字）	扩展的处理器特定关键字出现在非法的范围内，例如 interrupt inti
100	Undeclared identifier: ‘name’（未声明的识别符：‘name’）	存在对未声明的识别符的引用
101	First argument of ‘.’ operator must be of “struct” or “union”type（‘.’运算符的第一个参数必须具有：“struct” 或 “union”类型）	点运算符.被用于不是 struct（结构）或 union（联合）的参数。
102	First argument of ‘→’ was not pointer to “struct” or “union”（‘→’的第一个参数不是指向“struct” 或 “union”指针）	箭头运算符→被应用于不是指向 struct（结构）或 union（联合）指针的参数。
103	Invalid argument of “size of” operator	Size of 运算符.被应用于未知其大小

	(“size of”运算符的无效果参数)	的位域，函数或外部数组。
104	Initializer “string” exceeds array dimension（初始化“字符串”超出数组大小） 例如： char array [4]= “abcde”; /*invalid */	具有显式元素数的 char 数组用超出数据组大小的字符串初始化
105	Language feature not implemented（语言特性未执行）	In-line 函数需要常量参数或常数指针
106	Too many function parameters（max is 127）(函数参数太多（最大为 127）)	致命错误。在函数声明/定义中有太多的参数

107	Function parameter ‘name’ already declared（函数参数 ‘name’已经被声明）
	例如：
	/* K&R function */ int myfunc(i, i) /* invalid */
	int i;
	{
	}
	/* Prototyped function */
	int myfunc(int i, int i) /* invalid */
	{
	}
		函数定义头部中形式参数已多于一次被声明

108	Function parameter ‘name’ declared but not found in header（函数参数‘name’被声明但在头部中未找到）例如：
	int myfunc(i)
	int i, j /* invalid - j is not specified in the function header */
	{
	}
		在 K&R 函数定义中，参数已被声明，但在函数头部中未被规定

109	‘;’unexpected (有未预期的‘;’)	遇到未预期的定界符
110	‘)’unexpected (有未预期的‘)’)	遇到未预期的定界符
111	‘{’unexpected (有未预期的‘{’)	遇到未预期的定界符
112	‘,’unexpected (有未预期的‘,’)	遇到未预期的定界符
113	‘:’unexpected (有未预期的‘:’)	遇到未预期的定界符
114	‘[’unexpected (有未预期的‘[’)	遇到未预期的定界符
115	‘(’unexpected (有未预期的‘(’)	遇到未预期的定界符
116	Integral expression required（需要浮点表达式）	被求值的表达式产生错误类型的结果
117	Floating point expression required（需要浮点表达式）	被求值的表达式产生错误类型的结果
118	Scalar expression required（需要比例表达式）	被求值的表达式产生错误类型的结果
119	Point expression required（需用指针表达式）	被求值的表达式产生错误类型的结果
120	Arithmetic expression required（需用算术表达式）	被求值的表达式产生错误类型的结果
121	Lvalue required（需要存储器地址）	表达式结果不是存储器地址
122	Modifiable lvalue requierd（需要可修	表达式结果不是变量对象或 const（常

	改的 Lvalue）	量）
123	Prototyped function argument number mismatch（原型函数参数数目不符）	用许多不同于所声明数目的参数调用原型函数
124	Unknown “struct”or “union”member: ‘name’(未知的“struct”或“union”成员：‘name’)	企图引用不存在的 struct（结构）union（联合）成员
125	Attempt to take address or field（企图取域的地址，	&操作符不可用于位域
126	Attempt to take address or variable（企图取 “register”（寄存器）变量的地址）	&操作符不可用于具有 register（寄存器）存储类别的对象
127	Incompatible pointers（不兼容的指针）	指针指向的对象必须具有完全的兼容性

特别是，如果指针指向（直接或间接）原型函数，那么代码对返回值并对参数数目及其类型实行兼容性测试。这意味着非兼容性可以被深度隐藏，例如：

```
char (*(*p1)[8])(int);
char (*(*p2)[8])(float);

/* p1 and p2 are incompatible - the function
parameters have incompatible types */
```

如果数组元素个数出现在所指对象的描述中，那么兼容性测试也包括对其的检查，例如：

```
int (*p1)[8];
int (*p2)[9];

/* p1 and p2 are incompatible - array dimensions
differ */
```

128	Function argument incompatible with its declaration（函数参数与它的声明不兼容）	函数参数与声明中的参数不兼容
129	Imcompatible operands of binary operator（二元运算符操作数不兼容）	二元运算符的一个或多个操作数的类型与运算符不兼容
130	Imcompatible operands of ‘=’ operator（‘=’运算符操作数不兼容）	=的一个或多个操作数的类型与=不兼容
131	Imcompatible “return” expression(‘=’表达式不兼容)	表达式的结果与 return 值的声明不兼容
132	Imcompatible initializer（初始化不兼容）	初始化表达式的结果与被初始化的对象不兼容
133	Constant value required（要求常数值）	在 case 标号，#if, #elif,位域声明，数组声明或静态初始化中的表达式不是常数
134	Unmatching “struct” or “union” arguments to ‘?’ operator（‘?’运算符的“struct”或 “union”参数不匹配）	？运算符的第二个和第三个参数不同
135	“pointer+pointer” operation（“指针+指针” 操作）	指针不能相加
136	Redecllaration error（重新定义错误）	当前声明与同一对象早先的声明不一致
137	Reference to member of underfined “struct” or “union”（引用未定义的	对未定义的 struct（结构）或 union（联合）声明唯一允许的引用是指针

	“struct” 或“union”的成员）	
138	“-pointer” expression（“-pointer”表达式）	只有在两个操作对象均为指针时，-运算符才可用于指针，即 pointer-pointer（指针-指针）。此错误意味着发现了 non-pointer-pointer（非指针-指针）形式的表达式。
139	Too many “extern” symbols declared (max is 32767)(声明了太多的 “extern”符号（最大为 32767）	致命错误。超过编译器的极限。
140	“void” pointer not allowed in this context（在此范围内不允许 “void”指针）	指针表达式例如索引表达式包含 void（空）指针（元素数未知）
141	#error ‘any message’	致命错误。发现预处理器伪指令 #error，告知为了编译此模块在命令行上必须定义某些项。
142	“interrupt” function can only be “void” and have no arguments（中断函数只能为 void 且无参数）	中断函数声明具有非空(non-void)的结果和/或参数，这二者均不允许
143	Too large, negative or overlapping “interrupt” [value] in name（太大，为负或名字中 “interrupt” [value]重叠）	检查所声明的中断函数的[vector](矢量)值。
144	Bad context for storage modifier（storage-class or function)（存储修正符范围错（存储类或函数））	Non-init 关键字只能用于声明静态存储类别的变量，即,non-init 不能用于 typedef 语句或用于函数的自动变量。当找到函数声明时，#pragma memory=non-init 可引起这种错误
145	Bad context for function call modifier（函数调用修正符范围错）	关键字 interrupt, banked 或 monitor 只能用于函数声明。
146	Unkown #pragma identifier（未知的 #pragma 识别符）	发现未知的 pragma 识别符，如果只使用-g 选项，则此错误将结束目标代码的产生
147	Extension keyword “name” is already defined by user（扩展字 “name”已由用户定义）	在执行#pragma language=extended 时，编译器发现所命名的识别符具有与扩展关键字相同的名字。此错误仅在编译器运行在 ANSI 模式时才发出
148	“=” expected（期待“=”）	Sfrb 声明的（sfrb-declared）识别符必须后随=value
149	Attempt to take address of “sfrb” or “bit” variable（试图取“sfrb” 或 “bit” 变量的地址）	&运算符不能用于被声明为 bit 或 sfrb 的变量
150	Illegal range for “sfrb”or “bit” address（非法的“sfrb”或 “bit”地址范围）	地址表达式不是有效的 bit 或 sfrb 地址
151	Too many functions defined in a single modle（在单个模块中定义了太多的函数）	在模块中不能有多于 256 个还在使用的函数。注意：对被声明的函数的数目没有限制
152	‘.’expected（期待‘.’）	Bit 声明中遗漏了.
153	Illegal context for extended specifier	

	（非法的扩展指定符范围）	
MSP-430 专用错误消息		
编号	警告消息	建议
154	Constant argument reguired（要求常量参数）	非常量参数用于-OPC 内在函数

18.2 编译警告消息

下表列出了编译警告消息：

编号	警告消息	建议
0	Macro 'name' redefined（宏 'name'被重新定义）	用#define 定义的符号被用不同的参数或形式表重新定义
1	Macro formal partameter 'name' is never referenced（宏形式参数 'name' 被重新定义）	#define 形式参数从未在参数字符串中出现
2	Macro 'name' is already#undef（宏 'name'被重新定义）	#undef 被用于不是宏观的符号
3	Macro 'name' called with empty parameter(s)（用空参数调用宏 'name'）	用零长度(zero-length)的参数不清调用在#define 中定义的参数化宏观
4	Macro 'name' is called recursively; not expanded（宏 'name' 被递归调用；不扩展）	递归宏调用使预处理器停止该宏的进一步扩展
5	Undefined symbol 'name' in #if or #elif; assumed zero（在#if 或#elif 中未定义的符号 'name';假设为零）	在#if 或#elif 表达式中把非宏符号作为零来处理被认为是错误的编程实践。使用以下两个中任一个：#ifdef symbol 或#if defined(symbol)
6	Unkown escape seguence（'\c'）;assume' c'（未知的转义序列（'\c'）；假设为'c'）	在字符常数中发现反斜杠（\）或字符文字后随未知的转义字符
7	Nested comment found without using the'c' option（发现嵌套的注释未使用'c'选项）	在注释中发现字符序列/*且被忽略
8	Invalid type-specifier for field; assumed "int"（无效的域类型识别符；假设为（整型）"int"）	在此执行中，位域仅能被指定为 int 或 unsigned int（无符号整型）
9	Undeclared function parameter 'name'; assumed "int"（未声明的函数参数 'name';假设为 "int"）	缺省情况下，K&R 函数定义头部中未声明的识别符被给予类型 int（整型）
10	Dimension of array ignored; array assumed pointer（忽略数组的元素数；数组假设为指针）	具有显式元素数的数组被规定为形式参数，编译器把它作为指向对象的指针来处理
11	Storage class "static" ignored; 'name' declared "extern"(存储类"static"被忽略；'name'被声明为"extern")	对象或函数首先被声明为 extern（外部）(显式地或缺省)，后来被声明为 static（静态）。静态声明被忽略
12	Incompletely bracketed initializer（括号不完整的初始化）	为了避免模糊，初始化应当只使用一层{ }括号或者被{ }括号完整地包围
13	Unreferenced label 'name'（未引用的标号'name'）	括号被定义但从未被引用
14	Type specifier missing; assumed "int"（遗漏了类型指定符；假设为"int"）	在声明中未给出类型指定符—假设 int（整型）

15	Wrong usage of string operator（'#' or '##'）; ignored （字符串运算符（'#' '##'）错误使用；被忽略）此外，#运算符必须先于形式参数：	这限制了#和##运算符用于参数化宏的标记域（token-field）

```
#define mac(p1)    #p1             /* Becomes "p1" */
#define mac(p1,p2) p1+p2##add this /* Merged p2 */
```

16	Non-void function: "return" with <expression>; expected（非空函数：用<表达式>返回）	在所有地方非空（non-void）函数定义应当用已定义的返回值退出
17	Invalid storage class for function; assumed to be "extern"（无效的函数存储类别；假设为"extern"）	无效的函数存储类别一被忽略。有效的类别是 extern， static， 或 typedef
18	Redeclared parameter's storage class（重新声明的参数存储类别）	在后续的声明/定义中函数形式参数的存储类别从 register 变为 auto，或从 auto 变为 register。
19	Storage class "extern" ignored; 'name' was first declared as "static"（存储类 "extern"被忽略； 'name'首先被声明为 "static"	被声明为 static（静态）的识别符后来被显式或隐含地声明为 extern（外部），extern（外部）声明被忽略
20	Unreachable statement(s)（不能到达的语句）例如： Break; I=2; /*never executed */	无条件转移或返回在一条或多条语名之前，使得这条或多条语句从来不会被执行。
21	Unreachable statement(s) at unreachable label 'name'（在未卜先知被引用的标号'name'处不能到达的语句）例如： Break; Here: I=2; /*never executed */	无条件转移或返回在有标号的一条或多条语句之前，但是标号从未被引用，所以这条或多余语句从来不会被执行
22	Non-void function: explicit "return" <expression>; expected（非空函数：显式的 "return"<表达式>；被预期）	非空（non-void）函数产生隐含返回。这可能是从循环或开关语句中非预期的退出。注意：不带 default 的开关语句总是被编译器当作'可退出的'而不管 case 的结构如何
23	Undeclared function 'name'; assumed "extern" "int"（未声明的函数'name'；假设为"extern" "int"）	对未声明函数的引用导致使用缺省的声明。函数被假设为具有 K&R 类型，具有外部存储类别，并返回 int（整型）
24	Static memory option converts local "auto" or "register" to "static"(静态存储器选项把局部 "auto"或 "register"转换为 "static")	用于静态存储器分配的命令行选项使 auto（自动）和 register（寄存器）声明被当作 static（静态）来处理
25	Inconsistent use of K&R function-varying number of parameters(K&R 函数的不一致使用-改变了参数的数目)	用改变了的参数类型调用 K&R 函数
26	Inconsistent use of K&R function-changing type of parameter（K&R 函数	用改变了的参数类型调用 K&R 函数

	的不一致使用改变了参数的类型） 例如： myfunc(34); /*int argument*/ myfunc(34.6); /*float argument*/	
27	Size of “extern” object ‘name’ is unknown（外部对象‘name’的大小未知）	Extern（外部）数组应当用 size 声明
28	Constant [index] outside array bound（常数[索引]超出数组边界）	存在超出已声明数组边界的常数索引
29	Hexadecimal escape sequence larger than “char”（十六进制转义序列大于“char”	转义序列被截断以适合于放入 char（字符）中
30	Attribute ignored（属性被忽略） 例子：	因为 const（常量）或 volatile（易失的）是对象的属性，所以当它们与 structure（结构）,union（联合），或 enumeration(枚举)标签定义一起给出时将被忽略，上述定义没有和对象同时声明。而且，函数被认为不能返回 const 或 volatile

```
const struct s
{
  ...
}; /* no object declared, const ignored - warning */
const int myfunc(void);
/* function returning const int - warning */
const int (*fp)(void); /* pointer to function
returning const int - warning */
int (*const fp)(void);
/* const pointer to function returning int - OK,
no warning */
```

31	Incompatible parameters of K&R functions（K&R 函数的参数不兼容）在下列范围之一使用指针：	指向函数的指针（可能是间接的）或 K&R 函数声明具有不兼容的参数类型

```
pointer - pointer,
expression ? ptr : ptr,
pointer relational_op pointer
pointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter
```

32	Incompatible numbers parameters of K&R functions(K&R 函数的参数数目不兼容)在下列范围之一使用指针：	指向函数的指针（可能是间接的）或 K&R 函数声明具有不同数目的参数

```
pointer - pointer
expression ? ptr : ptr
pointer relational_op
pointerpointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter
```

33	Local or formal ‘name’ was never	在函数定义中未使用形式参数或局部变

	referenced(局部或形式参数‘name’从未被引用)	量对象。
34	Non-printable character ‘\xhh’ found in literal or character constant（在文字或字符常数中发现不可打印字符号‘\xhh’）	在字符串文字或字符常数中使用不可打印（non-printable）字符被认为是一种不好的编程习惯。为了得到同样的结果可使用‘\0xhhh’
35	Old-style(K&R) type of function declarator (老式(K&R)类型的函数声明)	发现老式(K&R)函数声明。只有正在使用-gA 选项时才发出这种警告
36	Floating point constant out of range(浮点常数超出范围)	浮点值太大或太小以致不能使用目标的浮点系统来表示
37	Illegal float operation :division by zero not allowed(非法浮点运算：不允许除以零)	在常数算术运算时发现除零
38	Tag identifier ‘name’ was never defined(从未定义标签识别符‘name’)	
39	Dummy statement. Optimized away!	发现多余的代码。这通常表示用户代码中打印错误或可能产生于使用有点不太通用的宏时（这不是错误）。 例如： a+b;

40	Possible bug! “if” statement terminated(可能是缺陷！“if”语句被中止)	这通常表示用户代码中的打印错误。例如：
		if (a= =b);
		{
		<if body>
		}

41	Possible bug! Unintialized variable（可能是缺陷！未初始化的变量）	在初始化之前使用变量（变量具有随机值）。
		例如：
		void func(p1)
		{
		short a;
		p1+=a;
		}

42		止消息被废弃

43	Possible bug! Integer promotion may cause problems. Use cast to avoid it（可能是缺陷！整数提升可能产生问题。使用 cast 以避免此问题）
	例如：
	short tst(unsigned char a)
	{
	if (-a)
	return (1);
	else
	return (-1);
	}
		整数提升规则指出所有整数运算必须产生这样的结果：当它们具有比 int（整型）低的精度时，就好像它们是 int（整型）一样。这有时可能导致未预期的结果

	此例将始终返回 1，即使对于数值 0xff 也是如此，其原因是整数提升首先使变量 a 变为 0x00ff，然后执行位非（bit not）。 整数提升被许多其他 C 编译器所忽略，因此当用 IAR 系统编译器重新编译已有的程序时，可能产生此警告。	

44	Possible bug! Single ‘=’ instead of ‘= =’used in “if” statement（可能是缺陷！在 “if”语句中用‘= =’ 代替单个‘=’）	这通常表示用户代码中的打印错误。
		例如：
		if (a=1)
		{
		<if body>
		}

45	Redundant expression. Example:multiply with 1. Add with 0(多余的表达式。例如：乘以 1，加上 0)	这可能表示用户代码中的打印错误，但是它也是可能是，由 case 工具产生的错误代码的结果。
46	Possible bug! Strange or faulty expression. Example: division by zero（可能是缺陷！奇怪或错误的表达式。） 例如：除以零）	这通常表示用户代码中的缺陷
47	Unreachable code deleted by the global optimizer（由全局优化删除不能到达的代码） 例如：除以零）	在用户代码中多余的代码块。它可能是 bug（缺陷）的结果，但通常仅是不完善代码的信号
48	Unreachable returns. The function will never return（不能到达的返回，函数将永远不返回）	函数将永远不能返回到调用的函数。这可能是程序缺陷的结果，但通常当在 RTOS 系统中具有永不结束循环时产生。

49	Unsigned compare always true/false（无符号的比较总是为真/假）	这表示用户代码中的缺陷！通常的原因是遗漏了-c 编译器开关
		例如：
		for (uc=10; uc>=uc--);
		{
		<loop body>
		}
		因为无符号的值永远大于或等于零，所以这是永不结束的循环。

50	Signed compare always true/false（有符号的比较总是为真/假）	这表示用户代码中的缺陷！

MSP 专用的警告消息

无。