KEIL SOFTWARE

Cx51 编译器

对传统和扩展的 8051 微处理器的优化的 C 编译器和库参考

翻译者
网名：jxlxh
E-mail: jx_lxh@163.com

原文件：
C51.pdf

www.c51bbs.com
网站协助发布

用户手册 09.2001

译者序

由于本人的英语水平有限，所以在使用 KEIL C51 的过程中，老要去看那英文的手册，总感到不是那么方便，老要用词霸查来查去的，烦的很。因此在看到 C51BBS 上的倡议后，就动了把它翻译出来的念头。我想这对自己和别人都会带来些好处。

利用工作之余的时间，经过几个月的努力，终于把它翻译完了。但由于水平所限，文中肯定有很多不是十分恰当的地方，或许没有用大家比较熟悉的惯用语，或许可能引起误解，所以在这里，我请大家能指出其中的错误和不当之处，请大家 EMAIL 告诉我，使我能够作出改正。对于大家的建议我会很高兴的接受。

我最大的愿望是希望我的翻译不会误导大家，且能对大家有所帮助。

不明之处可以参考英文原文。

感谢 C51BBS 版主龙啸九天的帮助。

欢迎大家与我交流，我的 e-mail：jx_lxh@163.com

Keil Software 声明：

本文档所述信息不属于我公司的承诺范围，其内容的变化也不会另行通知。本文档所述软件的出售必须经过授权或签订特别协议。本文档所述软件的使用必须遵循协议约定，在协议约定以外的任何媒体上复制本软件将触犯法律。购买者可以备份为目的而做一份拷贝。在未经书面许可之前，本手册的任何一部分都不允许为了购买者个人使用以外的目的而以任何形式和任何手段(电子的，机械的)进行复制或传播。

Keil C51™，Keil CX51™,和 uVision（TM）是 Keil Elektronik GmbH 的商标。
Microsoft®和 Windows™是 Microsoft Corporation 的商标或注册商标。
IBM®，PC®，和 PS/2®是 International Business Machines Corporation 的注册商标。
Intel®，MCS®51，MCS®251，ASM-51®，和 PL/M-51®是 Intel 的注册商标。

我们尽全力去做来保证这本手册的正确，从而保证我们个人，公司和在此提及的商标的形象。

前言

本手册讲述对 8051 的目标环境，如何使用 **C*x*51** 优化 C 编译器编译 C 程序。**C*x*51** 编译器包可以用在所有的 8051 系列处理器上，可以在 WINDOWS 32 位命令行中执行。本手册假定你熟悉 WINDOWS 操作系统，知道如何编程 8051 处理器，并会用 C 语言编程。

注意：

本手册用条件窗口来指明 32 位 WINDOWS 版本是 WINDOWS95，WINDOWS98，WINDOWS ME，WINDOWS NT，WINDOWS 2000 或 WINDWOS XP。

如果你对 C 编程有问题，或者你想知道 C 语言编程的更多信息，可参考 16 页的“关于 C 语言的书”。

手册中讨论的许多例子和描述是从 WINDOWS 命令提示符下调用的。这对在一个集成环境如**µVision2** 中运行 **C*x*51** 的情况是不适用的。本手册中的例子是通用的，可以应用到所有编程环境。

手册组织

本用户手册分成下面的章节和附录：

“第一章，介绍”，概述 **C*x*51** 编译器。

“第二章，用 **C*x*51** 编译”，解释怎样用 **C*x*51** 交叉编译器编译一个源文件。本章叙述控制文件处理，编译和输出的命令行提示。

“第三章，语言扩展”，叙述支持 8051 系统结构必须的 C 语言扩展。本章提供一个在 ANSI C 说明中没有的命令，函数，和控制的详细列表。

“第四章，预处理器”，叙述 **C*x*51** 编译器预处理器的组成和包含的例子。

“第五章，派生的 8051”，叙述 **C*x*51** 编译器支持的 8051 派生系列。本章还包括能帮助提高目标程序性能的技巧。

“第六章，高级编程技术”，对有经验的开发人员的重要信息。本章包括定制文件描述，优化器详细资料，和段名约定。本章还讨论了 **C*x*51** 编译器产生的程序和别的 8051 编程语言如何接口。

“第七章，错误信息”，列出了在使用 **C*x*51** 编译器时可能遇到的致命错误，语法错误和警告。

“第八章，库参考”，提高一个扩展的 **C*x*51** 库参考。分类列出了库例程和相关的包含文件。本章最后有一个按字母顺序的参考，包括每个库例程的例子代码。

附录中包含不同编译器版本间的差异，作品编号，和别的有些信息。

文档约定

本文档有下列约定：

例子	说明
README.TXT	粗体大写用在可执行程序名，数据文件名，源文件名，环境变量，和输入WINDOWS命令行的命令上。表示你必须手工输入的文本（不一定要大写）。 例：**CLS DIR BL51.EXE**
Language Elements	C 语言的构成包括关键词，操作符和库函数用粗体。 例：**if != long** **isdigit main >>**
`Courier`	这种字体的文本代表显示在屏幕上或打印出的信息。这字体也用在讨论或描述命令行中。
`Variables`	斜体字必须提供的信息。例如，在语法字符串中的 *projectfile* 表示需要提供实际的工程文件名。
重复的成分…	例子中使用的省略号（…）表示重复的成分。

省略代码
.
.
.
	垂直省略号用在源代码例子中，表示省略一段程序。
	例子：
	void main(void) {
	.
	.
	.
	while(1);

[*可选项*]	命令行中的可选参数和选择项用方括号表示。 例：**C51 TEST.C PRINT** [(***filename***)]
{*opt1*|*opt2*}	大括号中的文本，用竖线分隔，代表一组选项，必须从中选一项。 ■ 大括号中包含了所有选项。 ■ 竖线分隔选项。
Keys	Sans serif 字体的文本代表键盘的键。例如，“按 `Enter` 继续”

Contents

第一章．介绍

C 语言是一个通用的编程语言，它提供高效的代码，结构化的编程，和丰富的操作符。C 不是一种大语言，不是为任何特殊应用领域而设计。它一般来说限制较少，可以为各种软件任务提供方便和有效的编程。许多应用用 C 比其他语言编程更方便和有效。

优化的 **C*x*51** C 编译器完整的实现了 ANSI 的 C 语言标准。对 8051 来说，**C*x*51** 不是一个通用的 C 编译器。它首先的目标是生成针对 8051 的最快和最紧凑的代码。**C*x*51** 具有 C 编程的弹性和高效的代码和汇编语言的速度。

C 语言不能执行的操作（如输入和输出）需要操作系统的支持。这些操作作为标准库的一部分提供。因为这些函数和语言本身无关，所以 C 特别适合对多平台提供代码。

既然 **C*x*51** 是一个交叉编译器，C 语言的某些方面和标准库就有了改变或增强，以适应一个嵌套的目标处理器的特性。更多的细节参考 89 页的“第三章.语言扩展”。

支持所有的 8051 变种

8051 系列是增长最快的微处理器构架之一。从不同的芯片厂家提供了 400 多种芯片。新扩展的 8051 芯片，如 PHILIPS 8051MX 有几 M 字节的代码和数据空间，可被用到大的应用中。

为了支持这些不同的 8051 芯片，KEIL 提供了几种开发工具，如下表所列。一个新的输出文件格式（OMF2）允许支持最多 16MB 代码和数据空间。CX51 编译器适用于新的 PHILIPS 8051MX 结构。

开发工具	支持的微处理器，说明
C51编译器 A51宏汇编 BL51连接器	对传统的8051开发工具，包括支持32 x64KB的代码库
C51编译器（有OMF2输出） AX51宏汇编 LX51连接器	对传统的8051和扩展的8051芯片（如DALLAS 390）的开发工具。包括支持代码库，和最多16MB代码和XDATA存储区。
CX51编译器 AX51宏汇编 LX51连接器	对PHILIPS 8051MX的开发工具,支持最多16MB代码和XDATA存储区。

C*x*51 编译器在不同的包中提供。上表是完整的 8051 开发工具参考。

注意:
Cx51 指两种编译器：C51 编译器和 CX51 编译器。

C 语言的书

有许多书介绍 C 语言。有更多的书详细介绍用 C 完成的任务。下面的列表不是一个完整的列表。列表只是作为参考。

The C Programming Language, Second Edition

Kernighan & Ritchie
Prentice-Hall, Inc.
ISBN 0-13-110370-9

C: A Reference Manual, Second Edition

Harbison & Steel
Prentice-Hall Software Series
ISBN 0-13-109810-1

C and the 8051: Programming and Multitasking

Schultz
P T R Prentice-Hall, Inc.
ISBN 0-13-753815-4

第二章．用 Cx51 编译器编译

本章说明怎样编译 C 源文件，讨论编译器的控制命令。这些命令可以：

- 命令**C*x*51**编译器产生列表文件。
- 控制包含在OBJ文件中的信息的数量。
- 指定优化级别和存储模式。

注意:

*一般来说你应在**µVision2** IDE中使用**Cx51**。关于使用µVision2 IDE的更多信息，参考用户手册“Getting Started with **µVision2** and C51”.*

环境变量

如果在µVision2 IDE 中运行 Cx51 编译器，计算机不需要另外的设置。如果想要在命令行中运行 **Cx51** 编译器和工具，必须手工创建下面的环境变量。

变量	路径	环境变量说明
PATH	\C51\BIN	C51和CX51可执行程序的路径。
TMP		编译器产生的临时文件的路径。如果指定的路径不存在，编译器会生成错误并停止编译。
C51INC	\C51\INC	Cx51头文件的路径。
C51LIB	\C51\LIB	Cx51库文件的路径。

对WINSOWS NT，WINDOWS 2000和WINDOWS XP，这些环境变量在Control Panel – System – Advanced – Environment Variables 中输入。

对 WINDOWS 95，WINDOWS 98 和 WINDOWS ME，这些设置放在 AUTOEXEC.BAT 中：

```
PATH=C：\KEIL\C51\BIN;%PATH%
SET TMP=D:\
```

```
SET C51INC=C:\KEIL\C51\INC
SET C51LIB=C:\KEIL\C51\LIB
```

从命令行运行 Cx51

调用 C51 或 CX51 编译器，在命令行输入 **C51** 或 **CX51**。在命令行中，必须包含要编译的 C 源文件，和必需的编译控制命令。**Cx51** 命令行的格式：

```
C51 sourcefile [directives…]
CX51 sourcefile [directives…]
```

或：

```
C51 @commandfile
CX51 @commandfile
```

这里：

sourcefile 要编译的源文件名。

directives 用来控制编译器功能的命令。参考20页的“控制命令”。

commandfile 包含源文件名和命令的命令输入文件。当**Cx51**调用行较复杂，超过了WINDOWS命令行的限制时，使用*commandfile*。

下面的命令行例子调用 C51，指定源文件 **SAMPLE.C**，用控制 **DEBUG**，**CODE** 和 **PREPRINT**。

```
C51 SAMPLE.C DEBUG CODE PREPRINT
```

Cx51 编译器在成功编译后显示下面的信息：

```
C51 COMPILER V6.10

C51 COMPILATION COMPLETE. 0 WARNING（S），0 ERROR（S）
```

错误级别

在编译后，错误和警告的数目输出在屏幕上。**Cx51** 编译器设置 **ERRORLEVEL** 指示编译的状态。值如下表所列：

错误级别	意义
0	没有错误或警告
1	只有警告
2	错误和可能的警告
3	致命错误

可以在批处理文件中访问 **ERRORLEVEL** 变量。关于 **ERRORLEVEL** 或批处理文件可以参考 WINDOWS 命令索引或在线帮助。

Cx51输出文件

Cx51 编译器在编译时产生许多输出文件。缺省的，输出文件和源文件同名。但，文件的扩展名不同。下面的表列出了文件并有简短的说明。

文件扩展	说明
***Filename*.LST**	列表文件，包含格式化的源文件和编译中检测到的错误。列表文件可以选择包含所用的符号和生成汇编代码。更多的信息，参考**PRINT**命令。
Filename.OBJ	包含可重定位目标代码的OBJ模块。OBJ模块用**Lx51**连接器连接到一个绝对的OBJ模块。
Filename.I	包含由预处理器扩展的源文件。所有的宏都扩展了，所有的注释都删除了。可参考**PREPRINT**命令。
Filename.SRC	C源代码产生的汇编源文件。可以用A51汇编。可参考**SRC**命令。

控制命令

C*x*51 编译器提供许多控制命令控制编译。除了指定的，命令由一个或多个字母或数字组成，在命令行中在文件名后指定，或在源文件中用**#pragma** 命令。例如：

```
C51 testfile.c SYMBOLS CODE DEBUG

#pragma SYMBOLS CODE DEBUG
```

在说明的例子中，SYMBOLS，CODE，和 DEBUG 都是控制命令，testfile.C 是要编译的源文件。

注意：

*对命令行和#**pragma** 语法是相同的。在#**pragma** 可指定多个选项。*

典型的，每个控制命令只在源文件的开头指定一次。如果一个命令指定多次，编译器产生一个致命错误，退出编译。可以指定多次的命令在下面部分注明。

命令种类

控制命令可以分成三类：源文件控制，目标控制，和列表控制。

- 源文件控制定义命令行的宏，定义要编译的文件名。
- 目标控制影响产生的目标模块（*.OBJ）的形式和内容。这些命令指定优化级别或在OBJ文件中包含调试信息。
- 列表控制管理列表文件（*.LST）的各种样式，特别是格式和指定的内容上。

下表按字母顺序列出了控制命令。有下划线的字母表示命令的缩写。

命令	类	说明
AREGS，NOAREGS	Object	使能或不使能绝对寄存器（ARn）地址。
ASM，ENDASM	Source	标志内嵌汇编块的开始和结束。
BROWSE †	Object	产生浏览器信息。
CODE †	Listing	加一个汇编列表到列表文件。
COMPACT †	Object	设置COMPACT存储模式。
COND，NOCOND †	Listing	包含或执行预处理器跳过的源程序行。
DEBUG †	Object	在OBJ文件中包含调试信息。
DEFINE	Source	在Cx51调用行定义预处理器名。
DISABLE	Object	在一个函数内不允许中断。
EJECT	Listing	在列表文件中插入一个格式输入字符。
FLOATFUZZY	Object	在浮点比较中指定位数。
INCDIR †	Source	指定头文件的附加路径名。
INTERVAL †	Object	对SIECO芯片指定中断矢量间隔。
INTPROMOTE，NOINTPROMOTE †	Object	使能或不使能ANSI整数同时提升。
INTVECTOR，NOINTVECTOR †	Object	指定中断矢量的基地址或不使能矢量。
LARGE †	Object	选择LARGE存储模式。
LISTINCLUDE	Listing	在列表文件中显示头文件。
MAXAREGS †	Object	指定可变参数列表的大小。
MOD517，NOMOD517	Object	使能或不使能代码支持80C517和派生的额外的硬件特征。
MODA2，NOMODA2	Object	使能或不使能ATMEL 82x8252和变种的双DPTR寄存器。
MODAB2，NOMODAB2	Object	使能或不使能模拟设备ADuC B2系列支持双DPTR寄存器。
MODDA，NOMODDA	Object	使能或不使能DALLAS 80C390，80C400，和5240支持算法加速器。
MODDP2，NOMODDP2	Object	使能或不使能DALLAS的320，520，530，550和变种支持双DPTR寄存器。
MODP2，NOMODP2	Object	使能或不使能PHILIPS和ATMELWM派生的支持双DPTR寄存器。
NOAMAKE †	Object	不记录μVision2更新信息。
NOEXTEND†	Source	Cx51不扩展到ANSI C。
OBJECT，NOOBJECT†	Object	指定一个OBJ文件或禁止OBJ文件。
OBJECTEXTEND †	Object	在OBJ文件中包含变量类型信息。
ONEREGBANK	Object	假定在中断中只用寄存器组0。

命令	类	说明
OMF2†	Object	产生OMF2输出文件格式。
OPTIMIZE	Object	指定编译器的优化级别。
ORDER†	Object	按源文件中变量的出现顺序分配。
PAGELENGTH†	Listing	指定页的行数。
PAGEWIDTH†	Listing	指定页的列数。
PREPRINT†	Listing	产生一个预处理器列表文件，扩展所有宏。
PRINT , NOPRINT†	Listing	指定一个列表文件名或不使能列表文件。
REGFILE†	Object	对全局寄存器优化指定一个寄存器定义文件。
REGISTERBANK	Object	为绝对寄存器访问选择寄存器组。
REGPARMS, NOREGPARMS	Object	使能或不使能寄存器参数传递。
RET_PSTK† , RET_XSTK†	Object	用重入堆栈保存返回地址。
ROM†	Object	AJMP/ACALL指令产生控制。
SAVE, RESTORE	Object	保存和恢复 AREGS，REGPARMS和OPTIMIZE命令设置。
SMALL†	Object	选择SMALL存储模式（缺省）。
SRC†	Object	产生一个汇编源文件，不产生OBJ模块。
STRING†	Object	定位固定字符串到XDATA或远端存储区。
SYMBOLS†	Listing	模块中所有符号的列表文件。
USERCLASS†	Object	对可变的变量位置重命名存储区类。
VARBANKING†	Object	使能FAR存储类型变量。
WARNINGLEVEL†	Listing	选择警告检测级别。
XCROM†	Object	对CONST XDATA变量假定ROM空间。

† 这些命令在命令行或源文件开头的#pragma 中只指定一次。在一个源文件中不能使用多次。

控制命令和参数，除了用 **DEFINE** 命令的参数，是大小写无关的。

参考

本章的余下部分按字母顺序描述 **C*x*51** 编译器控制命令。他们分成如下部分：

缩写: 可以替代命令的缩写。

参数: 命令可选和要求的参数。

缺省: 命令的缺省设置。

µVision2控制: 怎样指定命令。

说明: 详细的说明命令和使用。

参考: 相关命令。

例子: 命令使用的例子，有时，也列出结果。

AREGS/NOAREGS

缩写: 无

参数: 无

缺省: **AREGS**

µVision2控制: Options – C51 – Don‘t use absolute register access。

说明: **AREGS**控制使编译器对寄存器R0到R7用绝对寄存器地址。绝对地址提高了代码的效率。例如，**PUSH**和**POP**指令只能用直接或绝对地址。用**AREGS**命令可以直接PUSH或POP寄存器。

可用**REGISTERBANK**命令定义使用的寄存器组。

NOAREGS命令对寄存器R0到R7不使能绝对寄存器地址。用**NOAREGS**编译的函数可以使用所有的8051寄存器组。命令可用在被别的函数用不同的寄存器组调用的函数中。

注意:
*虽然可能在一个程序中定义了几次，**AREGS/NOAREGS**选项只有定义在函数声明为有效。*

例子: 下面是一个使用**NOAREGS**和**AREGS**的源程序和代码的列表。

```
stmt level    source
   1          extern char func ();
   2          char k;
   3
   4          #pragma NOAREGS
   5          noaregfunc ()  {
   6   1        k = func () + func ();
   7   1      }
   8
   9          #pragma AREGS
  10          aregfunc ()  {
  11   1        k = func () + func ();
  12   1      }

     ; FUNCTION noaregfunc (BEGIN)
                        ; SOURCE LINE # 6
0000 120000 E   LCALL func
0003 EF         MOV   A,R7
0004 C0E0       PUSH  ACC
0006 120000 E   LCALL func
0009 D0E0       POP   ACC
000B 2F         ADD   A,R7
000C F500   R   MOV   k,A
                        ; SOURCE LINE # 7
000E 22         RET
      ; FUNCTION noaregfunc (END)

      ; FUNCTION aregfunc (BEGIN)
                        ; SOURCE LINE # 11
0000 120000  E  LCALL func
0003 C007       PUSH  AR7
0005 120000  E  LCALL func
0008 D0E0       POP   ACC
000A 2F         ADD   A,R7
000B F500    R  MOV   k,A
                        ; SOURCE LINE # 12
000D 22         RET
      ; FUNCTION aregfunc (END)
```

注意保存R7到堆栈中的不同方法。函数**noaregfunc**产生的代码是:

```
MOV    A，R7
PUSH   ACC
```

同时对 **aregfunc** 函数的代码是:

```
PUSH   AR7
```

ASM/ENDASM

缩写: 无

参数: 无

缺省: 无

µVision2控制: 本命令不能在命令行指定。

说明: **ASM**命令标志一块源程序的开始，它可以直接合并到由**SRC**命令产生的**.SRC**文件中。

这些源程序可以认为是内嵌的汇编。然而，它只输出到由**SRC**命令产生的源文件中。源程序不汇编和输出到OBJ文件中。

在µVision2应对C源文件中包含**ASM/ENDASM**段如下设置一个文件指定选项：

- 右键点击PROJECT窗口 – 文件表中的文件。
- 选择**Options for…**打开选项 – 属性页。
- 使能**Generate Assembler SRC file**
- 使能**Assemble SRC file**

用这些设置，µVision2产生一个汇编源文件（**.SRC**），并用汇编编译产生一个OBJ文件（**.OBJ**）。

ENDASM命令标志一个源程序块的结束。

注意:
***ASM*和*ENDASM*命令只能在源文件中使用，且作为#*pragma*命令的一部分。**

例子： #pragma asm / #pragma endasm

下面是C源文件：

```
.
.
.
stmt level  source
   1        extern void test ();
   2
   3        main ()  {
   4   1      test ();
   5   1
   6   1    #pragma asm
   7   1      JMP   $  ; endless loop
   8   1    #pragma endasm
   9   1    }
.
.
```

产生下面的.SRC文件：

```
; ASM.SRC generated from: ASM.C
NAME  ASM
?PR?main?ASM          SEGMENT CODE
EXTRN  CODE (test)
EXTRN  CODE (?C_STARTUP)
PUBLIC main
; extern void test ();
;
; main ()  {
        RSEG  ?PR?main?ASM
        USING 0
main:
                      ; SOURCE LINE # 3
;   test ();
                      ; SOURCE LINE # 4
                LCALL test
;
; #pragma asm
                JMP   $  ; endless loop
; #pragma endasm
; }
                      ; SOURCE LINE # 9
                RET   ; END OF main
        END
```

BROWSE

缩写: **BR**

参数: 无

缺省: 不创建浏览信息。

µVision2控制: Options – Output – Browse Information

说明: 用**BROWSE**，编译器产生浏览信息。浏览信息包括标识符（包含预处理器符号），他们的存储空间，类型，定义和参考列表。

信息可以在µVision2内显示。选择**View – Source Browser**打开µVision2源浏览器。*参考µVision2用户手册，第四章，µVision2功能，源浏览器。*

例子:

```
C51 SAMPLE.C BROWSE

#pragma browse
```

CODE

缩写: **CD**

参数: 无

缺省: 不产生汇编代码列表。

µVision2控制: Options – Listing – C Compiler Listing – Assembly Code

说明: **CODE**命令附加一个汇编助记符列表到列表文件。汇编程序代码代表源程序中的每个函数。缺省的，在列表文件中没有汇编代码。

例子:

```
C51 SAMPLE.C CD

#pragma code
```

下面例子显示C源程序和它产生的OBJ结果代码和助记符。在汇编间显示了产生代码的行号。字符**R**和**E**代表可重定位和外部的。

```
stmt level  source
   1        extern unsigned char a, b;
   2               unsigned char    c;
   3
   4        main()
   5        {
   6   1      c = 14 + 15 * ((b / c) + 252);
   7   1    }
.
.
.
ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                        ; SOURCE LINE # 5
                        ; SOURCE LINE # 6
0000 E500   E   MOV   A,b
0002 8500F0 R   MOV   B,c
0005 84         DIV   AB
0006 75F00F     MOV   B,#0FH
0009 A4         MUL   AB
000A 24D2       ADD   A,#0D2H
000C F500   R   MOV   c,A
                        ; SOURCE LINE # 7
000E 22         RET
      ; FUNCTION main (END)
```

COMPACT

缩写: **CP**

参数: 无

缺省: **SMALL**

µVision2控制: Options – Target – Memory Model

说明: 本命令选择**COMPACT**存储模式。

在**COMPACT**存储模式中，所有的函数和程序变量和局部数据段定位在8051系统的外部数据存储区。外部数据存储区可有最多256字节（一页）。在本模式中，外部数据存储区的短地址用@R0/R1。

不管什么存储类型，可以在任何8051的存储范围内声明变量。但是，把常用的变量（如循环计数器和数组索引）放在内部数据存储区可以显著的提高系统性能。

注意:
*函数调用所用的堆栈经常放在**IDATA**存储区。*

参考: **SAMLL，LARGE，ROM**

例子:

```
C51 SAMPLE.C COMPACT

#pragma compact
```

COND/NOCOND

缩写: **CO**

参数: 无

缺省: **COND**

µVision2控制: Options – Listing – C Compiler Listing - Conditional

说明: 本命令定义这些部分的受条件编译影响的源程序是否显示在列表文件中。

COND命令在列表文件中包含编译省略的行。行号和嵌套级不输出，以便于阅读。

本命令影响预处理器删除的行。

NOCOND命令不在列表文件中包含编译省略的行。

例子： 下面的例子显示用**COND**命令编译产生的一个列表文件。

```
.
.
.
stmt level  source
   1        extern unsigned char  a, b;
   2               unsigned char     c;
   3
   4        main()
   5        {
   6   1    #if defined (VAX)
              c = 13;
            #elif defined (_ _TIME_ _)
   9   1      b = 14;
  10   1      a = 15;
  11   1    #endif
  12   1    }
.
.
```

下面的例子用 NOCOND 命令编译产生的一个列表文件。

```
.
.
.
stmt level  source
   1        extern unsigned char  a, b;
   2               unsigned char     c;
   3
   4        main()
   5        {
   6   1    #if defined (VAX)
   9   1      b = 14;
  10   1      a = 15;
  11   1    #endif
  12   1    }
.
.
.
```

DEBUG

缩写: **DB**

参数: 无

缺省: 不产生调试信息。

µVision2控制: Options – Output – Debug Information

说明: **DEBUG**命令指示编译器在OBJ文件中包含调试信息。缺省，OBJ文件不包含调试信息。

对程序的符号测试必需有调试信息。信息包括全局和局部变量定义和地址，和函数名和行号。包含在目标模块中的调试信息在连接过程中仍有效。这些信息可以被µVision2调试器或任何INTEL兼容的模拟器使用。

注意:
OBJECTEXTEND*命令用来指示编译器在目标文件中包含附加的变量类型定义信息。*

参考: **OBJECTEXTEND**

例子:

```
C51 SAMPLE.C DEBUG

#pragma db
```

DEFINE

缩写: **DF**

参数: 一个或多个符合C语言约定的的名称，用逗号分隔。对每个名称可有一个参数，用**DEFINE**给出。

缺省: 无。

µVision2控制: 在Options –C*x*51 – Define输入名称。

说明: **DEFINE**命令定义调用行的名称，预处理器要用**#if**，**#ifdef**和**#ifndef**查询这些名称。定义的名称在输入后被复制。这些命令是大小写相关的。作为一个选项，每个名称可跟一个值。

注意:
DEFINE*命令只能在命令行中指定。在一个C源程序中用C预处理器命令****#define****。*

例子:

```
C51 SAMPLE.C DEFINE （check,NoExtRam）

C51 MYPROG.C DF (X1=“1+5”,iofunc=“getkey()”)
```

DISABLE

缩写: 无

参数: 无

缺省: 无

µVision2控制: 本命令不能在命令行中指定，只能在源文件中指定。

说明: **DISABLE**命令指示编译器在产生代码时，在一个函数内不使能所有中断。**DISABLE**必须在一个函数声明前一行用**#pragma**命令指定。**DISABLE**控制只用到一个函数，对每个新的函数必须重新指定。

注意:
DISABLE*只能用****#pragma****命令指定，不能在命令行指定。*

DISABLE *可在一个源文件中指定多次，对每个函数只能指定一次，执行后不使能中断。*

一个不使能中断的函数不能对调用者返回一个位值。

例子: 本例子是一个用**DISABLE**命令函数的源程序和代码列表。注意**EA**指定函数寄存器在函数进入时清除（**JBC EA，？C002**），在结尾时恢复（**MOV EA，C**）。

```
.
.
.
stmt level    source
   1          typedef unsigned char  uchar;
   2
   3          #pragma disable   /* Disable Interrupts */
   4          uchar dfunc (uchar p1, uchar p2)  {
   5   1        return (p1 * p2 + p2 * p1);
   6   1      }

      ; FUNCTION _dfunc (BEGIN)
0000 D3         SETB  C
0001 10AF01     JBC   EA,?C0002
0004 C3         CLR   C
0005  ?C0002:
0005 C0D0       PUSH  PSW
;---- Variable 'p1' assigned to register 'R7' ----
;---- Variable 'p2' assigned to register 'R5' ----
                      ; SOURCE LINE # 4
                      ; SOURCE LINE # 5
0007 ED         MOV   A,R5
0008 8FF0       MOV   B,R7
000A A4         MUL   AB
000B 25E0       ADD   A,ACC
000D FF         MOV   R7,A
                      ; SOURCE LINE # 6
000E  ?C0001:
000E D0D0       POP   PSW
0010 92AF       MOV   EA,C
0012 22         RET
      ; FUNCTION _dfunc (END)
.
.
.
```

EJECT

缩写: **EJ**

参数: 无

缺省: 无

µVision2控制: 本命令不能在命令行中指定，只能在源文件中指定。

说明: **EJECT**命令在列表文件中插入一个格式输入字符。

注意:
***EJECT** 只在源文件中出现，必须是#**pragma** 命令的一部分。*

例子:

```
#pragma eject
```

FLOATFUZZY

缩写: **FF**

参数: 0到7间的一个数字。

缺省: **FLOATFUZZY（3）**

µVision2控制: Options - C*x*51 – Bits to round for float compare

说明: **FLOATFUZZY**命令在一个浮点比较前定义位数。缺省值3指定最少有三个有效位。

例子:

```
C51 MYFILE.C FLOATFUZZY（2）

#pragma FF(0)
```

INCDIR

缩写: 无

参数: 指定头文件的路径。

缺省: 无

µVision2控制: Options - C*x*51 – Include Paths

说明: **INCDIR**命令指定**Cx51**编译器头文件的位置。编译器最多50个路径声明。
如果需要多个路径，路径名必须用分号分开。如果指定#include “filename.h”， **Cx51**编译器首先搜索当前目录，然后是源文件目录。当找不到或用了#include <filename.h>，就搜索**INCDIR**指定的路径。当仍找不到，就使用**C51INC**环境变量指定的路径。

例子: C51 SAMPLE.C INDIR（C：\KEIL\C51\MYINC;C:\CHIP-DIR）

INTERVAL

缩写: 无

参数: 对中断矢量表可选，用括号括住。

缺省: **INTERVAL（8）**

µVision2控制: Options - C*x*51 – Misc controls:enter the directive。

说明: **INTERVAL**命令指定中断矢量的间隔。指定间隔是SIECO-51派生系列要求的，它定义中断矢量在3字节间隔。用本命令，编译器定位中断矢量在绝对地址，如下计算：

***(interval*×*n)+offset*+3,**

这里：

interval **INTERVAL**命令的参数（缺省为8）。

n 中断号。

offset **INTVECTOR**命令的参数（缺省为0）。

参考: **INTVECTOR/NOINTVECTOR**

例子:

```
C51 SAMPLE.C INTERVAL（3）

#pragma interval(3)
```

INTPROMOTE/NOINTPROMOTE

缩写: **IP/NOIP**

参数: 无。

缺省: **INTPROMOTE**

µVision2控制: Options - C*x*51 – Enable ANSI integer promotion rules。

说明: **INTPROMOTE**命令使能ANSI整数提升规则。如果提升声明了，在比较前所用的表达式从小类型提升到整数表达式。这使MICROSOFT C和BORLAND C改动很少就可用到**Cx51**上。

因为8051是8位处理器，使用**INTPROMOTE**命令可能在某些应用中降低效率。

NOINTPROMOTE命令不使能自动整数提升。整数提升使**Cx51**和别的ANSI编译器间有更大的兼容性。然而，整数提升可能降低效率。

例子:

```
C51 SAMPLE.C INTPROMOTE

#pragma intpormote

C51 SAMPLE.C NOINTPROMOTE
```

下面的代码示范用**INTPROMOTE**和**NOINTPROMOTE**命令产生的代码。

```
stmt lvl  source

   1        char c;
   2        unsigned char  c1,c2;
   3        int  i;
   4
   5        main ()  {
   6  1       if (c == 0xff) c = 0;    /* never true! */
   7  1       if (c == -1) c = 1;      /* works */
   8  1       i = c + 5;
   9  1       if (c1 < c2 +4) c1 = 0;
  10  1     }
```

Code generated with INTPROMOTE

```
       ; FUNCTION main (BEGIN)
                ; SOURCE LINE # 6
0000 AF00           MOV  R7,c
0002 EF             MOV  A,R7
0003 33             RLC  A
0004 95E0           SUBB A,ACC
0006 FE             MOV  R6,A
0007 EF             MOV  A,R7
0008 F4             CPL  A
0009 4E             ORL  A,R6
000A 7002           JNZ  ?C0001
000C F500           MOV  c,A
000E            ?C0001:
                ; SOURCE LINE # 7
000E E500           MOV  A,c
0010 B4FF03         CJNE A,#0FFH,?C0002
0013 750001         MOV  c,#01H
0016            ?C0002:
                ; SOURCE LINE # 8
0016 AF00           MOV  R7,c
0018 EF             MOV  A,R7
0019 33             RLC  A
001A 95E0           SUBB A,ACC
001C FE             MOV  R6,A
001D EF             MOV  A,R7
001E 2405           ADD  A,#05H
0020 F500           MOV  i+01H,A
0022 E4             CLR  A
0023 3E             ADDC A,R6
0024 F500           MOV  i,A
                ; SOURCE LINE # 9
0026 E500           MOV  A,c2
0028 2404           ADD  A,#04H
002A FF             MOV  R7,A
002B E4             CLR  A
002C 33             RLC  A
002D FE             MOV  R6,A
002E C3             CLR  C
002F E500           MOV  A,c1
0031 9F             SUBB A,R7
0032 EE             MOV  A,R6
0033 6480           XRL  A,#080H
0035 F8             MOV  R0,A
0036 7480           MOV  A,#080H
0038 98             SUBB A,R0
0039 5003           JNC  ?C0004
003B E4             CLR  A
003C F500           MOV  c1,A
                ; SOURCE LINE # 10
003E            ?C0004:
003E 22             RET
                ; FUNCTION main (END)
```

CODE SIZE = 63 Bytes

Code generated with NOINTPROMOTE

```
                ; FUNCTION main (BEGIN)
                ; SOURCE LINE # 6
0000 AF00           MOV  R7,c
0002 EF             MOV  A,R7
0003 33             RLC  A
0004 95E0           SUBB A,ACC
0006 FE             MOV  R6,A
0007 EF             MOV  A,R7
0008 F4             CPL  A
0009 4E             ORL  A,R6
000A 7002           JNZ  ?C0001
000C F500           MOV  c,A
000E            ?C0001:
                ; SOURCE LINE # 7
000E E500           MOV  A,c
0010 B4FF03         CJNE A,#0FFH,?C0002
0013 750001         MOV  c,#01H
0016
                ; SOURCE LINE # 8
0016 E500           MOV  A,c
0018 2405           ADD  A,#05H
001A FF             MOV  R7,A
001B 33             RLC  A
001C 95E0           SUBB A,ACC
001E F500           MOV  i,A
0020 8F00           MOV  i+01H,R7
                ; SOURCE LINE # 9
0022 E500           MOV  A,c2
0024 2404           ADD  A,#04H
0026 FF             MOV  R7,A
0027 E500           MOV  A,c1
0029 C3             CLR  C
002A 9F             SUBB A,R7
002B 5003           JNC  ?C0004
002D E4             CLR  A
002E F500           MOV  c1,A
                ; SOURCE LINE # 10
0030            ?C0004:
0030 22             RET
                ; FUNCTION main (END)
```

CODE SIZE = 49 Bytes

INTVECTOR/NOINTVECTOR

缩写： **IV/NOIV**

参数： 对中断矢量表，一个可选的偏移，在括号中。

缺省： **INTVECTOR（0）**

µVision2控制： Options - C*x*51 – Misc controls:enter the directive

说明： **INTVECTOR**命令指示编译器对要求的函数产生中断矢量。如果矢量表不从0开始，需输入一个偏移。

用本命令，编译器产生一个中断矢量入口，根据**ROM**命令指定的程序存储区，用**AJMP**或**LJMP**指令跳转。

NOINTVECTOR命令禁止产生中断矢量表。这也许用户用别的编程工具提供中断矢量。

编译器通常用一个3字节跳转指令（**LJMP**）产生一个中断矢量。矢量用绝对地址表示：

(*interval* × *n*) + *offset* + 3,

这里：

n 中断号。

interval **INTERVAL**命令的参数（缺省为8）。

offset **INTVECTOR**命令的参数（缺省为0）。

参考： **INTERVAL**

例子：

```
C51 SAMPLE.C INTVECTOR（0x8000）

#pragma iv(0x8000)

C51 SAMPLE.C NOINTVECTOR

#pragma noiv
```

LARGE

缩写: **LA**

参数: 无

缺省: SMALL

µVision2控制: Options – Target – Memory Model

说明: 本命令选择**LARGE**存储模式。

在**LARGE**存储模式，所有函数和过程的变量和局部数据段都定位在8051系统的外部数据区。外部数据区最多可有64Kbytes。这，要求用**DPTR**数据指针访问数据。

和存储类型无关的，在任何8051存储范围内可以声明变量。然而，把常用的变量（如循环计数器和数组索引）放在内部数据区可以显著提高系统性能。

注意:
*函数调用所用的堆栈一般放在**IDATA**存储区。*

参考: **SMALL，COMPACT，ROM**

例子:

```
C51 SAMPLE.C LARGE

#pragma large
```

LISTINCLUDE

缩写: **LC**

参数: 无

缺省: **NOLISTINCLUDE**

µVision2控制: Options – Listing – C Compiler Listing - #include Files

说明: **LISTINCLUDE**命令在列表文件中显示头文件的内容。缺省的，不在列表文件中列出头文件。

例子:

```
C51 SAMPLE.C LISTINCLUDE

#pragma listinclude
```

MAXAREGS

缩写: 无

参数: 编译器为可变长度参数列表保留的字节数。

缺省: **MAXAREGS（15）** 对SMALL和COMPACT模式。

MAXAREGS（40） 对LARGE模式。

µVision2控制: Options - C*x*51 – Misc controls:enter the directive

说明: 用**MAXAREGS**命令，可对参数传递的可变长度参数列表指定缓冲区大小。**MAXAREGS**定义参数的最大数目。**MAXAREGS**命令必须在C函数前应用。本命令和传递给可重入函数的最大参数数目没有冲突。

例子: C51 SAMPLE.C MAXAREGS（20）

```
#pragma maxaregs (4)  /* allow 4 bytes for parameters */

#include <stdarg.h>

void func (char typ, ...)  {
  va_list ptr;
  char c;
  int i;

  va_start (ptr, typ);
  switch *typ)  {
    case 0:                      /* a CHAR is passed */
      c = va_arg (ptr, char);  break;

    case 1:                      /* an INT is passed */
      i = va_arg (ptr, int);  break;
  }
}


void testfunc (void)  {
  func (0, 'c');             /* pass a char variable */
  func (1, 0x1234);          /* pass an int variable */
}
```

MOD517/NOMOD517

缩写: 无

参数: 可选参数，在括号内，控制对80C517的独立组成的支持。

缺省: **NOMOD517**

µVision2控制: Options - Target – Use On-Chip Arithmetic Unit
Options – Target – Use multiple DPTR registers

说明: **MOD517**命令指示**Cx51**编译器对INFINEON C517或变种的附加的硬件组成（算法处理器和附加的数据指针）产生代码。本特征提高整型，长整型，和浮点数操作的性能，和利用附加数据指针的函数的性能。

下面的库函数利用外部数据指针：memcpy，memmove，memcmp，strcpy，和 strcmp。

利用算法处理器的库函数有一个 **517** 后缀。（参考 209 页的“第八章.库参考）

用 **MOD517** 来指定附加参数控制 **Cx51** 支持 INFINEON 设备的附加组成。当指定后，参数必须紧跟在 **MOD517** 命令后的括号内。如果没指定附加参数就不需要括号。

Directive	Description
NOAU	When specified, the **Cx51** Compiler uses only the additional data pointers of the Infineon device. The arithmetic processor is not used. The **NOAU** parameter is useful for functions that are called by an interrupt while the arithmetic processor is already being used.
NODP8	When specified, the **Cx51** Compiler uses only the arithmetic processor. The additional data pointers are not used. The **NODP8** parameter is useful for interrupt functions declared without the using function attribute. In this case, the extra data pointers are not used and, therefore, do not need to be saved on the stack during the interrupt.

用 **MOD517** 和 **NOMOD517** 指定这些附加的参数有相同的效果。

NOMOD517 不产生利用 C517 或变种的附加硬件组成的代码。

注意:

*虽然可在一个程序中定义几次，**MOD517** 命令只有定义在一个函数声明外才有效。*

参考： **MODA2，MODAD2，MODDA，MODDP2，MODP2**

例子:

```
C51 SAMPLE.C MOD517

#pragma MOD517 (NOAU)

#pragma MOD517 (NODP8)

#pragma MOD517 (NODP8,NOAU)

C51 SAMPLE.C NOMOD517

#pragma NOMOD517
```

MODA2/NOMODA2

缩写: 无

参数: 无

缺省: **NOMODA2**

µVision2控制: Options - Target – Use multiple DPTR registers

说明: **MODA2**命令指示**Cx51**编译器对ATMEL 80x8252或变种和兼容的附加硬件组成（特别的，附加的CPU数据指针）产生代码。对下面的库函数用附加数据指针可以提高性能：**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**

NOMODA2命令不对利用附加的CPU数据指针产生代码。

参考: **MOD517**，**MODAB2**，**MODDP2**，**MODP2**

例子:

```
C51 SAMPLE.C  MODA2

#pragma moda2

C51 SAMPLE.C  NOMODA2

#pragma  nomoda2
```

MODAB2/NOMODAB2

缩写: 无

参数: 无

缺省: **NOMODAB2**

µVision2控制: Options - Target – Use multiple DPTR registers

说明: **MODAB2**命令指示**Cx51**编译器对MICROCONVERTERS的模拟器件B2系列的附加硬件资源（特别是附加的CPU数据指针）生成代码。对下面的库函数用附加数据指针可以提高性能：**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**。

NOMODAB2命令对利用附加的CPU数据指针不生成代码。

参考: **MOD517**，**MODA2**，**MODDP2**，**MODP2**

例子:

```
C51 SAMPLE.C MODAB2

#pragma modab2

C51 SAMPLE.C NOMODAB2

#pragma nomodab2
```

MODDA2/NOMODDA2

缩写: 无

参数: 无

缺省: **NOMODDA2**

µVision2控制: Options - Target – Use On-Chip Arithmetic Accelerator

说明: **MODDA2** 命令指示 **Cx51** 编译器对 DALLAS 的 DS80C390，DS80C400和DS5240的附加硬件资源（算法加速器）生成代码。本特征提高整型和长整型操作的性能。

NOMODDA2命令对利用片内算法加速器不生成代码。

用下面的方法保证只有一个可执行线程使用算法加速器：

- 用**MODDA**命令编译保证只在主程序执行或只被一个中断服务程序函数使用的函数。
- 对其余的函数用**NOMODDA**命令编译。

参考: **MOD517**

例子:

```
C51 SAMPL390.C MODDA

#pragma modda

C51 SAMPL390.C NOMODDA

#pragma nomodda
```

MODDP2/NOMODDP2

缩写: 无

参数: 无

缺省: **NOMODDP2**

µVision2控制: Options - Target – Use multiple DPTR registers

说明: **MODDP2**命令指示**Cx51**编译器对DALLAS的80C320，C520，C530，C550，或变种及兼容器件的附加硬件资源（特别是附加的CPU数据指针）生成代码。对下面的库函数用附加数据指针可以提高性能：**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**。

NOMODDP2命令对利用附加的CPU数据指针不生成代码。

参考: **MOD517，MODA2，MODP2**

例子:

```
C51 SAMPL320.C MODDP2

#pragma moddp2

C51 SAMPL320.C NOMODDP2

#pragma nomoddp2
```

MODP2/NOMODP2

缩写: 无

参数: 无

缺省: **NOMODP2**

µVision2控制: Options - Target – Use multiple DPTR registers

说明: **MODP2**命令指示**Cx51**编译器对一些PHILIPS或ATMELWM的8051变种的附加DPTR寄存器（双数据指针）生成代码。对下面的库函数用附加数据指针可以提高性能：**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**。

NOMODP2命令对利用双DPTR指针不生成代码。

参考: **MOD517**，**MODA2**，**MODAB2**，**MODDP2**

例子:

```
C51 SAMPLE.C MODP2

#pragma modp2

C51 SAMPLE.C NOMODP2

#pragma nomodp2
```

NOAMAKE

缩写: **NOAM**

参数: 无

缺省: 产生AUTOMAKE信息。

μVision2控制: 本命令不能在μVision2.内使用。

说明: **NOAMAKE**不使能由**Cx51**编译器产生的AUTOMAKE PROJECT信息记录。同时不使能寄存器优化信息。

用**NOAMAKE**生成的OBJ文件可用在旧版本的8051开发工具上。

例子:

```
C51 SAMPLE.C NOAMAKE

#pragma NOAM
```

NOEXTEND

缩写: 无

参数: 无

缺省: 允许所有的语言扩展。

µVision2控制: 在Options – C51 – Misc Controls输入NOEXTEND。

说明: **NOEXTEND**控制命令编译器只处理ANSI C语言结构。**Cx51**语言扩展不使能。保留关键词如**bit**，**reentrant**和**using**将不认识，并产生编译错误或警告。

例子:

```
C51 SAMPLE.C NOEXTEND

#pragma NOEXTEND
```

OBJECT/NOOBJECT

缩写： **OJ/NOOJ**

参数： 一个任意的包括路径的文件名。

缺省： **OBJECT（*filename*.OBJ）**

µVision2控制： Options – Output - Select Folder for Objects

说明： **OBJECT(*filename*)**命令用提供的名称命名OBJ文件。缺省的，OBJ文件使用源文件的文件名和**.OBJ**扩展名。

NOOBJECT控制禁止产生一个OBJ文件。

例子：

```
C51 SAMPLE.C OBJECT（sample1.obj）

#pragma oj(sample1.obj)

C51 SAMPLE.C NOOBJECT

#pragma nooj
```

OBJECTADVANCE

缩写： **OA**

参数： 无

缺省： 无

μVision2控制： Options – C51 – Code Optimization – Linker Code Packing

说明： **OBJECTADVANCED**命令指示编译器在OBJ文件中包含连接器程序的优化信息。本命令用在用**OPTIMIZE**命令连接时，用来减少程序大小和执行速度。

当使能时，**OBJECTADVANCED**命令指示LX51连接器/定位器选择下面的优化：

优化级	连接器优化表现
0-7	**AJMP/ACALL最大化**：连接器重新安排代码段使**AJMP**和**ACALL**指令最大化，**AJMP**和**ACALL**指令比LJMP和LCALL只短的多。
8	**重复使用共同入口代码**：当一个函数有多个调用时设置代码可重复使用。重复使用共同的入口代码减少程序大小。本优化在全部的应用中执行。
9	**公共块子程序**：循环指令系列转换成子程序。这可减少程序大小但要稍增加执行时间。本优化在全部应用中执行。
10	**重排代码**：当检测到公共块子程序，代码重排以得到最多的循环序列。
11	**公共出口代码的复用**：固定的出口序列被复用。这可进一步减少公共块子程序的大小。本优化可产生最紧凑的程序代码。

参考： **OPTIMIZE，OMF2**

例子：

```
C51 SAMPLE.C OBJECTADVANCED DEBUG
```

OBJECTEXTEND

缩写： **OE**

参数： 无

缺省： 无

µVision2控制： Options – Output – Debug Information

说明： **OBJECTEXTEND**命令指示编译器在生成的OBJ文件中包含附加的变量类型，定义信息。这些附加的信息用来标识不同范围内相同名字的目标，以便于被各种模拟和仿真器区别。

注意：
*用本命令产生的OBJ文件包含一个可重定位的OBJ格式规格OMF-51的扩展集。仿真器必须提供增强的OBJ加载器来使用本特征。如果不能确定，不要使用**OBJECTEXTEND**。*

参考： **DEBUG，OMF2**

例子：

```
C51 SAMPLE.C OBJECTEXTEND DEBUG

#pragma oe db
```

ONEREGBANK

缩写: **OB**

参数: 无

缺省: 无

µVision2控制: 在Options – C51 – Misc controls输入ONEREGBANK

说明: 不用**using**属性指定，在中断入口**C*x*51**选择寄存器组0。这在中断服务程序的口头执行，用**MOV PSW，#0**指令。这确保没使用**using**属性的高优先级中断可以中断使用不同的寄存器组的低优先级中断。

如果应用中中断只用一个寄存器组，应使用**ONEREGBANK**命令。这模拟**MOV PSW，#0**指令。

例子:

```
C51 SAMPLE.C ONEREGBANK

#pragma OB
```

OMF2

缩写: **O2**

参数: 无

缺省: C51编译器缺省的生成INTEL OMF51文件格式。OMF2文件格式是**C*x*51**编译器的缺省。

µVision2控制: Project – Select Device – Use LX51 instead of BL51

说明: **OMF2**命令使能OMF2文件格式，它对模块提供详细的符号类型检查和排除INTEL OMF51文件格式的历史限制。

当要用下面的**C*x*51**编译器特征之一时需要OMF2文件格式:

- 可变组：**VARBANKING**命令使能使用**far**存储类型。
- XDATA ROM：用**const xdata**存储类型指定的XDATA变量定位在ROM。
- RAM字符串：**STRING**命令指定字符串常数定位在xdata或更远的空间。
- 相连模式：**ROM（D521K）**和**ROM（D16M）**命令使能DALLAS的390和变种的相连模式。

OMF2 文件格式要求扩展的 **LX51** 连接/定位器，不能用 BL51 连接/定位器。

参考: **OBJECTEXTEND**

例子:

```
C51 SAMPLE.C OMF2

#pragma O2
```

OPTIMIZE

缩写： **OT**

参数： 一个括号内的0到9间的十进制数。另外，**OPTIMIZE（SIZE）**或**OPTIMIZE（SPEED）**可用来选择优化重点是放在代码大小或放在执行速度。

缺省： **OPTIMIZE（8，SPEED）**

µVision2控制： Options – C51 – Code Optimization

说明： **OPTIMIZE**命令设置优化级别和重点。

注意：
每个更高的优化级别包含低优化级别的所有特征。

级别	说明
0	**常数合并**：编译器在计算时，只要可能，就用常数代替表达式。这包括运行地址计算。 **优化简单访问**：编译器优化访问8051系统的内部数据和位地址。 **跳转优化**：编译器经常扩展跳转最终目标。多次跳转被删除。
1	**死代码删除**：没用的代码段被删除。 **拒绝跳转**：严密的检查条件跳转，以确定是否可以倒置测试逻辑来改进或删除。
2	**数据覆盖**：适合静态覆盖的数据和位段是确定的，并内部标识。BL51连接/定位器可以，通过全局数据流分析，选择可被覆盖的段。
3	**PEEPHOLE优化**：清除多余的MOV指令。这包括不必要的存储区目标加载和常数加载。当存储空间或执行时间可节省时，用简单操作代替复杂操作。

级别	说明
4	**寄存器变量**：如有可能，自动变量和函数参数定位在寄存器上。为这些变量保留的存储区就省略了。 **优化扩展访问**：IDATA，XDATA，PDATA和CODE的变量直接包含在操作中。在多数时间中间寄存器是没必要的。 **局部公共子表达式删除**：如果用一个表达式重复的进行相同的计算，则保存第一次计算结果，后面有可能就用这结果。多余的计算就被删除。 **CASE/SWITCH优化**：包含SWITCH和CASE的代码优化为跳转表或跳转队列。
5	**全局公共子表达式删除**：一个函数内相同的子表达式有可能就只计算一次。中间结果保存在寄存器中，在一个新的计算中使用。 **简单循环优化**：用一个常数填充存储区的循环程序被修改和优化。
6	**回路循环**：如果结果程序代码更快和有效则程序回路循环。
7	**优化扩展的索引访问**：当适当时对寄存器变量用DPTR。指针和数组访问对执行速度和代码大小优化。
8	**公共的尾部合并**：当一个函数有多个调用，一些设置代码可以复用，因此减少程序大小。
9	**公共块子程序**：检测循环指令序列，并转换成子程序。Cx51甚至重排代码以得到更大的循环序列。

OPTIMIZE 级别 9 包括 0 到 8 的所有优化级别。

例子：

```
C51 SAMPLE.C OPTIMIZE（9）

C51 SAMPLE.C OPTIMIZE（0）

#pragma ot(6,SIZE)

#pragma ot(size)
```

ORDER

缩写: **OR**

参数: 无

缺省: 变量是无序的。

µVision2控制: Options – C51 – Keep Variables in Order

说明: **ORDER**命令指示**C*x*51**编译器根据变量在C源文件的定义顺序在存储区中排序。**ORDER**不使能C编译器使用HASH算法。本命令使**C*x*51**编译变慢。

例子:

```
C51 SAMPLE.C ORDER

#pragma OR
```

PAGELENGTH

缩写： **PL**

参数： 括号中一个十进制数，最大为65535。

缺省： **PAGELENGTH（60）**

µVision2控制： Options – Listing – Page Length

说明： **PAGELENGTH**命令指定列表文件中每页打印的行数。缺省使每页60行，包括头和空行。

参考： **PAGEWIDTH**

例子：

```
C51 SAMPLE.C PAGELENGTH（70）

#pragma pl(70)
```

PAGEWIDTH

缩写： **PW**

参数： 括号中的一个十进制数，范围在78到132间。

缺省： **PAGEWIDTH（132）**

µVision2控制： Options – Listing – Page Width

说明： **PAGEWIDTH**命令指定列表文件中每行的字符数。多于指定字符数的将被分成两行或多行。

参考： **PAGELENGTH**

例子：

```
C51 SAMPLE.C PAGEWIDTH（79）

#pragma pw(79)
```

PREPRINT

缩写: **PP**

参数: 括号中的一个可选的文件名。

缺省: 不产生预处理器列表

µVision2控制: Options – C51 – C Preprocessor Listing

说明: **PREPRINT**命令指示编译器产生一个预处理器列表。宏调用被扩展，注释被删除。如果用了不带参数的**PREPRINT**，就产生一个用源文件名和**.I**扩展名的文件。缺省，**Cx51**编译器不产生一个预处理器输出文件。

注意:
PREPRINT*命令只能在命令行指定。不能在C源文件中用****#pragma*** *命令指定。*

例子:

```
C51 SAMPLE.C PREPRINT

C51 SAMPLE.C PP（PREPRO.LST）
```

PRINT/NOPRINT

缩写: **PR/NOPR**

参数: 括号中的一个可选的文件名。

缺省: **PRINT（*filename*.LST）**

µVision2控制: Options – Listing – Select Folder for List Files

说明: 编译器对每个编译的程序产生一个列表，扩展名为**.LST**。用**PRINT**命令，可以重新指定列表文件名。

NOPRINT命令禁止编译器产生一个列表文件。

例子:

```
C51 SAMPLE.C PRINT（CON：）

#pragma pr(\usr\list\sample.lst)

C51 SAMPLE.C NOPRINT

#pragma nopr
```

REGFILE

缩写: **RF**

参数: 括号中的一个文件名。

缺省: 无

µVision2控制: Options – C51 – Global Register Coloring

说明: **REGFILE**命令指示**C*x*51**编译器用一个寄存器定义文件优化全局寄存器。寄存器定义文件指定外部函数对寄存器的使用。用这些信息，**C*x*51**编译器可以优化通用寄存器的使用。本特征使能全局寄存器的优化。

例子:

```
C51 SAMPLE.C REGFILE（sample.reg）

#pragma REGFILE(sample.reg)
```

REGISTERBANK

缩写： **RB**

参数： 括号中的一个0到3的数字。

缺省： **REGISTERBANK（0）**

µVision2控制： 在Options – C51 – Misc controls输入**REGISTERBANK**命令。

说明： **REGISTERBANK**命令对在源文件中声明的后来的函数选择哪组寄存器组。当绝对寄存器号可以计算时，结果代码可使用绝对形式的寄存器访问。**using**函数属性取代**REGISTERBANK**命令的效果。

注意：
*和**using**函数属性不同，**REGISTERBANK**控制不切换寄存器组。*

返回一个值给调用者的函数必须和调用者使用相同的寄存器组。如果寄存器组不同，返回值可能会在错误的寄存器组中。

***REGISTERBANK**命令在一个源程序中可以多次出现；然而，在一个函数声明中使用本命令将被忽略。*

例子：

```
C51 SAMPLE.C REGISTERBANK（1）

#pragma rb(3)
```

REGPARMS/NOREGPARMS

缩写: 无

参数: 无

缺省: **REGPARMS**

µVision2控制: 在Options – C51 – Misc controls中输入**REGPARMS**命令

说明: **REGPARMS**命令指示编译器为在寄存器中传递三个函数参数生成代码。这类参数传递类似于用汇编编写的代码，比存储函数参数在存储区中快的多。不能定位在寄存器中的参数用固定存储区传递。

NOREGPARMS命令强制所有函数参数在固定存储区中传递。本命令产生的参数传递代码和**C51**的版本2和版本1兼容。

注意:
*在一个源程序中可指定**REGPARMS**和**NOREGPARMS**命令多次。这允许建立一些程序段用寄存器参数，另外的段用老式的参数传递。*

*不必重新汇编或编译，用**NOREGPARMS**可访问旧的汇编函数或库文件。这在下面的例子程序中说明。*

```
#pragma NOREGPARMS                /*Parm passing-old method */
extern int old_func(int,char);

#pragma REGPARMS                  /* Parm passing-new method */
extern int new_func(int,char);

main()    {
      char a;
      int x1,x2;
      x1 = old_func(x2,a);
      x1 = new_func(x2,a);
}
```

例子：

```
C51 SAMPLE.C NOREGPARMS
```

RET_PSTK，REG_XSTK

缩写: **RP，RX**

参数: 无

缺省: 无

µVision2控制: 在Options – C51 – Misc controls中输入**RET_PSTK,RET_XSTK**命令。

说明: **RET_PSTK**和**RET_XSTK**命令引起返回地址使用**pdata**或**xdata**重入堆栈。正常情况，返回地址保存在8051的硬件堆栈中。这些命令指示编译器产生代码从硬件堆栈中POP出返回地址，并存储在指定的重入堆栈中。

RET_PSTK 使用COMPACT模式重入堆栈。
RET_XSTK 使用LARGE模式重入堆栈。

注意:
*可用 **RET_xSTK** 命令从片内或硬件堆栈卸载返回地址。这些命令可有选择的使用在包含深度堆栈嵌套的模块中。*

如果用了这些命令中的其中之一，必须在启动代码中初始化重入堆栈指针。如何初始化重入堆栈参考 151 页的“STARTUP.A51”。

```
   1                    #pragma RET_XSTK
   2                    extern void func2(void);
   3
   4                    void func(void)      {
   5     1              func2();
   6     1              }

  ASSEMBLY LISTING OF GENERATED OBJECT CODE
                       ;FUNCTION func(BEGIN)
  0000 120000              E        LCALL   ?C?CALL      XBP
;SOURCE LINE #5
  0003 120000              E        LCALL   func2
                                            ;SOURCE LINE #6
  0006 020000              E        LJMP    ?C?RET_XBP
                       ;FUNCTION func(END)
```

例子:

```
C51 SAMPLE.C RET_XSTK
```

ROM

缩写: 无

参数: **（SMALL），（COMPACT），（LARGE），（D521K），或（D16M）**

缺省: **ROM（LARGE）**

µVision2控制: Options – Target – Code Rom Size

说明: 使用**ROM**命令指定程序存储区的大小。本命令影响JMP和CALL指令的代码。

选项	说明
SMALL	**CALL**和**JMP**指令和**ACALL**和**AJMP**一样编码。最大程序大小为2K字节。整个程序必须定位在2K字节程序存储区内。
COMPACT	**CALL**指令和**LCALL**一样编码。**JMP**指令和同一函数内的**AJMP**一样编码。一个函数的大小不能超过2K字节。整个程序可最大到64K字节。应用的类型决定是否**ROM（COMPACT）**比**ROM（LARGE）**有利。用**ROM（COMPACT）**节省代码空间必须根据经验决定。
LARGE	**CALL**和**JMP**和**LCALL**和**LJMP**一样编码。这允许使用整个代码空间，没有如何限制。程序大小限制到64K字节。函数大小也限制到64K字节。
D521K†（Dallas 390和变种）	产生19位的**ACALL**和**AJMP**指令。最大的程序大小为521K字节。本模式只对DALLAS 390和兼容设备有效。
D16M†（Dallas 390和变种）	产生24位**LCALL**指令和19位**AJMP**指令。最大程序大小为16M字节。本模式只对DALLAS 390和兼容设备有效。

† **D521K** 和 **D16M** 选项要求 OMF2 命令。

参考: **SMALL，COMPACT，LARGE**

例子:

```
C51 SAMPLE.C ROM（SMALL）

#pragma ROM(SMALL)
```

SAVE/RESTORE

缩写: 无

参数: 无

缺省: 无

µVision2控制: 本命令不能在命令行指定。

说明: **SAVE**命令保存**AREGS**，**REGPARMS**的当前设置，和当前**OPTIMIZE**级和重点。例如，这些设置在一个**#include**命令前保存，后来用**RESTORE**命令恢复。

RESTORE命令从保存堆栈中恢复**SAVE**命令保存的值。

SAVE命令最大的嵌套深度是8级。

注意:
***SAVE*和*RESTORE*只能作为*#pragma*的一个参数指定。不能在命令行指定本控制选项。**

例子:

```
#pragma save
#pragma noregparms

extern void test1(char c,int I);
extern char test2(long l,float f);

#pragma restore
```

在上面的例子中，通过寄存器传递参数对两个外部函数**test1**和**test2**是不使能的。**SAVE**命令时的设置被**RESTORE**命令恢复。

SMALL

缩写: **SM**

参数: 无

缺省: **SMALL**

µVision2控制: Options – Target – Memory Model

说明: 本命令选择**SMALL**存储模式，把所有函数变量和局部数据段放在8051系统的内部数据存储区。这使访问数据非常快。但**SMALL**存储模式的地址空间受限。

无论什么存储模式，都可以声明变量在任何的8051存储区范围。然而，把最常用的命令（如循环计数器和队列索引）放在内部数据区可以显著的提高系统性能。

注意:
函数调用所用的堆栈通常放在IDATA存储区。

*在开始时常用**SMALL**存储模式。但应用变大时，在变量声明时用明确的的存储区定义，可以放置大的变量和数据在别的存储区。*

参考: **COMPACT**，**LARGE**，**ROM**

例子:

```
C51 SAMPLE.C SMALL

#pragma small
```

SRC

缩写: 无

参数: 括号内的一个可选的文件名。

缺省: 无

µVision2控制: 可以在µVision2中设置如下：

- 在PROJECT窗口-文件表中右键点击文件
- 选择**Options for…**打开Options – Properties page
- 使能**Generate Assembler SRC file**

说明: 用**SRC**建立一个汇编源文件，而不是一个OBJ文件。这源文件可用A51汇编器汇编。

如果括号中没指定外文件名，就用C源文件的名字和路径，和**.SRC**的扩展名。

注意:
编译器不能同时产生一个源文件和一个OBJ文件。

参考: **ASM，ENDASM**

例子:

```
C51 SAMPLE.C SRC

C51 SAMPLE.C SRC（SML.A51）
```

STRING

缩写: **ST**

参数: **（CODE），（XDATA），或（FAR）**

缺省: **STRING（CODE）**

µVision2控制: 在Options – C51 – Misc controls中输入命令**STRING**。

说明: **STRING**命令可以指定固定字符串的存储类型。缺省的，字符串固定的放在代码区。例如：“hello world”如下位于代码区：

```
void main(void) {
    printf(“hello world\n”);
}
```

用**STRING**命令可以改变字符串的位置。本选项必须谨慎使用，因为现有的程序可能使用存储区类型指针访问字符串。把字符串定位到**xdata**或**far**存储区，在应用中应避免使用code banking。本选项对扩展的8051系列如PHILIPS 80C51MX非常有用。

选项	说明
CODE	固定的字符串放在CODE区。这是Cx51的缺省设置。
XDATA	固定的字符串放在CONST XDATA区。
FAR†	固定的字符串放在CONST FAR区。

† XDATA 和 FAR 选项要求 OMF2 命令。

参考: OMF2，XCROM

例子:

```
C51 SAMPLE.C STRING(XDATA)

#pragma STRING(FAR)
```

SYMBOLS

缩写: **SB**

参数: 无

缺省: 不产生符号列表

µVision2控制: Options – Listing – C Compiler Listing – Symbols

说明: **SYMBOLS**控制编译器产生所有的程序模块的符号列表。列表包含在列表文件中。每个符号都列出存储种类，类型，偏移，和大小。

例子:

```
C51 SAMPLE.C SYMBOLS

#pragma SYMBOLS
```

下面的列表文件摘录了一段符号列表:

```
NAME             CLASS    MSPACE  TYPE     OFFSET  SIZE
==== ===== ====== ==== ====== ====
EA . . . . . .   ABSBIT   -----    BIT     00AFH   1
update . . . . . .  PUBLIC  CODE   PROC    -----    -----
dtime. . . . . .  PARAM   DATA    PTR     0000H   3
setime . . . . . .  PUBLIC  CODE   PROC    -----    -----
mode . . . . . .  PARAM   DATA    PTR     0000H   3
dtime. . . . . .  PARAM   DATA    PTR     0003H   3
setuptime. . . .  AUTO    DATA    STRUCT  0006H   3
time . . . . . .  * TAG *  -----   STRUCT  -----   3
hour . . . . . .  MEMBER  DATA    U_CHAR  0000H   1
min. . . . . .  MEMBER  DATA    U_CHAR  0001H   1
sec. . . . . .  MEMBER  DATA    U_CHAR  0002H   1
SBUF . . . . . .  SFR     DATA    U_CHAR  0099H   1
ring . . . . . .  PUBLIC  DATA    BIT     0001H   1
SCON . . . . . .  SFR     DATA    U_CHAR  0098H   1
TMOD . . . . . .  SFR     DATA    U_CHAR  0089H   1
TCON . . . . . .  SFR     DATA    U_CHAR  0088H   1
mnu. . . . . .  PUBLIC  CODE    ARRAY   00FDH   119
```

USERCLASS

缩写: **UCL**

参数: （***mspace = user_classname***）

mspace涉及变量或程序代码所使用的缺省的存储区，说明如下：

mspace	说明
CODE	程序代码。
CONST	CODE空间的变量（CONST类）。
XCONST	CONST XDATA空间的常数（XDATA类）。
XDATA	XDATA空间的变量（XDATA类）。
HDATA	扩展FAR空间的变量（HDATA类）。
HCONST	扩展CONST FAR空间的常数（HCONST类）。

User_classname 是一个存储类的名称。类名可以用如何可用的标识符。

缺省: 段接收缺省的类名。

µVision2控制: 在Options – C51 – Misc controls输入**USERCLASS**命令。

说明: **USERCLASS**命令对一个编译器产生的段指定一个类名。缺省的，**C*x*51**编译器用基类名来定义段。用户类名在扩展的LX51连接/定位器级被引用，用一个类名定位所有的段，例如**HDATA_FLASH**，来指定存储区。**USERCLASS**命令修改一个完整模块中的基类名，但不包括可覆盖段。

存储类只有在使用OMF2格式和扩展LX51连接/定位器中可用。

例子:

```
C51 UCL.C

#pragma userclass (xdat = flash)
#pragma userclass (hconst = patch)

    int xdata x1[10];                  //XDATA_FLASH
const char far tst[] = “Hello”;        //HCONST_PATCH
```

VARBANKING

缩写: **VB**

参数: 无 不修改中断代码。
（1） 在中断代码中保存SFR的扩展地址。

缺省: 使用标准C51库

µVision2控制: Options – Target – 'far' memory type support.
Options – Target – save address extension SFR in interrupts.

说明: **VARBANKING**命令允许在传统的8051中用**FAR**存储区。但使能时，就会选支持**FAR**存储区的不同的库函数集。

在**XBANKING.A51**中配置用**FAR**变量访问函数。参考154页的"XBANKING.A51"。

VARBANKING（1）在中断函数对SFR的扩展地址增加保存和恢复代码。**XBANKING.A51**中定义的符号**?C?XPAGE1SFR**指定SFR扩展地址的地址。SFR的初始值用符号**?C?XPAGE1RST**指定。在中断函数的开头加入**MOV ?C?XPAGE1SFR，?C?XPAGE1RST**指令。

注意:
*只有C模块中用**VARBANKING（1）**命令编译的中断函数保存和恢复SFR的扩展地址。如果应用中包含别的汇编模块或库的中断函数，必须仔细的检查这些函数。*

在**\KEIL\C51\EXAMPLES\FARMEMORY**中的例子说明如何使用C51 FAR存储类型在传统的8051。

例子:

```
C51 SAMPLE.C VARBANKING

C51 MYFILE.C VARBANKING（1）
```

WARNINGLEVEL

缩写: **WL**

参数: 数字0到2。

缺省: **WARNINGLEVEL（2）**

µVision2控制: Options – C51 – Warnings

说明: **WARNINGLEVEL**命令允许抑制编译器告警。参考“第7章.错误信息”。

告警级别	说明
0	不使能大多数的编译器告警
1	只列出可能产生不正确代码的告警
2（缺省）	列出所有的告警信息，包括未使用变量，表达式，或标号告警。

例子:

```
C51 SAMPLE.C WL（1）

#pragma WARNINGLEVEL(0)
```

XCROM

缩写: **XC**

参数: 无

缺省: 所有XDATA变量在起始代码中初始化。

µVision2控制: 在Options – C51 – Misc controls中输入**XCROM**命令。

说明: **XCROM**命令指示编译器保存常数变量在**XDATA**存储区，而不是在**CODE**区。这些变量必须用**CONST XDATA**声明。这可不占用代码存储区。

有些新的8051提供一个存储管理单元，允许影射ROM空间到XDATA存储区。对传统的8051，对XDATA空间，应该用ROM设备替代RAM。

参考: **OMF2**，**STRING**

例子:

```
#pragma XCROM          // Enable const xdata ROM

/*
 * The following text will be in a ROM that
 * is addressed in the XDATA space.
 */

const char xdata text[] =    “Hello world\n”;

void main(void)      {
      printf(text);
}
```

第三章. 语言扩展

Cx51 编译器提供几种 ANSI 标准 C 的扩展，以支持 8051 结构。这些扩展有：

- 存储区
- 存储区类型
- 存储模型
- 存储类型说明符
- 变量数据类型说明符
- 位变量和位可寻址数据
- SFR
- 指针
- 函数属性

下面各节详细说明。

关键字

为了利用8051的许多特性，**Cx51**编译器在C语言的范围内加了许多新的关键字：

at	**far**	**sbit**
alien	**idata**	**sfr**
bdata	**interrupt**	**sfr16**
bit	**large**	**small**
code	**pdata**	***task***
compact	***priority***	**using**
data	**reentrant**	**xdata**

可以用 **NOEXTEND** 控制命令取消这些扩展。参考 17 页的“第二章.用 Cx51 编译”。

存储区

8051 结构支持几个物理分开的程序和数据的存储区或存储空间。每个存储区都有有利的和不利的方面。这些存储空间可能：

- 可读不可写。
- 可读写。
- 读写比别的存储空间快。

和这些不同的是，对多数的大型机，小型机，和微型机来说，程序，数据和常数都放在机内相同的存储空间内。可参考*Intel 8位嵌入式处理器*手册或别的8051数据手册。

程序存储区

程序（CODE）存储区是只读的；他不能写。程序存储区可能在8051CPU内，或者在外部，或者都有，根据8051派生的硬件决定。

最多可以有64K字节的程序存储区。程序代码，包括所有的函数和库，保存在程序存储区。常数变量也是。8051可执行程序只保存在程序存储区。

在**Cx51**编译器中，可用**code**存储区类型标识符来访问程序存储区。

内部数据存储区

8051CPU内部的数据存储区是可读写的。8051派生系列最多可有256字节的内部数据存储区。低128字节内部数据存储区可直接寻址。高128字节数据区（从0x80到0xFF）只能间接寻址。从20H开始的16字节可位寻址。

因为可以用一个8位地址访问，所以内部数据区访问很快。然而，内部数据区最多只有256字节。

内部数据区可以分成三个不同的存储类型：**data**，**idata**，和**bdata**。

data存储类型标识符通常指低128字节的内部数据区。存储的变量直接寻址。

idata存储类型标识符指内部的256个字节的存储区；但是，只能间接寻址，速度比直接寻址慢。

bdata存储类型标识符指内部可位寻址的16字节存储区（20H到2FH）。可以在本区域声明可位寻址的数据类型。

外部数据存储区

外部数据区可读写。访问外部数据区比内部数据区慢，因为外部数据区是通过一个数据指针加载一个地址来间接访问的。

几种8051系列增加片内XRAM，用和传统的外部数据区一样的指令访问。这些空间用专用的SFR配置寄存器使能，和外部空间重叠。

外部数据区最多可有64K字节；当然，这些地址不是必须用做存储区。硬件设计可能把外围设备影射到存储区。如果是这种情况，程序可以访问外部数据区和控制外围设备。这可参考I/O的存储区影射。

Cx51编译器提供两种不同的存储类型访问外部数据：**xdata**和**pdata**。

xdata存储类型标识符指外部数据区64K字节内的任何地址。

pdata存储类型标识符仅指一（1）页或256字节的外部数据区。参考95页的“COMPACT模式”中关于**pdata**的资料。

FAR存储区

FAR存储区指许多新的8051变种的扩展地址空间。**C*x*51**编译器用普通的3字节指针访问扩展存储区。两个**C*x*51**存储类型，**far**和**const far**，访问扩展RAM中的变量和ROM中的常数。

PHILIPS 51MX结构用通用指针可支持8Mb **code**和**xdata**空间。**C*x*51**编译器用80C51MX结构的新指令访问**far**和**const far**变量。

DALLAS 390结构在CONTIGIOUS模式用一个24位DPTR寄存器和传统的MOVX和MOVC指令支持扩展的**code**和**xdata**地址空间。用**far**和**const far**定义的变量位于这些扩展的**xdata**和**code**地址空间。

传统的8051系列也可用**far**和**const far**变量，如果配置**XBANKING.A51**。这对提供一个地址扩展SFR或附加存储空间可以影射到**xdata**空间的系列很有用。也可用**xdata** banking硬件扩展传统8051系列的地址空间。参考154页的“XBANKING.A51”。

注意:
*需要指定C51指示OMF2和扩展LX51连接/定位器使用**far**和**far const**存储类型。*

特殊功能寄存器（SFR）存储区

8051提供128字节的SFR存储区。SFR有位，字节，或字寄存器，用来控制计时器，计数器，串口，并口，和外围设备。参考101页的“SFR”。

存储模式

存储模式定义缺省的存储类型，用在函数参数，自动变量，和没有直接声明存储类型的变量上。在**C*x*51**编译器命令行中用**SMALL**，**COMPACT**和**LARGE**控制命令指定存储模式。参考20页的“控制命令”。

注意:
*除了在很特殊的应用中，**SMALL**存储模式产生最快和最有效的代码。*

用明确的存储类型标识符声明一个变量，可以重载缺省的存储模式指定的存储类型。参考95页的“存储类型”。

SMALL模式

在本模式中，所有的变量，缺省的情况下，位于8051系统的内部数据区。（这和用**data**存储类型标识符明确声明的一样）在本模式中，变量访问非常有效。然而，所有的东西，包括堆栈必须放在内部RAM中。堆栈大小是不确定的，它取决于函数嵌套的深度。典型的，如果连接/定位器配置为内部数据区变量可覆盖，SMALL模式是最好的模式。

COMPACT模式

用COMPACT模式，所有变量，缺省的，都放在外部数据区的一页中。（这就象用pdata声明的一样）这存储模式可提供最多256字节的变量。限制是由于用了寻址计划，它通过寄存器R0和R1（@R0，@R1）间接寻址。本存储模式不如SMALL模式有效，因此，变量访问不是很快。然而，COMPACT模式比LARGE模式快。

当用COMPACT模式，**C*x*51**编译器用以@R0和@R1为操作数的指令访问外部存储区。R0和R1是字节寄存器，只提供地址的低字节。如果COMPACT模式使用多于256字节的外部存储区，高字节地址（或页）由8051的PORT2提供。这种情况，必须初始化PORT2以使用正确的外部存储页。这可在起始代码中实现。同时必须为连接器指定**PDATA**的起始地址。

参考151页“STARTUP.A51”中的配置P2为COMPACT模式。

LARGE模式

在LARGE模式，所有变量，缺省的，放在外部数据存储区（可到64K字节）。（这和用**xdata**存储类型标识符明确声明的一样。）数据指针（**DPTR**）用做寻址。通过这指针访问存储区是低效的，特别是对两个或多个字节的变量。这种访问机制比SMALL或COMPACT模式产生更多的代码。

存储类型

C*x*51编译器明确支持8051和派生系列结构，可访问8051的所有存储区。每个变量可以明确的和特定的存储空间相关连。

访问内部数据区比访问外部数据区快的多。基于这个原因，把频繁使用的变量放在内部数据区。把较大的，较少使用的变量放在外部存储区。

明确声明存储类型

在变量声明中包含一个存储类型标识符，可以指定变量的存放区域。

下面的表综合了可用的存储类型标识符。

存储类型	说明
code	程序存储区（64K字节）；用MOVC @A+DPTR访问
data	直接寻址内部数据区；访问变量速度最快（128字节）。
idata	间接寻址内部数据区；可访问全部内部地址空间（256字节）。
bdata	位寻址内部数据区；支持位和字节混合访问（16字节）。
xdata	外部数据区（64K字节）；由MOVX @DPTR访问。
far	扩展的RAM和ROM存储空间（最多16MB）；由用户定义程序或特定的芯片扩展（PHILIPS 80C51MX，DALLAS 390）访问。
pdata	分页（256字节）外部数据区；由MOVX @Rn访问。

用 signed 和 unsigned 属性，在变量声明中可以包括存储类型标识符。

例子：

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

注意：

为了和以前的 C51 编译器兼容，应该在数据类型前指定存储区。例如，下面的声明 data char x； 等同于 char data x；

*但是，这个特征可能不能用在新的程序中，因为它和将来新的 **Cx51** 编译器关联。*

在同时使用旧的 C51 语法和存储标识符指针时应小心。在这种情况下，定义：

*data char *x； 等同于 char *data x；*

暗含的存储类型

如果在变量声明中省略存储类型标识符，自动选择缺省或暗含的存储类型。不能置于寄存器中的函数参数和自动变量也存放在缺省存储区中。

缺省存储类型由 SMALL，COMPACT 和 LARGE 编译器控制命令定义。参考 94 页“存储模式”。

数据类型

C*x*51 编译器提供一些基本的数据类型用在 C 程序中。**C*x*51** 编译器支持标准 C 数据类型和 8051 平台独特的数据类型。下表列出了可用的数据类型。

数据类型	位	字节	值范围
bit †	1		0到1
signed char	8	1	-128到+127
usnigned char	8	1	0到125
enum	8/16	1或2	-128到+127或-32768到+32767
signed short	16	2	-32768到+32767
usnigned short	16	2	0到65535
signed int	16	2	-32768到+32767
unsigned int	16	2	0到65535
signed long	32	4	-2147483648到+2147483647
unsigned long	32	4	0到4294967295
float	32	4	±1.175494E-38到±3.402823E+38
sbit †	1		0到1
sfr †	8	1	0到255
sfr16 †	16	2	0到65535

† **bit**，**sbit**，**sfr**，和 **sfr16** 数据类型在 ANSI C 中没有，是 **C*x*51** 编译器中独有的。这些数据类型在下面的节中有详细描述。

BIT 类型

C*x*51 编译器提供一个 **bit** 数据类型，可能在变量声明，参数列表，和函数返回值中有用。一个 **bit** 变量和别的 C 数据类型的声明相似，例如：

```
static bit done_flag = 0;     /* bit variable */

bit testfunc (                /* bit function */
  bit flag1,                  /* bit arguments */
  bit flag2)
{
.
.
.
return (0);                   /* bit return value */
```

所有的 **bit** 变量放在 8051 内部存储区的位段。因为这区域只有 16 字节长，所以在某个范围内只能声明最多 128 个**位**变量。

存储类型应包含在一个 **bit** 变量的声明中。但是，因为 **bit** 变量存储在 8051 的内部数据区，只能用 **data** 和 **idata** 存储类型。别的存储类型不能用。

bit 变量和 **bit** 声明的限制如下：

- 禁止中断的函数（**#pragma disable**）和用一个明确的寄存器组（**using** *n*）声明的函数不能返回一个位值。**C*x*51**编译器对这类函数想要返回一个**bit**类型产生一个错误信息。

- 一个位不能被声明为一个指针。例如：

```
bit *ptr;                                   /* invalid */
```

- 不能用一个**bit**类型的数组。例如：

```
bit ware[5];                                /* invalid */
```

可位寻址目标

位可寻址目标是可作为字或位寻址的目标。只有占用 8051 内部存储区可位寻址的数据目标符合条件。**Cx51** 编译器在这位可寻址区域用 **bdata** 存储类型声明变量。而且，用 **bdata** 存储类型声明的变量必须是全局的。必须如下声明变量：

```
int bdata ibase;                    /* Bit-addressable int */

char bdata bary[4];                 /* Bit-addressable array */
```

变量 **ibase** 和 **bary** 是位可寻址的。因此，这些变量的每个位是可直接访问和修改的。用 **sbit** 关键词声明新的变量可访问用 **bdata** 声明的变量的位。例如：

```
sbit mybit0 = ibase^0;          /* bit o of ibase */
sbit mybit15 = ibase^15;        /* bit 15 of ibase */

sbit Ary07 = bary[0]^7;         /* bit 7 of bary[0] */
sbit Ary37 = bary[3]^7;         /* bit 7 of bary[3] */
```

上面的例子只是声明，并不分配上面声明的 **ibase** 和 **bary** 变量的位。例子中‘^’符号后的表达式指定位的位置。这表达式必须是常数。范围由声明中的基变量决定。**char** 和 **unsigned char** 的范围是 0 到 7，**int**，**unsigned int**，**short**，和 **unsigned short** 是 0 到 15，**long** 和 **unsigned long** 是 0 到 31。

在别的模块中，应该对 **sbit** 类型提供外部变量声明。例如：

```
extern bit mybit0;              /* bit 0 of ibase */
extern bit mybit15;             /* bit 15 of ibase */

extern bit Ary07;               /* bit 7 of bary[0] */
extern bit Ary37;               /* bit 7 of bary[3] */
```

声明中包含 **sbit** 类型，要求基目标用存储类型 **bdata** 声明。唯一的例外是特殊功能位变量。参考 101 页的“SFR”。

下面的例子显示用上面的声明如何改变 **ibase** 和 **bary** 位。

```
Ary37 = 0;              /* clear bit 7 in bary[3] */
bary[3] = 'a';          /* Byte addressing */
ibase = -1;             /* word addressing */
mybit15 = 1;            /* set bit 15 in ibase */
```

bdata 存储类型和 **data** 存储类型一样处理，除了用 **bdata** 声明的变量位于内部数据区的位寻址区。注意这个区域的总的大小不超过 16 个字节。

除了对数量类型用 **sbit** 声明变量，也可对结构和联合用 **sbit** 声明变量。例如：

```
union lft
  {
      float mf;
      long ml;
  };

bdata struct bad
  {
      char ml;
      union lft u;
}tcp;
sbit tcpf31 = tcp.u.ml^31;          /* bit 31 of float */
sbit tcpm10 = tcp.ml^0;
sbit tcpm17 = tcp.ml^7;
```

注意：

*不能对 **float** 指定 **bit** 变量。然而，在一个 **union** 中可以包含 **float** 和 **long**。因此，可以声明 **bit** 变量来访问 **long** 类型中的位。*

***sbit** 数据类型用一个指定的变量作为基地址，用位位置来得到一个实际的位地址。实际的位地址不等于特定数据类型的逻辑位地址。实际位地址 0 对应第一个字节的位地址 0。实际位地址 8 对应第二个字节的位地址 0。因为 **int** 变量先保存高字节，int 的位 0 在第二个字节的位 0。用 **sbit** 数据类型访问时是实际的位 8。*

特殊功能寄存器（SFR）

8051 系列微处理器提供一个特别的存储区作为特殊功能寄存器（SFR）。用在程序中的 SFR 可控制计时器，计数器，串口，并口，和外围设备。SFR 的地址从 0x80 到 0xFF，可以以位，字节，和字访问。可参考 *Intel 8-Bit Embedded Controllers* 手册或别的 8051 数据手册。

在 8051 系列中，SFR 的数量是不同的。**Cx51** 编译器没有预定义 SFR 名称。但是，在包含文件中有 SFR 的声明。

Cx51 编译器对各种 8051 的派生系列提供许多的包含文件。每个文件包含了派生系列中可用的 SFR。参考 228 页的“8051 特殊功能寄存器包含文件”。

Cx51 编译器提供 **sfr**，**sfr16**，和 **sbit** 数据类型访问 SFR。下面的节说明这些数据类型。

sfr

SFR 和别的 C 变量一样声明。唯一的不同点是数据类型是 **sfr** 而不是 **char** 或 **int**。例如：

```
sfr P0 = 0x80;          /* port-0,address 80h */
sfr P1 = 0x90;          /* port-1,address 90h */
sfr P2 = 0xA0;          /* port-2,address 0A0h */
sfr P3 = 0xB0;          /* port-3,address 0B0h */
```

P0，P1，P2，和 P3 是声明的 SFR 名。**sfr** 变量的名称和别的 C 变量一样定义。在 **sfr** 声明中可用任何符号名。

在等号（=）后指定的地址必须是一个常数值。（不允许是带操作数的表达式。）传统的 8051 系列支持 SFR 地址从 0x80 到 0xFF。PHILIPS 80C51MX 提供一个附加扩展的 SFR 空间，地址范围是 0x180 到 0x1FF。

sfr16

许多新的 8051 派生系列用两个连续地址的 SFR 来指定 16 位值。例如，8052 用地址 0xCC 和 0xCD 表示计时器/计数器 2 的低和高字节。**C*x*51** 编译器提供 **sfr16** 数据类型访问 2 个 SFR 作为一个 16 字节 SFR。

访问一个 16 位 SFR 只能低字节跟着高字节。低字节用做 **sfr16** 声明的地址。例如：

```
sfr16 T2 = 0xCC;          /* Timer 2:T2L 0cch,T2H 0CDh */
sfr16 RCAP2 = 0xCA;   /* RCAP2L 0Cah,RCAP2H 0CBh */
```

在这个例子中，T2 和 RCAP2 被声明为 16 位 SFR。

sfr16 声明和 **sfr** 声明遵循相同的原则。任何符号名可用在 **sfr16** 的声明中。等号（=）指定的地址必须是一个常数值。带操作数的表达式是不允许的。地址必须是 SFR 的低和高字节中的低字节。

sbit

对典型的 8051 应用中，访问 SFR 的位是必须的。**C*x*51** 编译器用 **sbit** 数据类型使这变为可能，它可访问可位寻址的 SFR 和别的位可寻址的目标。例如：

```
sbit EA = 0xAF;
```

声明定义 **EA 为**地址 0xAF 的 SFR 位。在 8051，这是中断使能寄存器中的**全部使能**位。

注意：

不是所有的 SFR 都是可位寻址的。只有地址可被 8 整除的 SFR 可位寻址。SFR 地址的低半字节必须是0 或8。例如，SFR 在0xA8 和0xD0 是可位寻址的，在0xC7 和 0xEB 的 SFR 是不能位寻址的。计算一个 SFR 的位地址，在 SFR 字节地址上加上位地址。因此，访问 0xC8 的 SFR 的位 6，SFR 的位地址是 0xCE（0xC8+6）。

在 **sbit** 声明中可用任何符号名。等号（=）右边的表达式指定符号名的绝对位地址。下面有三个变量指定了地址：

变量1: **sfr_name^int_constant**

这变量用一个已定义的 **sfr**（sfr_name）作为 **sbit** 的基地址。SFR 的地址必须能被 8 整除。‘^’后面的表达式指定了位的位置。位位置必须是 0 到 7 间的一个数字。例如：

```
sfr PSW = 0xD0;
sfr IE = 0xA8;
sbit OV = PSW^2;
sbit CY = PSW^7;
sbit EA = IE^7;
```

变量2: **int_constant^int_constant**

这变量用一个整数常数作为**sbit**的基地址。基地址值必须能被8整除。‘^’后面的表达式指定位的位置。位的位置必须在0到7间。例如：

```
sbit OV = 0xD0^2;
sbit CY = 0xD0^7;
sbit EA = 0xA8^7;
```

变量3: **int_constant**

这变量用一个**sbit**的绝对位地址。例如：

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

注意:

特殊功能位代表一个独立的声明类，它不能和别的位声明或位域互换。

***sbit** 数据类型声明可以用做访问用**bdata** 存储类型标识符声明的变量的位。参考99页的“位可寻址目标”。*

绝对变量定位

在 C 程序中用**_at_**关键词，变量可以定位在绝对存储地址。用法如下：

*type [memory_space] variable_name **_at_** constant；*

这里：

memory_space 变量的存储空间。如果在声明中没有，则使用缺省的存储空间。缺省的存储空间参考94页的“存储模式”。

type 变量类型。

variable_name 变量名。

constant 定位变量的地址。

_at_后面的绝对地址必须在可用的实际存储空间内。**C*x*51** 编译器检查无效的地址标识符。

注意：
*如果用**_at_**关键词声明一个变量来访问一个 XDATA 外围设备，应使用 **volatile** 关键词以确保 C 编译器不进行优化以便能访问到要访问的存储区。*

在绝对变量定位中有下面限制：

1.绝对变量不能初始化。
2.**bit** 类型的函数和变量不能定位到一个绝对地址。

下面的例子示范如何用**_at_**定位几个不同的变量类型。

```
struct link
  {
  struct link idata *next;
  char        code  *test;
  };

struct link list idata _at_ 0x40;    /* list at idata 0x40 */
char xdata text[256]   _at_ 0xE000;  /* array at xdata 0xE000 */
int xdata i1           _at_ 0x8000;  /* int at xdata 0x8000 */

void main ( void ) {
   link.next = (void *) 0;
   i1        = 0x1234;
   text [0]  = 'a';
```

可以在一个源代码模块中声明一个变量，而在另一个模块中使用。用下面的外部声明在另外的源文件中访问上面定义的**_at_**变量。

```
struct link
  {
  struct link idata *next;
  char        code  *test;
  };

extern struct link idata list;    /* list at idata 0x40 */
extern char xdata text[256];      /* array at xdata 0xE000 */
extern int xdata i1;              /* int at xdata 0x8000 */
```

指针

C*x*51 编译器用*字符支持变量指针的声明。**C*x*51** 指针可用在所有标准 C 中可用的操作中。但是，因为 8051 和派生系列的独特结构，**C*x*51** 编译器提供两个类型的指针：通用指针和指定存储区指针。这些指针类型和指针变换方法在下面的节中说明。

通用指针

通用指针和标准 C 指针的声明相同。例如：

```
char *s;          /* string ptr */
int *numptr;      /* int ptr */
long * state;     /* Texas */
```

通用指针用三个字节保存。第一个字节是存储类型，第二个是偏移的高字节，第三是偏移的低字节。通用指针可访问 8051 存储空间内的任何变量。许多 **C*x*51** 库函数因而用了这些指针类型。通过这些通用指针，函数可以访问存储区中的所有数据。

注意：
通用指针产生的代码比指定存储区指针的要慢，因为存储区在运行前是未知的。编译器不能优化存储区访问，必须产生可以访问任何存储区的通用代码。如果优先考虑执行速度，应该尽可能的用指定存储类型的指针而不是通用指针。

下面的代码和汇编列表显示在不同的存储区中分配给通用指针的值。注意第一个值是存储空间，然后是地址的高字节和低字节。

```
stmt level      source
    1           char *c_ptr;        /* char ptr */
    2           int  *i_ptr;        /* int ptr */
    3           long *l_ptr;        /* long ptr */
```

```
   4
   5          void main (void)
   6          {
   7     1    char data dj;          /* data vars */
   8     1    int  data dk;
   9     1    long data dl;
  10     1
  11     1    char xdata xj;         /* xdata vars */
  12     1    int  xdata xk;
  13     1    long xdata xl;
  14     1
  15     1    char code cj = 9;      /* code vars */
  16     1    int  code ck = 357;
  17     1    long code cl = 123456789;
  18     1
  19     1
  20     1    c_ptr = &dj;           /* data ptrs */
  21     1    i_ptr = &dk;
  22     1    l_ptr = &dl;
  23     1
  24     1    c_ptr = &xj;           /* xdata ptrs */
  25     1    i_ptr = &xk;
  26     1    l_ptr = &xl;
  27     1
  28     1    c_ptr = &cj;           /* code ptrs */
  29     1    i_ptr = &ck;
  30     1    l_ptr = &cl;
  31     1    }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                           ; SOURCE LINE # 5
                           ; SOURCE LINE # 6
                           ; SOURCE LINE # 20
0000 750000 R        MOV   c_ptr,#00H
0003 750000 R        MOV   c_ptr+01H,#HIGH dj
0006 750000 R        MOV   c_ptr+02H,#LOW dj
                           ; SOURCE LINE # 21
0009 750000 R        MOV   i_ptr,#00H
000C 750000 R        MOV   i_ptr+01H,#HIGH dk
000F 750000 R        MOV   i_ptr+02H,#LOW dk
                           ; SOURCE LINE # 22
0012 750000 R        MOV   l_ptr,#00H
0015 750000 R        MOV   l_ptr+01H,#HIGH dl
0018 750000 R        MOV   l_ptr+02H,#LOW dl
                           ; SOURCE LINE # 24
001B 750001 R        MOV   c_ptr,#01H
001E 750000 R        MOV   c_ptr+01H,#HIGH xj
0021 750000 R        MOV   c_ptr+02H,#LOW xj
                           ; SOURCE LINE # 25
0024 750001 R        MOV   i_ptr,#01H
0027 750000 R        MOV   i_ptr+01H,#HIGH xk
002A 750000 R        MOV   i_ptr+02H,#LOW xk
                           ; SOURCE LINE # 26
002D 750001 R        MOV   l_ptr,#01H
0030 750000 R        MOV   l_ptr+01H,#HIGH xl
0033 750000 R        MOV   l_ptr+02H,#LOW xl
                           ; SOURCE LINE # 28
0036 7500FF R        MOV   c_ptr,#0FFH
```

```
0039 750000 R   MOV   c_ptr+01H,#HIGH cj
003C 750000 R   MOV   c_ptr+02H,#LOW cj
                      ; SOURCE LINE # 29
003F 7500FF R   MOV   i_ptr,#0FFH
0042 750000 R   MOV   i_ptr+01H,#HIGH ck
0045 750000 R   MOV   i_ptr+02H,#LOW ck
                      ; SOURCE LINE # 30
0048 7500FF R   MOV   l_ptr,#0FFH
004B 750000 R   MOV   l_ptr+01H,#HIGH cl
004E 750000 R   MOV   l_ptr+02H,#LOW cl
                      ; SOURCE LINE # 31
0051 22               RET
      ; FUNCTION main (END)
```

在上面的列表例子中，通用指针 c_ptr，i_ptr，和 l_ptr 都存储在 8051 的内部数据存储区中。但是，应该用一个存储类型标识符指定一个通用指针的存储区。例如：

```
char * xdata strptr;          /* generic ptr stored in xdata */
int  * data numptr;           /* generic ptr stored in data */
long * idata varptr;          /* generic ptr stored in idata */
```

这些例子指向可能保存在任何存储区中的变量。但是，指针分别保存在 **xdata**，**data**，和 **idata** 中。

指定存储区的指针

指定存储区的指针在指针的声明中经常包含一个存储类型标识符，指向一个确定的存储区。例如：

```
char data *str;          /* ptr to string in data */
int xdata *numtab;       /* ptr to int(s) in xdata */
long code *powtab;       /* ptr to long(s) in code */
```

因为存储类型在编译时是确定的，通用指针所需的存储类型字节在指定存储区的指针是不需要的。指定存储区指针只能用一个字节（**idata**，**data**，**bdata**，和 **pdata** 指针）或两字节（**code** 和 **xdata** 指针）。

注意：
一个指定存储区指针产生的代码比一个通用指针产生的代码运行速度快。这是因为存储区在编译时而非运行时就知道。编译器可以用这些信息优化存储区访问。如果运行速度优先，就应尽可能的用指定存储区指针。

象通用指针一样，可以指定一个指定存储区指针的保存存储区。在指针声明前加一个存储类型标识符。例如：

```
char data * xdata str;       /* ptr in xdata to data char */
int xdata * data numtab;     /* ptr in data to xdata int */
long code * idata powtab;    /* ptr in idata to code long */
```

指定存储区指针只用来访问声明在 8051 存储区的变量。指定存储区指针提供更有效的方法访问数据目标，但代价是损失灵活性。

下面的代码和汇编列表显示指针值和指定存储区指针是如何关联的。注意这些指针产生的代码比前一节中通用指针产生代码要少的多。

```
stmt level  source

   1        char data  *c_ptr;      /* memory-specific char ptr */
   2        int  xdata *i_ptr;      /* memory-specific int ptr */
   3        long code  *l_ptr;      /* memory-specific long ptr */
   4
   5        long code powers_of_ten [] =
   6          {
   7          1L,
   8          10L,
   9          100L,
  10          1000L,
  11          10000L,
  12          100000L,
  13          1000000L,
  14          10000000L,
  15          100000000L
  16          };
  17
  18        void main (void)
  19        {
  20   1    char data strbuf [10];
  21   1    int xdata ringbuf [1000];
  22   1
  23   1    c_ptr = &strbuf [0];
  24   1    i_ptr = &ringbuf [0];
  25   1    l_ptr = &powers_of_ten [0];
  26   1    }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)
                        ; SOURCE LINE # 18
                        ; SOURCE LINE # 19
                        ; SOURCE LINE # 23
0000 750000 R     MOV   c_ptr,#LOW strbuf
                        ; SOURCE LINE # 24
0003 750000 R     MOV   i_ptr,#HIGH ringbuf
0006 750000 R     MOV   i_ptr+01H,#LOW ringbuf
                        ; SOURCE LINE # 25
0009 750000 R     MOV   l_ptr,#HIGH powers_of_ten
000C 750000 R     MOV   l_ptr+01H,#LOW powers_of_ten
                        ; SOURCE LINE # 26
000F 22             RET
      ; FUNCTION main (END)
```

指针转化

C*x*51 编译器可能在指定存储区指针和通用指针间转化。指针转化可以用类型转换的直接程序代码来强迫转化，或由编译器暗中强制转化。

当把指定存储区指针作为一个参数传递给一个要求通用指针的函数，**C*x*51** 编译器就把指定存储区指针转化为通用指针。这如 **printf**，**sprintf**，和 **gets** 等用通用指针作为参数的函数。例如：

```
extern int printf(void *format,…);

extern int myfunc(void code *p,int xdata *pq);

int xdata *px;

char code *fmt = *value = %d | %4XH\n“;

void debuf_print(void) {

    printf(fmt,*px,*px);          /* fmt is converted */
    myfunc(fmt,px);               /* no conversions */
}
```

在调用 **printf** 中，参数 **fmt** 代表一个 2 字节 **code** 指针，自动转化或强迫转化成一个 3 字节的通用指针。这是因为 **printf** 的原型要求一个通用指针作为第一个参数。

注意：

*一个指定存储区的指针作为一个函数的参数，如果没有函数原型，就经常被转化成一个通用指针。如果调用的函数确实希望一个短指针作为一个参数，这可能引起错误。要在程序中避免这种错误，可用**#include** 文件和所有外部函数的原型。这确保编译器进行必须的类型转化，确保编译器检测类型转化错误。*

下面的表详细列出了转化通用指针（generic *）为指定存储区指针（**code ***，**xdata ***，**idata ***，**data ***，**pdata ***）的过程。

转化类型	说明
generic *到**code ***	用了通用指针的偏移段（2字节）。
generic *到**xdata ***	用了通用指针的偏移段（2字节）。
generic *到**data ***	用了通用指针的低字节。高字节丢弃。
generic *到**idata ***	用了通用指针的低字节。高字节丢弃。
generic *到**pdata ***	用了通用指针的低字节。高字节丢弃。

下面的表详细列出了从指定存储区指针（**code ***，**xdata ***，**idata ***，**data ***，**pdata ***）转化到通用指针的过程（generic *）。

转化类型	说明
**code ***到generic *	对**code**，通用指针的存储类型设为0XFF，用了原**code ***的2字节偏移段。
**xdata ***到generic *	对**xdata**，通用指针的存储类型设为0X01，用了原**xdata ***的2字节偏移段。
**data ***到generic *	**idata */data ***的1字节偏移转化为一个**unsigned int**类型的偏移。
**idata ***到generic *	对**idata/data**，通用指针的的存储类型设为0X00。
**pdata ***到generic *	对**pdata**，通用指针的存储类型设为0XFE。**pdata ***的1字节偏移转化为一个**unsigned int**的偏移。

下面的列表例举了几个指针转化和结果代码：

```
stmt level  source
   1        int *p1;           /* generic ptr (3 bytes) */
   2        int xdata *p2;     /* xdata ptr (2 bytes) */
   3        int idata *p3;     /* idata ptr (1 byte) */
   4        int code *p4;      /* code ptr (2 bytes */
   5
   6        void pconvert (void) {
   7   1    p1 = p2;           /* xdata* to generic* */
   8   1    p1 = p3;           /* idata* to generic* */
   9   1    p1 = p4;           /* code*  to generic* */
  10   1
  11   1    p4 = p1;           /* generic* to code* */
  12   1    p3 = p1;           /* generic* to idata* */
  13   1    p2 = p1;           /* generic* to xdata* */
  14   1
  15   1    p2 = p3;           /* idata* to xdata* (WARN) */
*** WARNING 259 IN LINE 15 OF P.C: pointer: different mspace
  16   1    p3 = p4;           /* code*  to idata* (WARN) */
*** WARNING 259 IN LINE 16 OF P.C: pointer: different mspace
  17   1    }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
      ; FUNCTION pconvert (BEGIN)
                         ; SOURCE LINE # 7
0000 750001 R   MOV  p1,#01H
0003 850000 R   MOV  p1+01H,p2
0006 850000 R   MOV  p1+02H,p2+01H
                         ; SOURCE LINE # 8
0009 750000 R   MOV  p1,#00H
000C 750000 R   MOV  p1+01H,#00H
000F 850000 R   MOV  p1+02H,p3
                         ; SOURCE LINE # 9
0012 7B05       MOV  R3,#0FFH
0014 AA00   R   MOV  R2,p4
0016 A900   R   MOV  R1,p4+01H
0018 8B00   R   MOV  p1,R3
001A 8A00   R   MOV  p1+01H,R2
001C 8900   R   MOV  p1+02H,R1
                         ; SOURCE LINE # 11
001E AE02       MOV  R6,AR2
0020 AF01       MOV  R7,AR1
0022 8E00   R   MOV  p4,R6
0024 8F00   R   MOV  p4+01H,R7
                         ; SOURCE LINE # 12
0026 AF01       MOV  R7,AR1
0028 8F00   R   MOV  p3,R7
                         ; SOURCE LINE # 13
002A AE02       MOV  R6,AR2
002C 8E00   R   MOV  p2,R6
002E 8F00   R   MOV  p2+01H,R7
                         ; SOURCE LINE # 15
0030 750000 R   MOV  p2,#00H
0033 8F00   R   MOV  p2+01H,R7
                         ; SOURCE LINE # 16
0035 850000 R   MOV  p3,p4+01H
                         ; SOURCE LINE # 17
0038 22         RET
      ; FUNCTION pconvert (END)
```

绝对指针

绝对指针类型可访问任何存储区的存储区地址。也可用绝对指针调用定位在绝对或固定地址的函数。

下面的变量用代码例子说明绝对指针类型。

```
char xdata *px;        /* ptr to xdata */
char idata *pi;        /* ptr to idata */
char code  *pc;        /* ptr to code  */

char c;                /* char variable in data space */
int  i;                /* int variable in data space */
```

下面的例子关联 C 函数 **main** 的地址到一个指针（保存在 **data** 存储区），指针指向一个保存在 **code** 存储区的 **char**。

源码	pc = (void *)main;		
目标码	0000 750000 R	MOV	pc,#HIGH mian
	0003 750000 R	MOV	pc+01H,#LOW main

下面的例子指定变量 **i** 的地址（是一个 **int data ***）到一个在 **idata** 中的 **char** 指针。因为 **i** 保存在 **data** 中，因为间接访问 **data** 是 **idata**，这指针转化是有效的。

源码	pi = (char idata *)&i;	
目标码	0000 750000 R	MOV pi,#LOW I

下面的例子指定在 **xdata** 中的 **char** 指针，到一个在 **idata** 中的 **char** 指针。因为 **xdata** 指针占用 2 个字节，**idata** 指针占用一个字节，这个指针转化不能产生想得到的结果，因为 **xdata** 的高字节被忽略。参考 111 页的“指针转化”中的不同指针类型转化的内容。

源码	pi = (char idata *)px;	
目标码	0000 750000 R	MOV pi,px+01H

下面的例子指定 0x1234 为一个 **code** 存储区的 **char** 指针。

源码	pc = (char idata *)0x1234;	
目标码	0000 750012 R	MOV pc,#012H
	0003 750034 R	MOV pc+01H,#034H

下面的例子用 0XFF00 作为一个函数的指针，没有参数，返回一个 **int**，调用函数，把返回值给变量 **i**。本例子实行函数指针类型转换的部分是：((int(code *)(void)) 0xFF00)。加参数列表到函数指针的后面，编译器可以正确的调用函数。

源码	i = ((int (code *)(void)) 0xFF00)();
目标码	0000 12FF00 LCALL 0FF00H 0003 8E00 R MOV i,R6 0005 8F00 R MOV i+01H,R7

下面的例子把 0x8000 作为一个指向 **code** 存储区的 **char** 指针，提取指向的 **char** 值，并赋给变量 **c**。

源码	c = *((char code *) 0x8000);
目标码	0000 908000 MOV DPTR,#08000H 0003 E4 CLR A 0004 93 MOVC A,@A+DPTR 0005 F500 R MOV C,A

下面的例子把 0xF0 作为一个指向 **idata** 存储区的 **char** 指针，提取指向的 **char** 值，并赋给变量 **c**。

源码	c += *((char idata *) 0xF0);
目标码	0000 78F0 MOV DPTR,#0F0H 0002 E6 MOV A,@R0 0003 2500 R ADD A,C 0005 F500 R MOV C,A

下面的例子把 0xE8 作为一个指向 **pdata** 存储区的 **char** 指针，提取指向的 **char** 值，并赋给变量 **c**。

源码	c += *((char pdata *) 0xE8);
目标码	0000 78E8 MOV R0,#0E8H 0002 E2 MOVX A,@R0 0003 2500 R ADD A,C 0005 F500 R MOV C,A

下面的例子把 0x2100 作为一个指向 **code** 存储区的 **int** 指针，提取指向的 **int** 值，并赋给变量 **i**。

源码	i = *((int code *) 0x2100);
目标码	0000 902100 MOV DPTR,#02100H 0003 E4 CLR A 0004 93 MOVC A,@A+DPTR 0005 FE R MOV R6,A 0006 7401 MOV A,#01H 0008 93 MOVC A,@A+DPTR 0009 8E00 R MOV i,R6 000B F500 R MOV i+01H,A

下面的例子把 0x4000 作为一个在 **xdata** 的指针指向一个 **xdata** 存储区的 **char** 指针，提取指向的 **char** 值。

源码	px = *((char xdata * xdata *) 0x4000);
目标码	0000 904000 MOV DPTR,#04000H 0003 E0 MOVX A,@DPTR 0004 FE MOV R6,A 0005 A3 INC DPTR 0006 E0 MOVX A,@DPTR 0007 8E00 R MOV px,R6 0009 F500 R MOV px+01H,A

象上面的例子，本例子把 0x4000 作为一个在 **xdata** 中指向一个 **xdata** 存储区的 **char** 指针。但是，指针作为一个在 xdata 中的指针数组。访问数组元素 0（保存在 **xdata** 的 0x4000），提取指向的 **char** 值。

源码	px = ((char xdata * xdata *) 0x4000)[0];
目标码	0000 904000 MOV DPTR,#04000H 0003 E0 MOVX A,@DPTR 0004 FE MOV R6,A 0005 A3 INC DPTR 0006 E0 MOVX A,@DPTR 0007 8E00 R MOV px,R6 0009 F500 R MOV px+01H,A

下面的例子和前面的一样，只是访问数组的元素 1。因为目标指向一个在 **xdata** 中的指针（指向一个 **char**），每个元素的大小是 2 个字节。程序访问数组元素 1（保存在 **xdata** 的 0x4002），提取保存在 **xdata** 的 **char** 值。

源码	px = ((char xdata * xdata *) 0x4000)[1];
目标码	0000 904002 MOV DPTR,#04002H 0003 E0 MOVX A,@DPTR 0004 FE MOV R6,A 0005 A3 INC DPTR 0006 E0 MOVX A,@DPTR 0007 8E00 R MOV px,R6 0009 F500 R MOV px+01H,A

函数声明

C*x*51 编译器扩展了标准 C 函数声明。这些扩展有：

- 指定一个函数作为一个中断函数
- 选择所用的寄存器组
- 选择存储模式
- 指定重入
- 指定 ALIEN（PL/M51）函数

在函数声明中可以包含这些扩展或属性。用下面标准格式来声明 **C*x*51** 函数。

[*return_type*]*funcname*([*args*]) [{**small**|**compact**|**large**}]
[**reentrant**][**interrupt** ***n***][**using** ***n***]

这里：

return_type	函数返回值的类型。如果不指定，缺省是 int。
funcname	函数名。
args	函数的参数列表。
small，compact 或 large	函数的存储模式。
reentrant	表示函数是递归的或可重入的。
interrupt	表示是一个中断函数。
using	指定函数所用的寄存器组。

在下面详细说明这些属性和别的特征。

函数参数和堆栈

在传统的 8051 中堆栈指针只能访问内部数据区。**Cx51** 编译器把堆栈定位在内部数据区的所有变量的后面。堆栈指针间接访问内部存储区，可以使用 0xFF 前的所有内部数据区。

传统 8051 的总的堆栈空间是受限的：最多只有 256 字节。除了用堆栈传递函数参数，**Cx51** 编译器对每个函数参数分配一个特定地址。当函数被调用时，调用者在传递控制权前必须拷贝参数到分配好的存储区。函数就可从固定的存储区提取参数。在这个过程中只有返回地址保存在堆栈中。中断函数要求更多的堆栈空间，因为必须切换寄存器组，需要保存一些寄存器值在堆栈中。

注意：

***Cx51** 编译器用一些 8051 变种的扩展的堆栈空间。这样堆栈空间可以增加到几 K 字节。*

缺省的，**Cx51** 编译器可以最多用寄存器传递三个参数。这可以提高运行速度。可参考 120 页的“用寄存器传递参数”。

注意：

一些派生 8051 只提供 64 字节的片内数据区；大多数有 256 字节。在决定存储模式时应考虑这个因素，因为片内 data 和 idata 直接影响堆栈空间的大小。

用寄存器传递参数

C*x*51 编译器允许用 CPU 寄存器传递三个参数。这可以明显的提高系统性能。参数传递可以用 **REGPARMS** 和 **NOREGPARMS** 控制命令来控制。

下面的表列出了不同参数位置和数据类型所用的寄存器。

参数数目	char，1字节指针	int，2字节指针	long，float	通用指针
1	R7	R6&R7	R4-R7	R1-R3
2	R5	R4&R5	R4-R7	R1-R3
3	R3	R2&R3		R1-R3

如果没有寄存器可用来传递参数，固定存储区被使用。

函数返回值

CPU 寄存器经常用来返回函数值。下面的表列出了返回类型和所用的寄存器。

返回类型	寄存器	说明
bit	CF	
char，unsigned char，1字节指针	R7	
int，unsigned int，2字节指针	R6&R7	MSB在R6，LSB在R7
long，unsigned long	R4-R7	MSB在R4，LSB在R7
float	R4-R7	32位IEEE格式
通用指针	R1-R3	存储类型在R3，MSB R2，LSB R1

注意：

*如果函数的第一个参数是一个 **bit** 类型，那么别的参数不能用寄存器传递。这是因为寄存器传递参数不符合上面的计划。因此，**bit** 参数应该在参数的最后声明。*

指定函数的存储模式

一个函数的参数和局部变量保存在由存储模式指定的缺省存储空间中。参考 94 页的“存储模式”。

但是，对单个函数，可以在函数声明中用 **small**，**compact**，或 **large** 声明来指定存储模式。例如：

```
#pragma small         /* Default to small model */

extern int calc (char i, int b) large reentrant;
extern int func (int i, float f) large;
extern void *tcp (char xdata *xp, int ndx) small;


int mtest (int i, int y)            /* Small model */
  {
  return (i * y + y * i + func(-1, 4.75));
  }


int large_func (int i, int k) large /* Large model */
  {
  return (mtest (i, k) + 2);
```

函数使用 **SMALL** 存储模式的好处是局部变量和函数参数保存在 8051 内部 RAM。因此，数据访问很有效。内部存储区是有限的。偶尔，SMALL 模式不能满足程序的要求，就必须用别的存储模式。在这种情况，必须声明一个函数用别的存储模式。

在函数声明中指定函数模式属性，应该选择所用的三个可能的重入堆栈和帧指针。在 **SMALL** 模式，堆栈访问比 **LARGE** 模式效率高。

指定一个函数的寄存器组

所有 8051 系列的最低 32 个字节分成 4 组 8 寄存器组。作为寄存器 R0 到 R7 访问。寄存器组由 **PSW** 的两位选择。在处理中断或使用一个实时操作系统时寄存器组非常有用。不用保存 8 个寄存器，在中断中，CPU 可以切换到一个不同的寄存器组。

using 函数属性用来指定一个函数所用的寄存器组。例如：

```
void rb_function(void) using 3
{
.
.
.
}
```

using 属性为一个 0 到 3 的整常数。带操作数的表达式是不允许的。**using** 属性在函数原型中不允许。**using** 属性影响如下的函数的目标代码：

- 在函数入口保存当前选择的寄存器组在堆栈中。
- 设置指定的寄存器组。
- 在函数出口恢复前面的寄存器组。

下面的例子显示如何指定 **using** 函数属性，函数入口和出口产生的汇编代码。

```
stmt level  source

   1
   2          extern bit alarm;
   3          int alarm_count;
   4          extern void alfunc (bit b0);
   5
   6          void falarm (void) using 3  {
   7   1            alarm_count++;
   8   1            alfunc (alarm = 1);
   9   1          }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION falarm (BEGIN)
0000 C0D0        PUSH  PSW
0002 75D018      MOV   PSW,#018H
                       ; SOURCE LINE # 6
                       ; SOURCE LINE # 7
0005 0500    R   INC   alarm_count+01H
0007 E500    R   MOV   A,alarm_count+01H
0009 7002        JNZ   ?C0002
000B 0500    R   INC   alarm_count
000D  ?C0002:
                       ; SOURCE LINE # 8
000D D3          SETB  C
000E 9200    E   MOV   alarm,C
0010 9200    E   MOV   ?alfunc?BIT,C
0012 120000 E    LCALL alfunc
                       ; SOURCE LINE # 9
0015 D0D0        POP   PSW
0017 22          RET
      ; FUNCTION falarm (END)
```

在前面的例子中，在 0000h 开始的代码，保存原始的 **PSW** 在堆栈中，设置新的寄存器组。从 0015h 开始的代码，从堆栈中弹出 **PSW**，恢复原来的寄存器组。

注意：

using *属性不能用在用寄存器返回一个值的函数中。必须确保寄存器组切换在可控范围内。否则可能产生错误。即使使用相同的寄存器组，用* ***using*** *声明函数不能返回一个 bit 值。*

---

using 属性在 **interrupt** 函数最有用。通常对每个中断优先级指定一个不同的寄存器组。因此，可以分配一个寄存器组对所有非中断代码，另一个寄存器组为高级的中断，第三个寄存器组为低级中断。

寄存器组访问

C*x*51 编译器定义了一个函数的缺省寄存器组。**REGISTBANK** 控制命令在一个源文件中指定所有函数的缺省寄存器组。但是，本命令不能产生代码切换寄存器组。

在启动后，8051 加载 PSW 为 00h，选择寄存器组 0。缺省，所有非中断函数都用寄存器组 0。要改变这个，必须：

- 修改启动代码选择不同的寄存器组。
- 用 REGISTERBANK 控制命令指定新的寄存器组号。

缺省的，**C*x*51** 编译器用绝对地址访问寄存器 R0—R7。这可得到最高性能。绝对寄存器访问由 **AREGS** 和 **NOAREGS** 控制命令控制。

采用绝对寄存器访问的函数不能被用不同的寄存器组的函数调用。否则可能引起不可预知的结果，因为调用函数假定一个不同的寄存器组被选择。

为了使函数对当前寄存器组不受影响，函数必须用 **NOAREGS** 控制命令编译。这对一个从主程序和一个用不同的寄存器组的中断函数被调用的函数来说是有用的。

注意：

Cx51 *编译器不能检测函数间的寄存器组的不匹配。因此，使用交替寄存器组的函数，只能调用不设定一个缺省寄存器组的别的函数。*

参考 17 页的“第二章. 用 **C*x*51** 编译”中 **REGISTERBANK**，**AREGS** 和 **NOAREGS** 命令。

中断函数

8051 和派生系列提供许多硬件中断，可用来计数，计时，检测外部事件，和发送和接收串口数据。一个 8051 的标准中断如下表：

中断号	中断说明	地址
0	外部中断0	0003h
1	计时/计数器0	000Bh
2	外部中断1	0013h
3	计时/计数器1	001Bh
4	串口	0023h

8051 的新型号加了更多的中断。**C*x*51** 编译器支持 32 个中断函数。用下面表中的矢量地址决定中断号。

Interrupt Number	Address
0	0003h
1	000Bh
2	0013h
3	001Bh
4	0023h
5	002Bh
6	0033h
7	003Bh
8	0043h
9	004Bh
10	0053h
11	005Bh
12	0063h
13	006Bh
14	0073h
15	007Bh

Interrupt Number	Address
16	0083h
17	008Bh
18	0093h
19	009Bh
20	00A3h
21	00ABh
22	00B3h
23	00BBh
24	00C3h
25	00CBh
26	00D3h
27	00DBh
28	00E3h
29	00EBh
30	00F3h
31	00FBh

interrupt 函数属性，当包含在一个声明中，指定函数为一个中断函数。例如：

```
unsigned int  interruptcnt;
unsigned char second;

void timer0 (void) interrupt 1 using 2  {
  if (++interruptcnt == 4000)  {    /* count to 4000 */
    second++;                       /* second counter    */
    interruptcnt = 0;               /* clear int counter */
  }
```

interrupt 属性的参数为 0 到 31 的整常数值。带操作数的表达式和 **interrupt** 属性在函数原型中是不允许的。**interrupt** 属性影响如下函数的目标代码：

- SFR **ACC**，**B**，**DPH**，**DPL** 和 **PSW** 的内容，在需要时，在函数调用时保存在堆栈中。
- 在中断函数中所用的寄存器，如果不用 **using** 属性指定一个寄存器组，就保存在堆栈中。
- 保存在堆栈中的寄存器和 SFR 在退出函数前恢复。
- 函数由指令 **RETI** 终止。

另外，**Cx51** 编译器自动产生中断矢量。

下面的例子程序说明了如何使用 **interrupt** 属性。程序同时显示进入和退出中断函数的代码。**using** 函数属性用来选择和非中断程序不同的寄存器组。但是，因为在本函数中不需要工作寄存器，为切换寄存器组而产生的代码被优化排除了。

```
stmt level  source

   1        extern bit alarm;
   2        int alarm_count;
   3
   4
   5        void falarm (void) interrupt 1 using 3  {
   6   1      alarm_count *= 2;
   7   1      alarm = 1;
   8   1      }


ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION falarm (BEGIN)
0000 C0E0        PUSH  ACC
0002 C0D0        PUSH  PSW
                       ; SOURCE LINE # 5
                       ; SOURCE LINE # 6
0004 E500   R    MOV   A,alarm_count+01H
0006 25E0        ADD   A,ACC
0008 F500   R    MOV   alarm_count+01H,A
000A E500   R    MOV   A,alarm_count
000C 33          RLC   A
000D F500   R    MOV   alarm_count,A
                       ; SOURCE LINE # 7
000F D200   E    SETB   alarm
                       ; SOURCE LINE # 8
0011 D0D0        POP   PSW
0013 D0E0        POP   ACC
0015 32          RETI
      ; FUNCTION falarm (END)
```

在上面的例子中，注意，**ACC** 和 **PSW** 寄存器保存在 0000H 开始的存储区，从 0011H 中恢复。**RETI** 指令退出中断。

下面的规则适用于中断函数：

- 中断函数没有函数参数。如果中断函数声明中带参数，编译器就产生错误信息。

- 中断函数声明不能包含返回值。必须声明为 VOID（参考上面的例子）。如果定义了一个返回值，编译器就产生一个错误。暗含的 **int** 返回值，被编译器忽略。

- 编译器不允许直接的对中断函数的调用。对中断函数的直接调用是无意义的，因为退出程序指令 **RETI** 影响 8051 的硬件中断系统。因为没有硬件存在中断请求，本指令的结果是不确定的，通常是致命的。不要通过一个函数指针间接调用一个中断函数。

- 编译器对每个中断函数产生一个中断矢量。矢量的代码是跳转到中断函数的起始。在 **Cx51** 命令行可用 **NOINTVECTOR** 控制命令禁止产生中断矢量。在这种情况，必须从单独的汇编模块提供中断矢量。参考 **INTVECOTOR** 和 **INTERVAL** 控制命令。

- **Cx51** 编译器的中断号为 0-31。参考具体的派生的 8051 文件决定可用的中断。

- 从一个中断程序中调用函数，必须和中断使用相同的寄存器组。当没用 **NOAREGS** 明确的声明，编译器将使用绝对寄存器访问函数选择（用 **using** 或 **REGISTERBANK** 控制）的寄存器组。当函数假定的和实际所选的寄存器组不同时，将产生不可预知的结果。参考 124 页的“寄存器组访问”。

可重入函数

一个可重入函数可在同一时间被几个进程共享。当一个可重入函数运行时，别的进程可中断执行，并开始执行相同的可重入函数。正常情况，**C*x*51** 编译器中的函数不能重入。原因是函数参数和局部变量保存在固定的存储区中。**reentrant** 函数属性允许声明函数可重入，因此，可重复调用。例如：

```
int calc(char I,int b)              reentrant  {
    int x;
    x = table[i];
    return (x * b);
}
```

可重入函数可以被递归调用，可*同时*被两个或多个进程调用。可重入函数经常在实时应用或在中断和非中断必须共用一个函数的情况下被使用。

在上面例子中，可有选择的定义（用 **reentrant** 属性）函数为可重入。对每个可重入函数，一个可重入堆栈区同时在内部和外部存储区（由存储模式决定）模拟，如：

- SMALL 模式可重入函数在 **idata** 存储区模拟可重入堆栈。
- COMPACT 模式可重入函数在 **pdata** 存储区模拟可重入堆栈。
- LARGE 模式可重入函数在 **xdata** 存储区模拟可重入堆栈。

可重入函数用缺省存储模式决定哪块存储空间用做可重入堆栈。可指定函数的存储模式（用 **small**，**compact**，和 **large** 函数属性）。参考 121 页的“指定一个函数的存储模式”。

下面的规则适用于用 **reentrant** 属性声明的函数。

- **bit** 类型的函数参数不能使用。局部的 **bit** 变量也不可用。重入性能不支持位寻址变量。
- 可重入函数不能从 **alien** 函数调用。
- 可重入函数不能用 **alien** 属性标识符使能 PL/M-51 参数传递规则。
- 一个可重入函数可同时有别的属性，如 **using** 和 **interrupt**，可包含一个明确的存储模式属性（**small**，**compact**，**large**）。
- 返回地址保存在 8051 的硬件堆栈中。任何别的要求 **PUSH** 和 **POP** 的操作也影响 8051 硬件堆栈。
- 用不同的存储模式的可重入函数可混在一起。但是，每个可重入函数必须有完整原型，必须在原型中包含存储模式属性。这对调用程序，安排函数参数在正确的可重入堆栈是必须的。
- 三种可能的可重入模式中的每种包含自己的可重入堆栈区和堆栈指针。例如，如果 **small** 和 **large** 可重入函数在一个模块中声明，SMALL 和 LARGE 可重入堆栈和两个关联的堆栈指针（一个对 SMALL，一个对 LARGE）都建立。

可重入堆栈模拟结构是低效的，但是必需的，因为在 8051 中缺乏一种有效的寻址方法。因此，只能用可重入函数。

可重入函数所用的模拟堆栈有自己的堆栈指针，它独立于8051堆栈和堆栈指针。堆栈和堆栈指针在 **STARTUP.A51** 文件中定义和初始化。

下面的表详细的列出了堆栈指针汇编变量名，数据区，和三种存储模式的每个的大小。

模式	堆栈指针	堆栈区
SMALL	**?C_IBP(1 字节）**	间接访问内部存储区（idata），堆栈区最大 256 字节。
COMPACT	**?C_PBP(1 字节)**	外部页寻址存储区（pdata），堆栈区最大 256 字节。
LARGE	**?C_XBP(2 字节)**	外部可访问存储区（xdata），堆栈区最大 64K 字节。

可重入函数的模拟堆栈区从上到下生长。8051 硬件堆栈与之相反，是从下到上。当在 **SMALL** 存储模式，模拟堆栈和 8051 硬件堆栈分享共同的存储区，但方向相反。

模拟堆栈和堆栈指针在 **C*x*51** 起始代码 **STARTUP.A51**（在 **LIB** 子目录）中声明和初始化。必须修改起始代码以指定初始化哪个模拟堆栈给可重入函数使用。也可在起始代码中修改模拟堆栈的顶部。参考 151 页的“STARTUP.A51”。

ALIEN函数（PL/M-51接口）

可从C中调用用PL/M-51写的程序，在外部用**alien**函数类型标识符声明。例如：

```
extern alien char plm_func(int,char);

char c_func(void) {
    int i;
    char c;

    for( i=0;i<100;i++) {
        c = plm_func(i,c);              /* call PL/M func */
    }
    return( c );
}
```

可建立被 PL/M-51 程序调用的 C 函数。要这样，用 **alien** 函数类型标识符声明 C 函数。例如：

```
alien char c_func(char a,int b) {
        return( a*b);
}
```

PL/M-51 函数的参数和返回值可以是下面中的任何类型：**bit**，**char**，**unsigned char**，**int**，和 **unsigned int**。别的类型，包括 **long**，**float**，和所有类型的指针，可以在用 **alien** 类型标识符声明的函数中。但是，用这些类型需要小心，因为 PL/M-51 不直接支持 32 位二进制整数或浮点数。

在 PL/M-51 中声明的公共变量，在外部和 C 变量一样声明，也可被 C 程序使用。

实时任务函数

C*x*51 编译器提供对 **RTX51 完整版**和 **RTX51 简装版**实时多任务操作系统的支持，通过使用**_task_**和**_priority_**关键词。**_task_**关键词定义一个函数为一个实时任务。**_priority_**关键词指定任务的优先级。

例如：

```
void func(void) _task_ num _priority_ pri
```

这里：

num 对 **RTX51 完整版**，是 0 到 255 的任务 ID 号，对 **RTX51 简装版**是 0 到 15 的任务 ID 号。

pri 任务的优先级。*参考 RTX51 用户手册*，或 *RTX51 简装版用户手册*。

任务函数必须用一个 void 返回类型和一个 void 参数列表声明。

第四章. 预处理器

Cx51编译器内建的预处理器处理在源文件中发现的命令。**Cx51**编译器支持所有标准的ANSI C命令。本章描述了预处理器的主要部分。

命令

预处理器命令必须放在一行的开头。所有的命令都以‘#’为前缀。例如：

```
#pragma
#include <stdio.h>
#define DEBUF 1
```

下表列出了预处理器命令，给出了一个说明。

命令	说明
define	定义一个预处理器宏或常数
elif	当前面的if，ifdef，ifndef，或elif分支没有执行，发起一个if条件的选择分支。
else	当前面的if，ifdef，或ifndef分支没有执行，发起一个选择分支。
endif	结束一个if，ifdef，ifndef，elif，或else块。
error	输出一个由用户定义的错误信息。本命令指示编译器输出指定错误信息。
ifdef	计算一个条件编译的表达式。计算的参数是一个定义的名称。
ifndef	和ifdef相似，但如果没定义则计算成立。
if	计算一个条件编译的表达式。
include	从一个外部文件中读源文本。符号序列决定了包含文件的查找顺序。Cx51在包含文件目录中查找用大于/小于号（‘<’‘>’）指定的包含文件。在当前目录中查找用双引号（“”）指定的包含文件。
line	指定一个行号和一个可选的文件名。这用在错误信息中来标识错误的位置。
pragma	允许指定可包含在C51命令行的命令。程序可以在命令行中包含相同的命令。
undef	删除一个预处理器宏或常数定义。

字符化操作符

字符化或数字标记操作符（‘#’），当用在一个宏定义中时，把宏参数转化成一个字符常数。这操作符只能用在一个有一个特定参数或参数列表的宏中。

当字符化操作符在一个宏参数前时，传递给宏的参数包含在双引号内，作为一个字符序列。例如：

```
#define stringer(x)    printf(#x " \n" )

 stringer(text)
```

本例子的结果即预处理器的实际输出如下：

```
printf(" text\n" );
```

扩展显示参数转换成一个字符串。当预处理器字符化***x***参数，结果是：

```
printf(" text" " \n" )
```

因为字符被空格分开，在编译时连接，这样两个字符串就组合成了**“text\n”**。

如果作为参数传递的字符串包含需要转义（例如，“和\）的字符，所需的\字符自动增加。

符号连接操作符

在一个宏定义中的符号连接操作符（##）组合两个参数。它允许把两个独立的宏定义的符号连接成一个符号。

如果宏定义中的一个宏参数的名称被立即处理或紧跟着符号连接操作符，宏参数和符号连接操作符被传递的参数值替代。临近符号连接操作符但不是宏参数的名称的文本不受影响。例如：

```
#define paster(n) printf(“ token”  #n “ =%d” ,token##n)

 paster(9);
```

预处理器的实际输出如下：

```
printf(“ token9 = %d” ,token9);
```

本例子显示串联**token##n**成**token9**。字符化和字符连接操作符都在本例子中应用。

预定义宏常数

C*x*51编译器提供预定义常数在模块化的预处理器命令和C代码中使用。下面的表列出和说明了每个常数。

常数	说明
__C51__	**C51**编译器的版本号（例如，610对版本6.10）
__CX51__	**CX51**编译器的版本号（例如，610对版本6.10）
__DATA__	当编译开始时的ANSI格式的日期（月/日/年）。
__DATA2__	编译日期的省略格式（月/日/年）。
__FILE__	被编译文件的名称。
__LINE__	被编译文件的当前行号。
__MODEL__	所选的存储模式（0对**SMALL**，1对**COMPACT**，2对**LARGE**）。
__TIME__	编译开始的时间。
__STDC__	定义为1表示和ANSI C标准完全一致。

第五章. 8051 派生系列

许多8051器件提供更强的性能，同时仍旧和8051内核兼容。这些派生系列提供额外的数据指针，很快的数学运算，扩展或简化的指令集。

C*x*51编译器直接支持下面8051系列的微处理器的增强特性：

- 模拟器件ADuC 微转换器B2系列（2个数据指针和扩展的堆栈空间）。
- ATMEL 89x8252和变种（2个数据指针）。
- DALLAS 80C320，80C420，80C520，80C530，80C550和变种（2个数据指针）。
- DALLAS 80C390，5240和变种（连续地址模式，扩展的堆栈空间，和算术累加器）。
- INFINEON C517，C517A，C509，和变种（高速32位和16位二进制算术运算，8个数据指针）。
- PHILIPS 8xC750，8xC751，和8xC752（最大代码空间2K字节，没有**LCALL**，或**LJMP**指令，64字节内部数据区，没有外部数据区）。
- PHILIPS 80C51MX结构，扩展的指令和存储空间。
- PHILIPS和ATMEL WM支持的几个器件变种，2个数据指针。

通过使用指定的库，库程序，或另外的命令来使能**C*x*51**编译器产生利用上面提到的器件的增强性能，**C*x*51**编译器提供对这些CPU的支持。参考17页的“第二章.用**C*x*51**编译”。

模拟器件微转换器 B2 系列

微转换器的模拟器件B2系列提供2个数据指针，可用来访问存储区。使用多个数据指针可提高如**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等库函数的速度。

MODAB2命令指示**C*x*51**编译器在程序中使用两个数据指针。

C*x*51编译器在一个中断函数中至少使用一个数据指针。如果一个中断函数用**MODAB2**命令编译，两个数据指针都保存在堆栈中，即使中断函数只使用一个数据指针。

为了节省堆栈空间，应该用**NOMODAB2**命令编译中断函数。当这个命令使用时，**C*x*51**编译器不用第二个数据指针。

这些器件同时提供一个扩展的堆栈空间，在起始文件**START_AD.A51**文件中设置。

Atmel 89x8252 和变种

ATMEL 89x8252和变种提供2个数据指针，可以用做存储区访问。用多个数据指针可提高如**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等库函数的速度。

MODA2指令指示**Cx51**编译器在程序中使用两个数据指针。

Cx51编译器在一个中断函数中至少使用一个数据指针。如果一个中断函数用**MODA2**命令编译，两个数据指针都保存在堆栈中，即使中断函数只使用一个数据指针。

为了节省堆栈空间，应该用**NOMODA2**命令编译中断函数。当这个命令使用时，**Cx51**编译器不用第二个数据指针。

Dallas 80C320，420，520，和 530

DALLAS半导体80C320，80C420，80C520和80C530提供2个可用做存储区访问的数据指针。用多个数据指针可提高如**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等库函数的速度。

MODDP2指令指示**Cx51**编译器在程序中使用两个数据指针。

Cx51编译器在一个中断函数中至少使用一个数据指针。如果一个中断函数用**MODDP2**命令编译，两个数据指针都保存在堆栈中，即使中断函数只使用一个数据指针。

为了节省堆栈空间，应该用**NOMODDP2**命令编译中断函数。当这个命令使用时，**Cx51**编译器不用第二个数据指针。

DS80C420对双数据指针提供自动翻转，递减，和自动递增的特性。**\KEIL\C51\LIB\C51DS2A.LIB**库包含**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等使用这些特性的函数的增强版本。当使用这些器件的双DPTR特性时加这个库到PROJECT中。

DS80C550，DS80C390，和DS5240提供对双数据指针的自动翻转和递减特性。**\KEIL\C51\LIB\C51DS2T.LIB**库包含**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等使用这些特性的函数的增强版本。当使用这些器件的双DPTR特性时加这个库到PROJECT中。

Dallas 80C390，80C400，5240，和变种

DALLAS半导体80C390，80C400，5240，和变种提供额外的CPU模式，KEIL编译器完全支持这模式。

连续模式允许创建超过传统8051的64K限制的更大的程序。**ROM（D512K）**和**ROM（D16M）**命令指示**Cx51**编译器采用连续模式。**far**存储类型用来访问使用24位DPTR查寻模式的变量和常数。

注意:
连续模式要求扩展的LX51连接/定位器，和只在PK51专业开发者工具包有的扩展AX51宏汇编器。

除了扩展地址空间，DS80C390，DS80C400，和DS5240提供对双数据指针的自动翻转和递减特性。**\KEIL\C51\LIB\C51DS2T.LIB**库包含**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等使用这些特性的函数的增强版本。对非连续模式（传统8051模式）应用，为了使用这些器件的双DPTR必须加这个库到PROJECT中。连续模式C库已经包含对自动翻转和递减特性的库程序。

DS80C390，DS80C400，和DS5240提供一个扩展的堆栈空间，在起始文件**START390.A51**文件中设置。

算术累加器

C*x*51编译器对DS80C390，DS80C400和DS5240使用32位和16位的算术运算来提高大量的算术运算。当使用这些CPU时，C语言程序执行速度相当快。

用下面的建议来保证只有一个进程使用算术处理器：

- 用**MODDA**命令编译只在主程序或只被一个中断程序而不是两个使用的函数。
- 用**NOMODDA**命令编译余下的函数。

Infineon C517，C509，80C537，和变种

INFINEON C517，C517A，和C509提供高速的32位和16位算术运算，可改进许多**int**，**long**，和**float**运算。

C517，C517A，C509，和C515C提供8个数据指针，可以增加存储区到存储区的运算速度。

MOD517命令指示**Cx51**编译器利用这些增强特性。

数据指针

INFINEON C515C，C517，C517A，和C509提供8个数据指针，可加速存储区访问。使用多个数据指针可提高库函数的运行，如：**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**。C515C，C517，C517和C509的8个数据指针也可减少中断函数的堆栈负载。

Cx51编译器一次只使用8个数据指针中的2个。为了减少中断函数的堆栈负载，当切换寄存器组时**Cx51**切换到2个未使用的数据指针。寄存器**DPSEL**的内容保存在堆栈中，重新选了一对数据指针。不再要求在堆栈中保存数据指针。

如果一个中断程序没有切换到别的寄存器组（例如，函数没有用**using**属性声明），数据指针必须保存在堆栈中（用4字节的堆栈空间）。为了使堆栈空间越小越好，用**MOD517**（**NODP8**）命令编译中断程序和所调用的函数。这使产生的中断代码只使用一个数据指针，2个字节的堆栈空间。

高速运算

C*x*51编译器使用C517，C517A，和C509的32位和16位算术运算来提高大量算术运算的性能。当使用这些CPU时，C语言程序执行速度相当快。

用下面的建议来保证只有一个进程使用算术处理器：

- 用**MOD517**命令编译只在主程序或只被一个中断程序而不是两个使用的函数。
- 用**MOD517**（**NOAU**）命令编译余下的函数。

库函数

C517，C517A，和C509的另外的特性被使用在几个库函数中来提高性能。这些函数如下所列，在209页的“第八章。库参考”中说明。

acos517	**log10517**	**sqrt517**
asin517	**log517**	**sscanf517**
atan517	**printf517**	**strtod517**
atof517	**scanf517**	**tan517**
cos517	**sin517**	
exp517	**sprintf517**	

Philips 8xC750，8xC751，和 8xC752

PHILIPS 8xC750，8xC751，和8xC751派生器件支持最多2K字节的内部程序存储区。CPU不能执行**LCALL**和**LJMP**指令。在使用这些器件时必须考虑下面的因素：

- 一个特定的库，**80C751.LIB**，它不使用这些指令，对这些器件是必需的。
- **C*x*51**编译器不能设成能使用**LJMP**和**LCALL**指令。这用**ROM（SMALL）**命令完成。

当创建对8xC750，8xC751，和8xC752的程序时，注意下面的限制：

- 流函数如**printf**和**putchar**不能使用。这些函数通常对这芯片不是必需的，因为它仅有最多2K字节，没有串口。
- 浮点运算不能使用。只有使用**char**，**unsigned char**，**int**，**unsigned int**，**long**，**unsigned long**，和**bit**数据类型的操作是允许的。
- **C*x*51**编译器必须调用**ROM（SMALL）**命令。这个命令指示C51编译器仅使用**AJMP**和**ACALL**指令。
- 库文件**80C751.LIB**必须包含在连接的输入模块列表中。例如：

```
BL51 myprog.obj,startup751.obj,80c751.LIB
```

- 一个特定的起始模块，**START751.A51**，被要求。这个文件包含如**STARTUP.A51**的起始代码，但不包含**LJMP**或**LCALL**指令。参考150页的“定制文件”。

Philips 80C51MX 结构

PHILIPS 80C51MX结构提供一个扩展的指令集，和支持最多16MB存储空间的扩展寻址模式。通用的指针寄存器和相关的指令给出对一般指针的硬件支持。可以使用**far**存储类型放置变量在扩展存储空间的任何地方。对PHILIPS 80C51MX结构的程序例子在目录**C51\EXAMPLES\PHILIPS 80C51MX**中。

PHILIPS 80C51MX结构由扩展的CX51编译器，LX51连接/定位器，和AX51宏汇编支持。这些另外的程序在PK51专业开发者工具包中提供。

Philips 和 Atmel WM 双 DPTR

PHILIPS半导体和ATMEL无线和微处理器提供几个有双数据指针的兼容8051变种。用多个数据指针可提高如**memcpy**，**memmove**，**memcmp**，**strcpy**，和**strcmp**等库函数的速度。

MODP2命令指示C**x51**编译器在程序中使用双数据指针。

C**x51**编译器在一个中断函数中至少使用一个数据指针。如果一个中断函数用**MODP2**命令编译，两个数据指针都保存在堆栈中，即使中断函数只使用一个数据指针。

为了节省堆栈空间，应该用**NOMODP2**命令编译中断函数。当这个命令使用时，C**x51**编译器不用第二个数据指针。

第六章. 高级编程技术

本章说明高级编程资料，有经验的软件工程师会觉得很有用。大多数这些题目的知识对用**C*x*51**编译器成功的建立一个内嵌的8051目标程序不是必需的。但是，下面的章节提供一个深入非标准程序的建立的过程（例如，和PL/M-51的接口）。

本章叙述下面的标题：

- 可定制的起始程序文件。
- 在运行时可定制的库程序文件。
- **C*x*51**编译器命名代码和数据段的惯例。
- **C*x*51**函数如何和汇编和PL/M-51程序接口。
- 不同的**C*x*51**数据类型的数据存储格式。
- **C*x*51**优化编译器不同的优化特性。

定制文件

C*x*51编译器提供许多可以修改的源文件，使之能适合于特定硬件平台的目标程序。这些文件有：

- 起始执行的代码（**STARTUP.A51**）。
- 初始化静态变量的代码（**INIT.A51**）。
- 执行低级流I/O的代码。
- 存储区分配的代码。

这些文件包含的代码已经过编译或汇编，并包含在C库中。当连接程序时，库中的代码自动包含在内。

可以定制这些文件来配合你的要求。如果在μ Vision2 IDE中，建议复制定制文件到PROJECT目录中来完成修改。文件的修改版本可以和别的源文件一样加到PROJECT中。

当在命令行中时，必须在连接命令行中包含修改后的定制文件的OBJ文件。下面的例子显示如何对**STARTUP.A51**和**PUTCHAR.C**包含定制替代文件：

```
Lx51 MYMODUL1.OBJ,MYMODUL2.OBJ,STARTUP.OBJ,PUTCHAR.OBJ
```

XBANKING.A51文件允许改变扩展的FAR存储区访问程序的配置。

STARTUP.A51

STARTUP.A51文件包含**Cx51**目标程序的起始代码。这源文件在**LIB**目录中。在每个需要定制起始代码的8051PROJECT中包含一个该文件的拷贝。

起始代码在目标系统复位后立即执行，下面的操作可选，为了：

- 清除内部数据区。
- 清除外部数据区。
- 清除外部页数据区。
- 初始化SMALL模式可重入堆栈和指针。
- 初始化LARGE模式可重入堆栈和指针。
- 初始化COMPACT模式可重入堆栈和指针。
- 初始化8051硬件堆栈指针。
- 传递控制权给C函数MAIN。

STARTUP.A51文件提供汇编常数，可以改变汇编常数以控制启动时的动作。这些定义如下表。

常数名	说明
IDATALEN	表示idata的字节数初始化为0。缺省是80h，因为大多数派生的8051包含至少128字节的内部数据区。对8052和别的有256字节的内部数据区的值是100h。
XDATASTART	指定xdata的起始地址初始化为0。
XDATALEN	表示xdata的字节数初始化为0。缺省是0。
PDATASTART	指定pdata的起始地址初始化为0。
PDATALEN	表示xdata的字节数初始化为0。缺省是0。
IBPSTACK	表示SMALL模式的可重入堆栈指针（?C_IBP）是否初始化。值1表示指针需初始化，值0表示不用初始化。缺省是0。
IBPSTARTTOP	指定SMALL模式可重入堆栈区的顶部地址。缺省是idata的0xFF。**Cx51**编译器不检查可用的堆栈区是否满足应用的要求。用户需要自己做一个测试。

常数名	说明
XBPSTACK	指示LARGE模式可重入堆栈指针（?C_XBP）是否应该初始化。值1表示指针需初始化，值0表示不用初始化。缺省是0。
XBPSTACKTOP	指定LARGE模式可重入堆栈区的顶部地址。缺省是xdata的0xFFFF。 **C*x*51**编译器不检查可用的堆栈区是否满足应用的要求。用户需要自己做一个测试。
PBPSTACK	指示COMPACT模式可重入堆栈指针（?C_PBP）是否应该初始化。值1表示指针需初始化，值0表示不用初始化。缺省是0。
PBPSTACKTOP	指定COMPACT模式可重入堆栈区的顶部地址。缺省是pdata的0xFF。 **C*x*51**编译器不检查可用的堆栈区是否满足应用的要求。用户需要自己做一个测试。
PPAGEENABLE	使能（值1）或不使能（值0）8051器件的端口2的初始化。缺省是0。端口2的寻址允许影射任何专用的xdata页为256字节的变量存储区。
PPAGE	指定对pdata存储区访问写到8051端口2的值。值代表用做pdata的xdata存储页。这是pdata的绝对地址的高8位。 例如，如果pdata区从xdata的1000h（页10h）开始，**PPAGEENABLE**应该设为1，**PPAGE**应该设为10h。BL51连接/定位器必须在PDATA命令中包含一个1000h和10FFh间的一个值。例如： **BL51 <input modules> PDATA (1050H)** BL51和Cx51都不检查指定的**PDATA**命令和**PPAGE**汇编常数是否正确。

在8051系列中有许多器件要求指定起始代码。下面的列表提供了各种起始版本的概况：

起始文件	说明
STARTUP.A51	传统8051的标准起始代码。
START_AD.A51	模拟设备微转换器B2系列的起始代码。
STARTLPC.A51	PHILIPS LPC的起始代码。
START390.A51	DALLAS 80C320，80C400，5240连续模式的起始代码。
START_MX.A51	PHILIPS 80C51MX结构的起始代码。
START751.A51	PHILIPS 80C75x的起始代码。

INIT.A51

INIT.A51文件包含对指定需要初始化的变量的初始化程序。如果系统有看门狗计时器，可在初始化代码中用**watchdog**宏集成一个看门狗的刷新。只有初始化过程比看门狗的周期长时需要这个。例如，如果用一个INFINEON C515，宏可如下定义：

```
WATCHDOG    MACRO
            SETB    WDT
            SETB    SWDT
            ENDM
```

INIT_TNY.A51文件是**INIT.A51**的简减版本，对不包含XDATA存储区的PROJECT使用。当对单片器件时用本文件，如PHILIPS LPC系列，包含数据空间的变量的初始化。

XBANKING.A51

本文件提供程序支持**far**（HDATA）和**const far**（HCONST）存储类型。扩展的LX51连接/定位器用**far**和**const far**来寻址扩展的地址空间HDATA和HCONST。**Cx51**编译器用一个3字节通用指针访问这些存储区。用**far**存储类型定义的变量放在存储类**HDATA**中。用**const far**定义的变量放在存储类**HCONST**中。LX51连接/定位器允许定位存储类在物理的16MB代码或16MB xdata空间。对传统的8051器件和C51编译器，使用**far**存储区，必须如84页所说的使用“VARBANKING”命令。

存储类型**far**和**const far**提供对新的8051器件的大代码/xdata空间的支持。如果所用的CPU提供一个扩展的24位的DPTR寄存器，可以使用缺省的**XBANKING.A51**文件版本，定义如下表的符号列表。

常数名	说明
?C?XPAGE1SFR	包含DPTR位16-23的DPTR页寄存器的SFR地址。
?C?XPAGE1RST	?C?XPAGE1SFR地址X:0区的复位值。当用VARBANKING（1）命令时，这个设置被C51编译器使用。C51编译器用VARBANKING（1）时，在中断函数的开头保存?C?XPAGE1SFR，设置这个寄存器为?C?XPAGE1SFR的值。

far存储类型允许寻址到如EEPROM空间或代码BANKING ROM中的字符串等特定存储区。应用访问这些存储区就象是标准8051存储空间的一部分。目录**C51\EXAMPLES\FARMEMORY**中的例子程序显示如何在传统8051器件中使用C51far存储类型。如果没有如果满足要求的例子，可以调整如下表所列的访问程序。

访问程序	说明
?C?CLDXPTR，?C?CSTXPTR	在扩展存储区加载/保存一个字节（char）。
?C?ILDXPTR，?C?ISTXPTR	在扩展存储区加载/保存一个字（int）。
?C?PLDXPTR，?C?PSTXPTR	在扩展存储区加载/保存一个3字节指针。
?C?LLDXPTR，?C?LSTXPTR	在扩展存储区加载/保存一个双字（long）。

每个访问程序从CPU寄存器R1/R2/R3中得到一个3字节指针的存储区地址作为参数。寄存器R3保存存储类型值。对传统的8051器件，**C*x*51**编译器用下面的存储类型值：

R3值	存储类型	存储类	地址范围
0x00	data/idata	DATA/IDATA	I:0x00-I:0xFF
0x01	xdata	XDATA	X:0x0000-X:0xFFFF
0x02-0x7F	far	HDATA	X:0x010000-X:0x7E0000
0x80-0xFD	far const	HCONST	C:0x800000-C:0xFD0000(**far const**影射到BANK存储区)
0xFE	pdata	XDATA	XDATA存储区的一个256字节页
0xFF	code	CODE/CONST	C:0x0000-C:0xFFFF

R3的值0x00，0x01，0xFE和0xFF在运行库中早已处理。只有0x02-0xFE的值传递给上面所说的访问程序的XPTR。AX51宏汇编器提供MBYTE操作符计算需要传递给XPTR访问函数的R3的值。下面是一个AX51汇编器用XPTR访问函数的例子：

```
    MOV     R1,#LOW     (variable)      ;给出变量的LSB地址字节
    MOV     R1,#HIGH    (variable)      ;给出变量的MSB地址字节
    MOV     R1,#MBYTE   (variable)      ;给出变量的存储类型
    CALL    ?C?CLDXPTR                  ;加载BYTE变量到A
```

基本的 I/O 函数

下面的文件包含了低级流I/O程序的源代码。当使用μ Vision2 IDE时，只要在PROJECT中加如修改的版本。

C源文件	说明
PUTCHAR.C	被所有的流程序用来输出字符。可以把这个程序用在自己的硬件上（例如，LCD或LED显示）。 缺省版本通过串口输出字符。流控制用**XON/XOFF**协议。换行符（‘**\n**’）转换成回车/换行符序列（‘**\r\n**’）。
GETKEY.C	被所有的流程序用来输入字符。可以把这个程序用在自己的硬件上（例如，矩阵键盘）。 缺省版本通过串口读入字符。没有数据转换。

存储区分配函数

下面的文件包含存储区分配程序的源代码。

C源文件	说明
CALLOC.C	从存储区池为一个数组分配存储区。
FREE.C	释放一个前面分配的存储区块。
INIT_MEM.C	指定可以用**malloc**，**calloc**，和**realloc**函数分配的存储区池的位置和大小。
MALLOC.C	从存储区池分配存储区。
REALLOC.C	调整一个前面分配的存储块的大小。

优化器

C*x*51编译器是一个优化编译器。这意味着编译器采取特定的步骤来确保生成的代码和输出的OBJ文件是可能的最有效的代码（最小和/或最快）。编译器分析已生成的代码，产生更有效的指令序列。这确保**C*x*51**编译器程序运行的尽可能的快。

C*x*51编译器提供几种不同的优化级别。参考63页的“优化”。

优化	说明
常数压缩	在一个表达式有几个常数值或地址计算的组合成一个常数。
跳转优化	当可提高程序效率时跳转或扩展为最后的目标地址。
清除死代码	不可及代码（死代码）从程序中清除。
寄存器变量	自动变量和函数参数尽可能的置于寄存器中。为这些变量保留的数据区省略。
通过寄存器传递参数	可通过寄存器传递最多三个函数参数。
清除全局公用子表达式	在一个函数中出现多次的同样的子表达式或地址计算尽可能的只验证和计算一次。
复用公用入口代码	当对一个函数有多次调用时，一些设置代码可以复用，因此可以减少程序大小。
公用块子程序	检测重复出现的指令序列并转换成子程序。编译器甚至重新安排代码以得到更大的重复出现序列。

8051 特定优化

优化	说明
窥视孔优化	当可以节省存储空间或运行时间时，用简单的操作代替复杂的操作。
扩展访问优化	常数和变量直接变化在操作中。
数据覆盖	BL51连接/定位器把函数的数据和位段确定为OVERLAYABLE的，并可被别的数据和位段覆盖。
CASE/SWITCH优化	用一个跳转表或行跳转来优化SWITCH CASE声明。

生成代码选项

优化	说明
OPTIMIZE(SIZE）	公用C运算用子程序替代。因此减少程序代码。
NOAREGS	**Cx51**编译器不再使用绝对寄存器访问。程序代码独立于寄存器组。
NOREGPARMS	传递的参数在局部数据段运行。程序代码和早期的**Cx51**兼容。

段名转换

C*x*51编译器生成的目标代码（程序代码，程序数据，和常数数据）保存在代码段或数据段中。一个段可以是可重定位的或绝对的。每个可重定位段有一个类型和一个名称。本节说明**C*x*51**编译器命名这些段的惯例。

段名包括一个*module_name*，它是声明目标的源文件名。为了适应大量的现有的软件和硬件工具，所有的段名都转换和保存为大写。

每个段名有一个前缀，它对应于段所用的存储类型。前缀用问号（？）为界。下面是一个标准段名前缀的列表：

段前缀	存储类型	说明
?PR?	**program**	可执行的程序代码
?CO?	**code**	程序存储区的常数数据
?BI?	**bit**	内部数据区的位数据
?BA?	**bdata**	内部数据区的可位寻址数据
?DT?	**data**	内部数据区
?FD?	**far**	FAR存储区（RAM空间）
?FC?	**const far**	FAR存储区（常数ROM空间）
?ID?	**idata**	间接寻址内部数据区
?PD?	**pdata**	外部数据区的分页数据
?XD?	**xdata**	XDATA存储区（RAM空间）
?XC?	**const xdata**	XDATA存储区（常数ROM空间）

数据目标

数据目标是在C程序中声明的变量和常数。**Cx51**编译器对每个声明的变量的存储类型产生一个独立的段。下表列出了对不同的变量数据目标产生的段名。

段前缀	说明
?BA?*module_name*	可位寻址数据目标
?BI?*module_name*	位目标
?CO?*module_name*	常数（字符串和已初始化变量）
?DT?*module_name*	在data中声明的目标
?FC?*module_name*	在const far（要求OMF2命令）声明的目标
?FD?*module_name*	在far（要求OMF2命令）声明的目标
?ID?*module_name*	在idata声明的目标
?PD?*module_name*	在pdata声明的目标
?XC?*module_name*	在const xdata（要求OMF2命令）声明的目标
?XD?*module_name*	在xdata声明的目标

程序目标

程序目标包括由**C*x*51**编译器产生的C程序函数代码。在一个源模块中的每个函数和一个单独的代码段关联，用?PR?*function_name?module_name*命名。例如，在文件**SAMPLE.C**中的函数**error_check**的段名的结果是?PR?ERROR_CHECK?SAMPLE。

在一个函数体内声明的局部变量也建立段。这些段名遵循上面的惯例，但根据局部变量所保存的存储区有一个不同的前缀。

过去函数参数用固定的存储区传递。这对用PL/M-51编写的程序仍适用。但是，**C*x*51**可以在寄存器中传递3个函数参数。别的参数用传统的固定存储区传递。对所有的函数参数，无论这些参数是否通过寄存器传递，存储空间都保留。参数区对如何调用模块都必须是公共的。因此，他们用下面的段名公开定义：

```
?function_name?BYTE
 ?function_name?BIT
```

例如，如果**func1**是一个接受**bit**段和别的数据类型的函数，**bit**段从?FUNC1?BIT开始传递，所有的别的参数从?FUNC1?BYTE传递。参考163页的“C程序和汇编的接口”的函数参数段的例子。

有参数，局部变量，或**bit**变量的函数，包含所有这些变量的附加段。这些段可以被BL51连接/定位器覆盖。

他们根据所用的存储模式建立如下。

SMALL模式段命名规则		
信息	**段类型**	**段名**
程序代码	**code**	?PR?*function_name?module_name*
局部变量	**data**	?DT?*function_name?module_name*
局部位变量	**bit**	?BI?*function_name?module_name*

COMPACT模式段命名规则		
信息	**段类型**	**段名**
程序代码	**code**	?PR?*function_name?module_name*
局部变量	**pdata**	?PD?*function_name?module_name*
局部位变量	**bit**	?BI?*function_name?module_name*

LARGE模式段命名规则		
信息	**段类型**	**段名**
程序代码	**code**	?PR?*function_name?module_name*
局部变量	**xdata**	?XD?*function_name?module_name*
局部位变量	**bit**	?BI?*function_name?module_name*

对有寄存器参数和可重入属性的函数名有稍许修改以避免运行错误。下表列出了和标准段命名不同之处。

声明	符号	说明
void func(void)…	FUNC	没有参数或参数不通过寄存器传递的函数名没有改变。函数名改为大写。
void func1(char)…	_FUNC1	参数通过寄存器传递的函数，函数名前有一个下划线（‘_’）。这确定这些函数通过CPU寄存器传递参数。
void func2(void) reentrant…	_?FUNC2	可重入的函数，函数名前有一个字符串“_?”。这用来确定可重入函数。

C 程序和汇编的接口

程序可以很容易的和8051汇编接口。A51汇编器是一个8051宏汇编器，生成OMF-51格式的目标模块。遵循一些编程规则，可以从C调用汇编程序，反之亦然。在汇编模块中声明的公共变量在C程序中也可用。

有一个原因需要从C程序中调用汇编程序。

- 有一个早已写好的汇编代码。
- 需要提高一个特定函数的速度。
- 需要直接从汇编操纵SFR或存储影射I/O设备。

这节说明如何编写可以直接被C程序调用的汇编程序。

对一个被C调用的汇编程序，必须明确C函数所用的参数和返回值。对所有的实际目的，它必须看起来象一个C函数。

函数参数

缺省的，C函数在寄存器中最多传递三个参数。余下的参数通过固定存储区传递。可以用**NOREGPARMS**命令取消用寄存器传递参数。如果用寄存器传递参数取消，或参数太多，参数通过固定存储区传递。用寄存器传递参数的函数在生成代码时被**C*x*51**编译器在函数名前加了一个下划线（‘_’）的前缀。只在固定存储区传递参数的函数没有下划线。参考166页的“用SRC命令”。

参数通过寄存器传递

C函数可以通过寄存器和固定存储区传递参数。寄存器可传递最多3个参数。所有别的参数通过固定存储区传递。下表定义用来传递参数的寄存器。

参数数目	char，1字节指针	int，2字节指针	long，float	通用指针
1	R7	R6&R7(MSB在R6，LSB在R7)	R4—R7	R1—R3 （存储类型在R3，MSB在R2，LSB在R1）
2	R5	R4&R5(MSB在R4，LSB在R5)	R4—R7	R1—R3 （存储类型在R3，MSB在R2，LSB在R1）
3	R3	R2&R3(MSB在R2，LSB在R3)		R1—R3 （存储类型在R3，MSB在R2，LSB在R1）

下表例子说明如何选择传递参数的寄存器。

声明	说明
func1(int a)	唯一一个参数***a***在寄存器R6和R7传递。
func2(int b,int c,int *d)	第一个参数***b***在寄存器R6和R7传递。第二个参数***c***在寄存器R4和R5传递。第三个参数***d***在寄存器R1，R2和R3传递。
func3(long e,long f)	第一个参数***e***在寄存器R4，R5，R6和R7传递。第二个参数***f***，不能用寄存器，因为long类型可用的寄存器已被第一个参数所用。这个参数用固定存储区传递。
func4(float g,char h)	第一个参数***g***在寄存器R4，R5，R6，和R7中传递。第二个参数***h***，不能用寄存器传递，用固定存储区传递。

用固定存储区传递参数

用固定存储区传递参数给汇编程序，参数用段名?*function_name*?BYTE和?*function_name*?BIT保存传递给函数*function_name*的参数。位参数在调用函数前复制到?*function_name*?BIT段。别的参数复制到?*function_name*?BYTE段。即使通过寄存器传递参数，在这些段中也给所有的参数分配空间。参数按每个段中的声明的顺序保存。

用做参数传递的固定存储区可能在内部数据区或外部数据区，有存储模式决定。**SMALL**模式是最有效的，参数段用内部数据区。**COMPACT**和**LARGE**模式用外部数据区。

函数返回值

函数返回值通常用CPU寄存器传递。下表列出了可能的返回值和所用的寄存器。

返回类型	寄存器	说明
bit	CF	在CF中返回一个位
char/unsigned char,1字节指针	R7	在R7返回单个字节类型
ing/unsigned int,2字节指针	R6&R7	MSB在R6，LSB在R7
long/unsigned long	R4-R7	MSB在R4，LSB在R7
float	R4-R7	32位IEEE格式
通用指针	R1-R3	存储类型在R3，MSB在R2，LSB在R1

用 SRC 命令

C*x*51编译器用A51汇编器可以建立汇编源文件。想要确定C和汇编间的参数传递转换，这些文件是很有用的。

建立一个汇编源文件，必须在**C*x*51**编译器用**SRC**命令。例如：

```
#pragma SRC
#pragma SMALL

unsigned int asmfunc1(unsigned int arg)
{
    return(1+arg);
}
```

当用SRC命令编译时产生如下的汇编输出文件。

```
; ASM1.SRC generated from: ASM1.C

NAME    ASM1

?PR?_asmfunc1?ASM1   SEGMENT CODE
PUBLIC  _asmfunc1
; #pragma SRC
; #pragma SMALL
;
; unsigned int asmfunc1 (

                RSEG   ?PR?_asmfunc1?ASM1
                USING  0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
                       ; SOURCE LINE # 4
                       ; SOURCE LINE # 6
; return (1 + arg);
                       ; SOURCE LINE # 7
                MOV    A,R7
                ADD    A,#01H
                MOV    R7,A
                CLR    A
                ADDC   A,R6
                MOV    R6,A
; }
                       ; SOURCE LINE # 8
?C0001:
                RET
; END OF _asmfunc1

                END
```

注意，在这个例子中，函数名**asmfunc1**有一个下划线前缀，表示参数通过寄存器传递。**arg**参数用R6和R7传递。

下面的例子显示相同的函数产生的汇编源文件，用**NOREGPARMS**命令禁止寄存器传递参数。

```
; ASM2.SRC generated from: ASM2.C

NAME    ASM2

?PR?asmfunc1?ASM2    SEGMENT CODE
?DT?asmfunc1?ASM2    SEGMENT DATA
PUBLIC  ?asmfunc1?BYTE
PUBLIC  asmfunc1

                RSEG  ?DT?asmfunc1?ASM2
?asmfunc1?BYTE:
arg?00:          DS  2
; #pragma SRC
; #pragma SMALL
; #pragma NOREGPARMS
;
; unsigned int asmfunc1 (

                RSEG   ?PR?asmfunc1?ASM2
                USING  0
asmfunc1:
                      ; SOURCE LINE # 5
                      ; SOURCE LINE # 7
; return (1 + arg);
                      ; SOURCE LINE # 8
                MOV   A,arg?00+01H
                ADD   A,#01H
                MOV   R7,A
                CLR   A
                ADDC  A,arg?00
                MOV   R6,A
; }
                      ; SOURCE LINE # 9
?C0001:
                RET
; END OF asmfunc1

                END
```

注意本例子的函数名**asmfunc1**没有下划线前缀，**arg**参数通过**?asmfunc1?BYTE**段传递。

寄存器使用

汇编函数可以修改当前所选的寄存器组和寄存器**ACC**，**B**，**DPTR**和**PSW**的内容。当从汇编调用一个C函数，假设这些寄存器要被所调用的C函数修改。

可覆盖段

如果在程序连接和定位过程中运行可覆盖进程，则每个汇编程序需要有一个独立的程序段。这是必须的，只有这样，在可覆盖进程中，函数间的参考用单独的段参考计算。当有下面各点时，汇编子程序的数据区可能包含在覆盖分析中：

- 所有的段名必须用**C*x*51**编译器段命名规则建立。
- 每个有局部变量的汇编函数必须分配自己的数据段。别的函数只能通过传递参数访问这数据段。参数必须按顺序传递。

例子程序

下面的程序例子显示如何传递参数给汇编程序，和从汇编程序传递参数。下面的C函数用在所有的这些例子中：

```
int function(
    int v_a,            /*  在R6 & R7传递  */
    char v_b,           /*  在R5传递  */
    bit v_c,            /*  在固定存储区传递  */
    long v_d,           /*  在固定存储区传递  */
    bit v_e);           /*  在固定存储区传递  */
```

SMALL 模式例子

在SMALL模式，参数通过固定存储区传递，保存在内部数据区。变量的参数传递段在**data**区。

下面有两各汇编代码例子。第一个显示例子函数如何被汇编调用。第二个例子显示例子函数的汇编代码。

从汇编调用一个C函数。

```
.
.
.
EXTRN CODE    (_function)         ; Ext declarations for function names
EXTRN DATA    (?_function?BYTE)   ; Seg for local variables
EXTRN BIT     (?_function?BIT)    ; Seg for local bit variables
.
.
.
        MOV   R6,#HIGH intval               ; int  a
        MOV   R7,#LOW  intval               ; int  a
        MOV   R7,#charconst                 ; char b
        SETB  ?_function?BIT+0              ; bit  c
        MOV   ?_function?BYTE+3,longval+0   ; long d
        MOV   ?_function?BYTE+4,longval+1   ; long d
        MOV   ?_function?BYTE+5,longval+2   ; long d
        MOV   ?_function?BYTE+6,longval+3   ; long d
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C            ; bit  e
        LCALL _function
        MOV   intresult+0,R6                ; store int
        MOV   intresult+1,R7                ; retval
.
.
.
```

例子函数的汇编代码：

```
NAME          MODULE                  ; Names of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE    ; Seg for prg code in 'function'
?DT?FUNCTION?MODULE  SEGMENT  DATA OVERLAYABLE
                                      ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT  OVERLAYABLE
                                      ; Seg for local bit vars in 'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                      ; Public symbols for 'C' function call

RSEG          ?PD?FUNCTION?MODULE     ; Segment for local variables
?_function?BYTE:                      ; Start of parameter passing segment
v_a:    DS   2                        ;   int variable: v_a
v_b:    DS   1                        ;   char variable: v_b
v_d:    DS   4                        ;   long variable: v_d
.
.                                     ; Additional local variables
.

RSEG          ?BI?FUNCTION?MODULE     ; Segment for local bit variables
?_function?BIT:                       ; Start of parameter passing segment
v_c:    DBIT  1                       ;   bit variable: v_c
v_e:    DBIT  1                       ;   bit variable: v_e
.
.                                     ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE     ; Program segment
_function:    MOV  v_a,R6             ; A function prolog and epilog is
              MOV  v_a+1,R7           ; not necessary.  All variables can
              MOV  v_b,R5             ; immediately be accessed.
.
.
.
              MOV  R6,#HIGH retval    ; Return value
              MOV  R7,#LOW  retval    ; int constant
              RET                     ; Return
```

COMPACT 模式例子

在COMPACT模式，参数通过固定存储区传递，保存在内部数据区。变量的参数传递段在**pdata**区。

下面有两各汇编代码例子。第一个显示例子函数如何被汇编调用。第二个例子显示例子函数的汇编代码。

从汇编调用一个C函数。

```
EXTRN CODE    (_function)          ; Ext declarations for function names
EXTRN XDATA   (?_function?BYTE)    ; Seg for local variables
EXTRN BIT     (?_function?BIT)     ; Seg for local bit variables
.
.
.
        MOV   R6,#HIGH intval           ; int  a
        MOV   R7,#LOW  intval           ; int  a
        MOV   R5,#charconst             ; char b
        SETB  ?_function?BIT+0          ; bit  c
        MOV   R0,#?_function?BYTE+3     ; Addr of 'v_d' in the passing area
        MOV   A,longval+0               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+1               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+2               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        INC   R0                        ; Inc parameter passing address
        MOV   A,longval+3               ; long d
        MOVX  @R0,A                     ; Store parameter byte
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C        ; bit  e
        LCALL _function
        MOV   intresult+0,R6            ; Store int
        MOV   intresult+1,R7            ; Retval
.
.
.
```

例子函数的汇编代码：

```
NAME          MODULE                    ; Name of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE     ; Seg for program code in 'function';
?PD?FUNCTION?MODULE  SEGMENT  XDATA OVERLAYABLE IPAGE
                                        ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT   OVERLAYABLE
                                        ; Seg for local bit vars in
'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                        ; Public symbols for C function call

RSEG          ?PD?FUNCTION?MODULE       ; Segment for local variables
?_function?BYTE:                        ; Start of the parameter passing seg
v_a:    DS   2                          ;   int variable: v_a
v_b:    DS   1                          ;   char variable: v_b
v_d:    DS   4                          ;   long variable: v_d
.
.                                       ; Additional local variables
.

RSEG          ?BI?FUNCTION?MODULE       ; Segment for local bit variables
?_function?BIT:                         ; Start of the parameter passing seg
v_c:    DBIT  1                         ;   bit variable: v_c
v_e:    DBIT  1                         ;   bit variable: v_e
.
.                                       ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE           ; Program segment
_function:    MOV   R0,#?_function?BYTE+0   ; Special function prolog
              MOV   A,R6                    ; and epilog is not
              MOVX  @R0,A                   ; necessary.  All
              INC   R0                      ; vars can immediately
              MOV   A,R7                    ; be accessed
              MOVX  @R0,A
              INC   R0
              MOV   A,R5
              MOVX  @R0,A
.
.
.
              MOV   R6,#HIGH retval         ; Return value
              MOV   R7,#LOW  retval         ; int constant
              RET                           ; Return
```

LARGE 模式例子

在LARGE模式，参数通过固定存储区传递，保存在内部数据区。变量的参数传递段在**xdata**区。

下面有两各汇编代码例子。第一个显示例子函数如何被汇编调用。第二个例子显示例子函数的汇编代码。

从汇编调用一个C函数。

```
EXTRN CODE    (_function)           ; Ext declarations for function names
EXTRN XDATA   (?_function?BYTE)     ; Start of transfer for local vars
EXTRN BIT     (?_function?BIT)      ; Start of transfer for local bit vars
.
.
.
        MOV   R6,#HIGH intval       ; int  a
        MOV   R7,#LOW  intval       ; int  a
        MOV   R5,#charconst         ; char b
        SETB  ?_function?BIT+0      ; bit  c
        MOV   R0,#?_function?BYTE+3 ; Address of 'v_d' in the passing area
        MOV   A,longval+0           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+1           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+2           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        INC   DPTR                  ; Increment parameter passing address
        MOV   A,longval+3           ; long d
        MOVX  @DPTR,A               ; Store parameter byte
        MOV   C,bitvalue
        MOV   ?_function?BIT+1,C    ; bit  e
        LCALL _function
        MOV   intresult+0,R6        ; Store int
        MOV   intresult+1,R7        ; Retval
.
.
.
```

例子函数的汇编代码：

```
NAME          MODULE                  ; Name of the program module
?PR?FUNCTION?MODULE  SEGMENT  CODE    ; Seg for program code in 'functions'
?XD?FUNCTION?MODULE  SEGMENT  XDATA   OVERLAYABLE
                                      ; Seg for local vars in 'function'
?BI?FUNCTION?MODULE  SEGMENT  BIT     OVERLAYABLE
                                      ; Seg for local bit vars in 'function'

PUBLIC        _function, ?_function?BYTE, ?_function?BIT
                                      ; Public symbols for C function call

RSEG          ?XD?FUNCTION?MODULE     ; Segment for local variables
?_function?BYTE:                      ; Start of the parameter passing seg
v_a:    DS    2                       ;   int variable: v_a
v_b:    DS    1                       ;   char variable: v_b
v_d:    DS    4                       ;   long variable: v_l
.
.; Additional local variables from 'function'
.

RSEG          ?BI?FUNCTION?MODULE     ; Segment for local bit variables
?_function?BIT:                       ; Start of the parameter passing seg
v_c:    DBIT   1                      ;   bit variable: v_c
v_e:    DBIT   1                      ;   bit variable: v_e
.
.                                     ; Additional local bit variables
.

RSEG          ?PR?FUNCTION?MODULE             ; Program segment
_function:    MOV   DPTR,#?_function?BYTE+0   ; Special function prolog
              MOV   A,R6                      ; and epilog is not
              MOVX  @DPTR,A                   ; necessary.  All vars
              INC   R0                        ; can immediately be
              MOV   A,R7                      ; accessed.
              MOVX  @DPTR,A
              INC   R0
              MOV   A,R5
              MOVX  @DPTR,A
.
.
.
              MOV   R6,#HIGH retval           ; Return value
              MOV   R7,#LOW  retval           ; int constant
              RET                             ; Return
```

C 程序和 PL/M-51 的接口

INTEL的PL/M-51是一个通用编程语言，在许多方面和C类似。PL/M-51程序可以很容易和**C*x*51**编译器接口。

- 声明为**alien**函数类型，可以从C访问PL/M-51函数。
- 在PL/M-51中声明的公用变量可被C程序所用。
- PL/M-51编译器生成OMF-51格式的目标文件。

C*x*51编译器可以用PL/M-51参数传递规则生成代码。**alien**函数类型标识符用来声明和PL/M-51兼容的任何存储模式的公共或外部函数。例如：

```
extern alien char plm_func(int,char);

alien unsigned int c_func(unsigned char x,unsigned char y)  {
      return (x*y);
}
```

PL/M-51函数的参数和返回值可以是下面的任何类型：**bit**，**char**，**unsigned char**，**int**，和**unsigned int**。别的类型，包括**long**，**float**，和所有的指针类型，可以在**alien**声明的C函数中声明。但是，用这些类型需要小心，因为PL/M-51不直接支持32位二进制整数或浮点数。

注意：

*PL/M-51不支持可变长度参数列表。因此，用**alien**声明的函数必须有一个固定的参数数目。**alien**函数不允许用做可变长度参数列表的省略号，可能引起**Cx51**编译器产生一个错误信息。例如：*

```
extern alien unsigned int plm_i(char ,int,…);
*** ERROR IN LINE 1 OF A.C:’ plm_i’ :Var_parms on alien function
```

数据存储格式

本节说明在**Cx51**编译器中可用的数据类型的存储格式。**Cx51**编译器为C程序提供许多基本的数据类型。下表列出了这些数据类型和大小和值的范围。

Data Type	Bits	Bytes	Value Range
Bit	1	—	0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
Enum	8 / 16	1 or 2	-128 to +127 or -32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
Float	32	4	±1.175494E-38 to ±3.402823E+38
data *, idata *, pdata *	8	1	0x00 to 0xFF
code*, xdata *	16	2	0x0000 to 0xFFFF
generic pointer	24	3	Memory type (1 byte); Offset (2 bytes) 0 to 0xFFFF

别的数据类型，象结构和联合，可能包含本表中的标量。所有这些数据类型的元素顺序分配，根据8051的8位结构按字节排列。

位变量

bit类型的标量用单个位存储。位指针和数组是不允许的。位目标通常在8051CPU的内部位可寻址区。BL51连接/定位器可能覆盖位目标。

signed 和 unsigned 字符，data，idata，和 pdata 指针

char类型的标量保存在一个字节中（8位）。指定存储区指针的**data**，**idata**，和**pdata**参考也用一个字节保存。如果一个**enum**可以用一个8位值表示，enum也用一个字节保存。

signed 和 unsigned 整数，列举，xdata 和 code 指针

int，**short**，和**enum**的标量，和指定存储区指针的参考**xdata**或**code**都用两个字节保存。高字节先保存，低字节后保存。例如，一个0x1234的整数值在存储区如下保存：

地址	+0	+1
内容	0x12	0x34

signed 和 unsigned long 整数

long类型的标量用四个字节保存。字节从高到低保存。例如，long值0x12345678在存储区如下保存：

地址	+0	+1	+2	+3
内容	0x12	0x34	0x56	0x78

通用和 FAR 指针

通用指针没有声明明确的存储类型。他们可能指向8051的任何存储区。这些指针用三个字节保存。第一个字节包含的值表明存储区或存储类型。另外两个字节包含地址偏移，高字节在先。所用的格式如下：

地址	+0	+1	+2
内容	存储类型	偏移；高字节	偏移；低字节

根据所用的编译器版本，存储类型的值如下：

存储类型	idata/data/bdata	xdata	pdata	code
C51编译器（8051器件）	0x00	0x01	0xFE	0xFF
CX51编译器（PHILIPS 80C51MX）	0x7F	0x00	0x00	0x80

PHILIPS 80C51MX结构支持新CPU指令用一个通用指针操作。通用指针用C*x*51通用指针确定。

通用指针的格式也用在**far**存储类型指针。因此，任何别的存储类型值可用做**far**存储空间的地址。

下面的例子显示一个通用指针（在C51编译器）的存储区保存，参考地址为**xdata**存储区的0x1234。

地址	+0	+1	+2
内容	0x01	0x12	0x34

浮点数

float类型标量用四个字节保存。格式用下面的IEEE-754标准。

一个浮点数用两个部分表示：尾数和2的幂。例如：

$$\pm mantissa \times 2^{exponent}$$

尾数代表浮点上的数据二进制数。

二的幂代表指数。指数的保存形式是一个0到255的8位值。指数的实际值是保存值（0到255）减去127，一个范围在127到-128之间的值。

尾数是一个24位值（代表大约7个十进制数），最高位（MSB）通常是1，因此，不保存。一个符号位表示浮点数是正或负。

浮点数保存的字节格式如下：

地址	+0	+1	+2	+3
内容	SEEE EEEE	EMMM MMMM	MMMM MMMM	MMMM MMMM

这里：

S 代表符号位，1是负，0是正。

E 幂，偏移127。

M 24位的尾数（保存在23位中）。

零是一个特定值，表示幂是0，尾数是0。

浮点数-12.5作为一个十六进制数0xC1480000保存。在存储区中，这个值如下：

地址	+0	+1	+2	+3
内容	0xC1	0x48	0x00	0x00

浮点数和十六进制等效保存值之间的转换相当简单。下面的例子说明上面的值-12.5如何转换。

浮点保存值不是一个直接的格式。要转换为一个浮点数，位必须按上面的浮点数保存格式表所列的那样分开，例如：

地址	+0	+1	+2	+3
格式	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM
二进制	11000001	01001000	00000000	00000000
十六进制	00	00	48	C1

从这个例子，可以得到下面的信息：

- 符号位是1，表示一个负数。
- 幂是二进制10000010，或十进制130。130减去127是3，就是实际的幂。
- 尾数是后面的二进制数：10010000000000000000000

在尾数的左边有一个省略的二进制点和1。这个数在浮点数的保存中经常省略。加上一个1和点到尾数的开头，尾数值如下：

1.10010000000000000000000

接着，根据指数调整尾数。一个负的指数向左移动小数点。一个正的指数向右移动小数点。因为指数是三，尾数调整如下：

1100.10000000000000000000

结果是一个二进制浮点数。小数点左边的二进制数代表所处位置的二的幂。例如，1100代表 $(1\times 2^3)+(1\times 2^2)+(0\times 2^1)+(0\times 2^0)$，这里是12。

小数点的右边也代表所处位置的二的幂。但是，幂是负的。例如，.100…代表 $(1\times 2^{-1})+(0\times 2^{-2})+(0\times 2^{-3})+...$，结果等于.5。

这些值的和是12.5。因为有设置符号位，这数是负的。因此，十六进制值**0xC1480000**是**-12.5**。

浮点错误

8051没有包含一个中断矢量来捕捉浮点错误；因此，你的软件必须对这些错误条件指出作出适当的响应。

除了正常的浮点值，一个浮点数可能包含一个二进制错误值。这些值被定义为IEEE标准的一部分，并在任何正常的浮点运算的过程中出现错误时被使用。目标代码需要在每个浮点运算结束后检查可能的算术错误。

名称	值	意义
NaN	0xFFFFFFF	不是一个数
+INF	0x7F80000	正无穷大（正溢出）
-INF	0Xff80000	负无穷大（负溢出）

注意：

*Cx51库函数**_chkfloat_**可以快速的检查浮点状态。*

可以使用下面的**union**来保存浮点值。

```
union f {
    float          f;          /* Floating-point value */
    unsigned long  u1;         /* Unsigned long value */
};
```

这个**union**包含一个**float**和一个**unsigned long**，是为了执行浮点算术运算和响应IEEE错误状态。

例如：

```
#define NaN        0xFFFFFFFF    /* Not a number (error) */
#define plusINF    0x7F800000    /* Positive overflow    */
#define minusINF   0xFF800000    /* Negative overflow    */

union f {
  float           f;            /* Floating-point value */
  unsigned long   ul;           /* Unsigned long value */
};

void main (void)  {
  float a, b;
  union f x;

  x.f = a * b;
  if (x.ul == NaN  || x.ul == plusINF  || x.ul == minusINF)  {
    /* handle the error  */
  }
  else  {
    /* result is correct */
  }
}
```

访问绝对存储地址

C编程语言不支持直接指定一个静态或全局变量的存储地址。有三种方法涉及直接存储地址。可用：

- 绝对存储区访问宏
- 连接器定位控制
- **_at_**关键词

三种方法的每个说明如下。

绝对存储访问宏

首先，可以使用**Cx51**库提供的部分绝对存储访问宏。用下面的宏直接访问8051存储区。

CBYTE	**FCVAR**	**CWORD**
DBYTE	**FVAR**	**DWORD**
FARRAY	**PBYTE**	**PWORD**
FCARRAY	**XBYTE**	**XWORD**

参考212页的“绝对存储区访问宏”。

连接器定位控制

第二种方法是在一个独立的C模块中声明变量，用BL51连接/定位器的定位命令指定一个绝对存储区地址。

在下面的例子中，假设有一个叫**alarm_control**的结构，想要置与**xdata**的地址2000h。输入一个源文件命名为**ALMCTRL.C**，仅包含这个结构的声明。

```
.
.
.
struct alarm_st        {
      unsigned int alarm_number;
      unsigned char enable_flag;
      unsigned int time_delay;
      unsigned char status;
};

xdata struct alarm_st alarm_control;
.
.
.
```

Cx51编译器从**ALMCTRL.C**生成一个目标文件，包含一个**xdata**存储区的变量的段。因为是本模块中唯一声明的变量，**alarm_control**是段中唯一的变量。段名是?XD?ALMCTRL。**Lx51**连接定位器允许用定位命令指定任何段的基地址。

对BL51，必须用XDATA命令，因为**alarm_control**变量声明在**xdata**中：

```
BL51..almctrl.obj XDATA(?XD?ALMCTRL(2000h))…
```

对LX51，用SEGMENTS命令定位xdata空间的段：

```
LX51 …almctrl.obj SEGMENTS(?XD?ALMCTRL(X:0x2000))…
```

这指示连接器定位段名?XD?ALMCTRL在**xdata**存储区的地址2000h。

用相同的方法，也可以定位别的存储区如**code**，**xdata**，**pdata**，**idata**，和**data**的段。*参考A51宏汇编用户手册*。

_at_关键词

第三种方法是在C源文件中声明变量时用**_at_**关键词。下面的例子说明用**_at_**关键词如何定位几种不同类型的变量。

```
struct link  {
  struct link idata *next;
  char        code  *test;
};

struct link list idata _at_ 0x40;     /* list at idata 0x40 */
char xdata text[256]   _at_ 0xE000;   /* array at xdata 0xE000 */
int xdata i1           _at_ 0x8000;   /* int at xdata 0x8000 */

void main ( void ) {
  link.next = (void *) 0;
  i1        = 0x1234;
  text [0]  = 'a';
```

参考104页的“绝对变量定位”。

注意:

*如果用**_at_**关键词声明一个变量访问一个XDATA外设，要求用**volatile**关键词以确保C编译器不对必需的存储区访问优化。*

调试

当使用μ Vision2 IDE和μ Vision2调试器，使能**Options for Target – Output – Debug Information**可以得到完整的调试信息。对命令行工具用下面的规则。

缺省，C51编译器对目标文件使用INTEL目标格式（OMF-51），生成完整的符号信息。所有INTEL兼容的仿真器都可用来调试程序。**DEBUG**命令在目标文件中嵌入调试信息。另外，**OBJECTEXTEND**命令在目标文件中嵌入附加变量类型信息，这允许用特定的仿真器显示变量和结构的类型。

Cx51编译器用OMF2目标文件格式。当OMF2命令激活时， OMF2格式也被**Cx51**编译器所用。OMF2格式要求扩展的LX51连接/定位器，不能用BL51连接/定位器。OMF2目标文件格式提高广泛的调试信息，被μ Vision2调试器和一些仿真器支持。

第七章. 错误信息

本章列出了编程中可能遇到的致命错误，语法错误，和警告信息。每节包括一个信息的主要说明，和消除错误或警告条件可采取的措施。

致命错误

致命错误立即终止编译。这些错误通常是命令行指定的无效选项的结果。当编译器不能访问一个特定的源包含文件时也产生致命错误。

致命错误信息采用下面的格式：

C51 FATAL-ERROR –
ACTION: <current action>
LINE: <line in which the error is detected>
ERROR: <corresponding error message>
C51 TERMIANTED.

C51 FATAL-ERROR –
ACTION: <current action>
FILE: <file in which the error is detected>
ERROR: <corresponding error message>
C51 TERMIANTED.

下面说明**Action**和**Error**中可能的内容。

Actions

ALLOCATING MEMORY

编译器不能分配足够的存储区来编译指定的源文件。

CREATING LIST-FILE / OBJECT-FILE / WORKFILE

编译器不能建立列表文件，OBJ文件，或工作文件。这个错误的出现可能是磁盘满或写保护，或文件已存在和只读。

GENERATING INTERMEDIATE CODE

源文件包含的一个函数太大，不能被编译器编译成虚拟代码。尝试把函数分小或重新编译。

OPENING INPUT-FILE

编译器不能发现或打开所选的源或包含文件。

PARSING INVOKE-/#PRAGMA-LINE

当在命令行检测到参数计算，或在一个#**pragma**中检测到参数计算，就产生这样的错误。

PARSING SOURCE-FILE / ANALYZING DECLARATIONS

源文件包含太多的外部参考。减少源文件访问的外部变量和函数的数目。

WRITING TO FILE

当写入列表文件，OBJ文件，或工作文件时遇到的错误。

Errors

'(' AFTER CONTROL EXPECTED

一些控制参数需要用括号包含一个参数。当没有左括号时显示本信息。

')' AFTER PARAMETER EXPECTED

本信息表示包含没有参数的右括号。

BAD DIGIT IN NUMBER

一个控制参数的数字参数包含无效字符。只能是十进制数。

CAN'T CREATE FILE

在**FILE**行定义的文件名不能建立。

CAN'T HAVE GERERAL CONTROL IN INVOCATION LINE

一般控制（例如，**EJECT**）不能包含在命令行。把这些控制用**#pragma**声明放在源文件中。

FILE DOES NOT EXIST

没有发现定义在**FILE**行的文件。

FILE WRITE-ERROR

因为磁盘空间不够，写到列表，预打印，工作，或目标文件时出错。

IDENTIFIER EXPECTED

当**DEFINE**控制没有参数时产生本信息。**DEFINE**要求一个参数作为标识符。这和C语言的规则相同。

MEMORY SPACE EXHAUSTED

编译器不能分配足够的存储区来编译指定的源文件。如果始终出现这个信息，应该把源文件分成两个或多个小文件再重新编译。

MORE THAN 100 ERRORS IN SOURCE-FILE

在编译时检测到的错误超过100个。这使编译器终止。

MORE THAN 256 SEGMENTS/EXTERNALS

在一个源文件中的参考超过256个。单个的源文件不能有超过256个函数或外部参考。这是INTEL目标模块格式（OMF-51）的历史的限制。包含标量和/或**bit**声明的函数在OBJ文件中生成两个，有时候三个段定义。

NON-NULL ARGUMENT EXPECTED

所选的控制参数需要用括号包含一个参数（例如，一个文件名或一个数字）。

OUT OF RANGE NUMBER

一个控制参数的数字参数超出范围。例如，**OPTIMIZE**控制只允许数字0到6。值7就将产生本错误信息。

PARSE STACK OVERFLOW

解析堆栈溢出。如果源程序包含很复杂的表达式或如果块的嵌套深度超过31级，就会出现这个错误。

PREPROCESSOR：LINE TOO LONG（32K）

一个中间扩展长度超过32K字符。

PREPROCESSOR：MACROS TOO NESTED

在宏扩展期间，预处理器所用的堆栈太大。这个信息通常表示一个递归的宏定义，但也可表示一个宏嵌套太多。

RESPECIFIED OR CONFLICTING CONTROL

一个命令行参数指定了两次，或命令行参数冲突。

SOURCE MUST COME FROM A DISK-FILE

源和包含文件必须存在。控制台CON:，：CI:，或类似的设备不能作为输入文件。

UNKNOWN CONTROL

所选的控制参数不认识。

语法和语义错误

语法和语义错误一般出现在源程序中。它们确定实际的编程错误。当遇到这些错误时，编译器尝试绕过错误继续处理源文件。当遇到更多的错误时，编译器输出另外的错误信息。但是，不产生OBJ文件。

语法和语义错误在列表文件中生成一条信息。这些错误信息用下面的格式：

***** ERROR *number* IN LINE *line* OF *file*:*error message***

这里：

number 错误号。

line 对应源文件或包含文件的行号。

file 产生错误的源或包含文件名。

error message 对错误的叙述说明。

下表按错误号列出了语法和语义错误。错误信息列出了主要说明和可能的原因和改正。

号	错误信息和说明
100	**跳过不可打印字符0x??** 在源文件中发现一个非法字符。（注意不检查注释中的字符）
101	**字符串没结束** 一个字符串没有用双引号（”）终止。
102	**字符串太长** 一个字符串不能超过**4096**个字符。用串联符号（‘\’）在逻辑上可延长字符串超过4096个字符。这个模式的行终止符在词汇分析时是连续的。
103	**无效的字符常数** 一个字符常数的格式无效。符号‘**\c**’是无效的，除非**c**是任何可打印的ASCII字符。
125	**声明符太复杂（20）** 一个目标的声明可包含最多20个类型修饰符（‘[’，‘]’，‘*’，‘(’，‘)’）。这个错误经常伴随着错误126。

号	错误信息和说明
126	**类型堆栈下溢** 类型声明堆栈下溢。这个错误通常死错误125的副产品。
127	**无效存储类** 一个目标用一个无效的存储空间标识符声明。如果一个目标在一个函数外用存储类**auto**或**register**声明，就会产生本错误。
129	**在‘标记’前缺少‘;’** 本错误通常表示前一行缺少分号。当出现本错误时，编译器会产生很多错误信息。
130	**值超出范围** 在一个**using**或**interrupt**标识符后的数字参数是无效的。**using**标识符要求一个0到3之间的寄存器组号。**interrupt**标识符要求一个0到31之间的中断矢量号。
131	**函数参数重复** 一个函数有相同的参数名。在函数声明中参数名必须是唯一的。
132	**没在正式的参数列表** 一个函数的参数声明用了一个名称没在参数名列表中。例如： **char function(v0,v1,v2)** **char *v0,*v1,*v5;** **/* ‘v5’没在正式列表中 */** **{** **/* … */** **}**
134	**函数的xdata/idata/pdata/data不允许** 函数通常位于**code**存储区，不能在别的存储区运行。函数默认定义为存储类型**code**。
135	**bit的存储类错** **bit**标量的声明可能包含一个**static**或**extern**存储类。**register**或**alien**类是无效的。
136	**变量用了‘void’** **void**类型只允许作为一个不存在的返回值，或一个函数的空参数列表（**void func(void)**），或和一个指针组合（**void ***）。
138	**Interrupt()不能接受或返回值** 一个中断函数被定义了一个或多个正式的参数，或一个返回值。中断函数不能包含调用参数或返回值。
140	**位在非法的存储空间** **bit**标量的定义可以包含可选的存储类型**data**。如果没有存储类型，则默认为**data**，因为位通常在内部数据存储区。当试图对一个**bit**标量定义别的数据类型时会产生本错误。
141	**临近*标志*语法错误：期待*别的标志*，…** 编译器所见的***标志***是错误的。参考所显示的期待的内容。
142	**无效的基地址** 一个**sfr**或**sbit**声明的基地址是错误的。有效的基地址范围在0x80到0xFF之间。如果用符号**基地址^位号**声明，则基地址必须是8的倍数。

号	错误信息和说明
143	**无效的绝对位地址** **sbit**声明中的绝对位地址必须在0x80到0xFF之间。
144	**基地址^位号：无效的位号** **sbit**声明中定义的位号必须在0到7之间。
145	**未知的sfr**
146	**无效sfr** 一个绝对位（基地址^位号）的声明包含一个无效的基地址标识符。基地址必须是已经声明的**sfr**。任何别的名称是无效的。
147	**目标文件太大** 单个目标文件不能超过65535（64K字节-1）。
149	**struct/union包含函数成员** struct或union不能包含一个函数类型的成员。但是，指向函数的指针是可以的。
150	**struct/union包含一个bit成员** 一个union不能包含**bit**类型成员。这是8051的结构决定的。
151	**struct/union自我关联** 一个结构不能包含自己。
152	**位号超出位域** 位域声明中指定的位号超过给定基类的位号。
153	**命名的位域不能为零** 命名的位域为零。只要未命名的位域允许为零。
154	**位域指针** 指向位域的指针不允许。
155	**位域要求char/int** 位域的基类要求**char**或**int**。**unsigned char**和**unsigned int**类型也行。
156	**alien只允许对函数**
157	**alien函数带可变参数** 存储类**alien**只对外部PL/M-51函数允许。符号（**char *,…**）在alien函数中是非法的。PL/M-51函数通常要求一个固定的参数表。
158	**函数包含未命名的参数** 一个函数的参数列表定义包含一个未命名的抽象类型定义。这个符号只允许在函数原型中。
159	**void后面带类型** 函数的原型声明可包含一个空参数列表（例如，**int func(void)**）。在**void**后不能再有类型定义。
160	**void无效** **void**类型只在和指针组合，或作为一个函数的不存在的返回值中是合法的。
161	**忽视了正式参数** 在一个函数内，一个外部函数的声明用了一个没有类型标识符的参数名列表（例如，**extern yylex(a,b,c);**）。

号	错误信息和说明
180	**不能指向一个‘函数’** 指向一个函数的类型是无效的。尝试用指针指向一个函数。
181	**操作数不兼容** 对给定的操作符，至少一个操作数类型是无效的（例如，**~float_type**）。
183	**左值不能修改** 要修改的目标位于**code**存储区或有**const**属性，因此不能修改。
184	**sizeof：非法操作数** **sizeof**操作符不能确定一个函数或位域的大小。
185	**不同的存储空间** 一个目标声明的存储空间和前一个同样目标声明的存储空间不同。
186	**解除参照无效** 一个内部编译器问题会产生本信息。如果本错误重复出现请和技术支持接洽。
187	**不是一个左值** 所需的参数必须是一个可修改的目标地址。
188	**未知目标大小** 因为没有一个数组的维数，或间接通过一个**void**指针，一个目标的大小不能计算。
189	**‘&’对bit/sfr非法** 取地址符（’**&**’）不允许对**bit**目标或特殊函数寄存器（**sfr**）。
190	**‘&’：不是一个左值** 尝试建立一个指针指向一个未知目标。
193	**非法操作类型**
193	**对ptr非法add/sub**
193	**对bit的非法操作**
193	**错误操作数类型** 当对一个给定的操作符用了非法的操作数类型时产生本错误。例如，无效的表达式如：**bit*bit**，ptr+ptr，或ptr**anything**。这个错误信息包括引起错误的操作符。 下面的操作对bit类型的操作数是可行的： ■ 赋值（=） ■ OR/复合OR（|，|=） ■ AND/复合AND（**&**，**&**=） ■ XOR/复合XOR（^，^=） ■ bit比较（= =，**!**=） ■ 取反（~） bit操作数可和别的数据类型在表达式中混用。在这种情况类型转换自动执行。
194	**‘*’间接指向一个未知大小的目标** 间接操作符*不能和void指针合用，因为指针所指的目标的大小是未知的。
195	**‘*’间接非法** *操作符不能用到非指针参数。

号	错误信息和说明
196	**存储空间可能无效** 转换一个常数到一个指针常数产生一个无效的存储空间。例如 char *p=0x91234。
198	**sizeof返回零** sizeof操作符返回一个零。
199	**‘->’的左边要求struct/union指针** ->操作符的左边参数必须是一个**struct**指针或一个**union**指针。
200	**‘.’左边要求struct/union** .操作符的左边参数要求必须是**struct**或**union**类型。
201	**未定义的struct/union** 给定的**struct**或**union**名是未知的。
202	**未定义的标识符** 给定的标识符是未定义的。
203	**错误的存储类（参考名）** 本错误表示编译器的一个问题。如果重复出现请接洽技术支持。
204	**未定义的成员** 给定的一个**struct**或**union**成员名是未定义的。
205	**不能调用一个中断函数** 一个中断函数不能象一个正常函数一样调用。中断的入口和退出代码是特殊的。
207	**参数列表声明为‘void’** 参数列表声明为**void**的函数不能从调用者接收参数。
208	**太多的实参** 函数调用包含太多的实参。
209	**太少的实参** 调用函数包含太少的实参。
210	**太多的嵌套调用** 函数的嵌套调用不能超过10级。
211	**调用不是对一个函数** 一个函数的调用项不是对一个函数或函数指针求值。
212	**间接调用：寄存器的参数不匹配** 通过一个指针的间接函数调用不包含实际的参数。一个例外是当所有的参数可以通过寄存器传递。这是由于Cx51所用的传递参数的方法。被调用的函数名必须是已知的，因为参数写到被调用函数的数据段。但是，对间接调用来说，被调用函数的名称是未知的。
213	**赋值符的左边不是一个左值** 赋值符的左边要求一个可修改目标的地址。
214	**非法指针转换** bit，float或集合类型的目标不能转换为指针。
215	**非法类型转换** struct/union/void不能转换为任何别的类型。

号	错误信息和说明
216	**标号用在非数组中，或维数超出** 一个数组引用包含太大的维数，或目标不是一个数组。
217	**非整数索引** 一个数组的维数表达式必须是**char**，**unsigned char**，**int**，或**unsigned int**类型。别的类型都是非法的。
218	**控制表达式用了void类型** 在一个**while**，**for**，或**do**的限制表达式中不能用类型**void**。
219	**long常数缩减为int** 一个常数表达式的值必须能用一个**int**类型表示。
220	**非法常数表达式** 期望一个常数表达式。目标名，变量或函数不允许出现在常数表达式中。
221	**非常数case/dim表达式** 一个**case**或一个维数（**[ ]**）必须是一个常数表达式。
222	**被零除**
223	**被零取模** 编译器检测到一个被零除或取模。
225	**表达式太复杂，需简化** 一个表达式太复杂，必须分成两个或多个子表达式。
226	**重复的struct/union/enum标记** 一个**struct**，**union**，或**enum**名早已定义。
227	**表示一个union标记** 一个**union**名称早已定义为别的类型。
228	**表示一个struct标记** 一个**struct**名早已定义为别的类型。
229	**表示一个enum标记** 一个**enum**名早已定义为别的类型。
230	**未知的struct/union/enum标记** 指定的**struct**，**union**，或**enum**名未定义。
231	**重复定义** 指定的名称已被定义。
232	**重复标号** 指定的标号已定义。
233	**未定义标号** 表示一个标号未定义。有时候这个信息会在实际的标号的几行后出现。这是所用的未定义标号的搜索方法引起的。
234	**‘{’，堆栈范围溢出（31）** 超过了最多31个嵌套块。超出的嵌套块被忽略。
235	**参数<数字>：不同类型** 函数声明的参数类型和函数原型中的不同。

号	错误信息和说明
236	**参数列表的长度不同** 函数声明中的参数数目和函数原型中的不同。
237	**函数早已定义** 试图声明一个函数体两次。
238	**重复成员**
239	**重复参数** 试图定义一个已存在的**struct**成员或函数参数。
240	**超出128个局部bit** 在一个函数内不能超过128个**bit**标量。
241	**auto段太大** 局部目标所需的空间超过模式的极限。最大的段大小定义如下： **SMALL** 128字节 **COMPACT** 256字节 **LARGE** 65535字节
242	**太多的初始化软件** 初始化软件的数目超过初始化目标的数量。
243	**字符串超出范围** 字符串中的字符数目超出字符串初始化的数目。
244	**不能初始化，错误的类型或类** 试图初始化一个**bit**或**sfr**。
245	**未知的pragma，跳过本行** **#pragma**状态未知，所以整行被忽略。
246	**浮点错误** 当一个浮点参数超出32位的范围就产生本错误。32位IEEE值的范围是：±1.175494E-38到±3.402823E+38。
247	**非地址/常数初始化** 一个有效的初始化表达式必须是一个常数值求值或一个目标名加或减去一个常数。
248	**集合初始化需要大括号** 给定**struct**或**union**初始化缺少大括号（**{}**）。
249	**段<名>：段太大** 编译器检测到一个数据段太大。一个数据段的最大的大小由存储空间决定。
250	**‘\esc’；值超过255** 一个字符串常数中的转义序列超过有效值范围。最大值是255。
252	**非法八进制数** 指定的字符不是一个有效的八进制数。
252	**主要控制放错地方，行被忽略** 主要控制必须被指定在C模块的开头，在任何**#include**命令或声明前。

号	错误信息和说明
253	**内部错误（ASMGEN\CLASS）** 在下列情况下出现本错误： ■ 一个内在函数（例如，**_testbit_**）被错误激活。这种情况是在没有函数原型存在和实参数目或类型错误。对这种原因，必须使用合适的声明文件（**INTRINS.H，STRING.H**）。参考第八章中的**instrinsic**函数。 ■ Cx51确认一个内部一致性问题。请接洽技术支持。
255	**switch表达式有非法类型** 在一个**switch**表达式没有合法的数据类型。
256	**存储模式冲突** 一个包含**alien**属性的函数只能包含模式标识符**small**。函数的参数必须位于内部数据区。这适用于所有的外部**alien**声明和**alien**函数。例如： alien plm_func(char c) large {…} 产生错误256。
257	**alien函数不能重入** 一个包含**alien**属性的函数不能同时包含**reentrant**属性。函数参数不能跳过虚拟堆栈传递。这适用于所有的外部alien声明和**alien**函数。
258	**struct/union成员的存储空间非法** **非法空间的参数被忽略** 一个结构的成员或参数不能包含一个存储类型标识符。但，指针所指的目标可能包含一个存储类型。例如： struct vp{char code c;int xdata i; }; 产生错误258。 struct v1{char c;int xdata *i; }; 是正确的struct声明。
259	**指针：不同的存储空间** 一个空指针被关联到别的不同存储空间的空指针。例如： char xdata *p1; char idata *p2; p1 = p2; /* 不同的存储空间 */
260	**指针断开** 一个空指针被关联到一些常数值，这些值超过了指针存储空间的值范围。例如： char idata *p1 = 0x1234; /* 结果是0x34 */

号	错误信息和说明

261	**reentrant()内有bit**
	一个可重入属性的函数的声明中不能包含bit目标。例如：
	
	int func1(int i1) reentrant {
	bit b1,b2; /* 不允许！ */
	return(i1-1);
	}

262	**‘using/disable’：不能返回bit值**
	用**using**属性声明的函数和禁止中断（**#pragma disable**）的函数不能返回一个**bit**值给调用者。例如：
	
	bit test(void) using 3
	{
	bit b0;
	return(b0);
	}
	
	产生错误262。

263	**保存/恢复：堆栈保存溢出/下溢** **#pragma save**的最大嵌套深度是八级。堆栈的**pragma save**和**restore**工作根据LIFO（后进，先出）规则。
264	**内在的‘<内在的名称>’：声明/激活错误** 本错误表示一个内在的函数错误定义（参数数目或省略号）。如果用标准的**.H**文件就不会产生本错误。确认使用了Cx51所有的**.H**文件。不要尝试对内在的库函数定义自己的原型。
265	**对非重入函数递归调用** 非重入函数不能被递归调用，因为这样会覆盖函数的参数和局部数据。如果需要递归调用，需声明函数为可重入函数。
267	**函数定义需要ANSI类型的原型** 一个函数被带参数调用，但是声明是一个空的参数列表。原型必须有完整的参数类型，这样编译器就可能通过寄存器传递参数，和适合应用的调用机制。
268	**任务定义错误（任务ID/优先级/using）** 任务声明错误。
271	**‘asm/endasm’控制放错地方** asm和endasm声明不能嵌套。endasm要求一个汇编块，前面用asm开头。例如： #pragma asm . . . 汇编指令 . . . #pragma endasm

号	错误信息和说明
272	**‘asm’要求激活SRC控制** 在一个源文件中使用asm和endasm，要求文件用SRC控制编译。那么编译器就会生成汇编源文件，然后可以用A51汇编。
273	**‘asm/endasm’在包含文件中不允许** 在包含文件中不允许asm和endasm。为了调试，在包含文件不能有任何的可执行代码。
274	**非法的绝对标识符** 绝对地址标识符对位目标，函数，和局部函数不允许。地址必须和目标的存储空间一致。例如，下面的声明是无效的，因为间接寻址的范围是0x00到0xFF。 idata int _at_ 0x1000;
278	**常数太大** 当浮点参数超出32位的浮点值范围就产生本错误。32位IEEE值的范围是：±1.175494E-38到±3.402823E+38。
279	多次初始化 试图多次初始化一个目标。
280	没有使用符号/标号/参数 在一个函数中声明了一个符号，标号，或参数，但没有使用。
281	非指针类型转换为指针 引用的程序目标不能转换成一个指针。
282	不是一个SFR引用 本函数调用要求一个SFR作为参数。
283	asmparms：参数不适合寄存器 参数不适合可用的CPU寄存器。

284	**<名称>：在可覆盖空间，函数不再可重入** 一个可重入函数包含对局部变量的明确的存储类型标识符。函数不再完全可重入。
300	**注释未结束** 一个注释没有一个结束符（*/）。
301	**期望标识符** 一个预处理器命令期望一个标识符。
302	**误用#操作符** 字符操作符‘#’没有带一个标识符。
303	**期望正式参数** 字符操作符‘#’没有带一个标识符表示当前所定义的宏的一个正式参数名。
304	**错误的宏参数列表** 宏参数列表没有一个大括号，逗号分开的标识符列表。
305	**string/char 常数未结束** 一个字符串活字符常数是无效的。典型的，后引号丢失。
306	**宏调用未结束** 预处理器在收集和扩展一个宏调用的实际的参数时遇到输入文件的结尾。

号	错误信息和说明
307	**宏‘名称’：参数计算不匹配** 在一个宏调用中，实际的参数数目和宏定义的参数数目不匹配。本错误表示指定了太少的参数。
308	**无效的整数常数表达式** 一个 if/elif 命令的数学表达式包含一个语法错误。
309	**错误或缺少文件名** 在一个 include 命令中的文件名参数是无效的，或没有。
310	**条件嵌套过多（20）** 源文件包含太多的条件编译嵌套命令。最多允许 20 级嵌套。
311	**elif/else 控制放错地方**
312	**endif 控制放错地方** 命令 elif，else，和 endif 只有在 if，ifdef，或 ifndef 命令中是合法的。
313	**不能清除预定义的宏‘名称’** 试图清除一个预定义宏。用户定义的宏可以用#undef 命令删除。预定义的宏不能清除。
314	**#命令语法错误** 在一个预处理器命令中，字符‘#’必须跟一个新行或一个预处理器命令名（例如，if/define/ifdef，…）。
315	**未知的#命令‘名称’** 预处理器命令是未知的。
316	**条件未结束** 到文件结尾，endif 的数目和 if 或 ifdef 的数目不匹配。
318	**不能打开文件‘文件名’** 指定的文件不能打开。
319	**‘文件’不是一个磁盘文件** 指定的文件不是一个磁盘文件。文件不能编辑。
320	**用户自定义的内容** 本错误号未预处理器的#error 命令保留。#error 命令产生错误号 320，送出用户定义的错误内容，终止编译器生成代码。
321	**缺少<字符>** 在一个 **include** 命令的文件名参数中，缺少结束符。例如： #include<stdio.h
325	**正参‘名称’重复** 一个宏的正参只能定义一次。
326	**宏体不能以‘##’开始或结束** ‘##’不能是一个宏体的开始或结束。
327	**宏‘宏名’：超过 50 个参数** 每个宏的参数数目不能超过 50。

警告

警告产生潜在问题的信息，他们可能在目标程序的运行过程中出现。警告不妨碍源文件的编译。

警告在列表文件中生成信息。警告信息用下面的格式：

***** WARNING *number* IN LINE *line* OF *file*: *warning message***

这里：

number	错误号。
line	在源文件或包含文件中的对应行号。
file	错误产生的源或包含文件名。
warning message	警告的内容。

下表按号列出了警告。警告信息包括一个主要的内容和可能的原因和纠正措施。

号	警告信息和说明
173	**缺少返回表达式** 一个函数返回一个除了 int 类型以外的别的类型的值，必须包含一个返回声明，包括一个表达式。为了兼容旧的程序，对返回一个 int 值的函数不作检查。
182	**指针指向不同的目标** 一个指针关联了一个不同类型的地址。
185	**不同的存储空间** 一个目标声明的存储空间和前面声明的同样目标的存储空间不同。
196	**存储空间可能无效** 把一个无效的常数值分配给一个指针。无效的指针常数是 long 或 unsigned long。编译器对指针采用 24 位（3 字节）。低 16 位代表偏移。高 8 位代表选择的存储空间。
198	**sizeof 返回零** 一个目标的大小计算结果为零。如果目标是外部的，或如果一个数组的维数没有全知道，则值是错误的。

号	警告信息和说明
206	**缺少函数原型** 因为没有原型声明，被调用的函数是未知的。调用一个未知的函数通常是危险的，参数的数目和实际要求不一样。如果是这种情况，函数调用不正确。 没有函数原型，编译器不能检查参数的数目和类型。要避免这种警告，应在程序中包含函数的原型。 函数原型必须在函数被调用前声明。注意函数定义自动生成原型。
209	**实参太少** 在一个函数调用中包含太少的实参。
219	**long 常数被缩减为 int** 一个常数表达式的值必须能被一个 int 类型所表示。
245	**未知的 pragma，本行被忽略** #pragma 声明是未知的，因此整行程序被忽略。
258	**struct/union 成员的存储空间方法** 参数的存储空间被忽略 一个结构的成员或一个参数不能指定存储类型。但是，指针所指的目标可以包含一个存储类型。例如： struct vp{ char code c;int xdata i; }; 产生警告 258。 struct v1{ char c;int xdata *i; }; 对 **struct** 是正确的声明。
259	**指针：不同的存储空间** 两个要比较的指针没有引用相同的存储类型的目标。
260	**指针折断** 把一个指针转换为一个更小偏移区的指针。转换会完成，但大指针的偏移会折断来适应小指针。
261	**bit 在重入函数** 一个 reentrant 函数不能包含 bit，因为 bit 标量不能保存在虚拟堆栈中。
265	**‘名称’：对非重入函数递归调用** 发现对一个非重入函数直接递归。这可能是故意的，但对每个独立的情况进行功能性检查（通过生成的代码）。间接递归由连接/定位器检查。

号	警告信息和说明
271	**‘asm/endams’控制放错地方** asm 和 endasm 不能嵌套。endasm 要求一个以 asm 声明开头的汇编块。例如： #pragma asm . . . 汇编指令 . . . #pragma endasm

275	**表达式可能无效**
	编译器检测到一个表达式不生成代码。例如：
	
	void test(void) {
	int i1,i2,i3;
	i1,i2,i3; /* 死表达式 */
	i1 << i3; /* 结果未使用 */
	}

276	**常数在条件表达式**
	编译器检测到一个条件表达式有一个常数值。在大多数情况下是一个输入错误。例如：
	
	void test(void) {
	int i1,i2,i3;
	if(i1 = 1) i2 = 3; /* 常数被赋值 */
	while(i3 = 2); /* 常数被赋值 */
	}

277	**指针有不同的存储空间** 一个 typedef 声明的存储空间冲突。例如： typedef char xdata XCC; /* 存储空间 xdata */ typedef XCC idata PICC; /* 存储空间冲突 */
280	**符号/标号未使用** 一个符号或标号定义但未使用。
307	**宏‘名称’：参数计算不匹配** 一个宏调用的实参的数目和宏定义的参数数目不匹配。表示用了太多的的参数。过剩的参数被忽略。
317	**宏‘名称’：重新定义无效** 一个预定义的宏不能重新定义或清除。参考 138 页的“预定义宏常数”。
322	**未知的标识符** 在一个#if 命令行的标识符未定义（等效为 FALSE）。
323	**期望新行，发现多余字符** 一个#命令行正确，但包含多余的非注释字符。例如： #include <stdio.h> foo

号	警告信息和说明
324	**期望预处理器记号** 期望一个预处理器记号，但输入的是一个新行。例如：#line，这里缺少#line 命令的参数。

第八章. 库参考

C*x*51 运行库提供超过 100 个可用在 8051 C 程序中的预定义函数和宏。通过库所提供程序执行公共的程序任务，例如字符串和缓冲区操作，数据转换，和浮点算术运算，使得内嵌软件开发更容易，

典型的，本库的程序符合 ANSI C 标准。但是，为了利用 8051 结构的特性，一些程序有些不同。例如，函数 **isdigit** 返回一个 **bit** 值而不是一个 **int**。如有可能，函数的返回类型和参数类型调整为更小的数据类型。另外，unsigned data 类型比 signed 更有利。这些对标准库的改变可以提供最好的性能，同时减少程序的大小。

库中的所有程序和函数使用什么寄存器组无关。

固有程序

C*x*51 编译器支持许多固有的库函数。非固有函数用 **ACALL** 或 **LCALL** 指令调用库程序。固有函数生成内嵌代码运行库程序。生成的内嵌代码比调用一个程序更快，更有效。下面的函数就是固有函数：

crol	**_iror_**	**_nop_**
cror	**_lrol_**	**_testbit_**
irol	**_lror_**	

在下面各节中详细这些程序。

库文件

Cx51 库包括六个编译库，对各种功能性要求进行优化。这些库支持大多数的 ANSI C 函数调用。

库文件	说明
C51S.LIB	SMALL 模式库，没有浮点运算
C51FPS.LIB	SMALL 模式，浮点运算库
C51C.LIB	COMPACT 模式库，没有浮点运算
C51FPC.LIB	COMPACT 模式浮点运算库
C51L.LIB	LARGE 模式库，没有浮点运算
C51FPL.LIB	LARGE 模式浮点运算库
80C751.LIB	SIGNETICS 8xC751 和派生系列库

PHILIPS 80C51MX，DALLAS390 连续模式和可变代码 BANKING 要求一个不同的 Cx51 运行库。LX51 连接/定位器自动把相应的库加到 PROJECT 中。

几个库模块以源代码形式提供。这些程序用来运行和低级硬件相关的 I/O 的流 I/O 函数。可以在 **LIB** 目录发现这些程序的源文件。可以修改这些源文件，替换库中的程序。应用这些程序，可以快速的使用库来运行（用任何目标可用的硬件 I/O 设备）流 I/O。参考 224 页的“流输入和输出”。

标准类型

C*x*51 标准库包含许多标准类型的定义，它们可以用在库程序中。这些标准类型在包含文件中声明，可以被 C 程序访问。

jmp_buf

jmp_buf 类型在 **SETJMP.H** 中定义，指定了 **setjmp** 和 **longjmp** 程序用来保存和恢复程序环境的缓冲区。**jmp_buf** 类型如下定义：

```
#define    _JBLEN  7
typedef    char jmp_buf[_JBLEN];
```

va_list

va_list 数组类型在 **STDARG.H** 定义。本类型保存 **va_arg** 和 **va_end** 程序所需的数据。**va_list** 类型如下定义：

```
typedef    char *va_list;
```

绝对存储区访问宏

C*x*51 标准库包含许多允许访问直接存储地址的宏的定义。这些宏定义在 ABSACC.H 中。每个宏的用法和数组一样。

CBYTE

CBYTE 宏允许访问 8051 程序存储区的每个字节。可以在程序中这样使用宏：

```
rval = CBYTE[0x0002];
```

读程序存储区地址 0002h 字节的内容。

CWORD

CWORD 宏允许访问 8051 程序存储区的每个字节。可以在程序中这样使用宏：

```
rval = CWORD[0x0002];
```

读程序存储区地址 0004h（**2×sizeof(unsigned int)=4**）字的内容。

注意：
这个宏所用的索引不代表存储区地址的整数值。为了得到存储区地址，必须索引乘以一个整数（2 字节）的大小。

DBYTE

DBYTE 宏允许访问 8051 内部数据存储区的单个字节。可以在程序中这样使用宏：

```
rval = DBYTE[0x0002];
DBYTE[0x0002] = 5;
```

读或写内部数据区地址 0002h 字节的内容。

DWORD

DWORD 宏允许访问 8051 内部数据存储区的单个字节。可以在程序中这样使用宏：

```
rval = DWORD[0x0002];
DWORD[0x0002] = 57;
```

读或写内部数据区地址 0004h（**2×sizeof(unsigned int)=4**）字的内容。

注意：

这个宏所用的索引不代表存储区地址的整数值。为了得到存储区地址，必须索引乘以一个整数（2 字节）的大小。

FARRAY，FCARRAY

FARRAY 和 **FCARRAY** 宏可以用来访问一个 **far** 和 **const far** 存储区的数组类型目标。**FARRAY** 访问 **far** 空间（存储类 HDATA）。**FCARRAY** 访问 **const far** 空间（存储类 HCONST）。可以在程序中这样用宏：

```
int i;
long l;

l = FARRAY(long,0x8000)[i];
FARRAY(long,ox8000)[10] = 0x12345678;

#define DualPortRam FARRAY(int,0x24000)
DualPortRam[i] = 0x1234;

l = FCARRAY(long,0x18000)[5];
```

FARRAY 和 **FCARRAY** 宏把目标的类型大小加到结果地址，得到索引。最后的地址用来访问存储区。

注意：

绝对地址目标不能穿过 64KB 段边界。例如，不能访问一个起始地址为 0xFFF8，有 10 个元素的 long 数组。

FVAR，FCVAR

FVAR 和 **FCVAR** 宏可以用来访问 **far** 和 **const far** 存储区的绝对地址。**FVAR** 访问 **far** 空间（存储类 HDATA）。**FCVAR** 访问 **const far** 空间（存储类 HCONST）。可以在程序中这样用宏：

```
#define IOVL FVAR(long,0x14FFE)          // long 在 HDATA 地址 0x14FFE
var = IOVAL;                             /* 读 */
IOVAL = 0x10;                            /* 写 */

var = FCVAR(int,0x24002)                 /* 从 HCONST 地址 0x24002 读 int */
```

HVAR 宏用 **huge** 修饰符，用段和偏移访问存储区，但和 **MVAR** 的页和偏移不一样。

注意：

绝对地址目标不能穿过 64KB 段边界。例如，不能在地址 0xFFFE 访问一个 long 变量。

PBYTE

PBYTE 宏允许访问 8051 外部页数据区的单个字节。可以如下使用：

```
rval = PBYTE[0x0002];
PBYTE[0x002] = 38;
```

读或写 **pdata** 存储区地址 0002h 的字节内容。

PWORD

PWORD 宏允许访问 8051 外部页数据区的单个字。可以如下使用：

```
rval = PWORD[0x0002];
PWORD[0x002] = 57;
```

读或写 **pdata** 存储区地址 0004h（**2×sizeof(unsigned int)=4**）的字内容。

注意：

这个宏所用的索引不代表存储区地址的整数值。为了得到存储区地址，必须索引乘以一个整数（2 字节）的大小。

XBYTE

XBYTE 宏允许访问 8051 外部数据区的单个字节。可以如下使用：

```
rval = XBYTE[0x0002];
XBYTE[0x002] = 57;
```

读或写外部数据存储区地址 0002h 的字节内容。

XWORD

XWORD 宏允许访问 8051 外部数据区的单个字。可以如下使用：

```
rval = XWORD[0x0002];
XWORD[0x002] = 57;
```

读或写外部数据存储区地址 0004h（**2×sizeof(unsigned int)=4**）的字内容。

注意：

这个宏所用的索引不代表存储区地址的整数值。为了得到存储区地址，必须索引乘以一个整数（2 字节）的大小。

程序分类

本节给出 **Cx51** 标准库主要程序类的一个概况。参考 232 页的“参考”。

注意:
Cx51 标准库中的许多程序是可重入的，固有的，或两者都是。这些说明列在下表的属性中。除非特别说明，程序是非重入和非固有的。

缓冲区操作

程序	属性	说明
memccpy		从一个缓冲区拷贝数据字节到另一个缓冲区，直到一个指定的字符或字符数。
memchr	可重入	返回一个缓冲区中指定字符第一次出现的位置指针。
memcmp	可重入	比较两个不同缓冲区给定数目的字符。
memcpy	可重入	从一个缓冲区拷贝指定数目的数据到另一个缓冲区。
memmove	可重入	从一个缓冲区拷贝指定数目的数据到另一个缓冲区。
memset	可重入	初始化一个缓冲区的指定数目的数据字节为指定的字符值。

缓冲区操作程序用在存储缓冲区，以字符为基础。一个缓冲区就是一个字符数组，类似于字符串，但是，缓冲区不是用 NULL（‘**\0**’）字符结束。因此，这些程序要求一个缓冲区长度或计数。

所有的这些程序作为程序运行。函数原型在 **STRING.H** 包含文件中。

字符转换和分类

程序	属性	说明
isalnum	可重入	是否是一个字母或数字字符。
isalpha	可重入	是否是一个字母字符。
iscntrl	可重入	是否是一个控制字符。
isdigit	可重入	是否是一个十进制数。
isgraph	可重入	是否是一个除空格以外的可打印字符。
islower	可重入	是否是一个小写字母字符。
isprint	可重入	是否是一个可打印字符。
ispunct	可重入	是否是一个标点字符。
isspace	可重入	是否是一个空格。
isupper	可重入	是否是一个大写字母字符。
isxdigit	可重入	是否是一个十六进制数。
toascii	可重入	转换一个字符为一个 ASCII 码。
toint	可重入	转换一个十六进制数为一个十进制数。
tolower	可重入	测试一个字符，如果是大写则转换成小写。
_tolower	可重入	无条件的转换一个字符为小写。
toupper	可重入	测试一个字符，如果是大写则转换成小写。
_toupper	可重入	无条件的转换一个字符为大写。

字符转换和分类程序允许测试单个字符的各种属性，并转换成不同的格式。

_tolower，**_toupper**，和 **toascii** 程序作为宏来运行。所有的别的程序作为函数。所有的宏定义和函数原型在包含文件 **CTYPE.H** 中。

数据转换

程序	属性	说明
abs	可重入	取一个整数类型的绝对值。
atof/atof517		转换一个字符串为一个 float。
atoi		转换一个字符串为一个 int。
atol		转换一个字符串为一个 long。
cabs	可重入	取一个字符类型的绝对值。
labs	可重入	取一个 long 类型的绝对值。
strtod/strtod517		一个字符串转换成一个 float。
strtol		一个字符串转换成一个 long。
strtoul		一个字符串转换成一个 unsigned long。

数据转换程序把 ASCII 字符串转换成数字。所有的这些程序作为函数执行，大多数的原型在 **STDLIB.H** 中。**abs**，**cabs**，和 **labs** 函数的原型在 **MATH.H** 中。**atof517**，和 **strtod517** 函数的原型在 **80C517.H** 中。

数学程序

程序	属性	说明
acos/acos517		反余弦。
asin/asin517		反正弦。
atan/atan517		反正切。
atan2		分数的反正切。
ceil		取整。
cos/cos517		余弦。
cosh		双曲余弦。
exp/exp517		指数函数。
fabs	可重入	取绝对值。
floor		小于等于指定数的最大整数。
fmod		浮点数余数。
log/log517		自然对数。
log10/log10517		常用对数。
modf		取出整数和小数部分。
pow		幂。
rand		随机数。
sin/sin517		正弦函数。
sinh		双曲正弦。
srand		初始化随机数发生器。
sqrt/sqrt517		平方根。
tan/tan517		正切函数。
tanh		双曲正切。
chkfloat	固有的，可重入的	检查 float 数的状态。
crol	固有的，可重入的	一个 unsigned char 向左循环位移。
cror	固有的，可重入的	一个 unsigned char 向右循环位移。
irol	固有的，可重入的	一个 unsigned int 向左循环位移。
iror	固有的，可重入的	一个 unsigned int 向右循环位移。
lrol	固有的，可重入的	一个 unsigned long 向左循环位移。
lror	固有的，可重入的	一个 unsigned long 向右循环位移。

数学程序运行通用的算术计算。大多数这些程序用浮点数，因此包含浮点库和支持程序。

所有的这些程序作为函数。多数原型在包含文件 **MATH.H** 中。以 517 结尾的函数（**acos517**，**asin517**，**atan517**，**cos517**，**exp517**，**log517**，**log10517**，**sin517**，**sqrt517**，和 **tan517**）的原型在包含文件 **80C517.H** 中。**rand** 和 **srand** 函数的原型在 **STDLIB.H** 中。

chkfloat，**_crol_**，**_cror_**，**_irol_**，**_iror_**，**_lrol_**，和**_lror_**函数的原型在 **INTRINS.H** 中。

存储区分配程序

程序	属性	说明
calloc		从存储池给一个数组分配存储区。
free		释放用 calloc，malloc，或 realloc 分配彻底存储块。
init_mempool		初始化存储池的位置和大小。
malloc		从存储池分配一个块。
realloc		从存储池再分配一个块。

存储区分配函数提供一种方法来指定，分配，和释放存储池的存储块。所有的存储区分配函数作为函数执行，原型再 **STDLIB.H** 中。

在用这些函数分配存储区前，必须首先用 **init_mempool** 指定存储池的位置和大小，以满足后面要求的存储区。

calloc 和 **malloc** 程序从存储池分配存储块。**calloc** 程序分配一个数组，指定给定大小的元素的数目，初始化数组元素为 0。**malloc** 程序分配指定数目的字节。

realloc 程序改变一个已分配块的大小，**free** 程序释放一个前面分配的存储块回到存储池。

流输入和输出程序

程序	属性	说明
getchar	可重入	用_getkey 和 putchar 程序读和显示一个字符。
_getkey		用 8051 串口读一个字符。
gets		用 getchar 程序读和显示一个字符串。
printf/printf517		用 putchar 程序写格式化数据。
putchar		用 8051 串口写一个字符。
puts	可重入	用 putchar 程序写一个字符串和换行符（'\n'）。
scanf/scanf517		用 getchar 程序读格式化数据。
sprintf/sprintf517		写格式化数据到一个字符串。
sscanf/sscanf517		从一个字符串读格式化数据。
ungetchar		把一个字符放回到 getchar 输入缓冲区。
vprintf		用 putchar 函数写格式化数据。
vsprintf		写格式化数据到一个字符串。

流输入和输出程序允许从 8051 串口或一个用户定义的 I/O 口读和写数据。缺省的 **Cx51** 的**_getkey** 和 **putchar** 函数用 8051 串口读和写字符。可以在 **LIB** 目录发现这些函数的源文件。可以修改这些源文件，并替代库程序。如果这样，别的流函数就会用新的**_getkey** 和 **putchar** 程序输入输出。

如果想要使用已有的**_getkey** 和 **putchar** 函数，必须首先初始化 8051 串口。如果串口没有正确初始化，缺省的流函数就不起作用。初始化串口要求操作 8051 的特殊功能寄存器 SFR。包含文件 **REG51.H** 包含所需的 SFR 的定义。

下面的例子代码必须在启动后立即执行，应在任何流函数调用前。

```
.
.
.
#include <reg51.h>
.
.
.
SCON = 0x50;          /* Setup serial port control register */
                      /* Mode 1: 8-bit uart var. baud rate */
                      /* REN: enable receiver */

PCON &= 0x7F;         /* Clear SMOD bit in power ctrl reg */
                      /* This bit doubles the baud rate */

TMOD &= 0xCF          /* Setup timer/counter mode register */
                      /* Clear M1 and M0 for timer 1 */
TMOD |= 0x20;         /* Set M1 for 8-bit autoreload timer */

TH1 = 0xFD;           /* Set autoreload value for timer 1 */
                      /* 9600 baud with 11.0592 MHz xtal */

TR1 = 1;              /* Start timer 1 */

TI = 1;               /* Set TI to indicate ready to xmit */
.
.
```

流程序把输入和输出作为单个的字符流。有程序处理字符，同时有函数处理字符串。选择最适合要求的程序。

所有的这些程序作为函数执行。大多数原型在 **STDIO.H** 中。**printf517**，**scanf517**，**sprintf517**，和 **sscanf517** 函数的原型在 **80C517.H** 中。

字符串操作程序

程序	属性	说明
strcat		连接两个字符串。
strchr	可重入	返回一个字符串中指定字符第一次出现的位置指针。
strcmp	可重入	比较两个字符串。
strcpy	可重入	拷贝一个字符串到另一个。
strcspn		返回一个字符串中和第二个字符串的任何字符匹配的第一个字符的索引。
strlen	可重入	字符串长度。
strncat		从一个字符串连接指定数目的字符到另一个字符串。
strncmp		比较两个字符串中指定数目的字符。
strncpy		从一个字符串拷贝指定数目的字符到另一个字符串。
strpbrk		返回一个字符串中和第二个字符串的任何字符匹配的第一个字符的指针。
strpos	可重入	返回一个指定字符在一个字符串中第一次出现的索引。
strrchr	可重入	返回一个指定字符在一个字符串中最后出现的指针。
strrpbrk		返回一个字符串中和第二个字符串的任何字符匹配的最后一个字符的指针。
strrpos	可重入	返回一个指定字符在一个字符串中最后出现的索引。
strspn		返回一个字符串中和第二个字符串中的任何字符不匹配的第一个字符索引。
strstr		返回一个字符串中和另一个子字符串一样的指针。

字符串程序作为函数运行，原型在 **STRING.H** 中。它们执行下面的操作：

- 拷贝字符串。
- 添加一个字符串到另一个字符串的结尾。
- 比较两个字符串。
- 定位一个字符串中的一个或多个字符。

所有的字符串函数都对 NULL 结尾的字符串操作。对无结尾的字符串，用前一节所说的缓冲区操作程序。

可变长度参数列表程序

程序	属性	说明
va_arg	可重入	从一个参数列表返回一个参数。
va_end	可重入	重设参数指针。
va_start	可重入	设置一个指针到一个参数列表的开头。

可变长度参数列表程序作为宏，在 **STDARG.H** 中定义。这些程序提供一个简单的方法访问一个可变参数的函数的参数。这些宏遵循 ANSI C 标准的可变长度参数列表。

其他程序

程序	属性	说明
setjmp	可重入	保存当前堆栈条件和编程地址。
longjmp	可重入	恢复堆栈条件和编程地址。
nop	固有的，可重入	插入一个 8051 NOP 指令。
testbit	固有的，可重入	测试一个位值并清 0。

其他类中的程序不能和别的库程序类合并。**setjmp** 和 **longjmp** 程序为函数，原型在 **STDJMP.H** 中。

_nop_和**_testbit_**程序直接编译，分别产生一个 **NOP** 指令和一个 **JBC** 指令。这些程序的原型在 **INTRINS.H** 中。

包含文件

Cx51 标准库提供的包含文件在 **INC** 子目录下。这些文件包含常数和宏定义，类型定义，和函数原型。下面各节说明每个包含文件的使用和内容。同时列出了文件中的宏和函数。

8051 特殊功能寄存器包含文件

Cx51 编译器包许多包含文件，定义许多 8051 派生系的特殊功能寄存器的明显常数。这些文件在目录 **KEIL\C51\INC** 和子目录下。例如，PHILIPS 80C554 的特殊功能寄存器（SFR）定义在文件 **KEIL\C51\INC\PHILIPS\REG554.H** 中。

在μVision2 编辑器内容菜单，点击鼠标右键可以打开一个编辑窗口，可以插入所先的器件的 SFR 定义。

可以从www.keil.com下载所有 8051 变种的 SFR 定义。网站的设备数据库包含差不多所有 8051 器件的特殊功能寄存器的头文件。

80C517.H

80C517.H 包含文件包括使用 80C517 CPU 和派生系的增强功能的程序。这些程序有：

acos517	**log10517**	**sqrt517**
asin517	**log517**	**sscan517**
atan517	**printf517**	**strtod517**
atof517	**scanf517**	**tan517**
cos517	**sin517**	
exp517	**sprintf517**	

ABSACC.H

ABSACC.H 文件包含允许直接访问 8051 不同存储区的宏定义。

CBYTE	**FARRAY**	**PBYTE**
CWORD	**FCARRAY**	**PWORD**
DBYTE	**FCVAR**	**XBYTE**
DWORD	**FVAR**	**XWORD**

ASSERT.H

ASSERT.H 文件定义 **assert** 宏，可以用来建立程序的测试条件。

CTYPE.H

CTYPE.H 文件包含对 ASCII 字符分类和字符转换的程序和原型。下面是程序的列表：

isalnum	**isprint**	**toint**
isalpha	**ispunct**	**tolower**
iscntrl	**isspace**	**_tolower**
isdigit	**isupper**	**toupper**
isgraph	**isxdigit**	**_toupper**
islower	**toascii**	

INTRINS.H

INTRINS.H 文件包含指示编译器产生嵌入固有代码的程序的原型。

chkfloat	**_irol_**	**_lror_**
crol	**_iror_**	**_nop_**
cror	**_lrol_**	**_testbit_**

MATH.H

MATH.H 文件包含所有浮点数运算的程序的原型和定义。别的数学函数也在这个文件中。所有的数学程序如下：

abs	**exp**	**modf**
acos	**fabs**	**pow**
asin	**floor**	**sin**
atan	**fmod**	**sinh**
atan2	**fprestore**	**sqrt**
cabs	**fpsave**	**tan**
ceil	**labs**	**tanh**
cos	**log**	
cosh	**log10**	

SETJMP.H

SETJMP.H 包含文件定义 **jmp_buf** 类型和 **setjmp** 和 **longjmp** 程序的原型。

STDARG.H

STDARG.H 包含文件定义允许访问可变长度参数列表的函数的参数的宏。包括：

va_arg	**va_end**	**va_start**

另外，**va_list** 类型也在文件中定义。

STDDEF.H

STDDEF.H 包含文件定义 **offsetof** 宏，可以用来确定结构成员的偏移。

STDIO.H

STDIO.H 文件包含流 I/O 程序的原型和定义。有：

getchar	**putchar**	**sscanf**
_getkey	**puts**	**ungetchar**
gets	**scanf**	**vprintf**
printf	**sprintf**	**vsprintf**

STDIO.H 包含文件也定义了 **EOF** 常数。

STDLIB.H

STDLIB.H 文件包含下面类型转换和存储区分配程序的原型和定义：

atof	**init_mempool**	**strtod**
atoi	**malloc**	**strtol**
atol	**rand**	**strtoul**
calloc	**realloc**	
free	**srand**	

STDLIB.H 包含文件也定义了 **NULL** 常数。

STRING.H

STRING.H 文件包含下面字符串和缓冲区操作程序的原型：

memccpy	**strchr**	**strncpy**
memchr	**strcmp**	**strpbrk**
memcmp	**strcpy**	**strpos**
memcpy	**strcspn**	**strrchr**
memmove	**strlen**	**strrpbrk**
memset	**strncat**	**strrpos**
strcat	**strncmp**	**strspn**

STRING.H 包含文件也定义了 **NULL** 常数。

参考

下面是 **Cx51** 标准库函数的参考。标准库中包含的程序按字母顺序排列，每个分成几部分：

摘要： 简要的说明程序作用，列出包含声明和原型的头文件，举例说明语法和参数。

说明： 提供程序的详细说明，和如何使用。

返回值： 程序的返回值。

参考： 相关程序的名称。

例子： 给出一个函数或程序段说明函数正确的使用方法。

abs

摘要： **#include<math.h>**
int abs(
int *val***);** **/*** 取绝对值 ***/**

说明： **abs** 函数取整数参数 *val* 的绝对值。

返回值： **abs** 函数返回 *val* 的绝对值。

参考： **cabs，fabs，labs**

例子：

```
#include <math.h>
#include <stdio.h>                          /*  对 printf */

void tst_abs(void) {
    int x;
    int y;

    x = -42;
    y = abs(x);
    pirntf("ABS(%d) = %d\n",x,y);
}
```

acos/acos517

摘要：

```
#include<math.h>
float acos(
    float x);                    /* 计算反余弦 */
```

说明： **acos** 函数计算浮点数 *x* 的反余弦。*x* 的值必须在-1 和 1 之间。**acos** 返回的浮点数值在 0 到π之间。

acos517 函数和 **acos** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

返回值： **acos** 函数返回 *x* 的反余弦。

参考： **asin，atan，atan2**

例子：

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_acos (void)  {
  float x;
  float y;

  for (x = -1.0; x <= 1.0; x += 0.1)  {
    y = acos (x);

    printf ("ACOS(%f) = %f\n", x, y);
  }
}
```

asin/asin517

摘要:

```
#include<math.h>
float asin(
    float x);                    /* 计算反正弦 */
```

说明: **asin** 函数计算浮点数 *x* 的反正弦。*x* 的值必须在-1 和 1 之间。**asin** 返回的浮点数值在-π/2 到π/2 之间。

asin517 函数和 **asin** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

返回值: **asin** 函数返回 *x* 的反正弦。

参考: **acos，atan，atan2**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_asin (void)  {
  float x;
  float y;

  for (x = -1.0; x <= 1.0; x += 0.1)  {
    y = asin (x);

    printf ("ASIN(%f) = %f\n", x, y);
  }
}
```

assert

摘要: **#include<assert.h>**
float assert(
*expression***);**

说明: **assert** 宏测试 *expression*，如果为假，则用 **printf** 打印一个详细诊断。

返回值: 无

例子:

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

void check_parms (
  char *string)
{
  assert (string != NULL);  /* check for NULL ptr */
  printf ("String %s is OK\n", string);
}
```

atan/atan517

摘要：

```
#include<math.h>
float atan(
    float x);                    /* 计算反正切 */
```

说明： **atan** 函数计算浮点数 *x* 的反正切。*x* 的值必须在-1 和 1 之间。**atan** 返回的浮点数值在-π/2 到π/2 之间。

atan517 函数和 **atan** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

返回值： **atan** 函数返回 *x* 的反正切。

参考： **acos，asin，atan2**

例子：

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_atan (void)  {
  float x;
  float y;

  for (x = -10.0; x <= 10.0; x += 0.1)  {
    y = atan (x);

    printf ("ATAN(%f) = %f\n", x, y);
  }
}
```

atan2

摘要：

```
#include<math.h>
float atan2(
    float y,                /* 分母的反正切 */
    float x);               /* 分子的反正切 */
```

说明： **atan2** 函数计算浮点数 *y*/*x* 的反正切。函数用 *x* 和 *y* 的符号来确定返回值的象限。**atan2** 返回的浮点数值在-π/2 到π/2 之间。

返回值： **atan2** 函数返回 *y*/*x* 的反正切。

参考： **acos，asin，atan**

例子：

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_atan2 ()  {
  float x;
  float y;
  float z;

  x = -1.0;

  for (y = -10.0; y < 10.0; y += 0.1)  {
    z = atan2 (y,x);

    printf ("ATAN2(%f/%f) = %f\n", y, x, z);
  }

  /* z approaches -pi as y goes from -10 to 0 */
  /* z approaches +pi as y goes from +10 to 0 */

}
```

atof/atof517

摘要：

```
#include<stdlib.h>
float atof(
    void *string);                    /* 字符串转换 */
```

说明： **atof** 函数转换 *string* 为一个浮点数。*string* 是一个字符序列，可以解释为一个浮点数。如果 *string* 的第一个字符不能转换成数字，就停止处理。

atof517 函数和 **atof** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

atof 函数要求 *string* 有下面的格式：
[{+|-}]*数字*[.*数字*][{e|E}[{+|-}]*数字*]
这里:
数字: 可能是一个或多个十进制数。

返回值： **atof** 函数返回 *string* 的浮点值。

参考： **atoi，atol，strtod，strtol，strtoul**

例子：

```
#include <stdlib.h>
#include <stdio.h>                    /* for printf */

void tst_atof (void)  {
  float f;
  char s [] = "1.23";

  f = atof (s);
  printf ("ATOF(%s) = %f\n", s, f);
}
```

atoi

摘要:

```
#include<stdlib.h>
int atoi(
    void *string);                    /* 字符串转换 */
```

说明: **atoi** 函数转换 *string* 为一个整数值。*string* 是一个字符序列，可以解释为一个整数。如果 *string* 的第一个字符不能转换成数字，就停止处理。

atoi 函数要求 *string* 有下面的格式:
[*空格*][{+|-}]*数字*
这里:
数字: 可能是一个或多个十进制数。

返回值: **atoi** 函数返回 *string* 的整数值。

参考: **atof，atol，strtod，strtol，strtoul**

例子:

```
#include <stdlib.h>
#include <stdio.h>                    /* for printf */

void tst_atoi (void)  {
  int i;
  char s [] = "12345";

  i = atoi (s);
  printf ("ATOI(%s) = %d\n", s, i);
}
```

atol

摘要: **#include<stdlib.h>**
long atol(
void ****string*);** /* 字符串转换 */

说明: **atol** 函数转换 *string* 为一个长整数值。*string* 是一个字符序列，可以解释为一个长整数。如果 *string* 的第一个字符不能转换成数字，就停止处理。

atol 函数要求 *string* 有下面的格式:
[*空格*][{+|-}]*数字*
这里:
数字: 可能是一个或多个十进制数。

返回值: **atol** 函数返回 *string* 的长整数值。

参考: **atof，atoi，strtod，strtol，strtoul**

例子:

```
#include <stdlib.h>
#include <stdio.h>                    /* for printf */

void tst_atol (void)  {
  long l;
  char s [] = "8003488051";

  l = atol (s);
  printf ("ATOL(%s) = %ld\n", s, l);

}
```

cabs

摘要: **#include<math.h>**
char cabs(
char *val***);** /* 字符取绝对值 */

说明: **cabs** 函数取 *val* 的绝对值。

返回值: **cabs** 返回 *val* 的绝对值。

参考: **abs，fabs，labs**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_cabs (void)  {
  char x;
  char y;

  x = -23;

  y = cabs (x);

  printf ("CABS(%bd) = %bd\n", x, y);
}
```

calloc

摘要：

```
#include<stdlib.h>
void *calloc(
    unsigned int num,                /* 元素数目 */
    unsigned int len);               /* 每个元素的长度 */
```

说明： **calloc** 函数从一个数组分配 *num* 个元素的存储区。每个元素占用 *len* 字节，并清 0。字节总数为 *num*×*len*。

注意：

*在 **LIB** 目录提供程序的源代码。可以修改源程序，为硬件定制本函数。参考 149 页的“第六章.高级编程技术”。*

返回值： **calloc** 函数返回一个指针，指向分配的存储区，如果不能分配，则返回一个 NULL 指针。

参考： **free，inti_mempool，malloc，realloc**

例子：

```
#include <stdlib.h>
#include <stdio.h>                   /* for printf */

void tst_calloc (void)  {
  int xdata *p;       /* ptr to array of 100 ints */

  p = calloc (100, sizeof (int));

  if (p == NULL)
  printf ("Error allocating array\n");
  else
    printf ("Array address is %p\n", (void *) p);

}
```

ceil

摘要: **#include<math.h>**
float ceil(
float *val*); /* 计算下限 */

说明: **ceil** 函数计算大于或等于 *val* 的最小整数值。

返回值: **ceil** 函数返回一个 **float**，包含不小于 *val* 的最小整数值。

参考: **floor**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_ceil (void)  {
  float x;
  float y;

  x = 45.998;
  y = ceil (x);

  printf ("CEIL(%f) = %f\n", x, y);

  /* output is "CEIL(45.998) = 46" */

}
```

chkfloat

摘要: **#include <intrins.h>**
unsigned char _chkfloat_(
float *val*); /* 错误检查 */

说明: **_chkfloat_**函数检查浮点数的状态。

返回值: **_chkfloat_**函数返回一个 **unsigned char** 值，包含下面的状态信息:

返回值	意义
0	标准浮点数
1	浮点数 0
2	**+INF**（正溢出）
3	**-INF**（负溢出）
4	**NaN**（不是一个数）错误状态

例子:

```
#include <intrins.h>
#include <stdio.h>                      /* for printf */

char _chkfloat_ (float);

float f1, f2, f3;

void tst_chkfloat (void)  {
  f1 = f2 * f3;

  switch (_chkfloat_ (f1))  {
    case 0:
      printf ("result is a number\n"); break;
    case 1:
      printf ("result is zero\n");     break;
    case 2:
      printf ("result is +INF\n");     break;
    case 3:
      printf ("result is -INF\n");     break;
    case 4:
      printf ("result is NaN\n");      break;
  }
}
```

cos/cos517

摘要: **#include <math.h>**

float cos(

float *x*); /* 计算余弦 */

说明: **cos** 函数计算浮点数 *x* 的余弦。*x* 的值必须在-65535 和 65535 之间。超出的结果是 **NaN** 错误。

cos517 函数和 **cos** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

返回值: **cos** 函数返回 *x* 的余弦。

参考: **sin**，**tan**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_cos (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = cos (x);

    printf ("COS(%f) = %f\n", x, y);
  }
}
```

cosh

摘要: **#include <math.h>**
float cosh(
float *x*); /* 双曲余弦函数 */

说明: **cosh** 函数计算浮点数 *x* 的双曲余弦。

返回值: **cosh** 函数返回 *x* 的双曲余弦。

参考: **sinh**，**tanh**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_cosh (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = cosh (x);

    printf ("COSH(%f) = %f\n", x, y);
  }
}
```

crol

摘要:

```
#include <intrins.h>
unsigned char _crol_(
    unsigned char c,          /* 循环左移的字符 */
    unsigned char b);         /* 左移的位数 */
```

说明: **_crol_**程序左移 *c* 字符 *b* 位。本程序是固有函数。代码要求内嵌，而不是调用。

返回值: **_crol_**程序返回结果 *c*。

参考: **_cror_，_irol_，_iror_，_lrol_，_lror_**

例子:

```
#include <intrins.h>

void tst_crol (void)  {
  char a;
  char b;

  a = 0xA5;

  b = _crol_(a,3);              /* b now is 0x2D */

}
```

cror

摘要: **#include <intrins.h>**

unsigned char _cror_(

unsigned char *c*, /* 循环右移的字符 */

unsigned char *b*); /* 右移的位数 */

说明: **_cror_**程序右移 *c* 字符 *b* 位。本程序是固有函数。代码要求内嵌，而不是调用。

返回值: **_cror_**程序返回结果 *c*。

参考: **_crol_，_irol_，_iror_，_lrol_，_lror_**

例子:

```
#include <intrins.h>

void tst_cror (void)  {
  char a;
  char b;

  a = 0xA5;

  b = _crol_(a,1);               /* b now is 0xD2 */

}
```

exp/exp517

摘要: **#include <math.h>**
float exp(

float *x*); /* e^x 函数的幂 */

说明: **exp** 函数计算浮点数 *x* 的指数函数。

exp517 函数和 **exp** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运算。当使用这些函数，应包含 80C517.H 头文件。不支持这特征的 CPU 不要使用这些程序。

返回值: **exp**函数返回浮点值 e^x 。

参考: **log,log10**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_exp (void)  {
  float x;
  float y;

  x = 4.605170186;

  y = exp (x);                           /* y = 100 */

  printf ("EXP(%f) = %f\n", x, y);

}
```

fabs

摘要: **#include<math.h>**
float fabs(
float *val***);** /* 计算绝对值 */

说明: **fabs**函数确定浮点数*val*的绝对值。

返回值: **fabs**返回*val*的绝对值。

参考: **abs,cabs,labs**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_fabs (void)  {
  float x;
  float y;

  x = 10.2;
  y = fabs (x);
  printf ("FABS(%f) = %f\n", x, y);

  x = -3.6;
  y = fabs (x);
  printf ("FABS(%f) = %f\n", x, y);

}
```

floor

摘要: **#include<math.h>**
float floor(
float *val*); /* floor函数值 */

说明: **floor**函数计算小于等于*val*的最大整数值。

返回值: **floor**函数返回不大于*val*的最大整数值的一个**float**数。

参考: **ceil**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_floor (void)  {
  float x;
  float y;

  x = 45.998;

  y = floor (x);

  printf ("FLOOR(%f) = %f\n", x, y);  * prints 45 */
}
```

fmod

摘要:

```
#include <math.h>
float fmod(
    float x,                /* 求模的值 */
    float y);               /* 模的整数值 */
```

说明: **fmod** 函数返回一个值 *f*，*f* 符号和 *x* 相同，*f* 的绝对值小于 *y* 的绝对值，同时有一个整数 *k*，*k*y+f=x*。如果不能表示 *x/y* 的商，结果是不确定的。

返回值: **fmode** 函数返回 *x/y* 的浮点余数。

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_fmod (void)  {
  float f;

  f = fmod (15.0, 4.0);
  printf ("fmod (15.0, 4.0) = %f\n", f);

}
```

free

摘要:

```
#include <stdlib.h>
void free(
    void xdata *p);                    /* 释放的块 */
```

说明: **free** 函数返回一个存储块到存储池。*p* 参数指向用 calloc，malloc，或 realloc 函数分配的存储块。一旦块返回到存储池，就可被再分配。

如果 *p* 是一个 NULL 指针，被忽略。

注意:
本程序的源代码在\KEIL\C51\LIB 目录中。可以修改源程序，根据硬件来定制本程序。参考 149 页的“第六章.高级编程技术”。

返回值: 无

参考: **calloc**，**init_mempool**，**malloc**，**realloc**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

void tst_free (void)  {
  void *mbuf;

  printf ("Allocating memory\n");
  mbuf = malloc (1000);

  if (mbuf == NULL)  {
    printf ("Unable to allocate memory\n");
  }
  else  {
      free (mbuf);
      printf ("Memory free\n");
  }

}
```

getchar

摘要: **#include <stdio.h>**
char getchar(void);

说明: **getchar** 函数用**_getkey** 函数从输入流读一个字符。所读的字符用 **putchar** 函数显示。

注意:
*本函数基于**_getkey** 和/或 **putchar** 函数的操作。这些函数，在标准库中提供，用 8051 的串口读和写字符。定制函数可以用别的 I/O 设备。*

返回值: **getchar** 返回所读的字符。

参考: **_getkey**，**putchar**，**ungetchar**

例子:

```
#include <stdio.h>

void tst_getchar (void)  {
  char c;

  while ((c = getchar ()) != 0x1B)  {
    printf ("character = %c %bu %bx\n", c, c, c);
  }

}
```

_getkey

摘要: **#include <stdio.h>**
char _getkey(void);

说明: **_getkey** 函数等待从串口接收字符。

注意:
*本函数指定运行，功能可能和上面所说的不一样。**_getkey** 和 **putchar** 函数的源代码可以修改，提供针对硬件的字符级的 I/O。参考 150 页的“定制文件”。*

返回值: **_getkey** 返回接收到的字符。

参考: **getchar**，**putchar**，**ungetchar**

例子:

```
#include <stdio.h>

void tst_getkey (void)  {
  char c;

  while ((c = _getkey ()) != 0x1B)  {
    printf ("key = %c %bu %bx\n", c, c, c);
  }

}
```

gets

摘要:

```
#include <stdio.h>
char *gets(
    char *string,          /* 要读的字符串 */
    int len);              /* 最多字符数 */
```

说明: **gets** 函数调用 **getchar** 函数读一行字符到 *string*。这行包括所有的字符和换行符（‘**\n**’）。在 *string* 中换行符被一个 NULL 字符（‘**\0**’）替代。

len 参数指定可读的最多字符数。如果长度超过 *len*，**gets** 函数用 NULL 字符终止 *string* 并返回。

注意:
本函数指定执行，基于 ***_getkey*** *和/或* ***putchar*** *函数的操作。这些函数，在标准库中提供，用 8051 的串口读写。对别的 I/O 设备可以定制。*

返回值: **gets** 函数返回 *string*。

参考: **printf**，**puts**，**scanf**

例子:

```
#include <stdio.h>

void tst_gets (void)  {
  xdata char buf [100];

  do  {
    gets (buf, sizeof (buf));
    printf ("Input string \"%s\"", buf);
  } while (buf [0] != '\0');
}
```

init_mempool

摘要:

```
#include<stdio.h>
void init_mempool(
    void xdata *p,              /* 存储池的开始 */
    unsigned int size);         /* 存储池的长度 */
```

说明: **init_mempool** 函数初始化存储管理程序，提供存储池的开始地址和大小。*p* 参数指向一个 **xdata** 的存储区，用 **calloc**，**free**，**malloc**，和 **realloc** 库函数管理。*size* 参数指定存储池所用的字节数。

注意:
*本函数必须在如何其他的存储管理函数（**calloc**，**free**，malloc，**realloc** ）被调用前设置存储池。只在程序的开头调用 **init_mempool** 一次。*

本程序的源代码在目录**\KEIL\C51\LIB** 中。可以修改源程序以适合硬件环境。参考 149 页的“第六章.高级编程技术”。

返回值: 无。

参考: **calloc**，**free**，**malloc**，**realloc**

例子:

```
#include <stdlib.h>

void tst_init_mempool (void)  {
  xdata void *p;
  int i;

  init_mempool (&XBYTE [0x2000], 0x1000);
/* initialize memory pool at xdata 0x2000
   for 4096 bytes */

  p = malloc (100);
  for (i = 0; i < 100; i++) ((char *) p)[i] = i;
  free (p);

}
```

irol

摘要: **#include <intrins.h>**
unsigned int _irol_(
unsigned int *i*, /* 左移的整数 */
unsigned char *b*); /* 移的位数 */

说明: **_irol** 程序循环左移整数 *i* *b* 位。本程序是一个固有函数。代码要求内嵌而不是被调用。

返回值: **_irol_**程序返回左移后的值 *i*。

参考: **_cror_**，**_crol_**，**_iror_**，**_lrol_**，**_lror_**

例子:

```
#include <intrins.h>

void tst_irol (void)  {
  int a;
  int b;

  a = 0xA5A5;

  b = _irol_(a,3);             /* b now is 0x2D2D */

}
```

iror

摘要:

```
#include <intrins.h>
unsigned int _iror_(
    unsigned int i,          /* 右移的整数 */
    unsigned char b);        /* 移的位数 */
```

说明: **_iror** 程序循环右移整数 *i* *b* 位。本程序是一个固有函数。代码要求内嵌而不是被调用。

返回值: **_iror_**程序返回右移后的值 *i*。

参考: **_cror_**，**_crol_**，**_irol_**，**_lrol_**，**_lror_**

例子:

```
#include <intrins.h>

void tst_iror (void)  {
  int a;
  int b;

  a = 0xA5A5;

  b = _irol_(a,1);            /* b now is 0xD2D2 */

}
```

isalnum

摘要:

```
#include<ctype.h>
bit isalnum(
    char c);                    /* 测试的字符 */
```

说明: **isalnum** 函数测试 *c*，确定是否是一个字母或数字字符（‘**A**’-‘**Z**’，‘**a**’-‘**z**’，‘**0**’-‘**9**’）。

返回值: 如果 *c* 是一个字母或数字字符，**isalnum** 函数返回 1，否则返回 0。

参考: **isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isalnum (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isalnum (i) ? "YES" : "NO");

    printf ("isalnum (%c) %s\n", i, p);
  }

}
```

isalpha

摘要:

```
#include<ctype.h>
bit isalpha(
    char c);                    /* 测试的字符 */
```

说明: **isalpha** 函数测试 *c*，确定是否是一个字母字符（‘**A**’-‘**Z**’或‘**a**’-‘**z**’）。

返回值: 如果 *c* 是一个字母字符，**isalpha** 函数返回 1，否则返回 0。

参考: **isalnum**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isalpha (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isalpha (i) ? "YES" : "NO");

    printf ("isalpha (%c) %s\n", i, p);
  }

}
```

iscntrl

摘要:

```
#include<ctype.h>
bit iscntrl(
    char c);                    /* 测试的字符 */
```

说明: **iscntrl** 函数测试 *c*，确定是否是一个控制字符（0x00-0x1F 或 0x7F）。

返回值: 如果 *c* 是一个控制字符，**iscntrl** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_iscntrl (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (iscntrl (i) ? "YES" : "NO");

    printf ("iscntrl (%c) %s\n", i, p);
  }

}
```

isdigit

摘要:

```
#include<ctype.h>
bit isdigit(
    char c);                    /* 测试的字符 */
```

说明: **isdigit** 函数测试 *c*，确定是否是一个十进制数（‘**0**’ - ‘**9**’）。

返回值: 如果 *c* 是一个十进制数，**isdigit** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isdigit (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isdigit (i) ? "YES" : "NO");

    printf ("isdigit (%c) %s\n", i, p);
  }

}
```

isgraph

摘要:

```
#include<ctype.h>
bit isgraph(
    char c);                    /* 测试的字符 */
```

说明: **isgraph** 函数测试 *c*，确定是否是一个可打印字符（不包括空格）。字符值在 0x21-0x7E 之间。

返回值: 如果 *c* 是一个可打印字符，**isgraph** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isgraph (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isgraph (i) ? "YES" : "NO");

    printf ("isgraph (%c) %s\n", i, p);
  }

}
```

islower

摘要:

```
#include<ctype.h>
bit islower(
    char c);                    /* 测试的字符 */
```

说明: **islower** 函数测试 *c*，确定是否是一个小写字母字符（‘**a**’ - ‘**z**’）。

返回值: 如果 *c* 是一个小写字母字符，**islower** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**isprint**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_islower (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (islower (i) ? "YES" : "NO");

    printf ("islower (%c) %s\n", i, p);
  }

}
```

isprint

摘要:

```
#include<ctype.h>
bit isprint(
    char c);                    /* 测试的字符 */
```

说明: **isprint** 函数测试 *c*，确定是否是一个可打印字符（0x20-0x7E）。

返回值: 如果 *c* 是一个可打印字符，**isprint** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**ispunct**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isprint (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isprint (i) ? "YES" : "NO");

    printf ("isprint (%c) %s\n", i, p);
  }

}
```

ispunct

摘要:

```
#include<ctype.h>
bit ispunct(
    char c);                    /* 测试的字符 */
```

说明: **ispunct** 函数测试 *c*，确定是否是一个标点符号字符，下面的符号是标点符号字符：

```
!   "   #   $   %   &   '   (
)   *   +   ′   -   .   /   :
;   <   =   >   ?   @   [   \
]   ^   _   `   {   |   }   ~
```

返回值: 如果 *c* 是一个标点符号字符，**ispunct** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**isspace**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_ispunct (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (ispunct (i) ? "YES" : "NO");

    printf ("ispunct (%c) %s\n", i, p);
  }

}
```

isspace

摘要:

```
#include<ctype.h>
bit isspace(
    char c);                    /* 测试的字符 */
```

说明: **isspace** 函数测试 *c*，确定是否是一个空白字符（0x09-0x0D 或 0x20）。

返回值: 如果 *c* 是一个空白字符，**isspace** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isupper**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isspace (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isspace (i) ? "YES" : "NO");

    printf ("isspace (%c) %s\n", i, p);
  }

}
```

isupper

摘要:

```
#include<ctype.h>
bit isupper(
    char c);                    /* 测试的字符 */
```

说明: **isupper** 函数测试 *c*，确定是否是一个大写字母字符（‘**A**’-‘**Z**’）。

返回值: 如果 *c* 是一个大写字母字符，**isupper** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isxditit**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isupper (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isupper (i) ? "YES" : "NO");

    printf ("isupper (%c) %s\n", i, p);
  }

}
```

isxdigit

摘要:

```
#include<ctype.h>
bit isxdigit(
    char c);                    /* 测试的字符 */
```

说明: **isxdigit** 函数测试 *c*，确定是否是一个十六进制数（‘**A**’ - ‘**Z**’，‘**a**’ - ‘**z**’，‘**0**’ - ‘**9**’）。

返回值: 如果 *c* 是一个十六进制数，**isxdigit** 函数返回 1，否则返回 0。

参考: **isalnum**，**isalpha**，**iscntrl**，**isdigit**，**isgraph**，**islower**，**isprint**，**ispunct**，**isspace**，**isupper**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_isxdigit (void)  {
  unsigned char i;
  char *p;

  for (i = 0; i < 128; i++)  {
    p = (isxdigit (i) ? "YES" : "NO");

    printf ("isxdigit (%c) %s\n", i, p);
  }

}
```

labs

摘要: **#include<math.h>**

long labs(

long *val*); /* 计算绝对值 */

说明: **labs** 函数确定长整数 *val* 的绝对值。

返回值: **labs** 函数返回 *val* 的绝对值。

参考: **abs**，**cabs**，**fabs**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_labs (void)  {
  long x;
  long y;

  x = -12345L;

  y = labs (x);

  printf ("LABS(%ld) = %ld\n", x, y);

}
```

log/log517

摘要:

```
#include<math.h>
float log(
    float val);                /* 计算自然对数 */
```

说明: **log** 函数计算浮点数 *val* 的自然对数。自然对数用基数 *e* 或 2.718282…

log517 函数和 **log** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

返回值: **log** 函数返回 *val* 的浮点自然对数。

参考: **exp**，**log10**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_log (void)  {
  float x;
  float y;

  x = 2.71838;
  x *= x;

  y = log (x);                              /* y = 2 */

  printf ("LOG(%f) = %f\n", x, y);

}
```

log10/log10517

摘要:

```
#include<math.h>
float log10(
    float val);                /* 计算常用对数 */
```

说明: **log10** 函数计算浮点数 *val* 的常用对数。自然对数用基数 10。

log10517 函数和 **log10** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

返回值: **log10** 函数返回 *val* 的浮点常用对数。

参考: **exp**，**log**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_log10 (void)  {
  float x;
  float y;

  x = 1000;

  y = log10 (x);                          /* y = 3 */

  printf ("LOG10(%f) = %f\n", x, y);

}
```

longjmp

摘要:

```
#include<setjmp.h>
void longjmp(
    jmp_buf env,          /* 恢复的环境 */
    int retval);          /* 返回值 */
```

说明: **longjmp** 函数恢复前面 **setjmp** 函数保存在 *env* 的状态。*retval* 参数指定从 **setjmp** 函数调用返回值。

longjmp 和 **setjmp** 函数可以用来执行非局部跳转，通常用来传递控制给一个错误恢复程序。

只有用 **volatile** 属性声明的局部变量和函数参数被恢复。

返回值: 无。

参考: **setjmp**

例子:

```
#include <setjmp.h>
#include <stdio.h>                  /* for printf */

jmp_buf env; /* jump environment (must be global) */
bit error_flag;


void trigger (void)  {
  .
  .
  .
  /* put processing code here */
  .
  .
  .
  if (error_flag != 0)  {
    longjmp (env, 1);       /* return 1 to setjmp */
  }
  .
  .
  .
}


void recover (void)  {
            /* put recovery code here */
}


void tst_longjmp (void)  {
  .
  .
  .
  if (setjmp (env) != 0)  { /* setjmp returns a 0 */
    printf ("LONGJMP called\n");
    recover ();
  }
  else  {
    printf ("SETJMP called\n");

    error_flag = 1;             /* force an error */

    trigger ();
  }
}
```

lrol

摘要:

```
#include <intrins.h>
unsigned long _lrol_(
    unsigned long l,          /* 32 位整数左移 */
    unsigned char b);         /* 移的位数 */
```

说明: **_lrol** 程序循环左移长整数 *l* *b* 位。本程序是一个固有函数。代码要求内嵌而不是被调用。

返回值: **_lrol_**程序返回左移后的值 *l*。

参考: **_cror_，_crol_，_irol_，_iror_，_lror_**

例子:

```
#include <intrins.h>

void tst_lrol (void)  {
  long a;
  long b;

  a = 0xA5A5A5A5;

  b = _lrol_(a,3);          /* b now is 0x2D2D2D2D */

}
```

lror

摘要:

```
#include <intrins.h>
unsigned long _lror_(
    unsigned long l,          /* 32 位整数右移 */
    unsigned char b);         /* 移的位数 */
```

说明: **_lror** 程序循环右移长整数 *l* *b* 位。本程序是一个固有函数。代码要求内嵌而不是被调用。

返回值: **_lror_**程序返回右移后的值 *l*。

参考: **_cror_，_crol_，_irol_，_iror_，_lrol_**

例子:

```
#include <intrins.h>

void tst_lror (void)  {
  long a;
  long b;

  a = 0xA5A5A5A5;

  b = _lrol_(a,1);         /* b now is 0xD2D2D2D2 */

}
```

malloc

摘要:

```
#include<stdlib.h>
void *malloc(
    unsigned int size);                    /* 分配的块大小 */
```

说明: **malloc** 函数从存储池分配 *size* 字节的存储块。

> ***注意:***
> *本程序的源代码在**\KEIL\C51\LIB** 目录中。可以根据硬件环境修改源文件。参考 149 页的“第六章.高级编程技术”。*

返回值: **malloc** 返回一个指向所分配的存储块的指针，如果没有足够的空间，则返回一个 NULL 指针。

参考: **calloc**，**free**，**init_mempool**，**realloc**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

void tst_malloc (void)  {
  unsigned char xdata *p;

  p = malloc (1000);       /* allocate 1000 bytes */

  if (p == NULL)
    printf ("Not enough memory space\n");
    else
      printf ("Memory allocated\n");

}
```

memccpy

摘要:

```
#include<string.h>
void *memccpy(
    void *dest,                 /* 目标缓冲区 */
    void *src,                  /* 源缓冲区 */
    char c,                     /* 结束拷贝的字符 */
    int len);                   /* 拷贝的最多字节数 */
```

说明: **memccpy** 函数从 *src* 到 *dest* 复制 0 或更多的字符，直到字符 *c* 被复制，或 *len* 字节被复制，那个条件先遇到就执行那个条件。

返回值: **memccpy** 函数返回一个指针，指向 *dest* 最后一个复制的字符的后一个字节，如果最后一个字符是 *c* 则返回一个 NULL 指针。

参考: **memchr，memcmp，memcpy，memmove，memset**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memccpy (void)  {
  static char src1 [100] = "Copy this string
                            to dst1";
  static char dst1 [100];

void *c;

  c = memccpy (dst1, src1, 'g', sizeof (dst1));

  if (c == NULL)
    printf ("'g' was not found in the src
             buffer\n");
    else
      printf ("characters copied up to 'g'\n");

}
```

memchr

摘要:

```
#include<string.h>
void *memchr(
    void *buf,          /* 搜索的缓冲区 */
    char c,             /* 查找的字节 */
    int len);           /* 最大缓冲区长度 */
```

说明: **memchr** 函数扫描 buf 的第一个 len 字节，查找字符 *c*。

返回值: **memchr** 函数返回字符 *c* 在 *buf* 中的指针，如没有则返回一个 NULL 指针。

参考: **memccpy**，**memcmp**，**memcpy**，**memmove**，**memset**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memchr (void)  {
  static char src1 [100] =
  "Search this string from the start";

  void *c;

  c = memchr (src1, 'g', sizeof (src1));

  if (c == NULL)
    printf ("'g' was not found in the buffer\n");
  else
    printf ("found 'g' in the buffer\n");

}
```

memcmp

摘要:

```
#include<string.h>
char memcmp(
    void *buf1,          /* 第一个缓冲区 */
    void *buf2,          /* 第二个缓冲区 */
    int len);            /* 比较的最大字节数 */
```

说明: **memcmp** 函数比较两个缓冲区 *buf1* 和 *buf2* *len* 字节，并返回一个值表示关系如下：

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

返回值: **memcmp** 函数返回一个正，负，或零值表示 *buf1* 和 *buf2* 的关系。

参考: **memccpy，memchr，memcpy，memmove，memset**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memcmp (void)  {
  static char hexchars  [] = "0123456789ABCDEF";
  static char hexchars2 [] = "0123456789abcdef";

  char i;

  i = memcmp (hexchars, hexchars2, 16);

  if (i < 0)
    printf ("hexchars < hexchars2\n");

  else if (i > 0)
    printf ("hexchars > hexchars2\n");

  else
    printf ("hexchars == hexchars2\n");

}
```

memcpy

摘要:

```
#include<string.h>
void *memcpy(
    void *dest,                 /* 目标缓冲区 */
    void *src,                  /* 源缓冲区 */
    int len);                   /* 拷贝的最多字节数 */
```

说明: **memcpy** 函数从 *src* 到 *dest* 复制 *len* 字节。如果存储缓冲区重叠，**memcpy** 函数不能保证 *src* 中的那个字节在被覆盖前复制到 *dest*。如果缓冲区重叠，用 **memmove** 函数。

返回值: **memcpy** 函数 *dest*。

参考: **memccpy**，**memchr**，**memcmp**，**memmove**，**memset**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memcpy (void)  {
  static char src1 [100] =
  "Copy this string to dst1";

  static char dst1 [100];

  char *p;

  p = memcpy (dst1, src1, sizeof (dst1));

  printf ("dst = \"%s\"\n", p);

}
```

memmove

摘要:

```
#include<string.h>
void *memmove(
    void *dest,                 /* 目标缓冲区 */
    void *src,                  /* 源缓冲区 */
    int len);                   /* 移动的最多字节数 */
```

说明: **memmove** 函数从 *src* 到 *dest* 复制 *len* 字节。如果存储缓冲区重叠，**memmove** 函数保证 *src* 中的那个字节在被覆盖前复制到 *dest*。

返回值: **memmove** 函数返回 *dest*。

参考: **memccpy**，**memchr**，**memcmp**，**memcpy**，**memset**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memmove (void)  {
  static char buf [] = "This is line 1  "
                       "This is line 2  "
                       "This is line 3  ";

  printf ("buf before = %s\n", buf);

  memmove (&buf [0], &buf [16], 32);

  printf ("buf after  = %s\n", buf);

}
```

memset

摘要:

```
#include<string.h>
void *memset(
    void *buf,                  /*  初始化的缓冲区  */
    char c,                     /*  设置的值  */
    int len);                   /*  缓冲区长度  */
```

说明: **memset** 函数设置 *buf* 的第一个 *len* 字节为 c。

返回值: **memset** 函数返回 *dest*。

参考: **memccpy**，**memchr**，**memcmp**，**memcpy**，**memmove**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_memset (void)  {
  char buf [10];

  memset (buf, '\0', sizeof (buf));
              /* fill buffer with null characters */

}
```

modf

摘要:

```
#include<math.h>
float modf(
    float val,                  /* 求模的值 */
    float *ip);                 /* 模的整数部分 */
```

说明: **modf** 函数把浮点数 *val* 分成整数和小数部分。*val* 的小数部分为一个带符号的浮点数。整数部分保存在浮点数 *ip* 中。

返回值: **modf** 函数返回带符号小数部分 *val*。

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_modf (void)  {
  float x;
  float int_part, frc_part;

  x = 123.456;

  frc_part = modf (x, &int_part);

  printf ("%f = %f + %f\n", x, int_part,frc_part);

}
```

nop

摘要: **#include<intrins.h>**
void _nop_(void);

说明: **_nop_**程序插入一个 8051 NOP 指令到程序。本程序可以用来停顿 1 个 CPU 周期。本程序是一个固有函数。代码要求内嵌而不是调用。

返回值: 无。

例子:

```
#include <intrins.h>
#include <stdio.h>                    /* for printf */

void tst_nop (void)  {

  P1 = 0xFF;

  _nop_ ();                  /* delay for hardware */
  _nop_ ();
  _nop_ ();

  P1 = 0x00;

}
```

offsetof

摘要:

```
#include<stddef.h>
int offsetof(
    structure,          /* 使用的结构 */
    member);            /* 取偏移的成员 */
```

说明: **offsetof** 宏计算结构元素 *member* 从结构开头的偏移。*structure* 参数必须指定一个结构的名称。*member* 参数必须指定结构的成员的名称。

返回值: **offsetof** 宏返回 *member* 元素从 **struct** *structure* 开头的偏移的字节数。

例子:

```
#include <stddef.h>

struct index_st
  {
  unsigned char type;
  unsigned long num;
  unsigned ing len;
  };

typedef struct index_st index_t;

void main (void)
{
int x, y;

x = offsetof (struct index_st, len);  /* x = 5 */
y = offsetof (index_t, num);          /* x = 1 */
}
```

pow

摘要:

```
#include<math.h>
float pow(
    float x,          /* 基数 */
    float y);         /* 指数 */
```

说明: **pow** 函数计算 *x* 的 *y* 次幂。

返回值: **pow** 函数返回值 x^y。如果 $x \neq 0$ 和 $y = 0$，**pow** 返回值 1。如果 $x = 0$ 和 $y \leq 0$，**pow** 返回 **NaN**。如果 $x < 0$ 和 *y* 不是一个整数，**pow** 返回 **NaN**。

参考: **sqrt**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_pow (void)  {
  float base;
  float power;
  float y;

  base  = 2.0;
  power = 8.0;

  y = pow (base, power);               /* y = 256 */

  printf ("%f ^ %f = %f\n", base, power, y);

}
```

printf/printf517

摘要:

```
#include <stdio.h>
int printf(
    const char *fmtstr              /* 格式化字符串 */
    [,arguments]..);                /* 附加的参数 */
```

说明:

printf 函数格式化一系列的字符串和数值，生成一个字符串用 **putchar** 写到输出流。*fmtstr* 参数是一个格式化字符串，可能是字符，转义系列，和格式标识符。

普通的字符和转义系列按说明的顺序复制到流。格式标识符通常以百分号（‘%’）开头，要求在函数调用中包含附加的参数 *arguments*。

格式字符串从左向右读。第一个格式标识符使用 *fmtstr* 后的第一个参数，用格式标识符转换和输出。第二个格式标识符访问 *fmtstr* 后的第二个参数，等等。如果参数比格式标识符多，多出的参数被忽略。如果参数不够，结果是不可预料的。

格式标识符用下面的格式:

%[*flags*][*width*][.*precision*][{**b**|**B**|**l**|**L**}]*type*

格式标识符中的每个域可以是一个字符或数字，指定特殊的格式选项。

type 域是一个字符，指定参数是否解释为一个字符，字符串，数字，或指针，如下表所示。

字符	参数类型	输出格式
d	**int**	带符号十进制数
u	**unsigned int**	不带符号十进制数
o	**unsigned int**	不带符号八进制数
x	**unsigned int**	不带符号十六进制数，用“0123456789abcdef”
X	**unsigned int**	不带符号十六进制数，用“0123456789ABCDEF”
f	**float**	浮点数用格式[-]*dddd.dddd*
e	**float**	浮点数用格式[-]*d.dddde*[-]*dd*
E	**float**	浮点数用格式[-]*d.ddddE*[-]*dd*
g	**float**	浮点数用 e 或 f 格式，无论那个对指定的值或精度更简洁。
G	**float**	和 g 格式一样，除了（可能）指数前为 E 而不是 e
c	**char**	单个字符
s	**通用 ***	用 NULL 字符结尾的字符串
p	**通用 ***	指针，用 *t:aaaa* 格式，这里 *t* 是指针索引的存储类型（c:code，i:data/idata，x:xdata，p:pdata），*aaaa* 是十六进制地址

可选的字符 **b** 或 **B** 和 **l** 和 **L** 可直接放在类型字符前，分别指定整数类型 **d**，**i**，**u**，**o**，**x**，和 **X** 的 **char** 或 **long** 版本。

flags 域是单个字符，用来对齐输出，和打印+/-号和空白，小数点，和八进制和十六进制的前缀，如下表所示。

标记	意义
-	对指定的域宽度，左对齐输出。
+	如果输出是一个带符号类型，用+或-符号为输出值的前缀。
空白（‘ ’）	如果是一个带符号正值，输出值的前缀是空格。否则没有空格。
#	当用 o，x，和 X 域类型时，对非零输出值分别用 0，0x，或 0X 为前缀。 当用 e，E，f，g，和 G 域类型时，#标记强迫输出值包含一个小数点。 在其他的情况#标记被忽略。
*	忽略格式标识符。

width 域时一个非负数字，指定显示的最小字符数。如果输出值的字符数小于 *width*，空白会加到左边或右边（当指定了-标记）以达到最小的宽度。如果 *width* 用一个‘**0**’前缀，则填充的是零而不是空白。*width* 域不会截短一个域。如果输出枝的长度超过指定宽度，则输出所有的字符。

width 域可能是一个星号（‘*’），在这种情况，参数列表的一个 **int** 参数提供宽度值。如果参数使用的是一个 **unsigned char**，在星号标识符前指定一个‘**b**’。

precision 域是一个非负数字，指定显示的字符数，小数位数，或有效位。*precison* 域可能使输出值切断或舍入，在一个浮点数入下表所指定。

类型	意义
d,u,o,x,X	*precison* 域用来指定输出值的数字的最小数目。如果参数中的数字数目超过 *precision* 域定义的，数字就不会被切断。如果参数的数字的数目小于 *precison* 域，输出值在左边用零填充。
f	*precision* 域用来指定小数点后面的数字的位数。最后一位舍入。
e,E	*precision* 域用来指定小数点后面的数字的位数。最后一位舍入。
g,G	*precision* 域用来指定输出值的有效位的最大数目。
c,p	*precision* 域无效。
s	*precision* 域指定输出值的最多字符数。超过的不输出。

precision 域可能是一个星号（‘*’），在这种情况，参数列表的一个 **int** 参数提供宽度值。如果参数使用的是一个 **unsigned char**，在星号标识符前指定一个‘**b**’。

printf517 函数和 **printf** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

注意:

本函数指定执行，基于 ***putchar*** *函数的操作。本函数，作为标准库提供，用 8051 的串口写字符。用别的 I/O 设备可以定制函数。*

必须确保参数类型和指定的格式匹配。可用类型影射确保正确的类型传递到 ***printf****。*

可传递给 ***printf*** *的总的字节数受到 8051 的存储区的限制。****SMALL*** *模式或* ***COMPACT*** *模式最多 15 字节，****LARGE*** *模式最多 40 字节。*

返回值: **printf** 函数返回实际写到输出流的字符数。

参考: **gets,puts,scanf,sprintf,sscanf,vprintf,vsprintf**

例子:

```
#include <stdio.h>

void tst_printf (void)  {
  char a;
  int  b;
  long c;
  unsigned char x;
  unsigned int  y;
  unsigned long z;
  float f,g;
  char buf [] = "Test String";
  char *p = buf;

  a = 1;
  b = 12365;
  c = 0x7FFFFFFF;
  x = 'A';
  y = 54321;
  z = 0x4A6F6E00;
  f = 10.0;
  g = 22.95;

  printf ("char %bd  int %d  long %ld\n",a,b,c);
  printf ("Uchar %bu Uint %u Ulong %lu\n",x,y,z);
  printf ("xchar %bx xint %x xlong %lx\n",x,y,z);
  printf ("String %s is at address %p\n",buf,p);
  printf ("%f != %g\n", f, g);
  printf ("%*f != %*g\n", 8, f, 8, g);
}
```

putchar

摘要:

```
#include<stdio.h>
char putchr(
    char c);                    /* 输出的字符 */
```

说明: **putchar** 函数用 8051 的串口输出字符 *c*。

> ***注意:***
> *本程序指定执行，功能可能和上面所说的有所区别。提供了 _getkey 和 putchar 函数的源程序，可以根据任何硬件环境修改以提供字符级的 I/O。参考 150 页的“定制文件”。*

返回值: **putchar** 函数返回输出的字符 *c*。

参考: **getchar,_getkey,ungetchar**

例子:

```
#include <stdio.h>

void tst_putchar (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)
    putchar (i);
}
```

puts

摘要:

```
#include<stdio.h>
int puts(
    const char *string);              /* 输出的字符串 */
```

说明: **puts** 函数用 putchar 函数写 *string* 和一个换行符（‘**\n**’）到输出流。

注意:
本函数指定执行，基于 ***putchar*** *函数的操作。本函数，作为标准库提供，写字符到 8051 的串口。用别的 I/O 口可以定制函数。*

返回值: 如果重新错误，**puts** 函数返回 **EOF**；如果没有则返回一个 0。

参考: **gets,printf,scanf**

例子:

```
#include <stdio.h>

void tst_puts (void)  {

  puts ("Line #1");
  puts ("Line #2");
  puts ("Line #3");

}
```

rand

摘要: **#include<stdlib.h>**
int rand(void);

说明: **rand** 函数产生一个 0 到 32767 之间的虚拟随机数。

返回值: **rand** 函数返回一个虚拟随机数。

参考: **srand**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

void tst_rand (void)  {
  int i;
  int r;

  for (i = 0; i < 10; i++)  {
    printf ("I = %d, RAND = %d\n", i, rand ());
  }

}
```

realloc

摘要:

```
#include<stdlib.h>
void *realloc(
    void xdata *p,          /* 前面已分配的块 */
    unsigned int size);     /* 新块的大小 */
```

说明: **realloc** 函数改变前面已分配的存储块的大小。*p* 参数指向已分配块，*size* 参数指定新块的大小。原块的内容复制到新块。新块中的任何其他区，如果是一个更大的块，不初始化。

注意:
*本程序的源代码在目录**\KEIL\C51\LIB** 中。可以根据硬件环境定制本函数。参考 149 页的“第六章.高级编程技术”。*

返回值: **realloc** 返回一个指向新块的指针。如果存储池没有足够的存储区，返回一个 NULL 指针，用来的存储块不受影响。

参考: **calloc,free,init_mempool,malloc**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

void tst_realloc (void)  {
  void xdata *p;
  void xdata *new_p;

  p = malloc (100);
  if (p != NULL)  {
    new_p = realloc (p, 200);

    if (new_p != NULL)  p = new_p;
    else  printf ("Reallocation failed\n");
  }
}
```

scanf

摘要:

```
#include<stdio.h>
int scanf(
    const char *fmtstr          /* 格式字符串 */
    [,argument]…);              /* 附加参数 */
```

说明:

scanf 函数用 **getchar** 程序读数据。输入的数据保存在由 *argument* 根据格式字符串 fmtstr 指定的位置。每个 *argument* 必须是一个指针，指向一个变量，对应 *fmtstr* 定义的类型，*fmtstr* 控制解释输入的数据。*fmtstr* 参数由一个或单个空白字符，非空白字符，和下面定义的格式标识符组成。

scanff517 函数和 **scanf** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

- 空白字符，空白（‘ ’），制表（‘**\t**’），或换行（‘**\n**’），使 **scanf** 跳过输入流中的空白字符。格式字符串中的单个的空白字符匹配输入流的 0 或多个空白字符。

- 非空白字符，除了百分号（‘**%**’），使 **scanf** 从输入流读但不保存一个匹配字符。如果输入流的下一个字符和指定的非空白字符不匹配，**scanf** 函数终止。

- 格式标识符以百分号（‘**%**’）开头，使 **scanf** 从输入流读字符，并转换字符到指定的类型值。转换后的值保存在参数列表的 *argument* 中。百分号后面的字符不被认为是一个格式标识符，只作为一个普通字符。例如，**%%**匹配输入流的一个百分号。

格式字符串从左向右读。不是格式标识符的字符必须和输入流的字符匹配。这些字符从输入流读入，但不保存。如果输入流的一个字符和格式字符串冲突，**scanf** 终止。任何冲突的字符仍保留在输入流中。

在格式字符串中的第一个格式标识符引用 *fmtstr* 后面的第一个参数，并转化输入字符，用格式标识符保存值。第二个格式标识符访问 *fmtstr* 后面的第二个参数，等等。如果参数比格式标识符多，多出的参数被忽略。如果没有足够的参数匹配格式标识符，结果是不可预料的。

输入流中的值被输入域调用，用空白字符隔开。在转换输入域时，**scanf** 遇到一个空白字符就结束一个参数的转换。而且，任何当前格式标识符不认识的字符会结束一个域转换。

格式标识符的格式：

%[*****][*width*][{**b**|**h**|**l**}]*type*

格式标识符中的每个域可以是单个字符或数字，用来指定一个特殊的格式选项。

type 域是单个字符，指定输入字符是否解释为一个字符，字符串，或数字。本域可以是下表中的任何值。

字符	参数类型	输入格式
d	**int ***	带符号十进制数
l	**int ***	带符号十进制，十六进制，或八进制整数
u	**unsigned int ***	不带符号十进制数

字符	参数类型	输入格式
o	**unsigned int ***	不带符号八进制数
x	**unsigned int ***	不带符号十六进制数
e	**float ***	浮点数
f	**float ***	浮点数
g	**float ***	浮点数
c	**char ***	单个字符
s	**char ***	一个字符串，以空白结尾

以星号（*）作为格式标识符的第一个字符，会使输入域被扫描但不保存。星号禁止和格式标识符关联。

width 域是一个非负数，指定从输入流读入的最多字符数。从输入流读入的字符不超过 *width*，并根据相应的 *argument* 转换。然而，如果一个空白字符或一个不认识字符先遇到，则读入的字符数小于 *width*。

可选字符 **b**，**h**，和 **l** 可以直接放在类型字符前面，分别指定整数类型 **d**，**i**，**u**，**o**，和 **x** 的 **char**，**short**，或 **long** 版本。

注意:
本函数指定执行，基于 ***_getkey*** *和/或* ***putchar*** *函数的操作。这些函数，作为标准库提供，用 8051 的串口读写。可对别的 I/O 设备定制函数。*

*可以传递给 scanf 的字节数受 8051 存储区的限制。****SMALL*** *模式或* ***COMPACT*** *模式最多为 15 字节。****LARGE*** *模式最多为 40 字节。*

返回值: **scanf** 函数返回成功转换的输入域的数目。如果有错误则返回一个 **EOF**。

参考: **gets,printf,puts,sprintf,sscanf,vprintf,fsprintf**

例子:

```
#include <stdio.h>

void tst_scanf (void)  {
  char a;
  int  b;
  long c;

  unsigned char x;
  unsigned int  y;
  unsigned long z;

  float f,g;

  char d, buf [10];

  int argsread;

  printf ("Enter a signed byte, int, and long\n");
  argsread = scanf ("%bd %d %ld", &a, &b, &c);
  printf ("%d arguments read\n", argsread);

  printf ("Enter an unsigned byte, int, and long\n");
  argsread = scanf ("%bu %u %lu", &x, &y, &z);
  printf ("%d arguments read\n", argsread);

  printf ("Enter a character and a string\n");
  argsread = scanf ("%c %9s", &d, buf);
  printf ("%d arguments read\n", argsread);

  printf ("Enter two floating-point numbers\n");
  argsread = scanf ("%f %f", &f, &g);
  printf ("%d arguments read\n", argsread);

}
```

setjmp

摘要:

```
#include<setjmp.h>
int setjmp(
 jmp_buf env);                    /* 当前环境 */
```

说明: **setjmp** 函数保存当前 CPU 的状态在 *env*。状态可以调用并发 **longjmp** 函数来恢复。当同时使用时，**setjmp** 和 **longjmp** 函数提供一种方法实行非局部跳转。

setjmp 函数保存当前指令地址和别的 CPU 寄存器。一个 **longjmp** 的并发调用恢复指令指针和寄存器，在 **setjmp** 调用后面恢复运行。

只有声明了 **volatile** 属性的局部变量和函数参数被恢复。

返回值: 当 CPU 的当前状态被复制到 *env*，**setjmp** 函数返回一个 0。一个非零值表示执行了 **longjmp** 函数来返回 **setjmp** 函数调用。在这种情况，返回值是传递给 **longjmp** 函数的值。

参考: **longjmp**

例子: 看 **longjmp**

sin/sin517

摘要:

```
#include<math.h>
float sin(
    float x);                    /* 计算正弦 */
```

说明: **sin** 函数计算浮点数 *x* 的正弦。值 *x* 必须在-65535 到+65535 之间，或产生一个 **NaN** 错误。

sin517 函数和 **sin** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

返回值: **sin** 函数返回 *x* 的正弦。

参考: **cos**，**tan**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_sin (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = sin (x);

    printf ("SIN(%f) = %f\n", x, y);
  }

}
```

sinh

摘要:

```
#include<math.h>
float sinh(
    float x);                    /*  计算正弦  */
```

说明: **sinh** 函数计算浮点数 *x* 的双曲正弦。值 *x* 必须在-65535 到+65535 之间，或产生一个 **NaN** 错误。

返回值: **sinh** 函数返回 *x* 的双曲正弦。

参考: **cosh**，**tanh**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_sinh (void)  {
  float x;
  float y;

  for (x = 0; x < (2 * 3.1415); x += 0.1)  {
    y = sinh (x);
    printf ("SINH(%f) = %f\n", x, y);
  }

}
```

sprintf/sprintf517

摘要:

```
#include <stdio.h>
int sprintf(
    char *buffer,                /* 保存缓冲区 */
    const char *fmtstr           /* 格式化字符串 */
    [,arguments]…);              /* 附加的参数 */
```

说明: **sprintf** 函数格式化一系列的字符串和数值，并保存结果字符串在 *buffer*。*fmtstr* 参数是一个格式字符串，和 **printf** 函数指定的要求相同。参考 290 页的“printf/printf517”。

sprintf517 函数和 **sprintf** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

注意:
*可传递给 **sprintf** 的总的字节数受到 8051 的存储区的限制。**SMALL** 模式或 **COMPACT** 模式最多 15 字节，**LARGE** 模式最多 40 字节。*

返回值: **sprintf** 函数返回实际写到 *buffer* 的字符数。

参考: **gets,printf,puts,scanf,sscanf,vprintf,vsprintf**

例子：

```
#include <stdio.h>

void tst_sprintf (void)  {
  char buf [100];
  int n;

  int a,b;
  float pi;

  a = 123;
  b = 456;
  pi = 3.14159;

  n =  sprintf (buf,   "%f\n", 1.1);
  n += sprintf (buf+n, "%d\n", a);
  n += sprintf (buf+n, "%d %s %g", b, "---", pi);
  printf (buf);
}
```

sqrt/sqrt517

摘要:

```
#include<math.h>
float sqrt(
    float x);                    /* 计算平方根 */
```

说明: **sqrt** 函数计算 *x* 的平方根。

sqrt517 函数和 **sqrt** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

返回值: **sqrt** 函数返回 *x* 的正平方根。

参考: **exp,log,pow**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_sqrt (void)  {
  float x;
  float y;

  x = 25.0;

  y = sqrt (x);                          /* y = 5 */

  printf ("SQRT(%f) = %f\n", x, y);

}
```

srand

摘要:

```
#include<stdlib.h>
void srand(
    int seed);                    /* 随机数发生器初始值 */
```

说明: **srand** 函数设置 **rand** 函数所用的虚拟随机数发生器的起始值 *seed*。随机数发生器对任何确定值 *seed* 产生相同的虚拟随机数序列。

返回值: 无。

参考: **rand**

例子:

```
#include <stdlib.h>
#include <stdio.h>                    /* for printf */

void tst_srand (void)  {
  int i;
  int r;

  srand (56);

  for (i = 0; i < 10; i++)  {
    printf ("I = %d, RAND = %d\n", i, rand ());
  }

}
```

sscanf/sscanf517

摘要:

```
#include<stdio.h>
int sscanf(
    char *buffer,          /* 扫描输入缓冲区 */
    const char *fmtstr     /* 格式字符串 */
    [,argument]…);         /* 附加参数 */
```

说明:

sscanf 函数从 *buffer* 读字符串。输入的数据保存在由 *argument* 根据格式字符串 fmtstr 指定的位置。每个 *argument* 必须是一个指向变量的指针，对应定义在 *fmtstr* 的类型，控制输入数据的解释。*fmtstr* 参数由一个或多个空白字符，非空白字符，和格式标识符组成，如同 **scanf** 函数所定义。格式字符串和附加参数的完整的说明参考 300 页的“scanf”。

sscanf517 函数和 **sscanf** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

注意:

*可传递给 **sscanf** 的总的字节数受到 8051 的存储区的限制。**SMALL** 模式或 **COMPACT** 模式最多 15 字节，**LARGE** 模式最多 40 字节。*

返回值:

sscanf 函数返回成功转换的输入域的数目。如果出现错误则返回一个 **EOF**。

参考:

gets,printf,puts,scanf,sprintf,vprintf,vsprintf

例子:

```
#include <stdio.h>

void tst_sscanf (void)  {
  char a;
  int  b;
  long c;

  unsigned char x;
  unsigned int  y;
  unsigned long z;

  float f,g;

  char d, buf [10];

  int argsread;

  printf ("Reading a signed byte, int,and long\n");
  argsread = sscanf ("1 -234 567890",
                     "%bd %d %ld", &a, &b, &c);
  printf ("%d arguments read\n", argsread);

  printf ("Reading an unsigned byte, int, and long\n");
  argsread = sscanf ("2 44 98765432",
                      "%bu %u %lu", &x, &y, &z);
  printf ("%d arguments read\n", argsread);

  printf ("Reading a character and a string\n");
  argsread = sscanf ("a abcdefg", "%c %9s", &d, buf);
  printf ("%d arguments read\n", argsread);

  printf ("Reading two floating-point numbers\n");
  argsread = sscanf ("12.5 25.0", "%f %f", &f, &g);
  printf ("%d arguments read\n", argsread);

}
```

strcat

摘要:

```
#include<string.h>
char *strcat(
    char *dest,          /* 目标字符串 */
    char *src);          /* 源字符串 */
```

说明: **strcat** 函数连接或添加 *src* 到 *dest*，并用一个 NULL 字符终止 *dest*。

返回值: **strcat** 函数返回 *dest*。

参考: **strcpy,strlen,strncat,strncpy**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strcat (void)  {
  char buf [21];
  char s [] = "Test String";

  strcpy (buf, s);
  strcat (buf, " #2");

  printf ("new string is %s\n", buf);

}
```

strchr

摘要:

```
#include<string.h>
char *strchr(
    const char *string,        /* 搜索的字符串 */
    char c);                   /* 搜索的字符 */
```

说明: **strchr** 函数搜索 *string* 中第一个出现的 *c*。*string* 中的 NULL 字符终止搜索。

返回值: **strchr** 函数返回 *string* 中指向 *c* 的一个指针，如没有发现则返回一个 NULL 指针。

参考: **strcspn,strpbrk,strpos,strrchr,strrpbrk,strrpos,strspn,strstr**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strchr (void)  {
  char *s;
  char buf [] = "This is a test";

  s = strchr (buf, 't');

  if (s != NULL)
    printf ("found a 't' at %s\n", s);
}
```

strcmp

摘要:

```
#include<string.h>
char strcmp(
    char *string1,          /* 第一个字符串 */
    char *string2);         /* 第二个字符串 */
```

说明: **strcmp** 函数比较 *string1* 和 *string2* 的内容，并返回一个值表示他们的关系。

返回值: **strcmp** 函数返回一个下面的值表示 *string1* 和 *string2* 的关系：

值	意义
<0	*string1* 小于 *string2*
=0	*string1* 等于 *string2*
>0	*string1* 大于 *string2*

参考: **memcmp,strncmp**

例子:

```
#include <string.h>
#include <stdio.h>                    /* for printf */

void tst_strcmp (void)  {
  char buf1 [] = "Bill Smith";
  char buf2 [] = "Bill Smithy";
  char i;

  i = strcmp (buf1, buf2);

  if (i < 0)
    printf ("buf1 < buf2\n");

  else if (i > 0)
    printf ("buf1 > buf2\n");

  else
    printf ("buf1 == buf2\n");
}
```

strcpy

摘要:

```
#include<string.h>
char *strcpy(
     char *dest,          /* 目标字符串 */
     char *src);          /* 源字符串 */
```

说明: **strcpy** 函数复制 *src* 到 *dest*，并用 NULL 字符结束 *dest*。

返回值: **strcpy** 函数返回 *dest*。

参考: **strcat,strlen,strncat,strncpy**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strcpy (void)  {
  char buf [21];
  char s [] = "Test String";

  strcpy (buf, s);
  strcat (buf, " #2");

  printf ("new string is %s\n", buf);
}
```

strcspn

摘要:

```
#include<string.h>
int strcspn(
    char *src,                    /*  源字符串  */
    char *set);                   /*  查找的字符  */
```

说明: **strcspn** 函数在 *src* 字符串中查找 *set* 字符串中的任何字符。

返回值: **strcspn** 函数返回 *src* 中和 *set* 匹配的第一个字符的索引。如果 *src* 的第一个字符和 *set* 中的一个字符匹配，返回 0。如果 *src* 中没有字符匹配，返回字符串的长度。

参考: **strchr,strpbrk,strpos,strrchr,strrpbrk,strrpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strcspn (void)  {
  char buf [] = "13254.7980";
  int i;

  i = strcspn (buf, ".,");

  if (buf [i] != '\0')
    printf ("%c was found in %s\n", (char)
    buf [i], buf);
}
```

strlen

摘要: **#include<string.h>**
int strlen(
char **src*); /* 源字符串 */

说明: **strlen** 函数计算 *src* 的字节数。不包括 NULL 结束符。

返回值: **strlen** 函数返回 *src* 的长度。

参考: **strcat,strcpy,strncat,strncpy**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strlen (void)  {
  char buf [] = "Find the length of this string";
  int len;

  len = strlen (buf);

  printf ("string length is %d\n", len);

}
```

strncat

摘要:

```
#include<string.h>
char *strncat(
    char *dest,          /* 目标字符串 */
    char *src,           /* 源字符串 */
    int len);            /* 连接的最多字符数 */
```

说明: **strncat** 函数从 *src* 添加最多 *len* 个字符到 *dest*，并用 NULL 结束。如果 *src* 的长度小于 *len*，*src* 连带 NULL 全部复制。

返回值: **strncat** 函数返回 *dest*。

参考: **strcat,strcpy,strlen,strncpy**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strncat (void)  {
  char buf [21];

  strcpy  (buf, "test #");
  strncat (buf, "three", sizeof (buf) - strlen
           (buf));
}
```

strncmp

摘要:

```
#include<string.h>
char *strncmp(
    char *string1,          /* 第一个字符串 */
    char *string2,          /* 第二个字符串 */
    int len);               /* 比较的最多字符数 */
```

说明: **strncmp** 函数比较 *string1* 的第一个 *len* 字节和 *string2*，返回一个值表示他们的关系。

返回值: **strncmp** 函数返回一个下面的值表示 *string1* 的第一个 *len* 字节和 *string2* 的关系:

值	意义
<0	*string1* 小于 *string2*
=0	*string1* 等于 *string2*
>0	*string1* 大于 *string2*

参考: **memcmp,strcmp**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strncmp (void)  {
  char str1 [] = "Wrodanahan     T.J.";
  char str2 [] = "Wrodanaugh     J.W.";

  char i;

  i = strncmp (str1, str2, 15);

  if (i < 0)       printf ("str1 < str2\n");
  else if (i > 0)  printf ("str1 > str2\n");
  else             printf ("str1 == str2\n");
}
```

strncpy

摘要:

```
#include<string.h>
char *strncpy(
    char *dest,          /* 目标字符串 */
    char *src,           /* 源字符串 */
    int len);            /* 复制的最多字符数 */
```

说明: **strncpy** 函数从 *src* 复制最多 *len* 个字符到 *dest*。

返回值: **strncpy** 函数返回 *dest*。

参考: **strcat,strcpy,strlen,strncat**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strncpy ( char *s)  {
  char buf [21];

  strncpy (buf, s, sizeof (buf));
  buf [sizeof (buf)] = '\0';
}
```

strpbrk

摘要:

```
#include<string.h>
char *strpbrk(
    char *string,          /* 查找的字符串 */
    char *set);            /* 要查找的字符 */
```

说明: **strpbrk** 函数查找 string 中第一个出现的 *set* 中的任何字符。不包括 NULL 结束符。

返回值: **strpbrk** 函数返回 *string* 匹配的字符的指针。如果 *string* 没有字符和 *set* 匹配，返回一个 NULL 指针。

参考: **strchr,strcspn,strpos,strrchr,strrpbrk,strrpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strpbrk (void)  {
  char vowels [] ="AEIOUaeiou";
  char text [] = "Seven years ago...";

  char *p;

  p = strpbrk (text, vowels);

  if (p == NULL)
    printf ("No vowels found in %s\n", text);

  else
    printf ("Found a vowel at %s\n", p);

}
```

strpos

摘要:

```
#include<string.h>
int strpos(
    const char *string,          /* 被搜索字符串 */
    char c);                     /* 查找的字符 */
```

说明: **strpos** 函数查找 *string* 中 *c* 的第一次出现。包括 *string* 的 NULL 结束符。

返回值: **strpos** 函数返回 string 中和 *c* 匹配的字符的索引，如没匹配则返回-1。*string* 中第一个字符的索引是 0。

参考: **strchr,strcspn,strpbrk,strrchr,strrpbrk,strrpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strpos (void)  {
  char text [] = "Search this string for
  blanks";

  int i;

  i = strpos (text, ' ');

  if (i == -1)
    printf ("No spaces found in %s\n", text);

  else
    printf ("Found a space at offset %d\n", i);

}
```

strrchr

摘要:

```
#include<string.h>
char *strrchr(
    const char *string,          /*  被搜索字符串  */
    char c);                     /*  查找的字符  */
```

说明: **strrchr** 函数查找 *string* 中 *c* 的最后一次出现。包括 *string* 的 NULL 结束符。

返回值: **strrchr** 函数返回 string 中和 *c* 匹配的字符的指针，如没匹配则返回 NULL。

参考: **strchr,strcspn,strpbrk,strpos,strrpbrk,strrpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strrchr (void)  {
  char *s;
  char buf [] = "This is a test";

  s = strrchr (buf, 't');

  if (s != NULL)
    printf ("found the last 't' at %s\n", s);

}
```

strrpbrk

摘要:

```
#include<string.h>
char *strrpbrk(
    char *string,          /* 查找的字符串 */
    char *set);            /* 要查找的字符 */
```

说明: **strrpbrk** 函数查找 string 中最后一个出现的 *set* 中的任何字符。不包括 NULL 结束符。

返回值: **strrpbrk** 函数返回 *string* 最后匹配的字符的指针。如果 *string* 没有字符和 *set* 匹配，返回一个 NULL 指针。

参考: **strchr,strcspn,strpos,strrchr,strpbrk,strrpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strrpbrk (void)  {
  char vowels [] ="AEIOUaeiou";
  char text [] = "American National Standards
                  Institute";

  char *p;

  p = strpbrk (text, vowels);

  if (p == NULL)
    printf ("No vowels found in %s\n", text);

  else
    printf ("Last vowel is at %s\n", p);

}
```

strrpos

摘要:

```
#include<string.h>
int strrpos(
    const char *string,          /* 被搜索字符串 */
    char c);                     /* 查找的字符 */
```

说明: **strrpos** 函数查找 *string* 中 *c* 的最后一次出现。包括 *string* 的 NULL 结束符。

返回值: **strrpos** 函数返回 string 中和 *c* 匹配的最后字符的索引，如没匹配则返回-1。*string* 中第一个字符的索引是 0。

参考: **strchr,strcspn,strpbrk,strrchr,strrpbrk,strpos,strspn**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strrpos (  char *s)  {

  int i;

  i = strpos (s, ' ');

  if (i == -1)
    printf ("No spaces found in %s\n", s);

  else
    printf ("Last space in %s is at offset %d\n",
             s, i);
}
```

strspn

摘要:

```
#include<string.h>
int strspn(
    char *string,          /* 查找的字符串 */
    char *set);            /* 要查找的字符 */
```

说明: **strspn** 函数查找 string 中 *set* 没有的字符。

返回值: **strspn** 函数返回 *string* 第一个和 *set* 不匹配匹配的字符的索引。
如果 *string* 中的第一个字符和 *set* 中的字符不匹配，返回一个 0。
如果 *string* 中的所有字符 *set* 中都有，返回 *string* 的长度。

参考: **strchr,strcspn,strpos,strrchr,strpbrk,strrpos,strrpbrk**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

void tst_strspn ( char *digit_str)  {
  char octd [] = "01234567";
  int i;

  i = strspn (digit_str, octd);

  if (digit_str [i] != '\0')
    printf ("%c is not an octal digit\n",
             digit_str [i]);

}
```

strstr

摘要:

```
#include<string.h>
char *strstr(
    const char *src,          /* 要查找的字符串 */
    char *sub);               /* 查找的子字符串 */
```

说明: **strstr** 函数确定字符串 *sub* 在 *src* 中第一次出现的位置，返回一个指针指向第一次出现的开头。

返回值: **strstr** 函数返回 *src* 中和 *sub* 一样的点的指针。如果 *src* 中不存在 *sub*，则返回一个 NULL 指针。

参考: **strchr,strpos**

例子:

```
#include <string.h>
#include <stdio.h>                  /* for printf */

char s1 [] = "My House is small";
char s2 [] = "My Car is green";

void tst_strstr (void)  {
  char *s;

  s = strstr (s1, "House");
  printf ("substr (s1, \"House\") returns %s\n", s);
```

strtod/strtod517

摘要:

```
#include<stdlib.h>
unsigned long strtod(
    const char *string,        /* 要转换的字符串 */
    char **ptr);               /* 指向后来的字符的指针 */
```

说明: **strtod** 函数转换 *string* 为一个浮点数。输入 *string* 是一个字符序列，可以解释为一个浮点数。字符串开头的空白字符被忽略。

strtod517 函数和 **strtod** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

strtod 函数要求 *string* 有下面的格式:

[{+|-}]*digits*[.*digits*][{e|E}[{+|-}]*digits*]

这里:

digits 可能是一个或多个十进制数。

ptr 的值设置指针到 *string* 中转换部分的第一个字符。如果 *ptr* 是 NULL，没有值和 *ptr* 关联。如果不能转换，则 *ptr* 就设为 *string* 的值，**strtoul** 返回 0。

返回值: **strtod** 函数返回由 *string* 生成的浮点数。

参考: **atof,atoi,atol,strtol,strtoul**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

void tst_strtod (void)  {
  float f;
  char s [] = "1.23";

  f = strtod (s, NULL);
  printf ("strtod(%s) = %f\n", s, f);
}
```

strtol

摘要:

```
#include<stdlib.h>
long strtol(
    const char *string,          /* 要转换的字符串*/
    char **ptr,                  /* 指向后来的字符的指针 */
    unsigned char base);         /* 转换的基数 */
```

说明: **strtol** 函数转换 *string* 为一个 long 值。输入 *string* 是一个字符序列，可以解释为一个整数。字符串开头的空白字符被忽略。符号可选。

strtol 函数要求 *string* 有下面的格式:

[*whitespace*][{+|-}]*digits*

这里:

digits 可能是一个或多个十进制数。

如果 *base* 是零，数值应该有一个十进制常数，八进制常数，或十六进制常数的格式。数值的基数从格式推出。如果 *base* 在 2 到 36 之间，数值必须是一个字母或数字的非零序列，表示指定基数的一个整数。字母 a 到 z（或 A 到 Z）分别表示值 10 到 36。只有小于 *base* 的字母表示的值是允许的。如果 *base* 是 16，数值可能以 0x 或 0X 开头，0x 或 0X 被忽略。

ptr 的值设置指针指向 *string* 中转换部分的第一个字符。如果 *ptr* 是 NULL，没有值和 *ptr* 关联。如果不能转换，*ptr* 设置为 *string* 的值，**strtol** 返回 0。

返回值: **strtol** 函数返回 *string* 生成的整数值。如溢出则返回 LONG_MIX 或 LONG_MAX。

参考： **atof,atoi,atol,strtod,strtoul**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

char s [] = "-123456789";

void tst_strtol (void)  {
  long l;

  l = strtol (s, NULL, 10);
  printf ("strtol(%s) = %ld\n", s, l);
}
```

strtoul

摘要:

```
#include<stdlib.h>
unsigned long strtoul(
    const char *string,          /* 要转换的字符串*/
    char **ptr,                  /* 指向后来的字符的指针 */
    unsigned char base);         /* 转换的基数 */
```

说明: **strtoul** 函数转换 *string* 为一个 unsigned long 值。输入 *string* 是一个字符序列，可以解释为一个整数。字符串开头的空白字符被忽略。符号可选。

strtoul 函数要求 *string* 有下面的格式:

[*whitespace*][{+|-}]*digits*

这里:

digits 可能是一个或多个十进制数。

如果 *base* 是零，数值应该有一个十进制常数，八进制常数，或十六进制常数的格式。数值的基数从格式推出。如果 *base* 在 2 到 36 之间，数值必须是一个字母或数字的非零序列，表示指定基数的一个整数。字母 a 到 z（或 A 到 Z）分别表示值 10 到 36。只有小于 *base* 的字母表示的值是允许的。如果 *base* 是 16，数值可能以 0x 或 0X 开头，0x 或 0X 被忽略。

ptr 的值设置指针指向 *string* 中转换部分的第一个字符。如果 *ptr* 是 NULL，没有值和 *ptr* 关联。如果不能转换，*ptr* 设置为 *string* 的值，**strtoul** 返回 0。

返回值: **strtoul** 函数返回 *string* 生成的整数值。如溢出则返回 ULONG_MAX。

参考： **atof,atoi,atol,strtod,strtol**

例子:

```
#include <stdlib.h>
#include <stdio.h>                  /* for printf */

char s [] = "12345AB";

void tst_strtoul (void)  {
  unsigned long ul;

  ul = strtoul (s, NULL, 16);
  printf ("strtoul(%s) = %lx\n", s, ul);
}
```

tan/tan517

摘要:

```
#include<math.h>
float tan(
    flaot x);                    /*  计算正切  */
```

说明: **tan** 函数计算浮点数 *x* 的正切。值 *x* 必须在-65535 到+65535 之间，或错误值 **NaN**。

tan517 函数和 **tan** 一样，但使用 INFINEON C517x，C509 的算术单元，提供更快的运行速度。当使用整个函数时，应包含 80C517.H 头文件。不要对不支持的 CPU 用本程序。

返回值: **tan** 函数返回 *x* 的正切。

参考: **cos,sin**

例子:

```
#include <math.h>
#include <stdio.h>                    /* for printf */

void tst_tan (void)  {
  float x, y, pi;

  pi = 3.14159;

  for (x = -(pi/4); x < (pi/4); x += 0.1)  {
    y = tan (x);
    printf ("TAN(%f) = %f\n", x, y);
  }
}
```

tanh

摘要:

```
#include<math.h>
float tanh(
    float x);                    /* 计算双曲正切 */
```

说明: **tanh** 函数计算浮点数 *x* 的双曲正切。

返回值: **tanh** 函数返回 *x* 的双曲正切。

参考: **cosh**,**sinh**

例子:

```
#include <math.h>
#include <stdio.h>                  /* for printf */

void tst_tanh (void)  {
  float x;
  float y;
  float pi;

  pi = 3.14159;

  for (x = -(pi/4); x < (pi/4); x += 0.1)  {
    y = tanh (x);
    printf ("TANH(%f) = %f\n", x, y);
  }

}
```

testbit

摘要:

```
#include<instrins.h>
bit _testbit_(
    bit b);                    /* 测试和清除位 */
```

说明: **_testbit_**程序在生成的代码中用 JBC 指令来测试位 *b*，并清零。本程序只能用在直接寻址位变量，对任何类型的表达式无效。本程序作为一个固有函数。代码要求内嵌，而非调用。

返回值: **_testbit_**程序返回值 *b*。

例子:

```
#include <intrins.h>
#include <stdio.h>                  /* for printf */

void tst_testbit (void){
  bit test_flag;

  if (_testbit_ (test_flag))
    printf ("Bit was set\n");

  else
    printf ("Bit was clear\n");
}
```

toascii

摘要: **#include<ctype.h>**
char toascii(
char *c*); /* 字符转换 */

说明: **toascii** 宏转换 *c* 为一个 7 位 ASCII 字符。宏只变量 *c* 的低 7 位。

返回值: **toascii** 宏返回 *c* 的 7 位 ASCII 字符。

参考: **toint**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_toascii ( char c)  {
  char k;

  k = toascii (c);

  printf ("%c is an ASCII character\n", k);
}
```

toint

摘要:

```
#include<ctype.h>
char toint(
    char c);                    /* 字符转换 */
```

说明: **toint** 函数解释 *c* 为一个十六进制值。ASCII 字符‘**0**’到‘**9**’生成值 0 到 9。ASCII 字符‘**A**’到‘**F**’和‘**a**’到‘**f**’生成值 10 到 15。如果 *c* 表示一个十六进制数，函数返回-1。

返回值: **toint** 宏返回 *c* 的十六进制 ASCII 值。

参考: **toascii**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_toint (void)  {
  unsigned long l;
  char k;

  for (l = 0; isdigit (k = getchar ());
       l *= 10)  {

     l += toint (k);
  }
}
```

tolower

摘要: **#include<ctype.h>**
char tolower(
char *c*); /* 字符转换 */

说明: **tolower** 函数转换 *c* 为一个小写字符。如果 *c* 表示一个字母，**tolower** 函数无效。

返回值: **tolower** 宏返回 *c* 的小写。

参考: **_tolower,toupper,_toupper**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_tolower (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)  {
    printf ("tolower(%c) = %c\n", i, tolower(i));
  }
}
```

_tolower

摘要: **#include<ctype.h>**
char _tolower(
char *c*); /* 字符转换 */

说明: **_tolower** 宏是在已知 *c* 是一个大写字符的情况下可用的 **lower** 的一个版本。

返回值: **_tolower** 宏返回 *c* 的小写。

参考: **tolower,toupper,_toupper**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst__tolower (  char k)  {

  if (isupper (k))   k = _tolower (k);
}
```

toupper

摘要: **#include<ctype.h>**
char toupper(
char *c*); /* 字符转换 */

说明: **toupper** 函数转换 *c* 为一个大写字符。如果 *c* 表示一个字母，**toupper** 函数无效。

返回值: **toupper** 宏返回 *c* 的大写。

参考: **_tolower,tolower,_toupper**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst_toupper (void)  {
  unsigned char i;

  for (i = 0x20; i < 0x7F; i++)  {
    printf ("toupper(%c) = %c\n", i, toupper(i));
  }
}
```

_toupper

摘要: **#include<ctype.h>**
char _toupper(
char *c*); /* 字符转换 */

说明: **_toupper** 宏是在已知 *c* 是一个小写字符的情况下可用的 **toupper** 的一个版本。

返回值: **_toupper** 宏返回 *c* 的大写。

参考: **tolower,toupper,_tolower**

例子:

```
#include <ctype.h>
#include <stdio.h>                  /* for printf */

void tst__toupper (  char k)  {
  if (islower (k))  k = _toupper (k);
}
```

ungetchar

摘要: **#include<stdio.h>**
char ungetchar(
char *c*); /* 退回字符 */

说明: **ungetchar** 函数把字符 *c* 放回到输入流。子程序被 **getchar** 和别的返回 *c* 的流输入函数调用。**getchar** 在调用时只能传递一个字符给 **ungetchar**。

返回值: 如果成功，**ungetchar** 函数返回字符 *c*。如果调用者在读输入流时调用 **ungetchar** 多次，返回 **EOF** 表示一个错误条件。

参考: **_getkey,putchar,ungetchar**

例子:

```
#include <stdio.h>

void tst_ungetchar (void)  {
  char k;

  while (isdigit (k = getchar ()))  {
      /* stay in the loop as long as k is a digit */
  }
  ungetchar (k);
}
```

va_rag

摘要:

```
#include<stdarg.h>
type va_arg(
    argptr,                 /* 可选的参数列表 */
    type);                  /* 下一个参数的类型 */
```

说明: **va_arg** 宏用来从一个可变长度参数列表索引 *argptr* 提取并列参数。*type* 参数指定提取参数的数据类型。本宏对每个参数只能调用一次，且必须根据参数列表中的参数顺序调用。

第一次调用 **va_arg** 返回 **va_start** 宏中指定的 *prevparm* 参数后的第一个参数。后来对 **va_arg** 的调用依次返回余下的参数。

返回值: **va_arg** 宏返回指定参数类型的值。

参考: **va_end,va_start**

例子:

```
#include <stdarg.h>
#include <stdio.h>                  /* for printf */

int varfunc (char *buf, int id, ...)  {
  va_list tag;

  va_start (tag, id);


  if (id == 0)  {
    int arg1;
    char *arg2;
    long arg3;

    arg1 = va_arg (tag, int);
    arg2 = va_arg (tag, char *);
    arg3 = va_arg (tag, long);
  }
  else  {
    char *arg1;
    char *arg2;
    long arg3;

    arg1 = va_arg (tag, char *);
    arg2 = va_arg (tag, char *);
    arg3 = va_arg (tag, long);
  }
}


void caller (void)  {
  char tmp_buffer [10];

  varfunc (tmp_buffer, 0, 27, "Test Code", 100L);
  varfunc (tmp_buffer, 1, "Test", "Code", 348L);
}
```

va_end

摘要: **#include<stdarg.h>**
void va_end(
argptr); /* 可选的参数列表 */

说明: **va_end** 宏用来终止可变长度参数列表指针 *argptr* 的使用，*argptr* 用 **va_start** 宏初始化。

返回值: 无

参考: **va_arg,va_start**

例子: 看 **va_arg**。

va_start

摘要:

```
#include<stdarg.h>
void va_start(
    argptr,                 /* 可选的参数列表 */
    prevparm);              /* 可选参数前的参数 */
```

说明: **va_start** 宏用在一个可变长度参数列表的函数中时，用 **va_arg** 和 **va_end** 宏初始化 *argptr*。*prevparm* 参数必须是用省略号（…）指定的可选参数前紧挨的函数参数。着函数必须在用 **va_arg** 宏访问前初始化可变长度参数列表指针。

返回值: 无

参考: **va_arg,va_end**

例子: 看 **va_arg**。

vprintf

摘要:

```
#include<stdio.h>
void vprintf(
    const char *fmtstr,          /* 指向格式字符串的指针 */
    char *argptr);               /* 指向参数列表的指针 */
```

说明: **vprintf** 函数格式化一系列字符串和数字值，并建立一个用 **puschar** 函数写到输出流的字符串，函数类似于 printf 的副本，但使用参数列表的指针，而不是一个参数列表。

fmtstr 参数是一个指向一个格式字符串的指针，和 **printf** 函数的 *fmtstr* 参数有相同的形式和功能。参考 290 页的“printf/printf517”中的格式字符串的说明。*argptr* 参数指向一系列参数，根据格式中指定的对应格式转换和输出。

注意:
*本函数是指定执行的，基于 **putchar** 函数的操作。本函数作为标准库提供，用 8051 的串口写字符。别的 I/O 设备可以定制函数。*

返回值: **vprintf** 函数返回实际写到输出流的字符数。

参考: **gets,puts,printf,scanf,sprintf,sscanf,vsprintf**

例子:

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

void error (char *fmt, ...)  {
  va_list arg_ptr;

  va_start (arg_ptr, fmt);       /* format string */
  vprintf (fmt, arg_ptr);
  va_end (arg_ptr);
}

void tst_vprintf (void)  {
  int i;
  i = 1000;
                /* call error with one parameter */
  error ("Error: '%d' number too large\n", i);
         /* call error with just a format string */
  error ("Syntax Error\n");
}
```

vsprintf

摘要:

```
#include<stdio.h>
void vsprintf(
    char *buffer;          /* 指向保存缓冲区的指针 */
    const char *fmtstr,    /* 指向格式字符串的指针 */
    char *argptr);         /* 指向参数列表的指针 */
```

说明: **vsprintf** 函数格式化一系列字符串和数字值，并保存字符串在 *buffer*。函数类似于 sprintf 的副本，但使用参数列表的指针，而不是一个参数列表。

fmtstr 参数是一个指向一个格式字符串的指针，和 **printf** 函数的 *fmtstr* 参数有相同的形式和功能。参考 290 页的“printf/printf517”中的格式字符串的说明。*argptr* 参数指向一系列参数，根据格式中指定的对应格式转换和输出。

返回值: **vsprintf** 函数返回实际写到输出流的字符数。

参考: **gets,puts,printf,scanf,sprintf,sscanf,vprintf**

例子:

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

xdata char etxt[30];               /* text buffer */


void error (char *fmt, ...)  {
  va_list arg_ptr;

  va_start (arg_ptr, fmt);       /* format string */
  vsprintf (etxt, fmt, arg_ptr);
  va_end (arg_ptr);
}

void tst_vprintf (void)  {
  int i;
  i = 1000;

                /* call error with one parameter */
  error ("Error: '%d' number too large\n", i);

         /* call error with just a format string */
  error ("Syntax Error\n");
}
```

附录 A. 和 ANSI C 的差异

C*x*51 编译器和 ANSI C 标准只在一些小的方面有差异。这些差异可以分成和编译器相关的差异和和库相关的差异。

和编译器相关的差异

- **宽字符**
  宽 16 位字符不被 **C*x*51** 支持。ANSI 对将来的一个国际字符集提供宽字符。

- **递归函数调用**
  缺省的情况不支持递归函数调用。递归函数必须用 **reentrant** 函数属性声明。可重入函数可被递归调用，因为局部数据和参数保存在可重入堆栈中。比较而言，不要用 **reentrant** 属性声明的函数对函数的局部数据使用静态存储段。对这些函数的递归调用会改写前面函数调用例程的局部数据。

和库相关的差异

ANSI C 标准库包括大量的程序，大多数在 **C*x*51** 中。但是，许多不能应用到一个嵌入应用中，被排除在 **C*x*51** 库之外。

下面 ANSI 标准库程序包含在 **C*x*51** 库中：

abs	**cosh**	**isdigit**
acos	**exp**	**isgraph**
asin	**fabs**	**islower**
atan	**floor**	**isprint**
atan2	**fmod**	**ispunct**
atof	**free**	**isspace**
atoi	**getchar**	**isupper**
atol	**gets**	**isxdigit**
calloc	**isalnum**	**labs**
ceil	**isalpha**	**log**
cos	**iscntrl**	**log10**

logjmp
malloc
memchr
memcmp
memcpy
memmove
memset
modf
pow
printf
putchar
puts
rand
realloc
scanf
setjmp
sin
sinh
sprintf
sqrt
srand
sscanf
strcat
strchr
strcmp
strcpy
strcspn
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtol
strtoul
tan
tanh
tolower
toupper
va_arg
va_end
va_start
vprintf
vsprintf

下面的 ANSI 标准库程序不在 **C*x*51** 库中：

abort
asctime
atexit
bsearch
clearerr
clock
ctime
difftime
div
exit
fclose
feof
ferror
fflush
fgetc
fgetpos
fgets
fopen
fprintf
fputc
fputs
fread
freopen
frexp
fscanf
fseek
fsetpos
ftell
fwrite
getc
getenv
gmtime
ldexp
ldiv
localeconv
localtime
mblen
mbstowcs
mbtowc
mktime
perror
putc
qsort
raise
remove
rename
rewind
setbuf
setlocale
setvbuf
signal
strcoll
strerror
strftime
strtok
strxfrm
system
time
tmpfile
tmpnam
ungetc
vfprintf
wcstombs
wctomb

下面的程序不包括在 ANSI 标准库中，但在 **C*x*51** 库中：

acos517
asin517
atan517
atof517
cabs
chkfloat
cos517
crol
cror
exp517
_getkey
init_mempool
irol
iror
log10517
log517
lrol
lror
memccpy
nop
printf517
scanf517
sin517
sprintf517
sqrt517
sscanf517
strpos
strrpbrk
strrpos
strtod517
tan517
testbit
toascii
toint
_tolower
_toupper
ungetchar

附录 B. 版本差异

本附录列出一个主要产品改进的概况，当前 **Cx51** 编译器版本和前面版本的差异。当前版本包含下面的*所有*改进：

版本 6.0 差异

- **取除外部和段的限制**
  每个模块的外部符号和段的数目不再限制在 256 内。这个历史的限制是旧的 INTEL 目标文件格式强制的。

- **变量名可以有 256 个字符**
  现在一个变量名可以有 256 个字符，以前只能有 32 个字符。

- **支持 PHILIPS 80C51MX 和 DALLAS 连续模式**
  C*x*51 提供对 PHILIPS 80C51MX 结构和 DALLAS 连续模式（对 DALLAS 390 和变种）的支持。

- **支持 OMF2 命令和 far 存储类**型
  OMF2 命令选择一个新的 OMF 文件格式，提供对模块内的详细的符号类型检查，支持最多 16MB 的代码和 xdata 存储区。当用 STRING，VARBANKING，和 XCROM 命令时要求这格式。

- **STRING 命令**
  C*x*51 允许定位常数字符串在 **const xdata** 或 **const far** 空间，为程序代码保留更多的代码空间。

- **USERCLASS 命令**
  分配用户定义的类名给编译器生成的段。类名可以被 LX51 连接器引用来定位所有的有指定类名的段。

- **VARBANKING 命令和 far 存储类型支持**
  两个新存储类，**far** 和 **const far**，和用户可配置的访问程序，提供对最多 16MB 扩展代码和 xdata 存储的支持。VARBANKING 命令使能 **far** 存储类型支持。

- **XCROM 命令**
  XCROM 命令定位常数到 **xdata** ROM，为程序代码空除 **code** ROM 空间。

- **支持模拟器件 B2 系列的微转换器**
  B2 系列 ADuC 器件包含双 DPTR 和一个扩展的堆栈。

注意:
只有 PK51 专业开发工具支持 OMF2 输出文件格式，PHILIPS 80C51MX，DALLAS 连续模式，和 VARBANKING。这些选项不包括在 CA51 和 DK51 包中。

版本 5 的差异

- **优化级别 7，8，和 9**
  C51 提供三个新的优化级别。这些新的优化主要集中于代码密度。参考 157 页的“优化”。

- **支持双 DPTR 的命令**
  C51 对 ATMEL，ATMEL WM，和 PHILIPS 提供双 DPTR 支持，用命令 **MODA2** 和 **MODP2**。

- **data，pdata，xdata 自动变量在所有存储模式中可覆盖**
  C51 现在可覆盖所有的 data，pdata，和 xdata 自动变量，无论选了什么存储模式。在以前的 C51 版本中，只有缺省存储模式的自动变量可覆盖。例如，如果一个函数用 SMALL 存储模式编译，C51 版本 5 不覆盖 pdata 或 xdata 变量。

- **enum 数据类型自动调整为 8 或 16 位**
  如果 enum 范围允许，C51 现在用一个 char 变量表示一个 enum。

- **modf，strtod，strtol，strtoul 库函数**
  C51 现在包括 ANSI 标准库函数 **modf**，**strtod**，**strtol**，**strtoul**。

- **BROWSE，INCDIR，ONEREGBANK，RET_XSTK，和 RET_PSTK 命令**
  C51 支持新命令产生浏览信息，指定包含命令，优化中断代码，和多返回地址用可重入堆栈。参考 17 页的“第二章.用 Cx51 编译”。

版本 4 差异

- **浮点数的字节顺序**
  现在浮点数按大 ENDIAN 顺序保存。以前的 C51 编译器用小 ENDIAN 格式保存浮点数。参考 179 页的“浮点数”。

- **_chkfloat_库函数**
  固有函数**_chkfloat_**允许对浮点数快速测试，可测试错误（**NaN**），±INF，零和正常数。参考 245 页的“_chkfloat_”。

- **FLOATFUZZY 命令**
  现在 C51 支持 **FLOATFUZZY** 命令。这个命令控制在浮点数比较中被忽略的位数。参考 38 页的“FLOATFUZZY”。

- **浮点数运算完全可重入**
  固有的浮点数运算操作（加，减，乘，除，和比较）现在完全可重入。C 库程序 **fpsave** 和 **fprestore** 不再需要。几个库函数也可重入。参考 218 页的“程序分类”。

- **long 和浮点数操作不再使用一个算术堆栈**
  long 和浮点运算更有效；现在产生的代码是基于寄存器的，不再使用模拟算术堆栈。这也减少了所需的存储空间。

- **存储类型**
  存储类型已经被改变，以在运行库中达到更优的性能，和反映 MCS251 结构的存储影射。

- **通用指针的存储类型字节**
  通用指针中所用的存储类型字节以改变。下表包含存储类型值和所关联的存储类型。

存储类型	idata	data	bdata	xdata	pdata	code
C51 V5 值	**0x00**	**0x00**	**0x00**	**0x01**	**0xFE**	**0xFF**
C51 V4 值	**0x01**	**0x04**	**0x04**	**0x02**	**0x03**	**0x06**

- **WARNINGLEVEL 命令**
  现在 C51 支持 **WARNINGLEVEL** 命令，可用来指定 C51 编译器检测警告的强度。

现在 C51 编译器也检查未用的局部变量，标号，和表达式。参考 85 页的“WARNINGLEVEL”。

版本 3.4 差异

- **_at_关键词**
  C51 支持用**_at_**关键词定位变量。这个新的关键词允许在声明中指定一个变量的地址。参考 186 页的“_at_关键词”。

- **NOAMAKE 命令**
  C51 支持 **NOAMAKE** 命令。本命令使 C51 生成的目标模块中不包含 PROJECT 信息和寄存器优化记录。这仅对想用旧版本的 C51 代码的目标文件时是必须的。

- **OH51 十六进制文件转换器**
  以前版本的 OHS51 OBJ-HEX-符号转换器已被 OH51 替代。

- **优化级别 6**
  C51 支持优化级别 6，提供循环旋转。结果代码更有效和运行更快。

- **ORDER 命令**
  当指定 **ORDER** 命令，C51 按源文件声明的顺序在存储区定位变量。参考 65 页的“ORDER”。

- **REGFILE 命令**
  C51 支持 REGFILE 命令，允许指定由连接器产生的寄存器定义文件名。这个文件包含用来优化不同模块间的函数使用寄存器的信息。参考 70 页的“REGFILE”。

- **vprintf 和 vsprintf 库函数**
  增加了 **vprintf** 和 **vsprintf** 库函数。参考 349 页的“vprintf”和 351 页的“vsprintf”。

版本 3.2 差异

- **ANSI 标准自动整数提升**
  ANSI C 标准的最新版本中，如果 **char** 或 **unsigned char** 值在计算中可能溢出，要求用 **int** 值计算。这个新的要求基于 **int** 和 **char** 操作类似于 16 位 CPU 的前提下。C51 缺省支持这个特征，提供两个新命令，**INTPROMOTE** 和 **NOINTPROMOTE**，来使能和不使能整数提升。

  在 8 位的 8051 中，根据代码大小和运行速度，8 位和 16 位之间有很大的差异。因此，应用 **NOINTPROMOTE** 命令禁止整数提升。

  但是，如果保留和别的 C 编译器和平台的最大的兼容性，应允许整数提升使能。

- **用在线汇编器产生汇编源代码**
  可用新的 **ASM** 和 **ENDASM** 命令包含源文本输出到用 **SRC** 命令行命令生成的**.SRC** 文件。

- **新命令**
  增加或改进了命令：**ASM**，**ENDASM**，**INTERVAL**，**INTPROMOTE**，**INTVECTOR**，**MAXARGS**，和 **NOINTPROMOTE**。

- **中断矢量的偏移和间隔可指定**
  可指定中断矢量表的偏移和间隔。这些特征提供对 SIECO-51 命令的支持，允许指定中断矢量在一个不同的位置，中断表不从地址 0000h 开始。

- **参数传递给间接被调用函数**
  如果所有的参数可以在 CPU 寄存器中传递，函数参数可以传递给间接被调用函数。这些函数不必用重入属性声明。

- **存储分配函数提供源代码**
  C51 编译器提供了存储分配程序的 C 源代码。可以很容易的根据嵌入系统的硬件结构调整这些函数。

- **三维**
  C51 支持三维序列。

- **所有函数的可变长度参数列表**
  对所有函数类型支持可变长度参数列表。有可变长度参数列表的函数不必用 **reentrant** 属性声明。新的命令行命令 **MAXARGS** 确定参数传递区的大小。

版本 3.0 差异

- **汇编源文件输出中增加的新命令**
  增加了 **SRC** 命令，指示编译器生成一个汇编源文件带头一个 OBJ 文件。

- **新的库函数**
  增加的库函数：**calloc**，**free**，**init_mempool**，**malloc**，和 **realloc**。

版本 2 差异

- **绝对寄存器寻址**
  C51 生成执行绝对寄存器寻址的代码。这可提高运行速度。命令 **AREGS** 和 **NOAREGS** 分别使能和不使能本特征。

- **位可寻址存储类型**
  char 和 **int** 变量类型现在可以用 **bdata** 存储标识符声明为位于位可寻址内部存储区。

- **固有函数**
  在库中增加了固有函数，用来支持 8051 内建的特殊指令。

- **混合存储模式**
  支持调用和被调用函数有不同的存储模式。

- **新的优化级别**
  C51 编译器增加了两个新的优化级别。这些新级别支持寄存器变量，取除局部共用子表达式，循环优化，和全局共用子表达式取除，to name a few。

- **新的预定义宏**
  宏_ _**C51**_ _和_ _**MODEL**_ _在编译时由预处理器定义。

- **可重入和递归函数**
  可用 **reentrant** 函数属性定义单个函数为可重入或递归。

- **用寄存器传递参数**
  C51 可以用寄存器传递 3 个函数参数。**REGPARMS** 和 **NOREGPARMS** 命令使能和不使能本特征。

- **支持指定存储指针**
  可以定义指针引用在特定存储区的数据。

- **支持 PL/M-51 函数**
  增加了 **alien** 关键词支持 PL/M-51 兼容函数和函数调用。

- **volatile 类型标识符**
  volatile 变量属性可用来强制变量访问和禁止变量优化。

附录 C. 编写适宜的代码

本章列出许多方法来提高由 **Cx51** 编译器生成的 8051 代码的效率（例如，更紧凑的代码和更快的速度）。下面的这些肯定不是所有的方法。但是，是提高程序的速度和减少代码大小的大多数情况。

存储模式

代码大小和运行速度的最大的冲突是存储模式。用 SMALL 模式编译通常生成可能的最小，最快的代码。**SMALL** 命令指示 **Cx51** 编译器使用小存储模式。在小模式，除了特别声明，所有的变量位于 8051 的内部存储区。访问内部数据区很快（典型的为 1 或 2 个时钟周期），生成的代码必 COMPACT 或 LARGE 模式生成的代码更小。例如，下面的循环：

```
for( i=0;i<100;i++) {
    do_nothing();
}
```

分别用 SMALL 模式和 LARGE 模式编译，示例生成的代码的差异。下面是 SMALL 模式的编译：

```
stmt level   source

   1         #pragma small
   2
   3         void do_nothing (void);
   4
   5
   6         void func (void)
   7         {
   8   1     unsigned char i;
   9   1
  10   1     for (i = 0; i < 100; i++)
  11   1       {
  12   2       do_nothing ();
  13   2       }
  14   1     }
       ; FUNCTION func (BEGIN)
                                  ; SOURCE LINE # 10
0000 E4                 CLR   A
0001 F500    R          MOV   i,A
0003            ?C0001:
0003 E500    R          MOV   A,i
0005 C3                 CLR   C
0006 9464               SUBB  A,#064H
```

```
0008 5007               JNC   ?C0004
                                ; SOURCE LINE # 12
000A 120000  E          LCALL do_nothing
                                ; SOURCE LINE # 13
000D 0500    R          INC   i
000F 80F2               SJMP  ?C0001
                                ; SOURCE LINE # 14
0011            ?C0004:
0011 22                 RET
```

在 SMALL 模式，变量 i 保存在内部数据区。访问 i 的指令，**MOV A，i** 和 **INC i**，仅要求两个字节的代码空间。另外，每个指令仅用一个时钟周期。用小模式编译主函数生成的代码的大小是 11h 或 17 个字节。
下面是系统的代码用 LARGE 模式编译：

```
        ; FUNCTION func (BEGIN)
                                ; SOURCE LINE # 10
0000 E4                 CLR   A
0001 900000  R          MOV   DPTR,#i
0004 F0                 MOVX  @DPTR,A
0005            ?C0001:
0005 900000  R          MOV   DPTR,#i
0008 E0                 MOVX  A,@DPTR
0009 C3                 CLR   C
000A 9464               SUBB  A,#064H
000C 500B               JNC   ?C0004
                                ; SOURCE LINE # 12
000E 120000  E          LCALL do_nothing
                                ; SOURCE LINE # 13
0011 900000  R          MOV   DPTR,#i
0014 E0                 MOVX  A,@DPTR
0015 04                 INC   A
0016 F0                 MOVX  @DPTR,A
0017 80EC               SJMP  ?C0001
                                ; SOURCE LINE # 14
0019            ?C0004:
0019 22                 RET
```

在 LARGE 模式，变量 i 保存在外部数据区。要访问 i，编译器必须首先加载数据指针，然后执行外部存储区访问（看上面列表的 0001h 到 0004h）。这两个指令需要 4 个时钟周期。增加 i 的代码在偏移 0011h 到 0016h。这个操作用了 6 个字节的空间和 7 个时钟周期。用 LARGE 模式编译的主函数的大小是 19h 或 25 字节。

变量定位

经常访问的数据应该定位在 8051 的内部数据区。访问内部数据区比外部数据区更有效。内部数据区有寄存器组，位数据区，堆栈，和别的用户用 **data** 存储类型定义的变量。

因为内部数据区有限（128 到 256 字节），不能把所有的程序变量都放在这个存储区。因此，必须把一些变量放在别的存储区。这里有两个方法。

一个方法是改变存储模式，让编译器决定。这是最简单的方法，但也是最费代码和系统性能的。参考 367 页的“存储模式”。

另外一个方法是手工选择放在外部数据区的变量，用 **xdata** 存储标识符声明。通常，字符串缓冲区和别的大数组可以用 **xdata** 存储类型声明，而不会在性能和代码大小上有大的降低。

变量大小

8051 系列的成员都是 8 位 CPU。用 8 位类型运算（如 **char** 和 **unsigned char**）比用 **int** 或 **long** 类型更有效。因此，仅可能的使用最小的数据类型。

Cx51 编译器直接支持所有的字节操作。字节类型如不要求不会提升为整数。参考 **INTPROMOTE** 命令。

用一个乘法运算的例子可以说明。两个 **char** 目标的乘法用 8051 指令 **MUL AB**。完成用 **int** 或 **long** 类型的相同的运算要求调用编译器库函数。

unsigned 类型

8051 系列处理器不明确支持带符号数运算。对符号扩展编译器必须生成额外的代码。如仅可能的用 unsigned 目标可以减少代码。

局部变量

如可能，在循环和别的临时计算中用局部变量。作为优化的一部分，编译器尝试保存局部变量在寄存器中。寄存器访问是最快的存储访问。这通常用 **unsigned char** 和 **unsigned int** 变量类型达到。

别的来源

编译器生成代码的质量通常受程序所用的算法的影响。有时，用一个不同的算法就可以提高性能或减少代码大小。例如，一个堆排序算法通常比起泡排序法有效。
更多的关于如何写高效程序的方法，可以参考下面的书：

The Elements of Programming Style,**Second Edititon**
Kernighan & Plauger
McGraw-Hill
ISBN 0-07-034207-5

Writing Efficient Programs
Jon Louis Bentley
Prentice-Hall Software Series
ISBN 0-13-970244-X

Efficient C
Plum & Brodie
Plum Hall,Inc.
ISBN 0-911537-05-8

附录 D. 编译器限制

C*x*51 编译器包含了下面列出的已知的限制。对大部分的 C 语言的成员没有限制；例如，在一个 **switch** 块中，**case** 可以指定无限数目的符号或数字。如果有足够的地址空间，可以定义几千个符号。

- 支持最多 19 级对任何标准数据类型的间接（访问修饰符）访问。这包括数组描述符，间接操作符，和函数描述符。
- 名称最多为 256 个字符。C 语言是大小写敏感的。但是为了兼容，目标文件中的所有名称但是大写字母。因此一个源程序的外部目标名的大小写是无关紧要的。
- **switch** 块中 **case** 的最大数目是不固定的。只受可用的存储区大小和单个函数的最大大限制。
- 一个调用参数列表中最大可嵌套的函数调用是 10。
- 可嵌套的包含文件最多为 9。这和列表文件，预处理器文件，或是否生成一个目标文件无关。
- 条件的最大深度为 20。这是一个预处理器限制。
- 指令块（{..} ）可嵌套到 15 级。
- 宏可以嵌套到 8 级。
- 在一个宏或函数调用中可以传递最多 32 个参数。
- 一行或一个宏最长为 2000 个字符。即使一个宏扩展后，结果也不能超过 2000 个字符。

附录 E. 字节顺序

大多数微处理器的存储结构是 8 位地址空间（字节）。许多数据项（地址，数字，和字符串）太长，不能用单个字节保存，必须用一系列连续字节保存。

当使用用多字节保存的数据时，字节顺序就成为一个问题。不幸的是，多字节数据保存的标准不只一个。有对字节顺序的两个通行的方法被广泛使用。

第一个方法是调用小 ENDIAN，就是通常指的 INTEL 顺序。在小 ENDIAN 中，低字节首先保存。例如，一个 16 位整数值 0x1234（十进制 4660），用小 ENDIAN 方法保存，两个连续的字节如下：

地址	+0	+1
内容	0x34	0x12

一个 32 位整数值 0x57415244（十进制 1463898692），用小 ENDIAN 方法保存如下：

地址	+0	+1	+2	+3
内容	0x44	0x52	0x41	0x57

第二个方法是调用大 ENDIAN，就是通常的 MOTOROLA 顺序。在大 ENDIAN 中，高字节先保存，低字节后保存。例如，一个 16 位整数值 0x1234，用大 ENDIAN 方法，两个连续的字节如下：

地址	+0	+1
内容	0x12	0x34

一个 32 位的整数值 0x004A4F4E 用大 ENDIAN 方法如下：

地址	+0	+1	+2	+3
内容	0x00	0x4A	0x4F	0x4E

8051 是 8 位机制，没有直接操作大于 8 位数据的指令。多字节数据根据下面的规则保存。

- 8051 的 **LCALL** 指令保存下一个指令在堆栈中。地址的低字节首先推入堆栈。因此，地址是以小 ENDIAN 格式保存。

- 所有别的 16 位和 32 位值以大 ENDIAN 格式保存，高字节先保存。例如，**LJMP** 和 **LCALL** 指令，期望地址以大 ENDIAN 格式保存。

- 浮点数根据 IEEE-754 格式保存，以大 ENDIAN 格式首先保存高字节。

如果 8051 嵌入应用平台和别的微处理器通讯，必须知道别的 CPU 所用的字节顺序。当传输原始的二进制数据时不用考虑。

附录 F. 提示，标题和技术

本章列出了通常要求进一步解释的许多图表和标题。本章的项没有按特定的顺序排列，仅仅是相似问题的参考。

递归代码参考错误

下面的程序例子：

```
#pragma code symbols debug oe

void func1(unsigned char *msg ) { ; }

void func2( void ) {
  unsigned char uc;
  func1("xxxxxxxxxxxxxx");
}

code void (*func_array[])() = { func2 };

void main( void ) {
  (*func_array[0])();
}
```

当用下面的命令行编译和连接：

```
C51 EXAMPLE1.C

BL51 EXAMPLE1.OBJ IX
```

失败并显示下面的错误信息：

```
***    WARNING 13: RECURSIVE CALL TO SEGMENT
       SEGMENT:       ?CO?EXAMPLE1
       CALLER:        ?PR?FUNC2?EXAMPLE1
```

在这个程序例子中，**func2** 定义了一个常数字符串（“**xxx…xxx**”），直接放在常数代码段 ?CO?EXAMPLE1。定义 **code void(*func_array[]) () = {func2}**；在段**?CO?EXAMPLE1**（代码表的位置）和可允许代码段**?PR?FUNC2?EXAMPLE1** 之间生成了一个引用。因为 **func2** 也指向段**?CO?EXAMPLE1**，BL51 假定这是一个递归调用。

为了避免这个问题，用下面的命令行连接：

```
LX51 EXAMPLE1.OBJ IX OVERLAY &
(?CO?EXAMPLE1 ~ FUNC2,MAIN ! FUNC2)
```

?CO?EXAMPLE1 ~ FUNC2 删除了例子中 **func2** 和代码常数段之间的暗含的调用。然后，**MAIN ！FUNC2** 增加了一个额外的 **MAIN** 和 **FUNC2** 之间的索引列表的调用。*参考 Ax51 宏汇编用户手册。*
总的来说，当通过函数指针引用时，自动覆盖分析不能成功的完成。这个类型的引用必须手工完成，如上面的例子。

用 printf 程序的问题

printf 函数使用一个可变长度的参数列表。在格式字符串后面指定的参数用内在的数据类型传递。当格式标识符期望传递一个不同类型的数据时就会引起问题。例如，下面的代码：

```
printf("%c %d %u %bu",'A',1,2,3);
```

不会打印字符串“A 1 2 3”。这是因为 **Cx51** 编译器把参数 **1**，**2**，和 **3** 作为 8 位字节类型传递。格式标识符**%d** 和**%u** 都期望 16 位 int 类型。
为了避免这类问题，必须明确定义传递给 printf 函数的数据类型。因此，必须对上面的值进行类型转换，例如：

```
printf("%c %d %u %bu",'A',(int)1,(unsigned int)2,(char)3 );
```

如果不能确定要传递的参数的大小，可以转换值到希望的大小。

未调用函数

在开发过程中，通常会有写了但不调用的函数。编译器允许这样而不产生错误，但同时连接/定位器也暂时不处理这些代码，因为支持数据覆盖，只生成一个警告。
中断函数不被调用，他们由硬件调用。一个未调用的程序被连接器作为一个可能的中断程序。这意味着函数的局部变量被分配在一个不可覆盖数据区。这会很快耗尽所有可用的数据区（根据所用的存储类型决定）。
如果不希望用尽存储区，必须检查和未调用或未使用程序相关的连接器警告。可以用连接器的 **IXREF** 命令在连接器影射（**.M51**）文件包含一个交叉参考列表。

使用 Monitor-51

如果想要用 Monitor-51 测试一个 C 程序，如果 Monitor-51 被安装在地址 0，考虑下面的规则（指定目标系统用户程序可用的代码存储区在地址 8000H）：

- 所有的 C 模块，包含中断函数，必须用 **INTVECTOR**（**0x8000**）编译。

- 在 STARTUP.A51 中，声明 CSEG AT 0 必须用 CSEG AT 8000H 替代。然后，这个文件必须汇编和连接到目标程序，如文件头所指定。

bdata 存储类型的问题

一些用户说使用 **bdata** 存储类型很困难。**bdata** 的使用类似于 **sfr** 修饰符。最多的错误在引用一个定义在别的模块中的 **bdata** 变量时遇到。例如：

```
extern bdata char xyz_flag;

sbit xyz_bit1 = xyz_flag^1;
```

为了产生适当的指令，编译器必须生成引用的绝对值。在上面的例子中，这不能做到，在编译完成前 **xyz_flag** 的地址是未知的。遵循下面的规则可以避免这个问题。

1. 一个 **bdata** 变量（和一个 **sfr** 同样定义和使用）必须定义在全局空间；不用局限在一个程序范围内。

2. 一个 **bdata bit** 变量（和一个 **sfr** 同样定义和使用）必须也定义在全局空间，不能局限在一个程序内。

3. **bdata** 变量的定义和它的 **sbit** 访问成员名的建立在编译器遇到变量和成员前必须完成。

例如，在同一源模块内声明 bdata 变量和 bit 成员：

```
bdata char xyz_flag;
sbit xyz_bit1 = xyz_flag^1;
```

然后，声明外部位成员：

```
extern bit xyz_bit1;
```

和任何保留空间的别的声明和命名的 C 变量一样，仅仅在一个模块中定义 **bdata** 变量和 **sbits** 成员。然后，在需要的地方用 **extern bit** 标识符引用。

函数指针

函数指针是 C 中最难理解和使用的方面之一。函数指针中的最大的问题是由不正确的函数指针的声明，不正确的关联，和不正确的解除参照引起的。
下面简短的例子示例如何声明一个函数指针（f），如何和它关联一个函数地址，和如何通过指针调用函数。运行 DS51 模拟程序执行时把 **printf** 程序作为例子。

```
#pragma code symbols debug oe

#include <reg51.h>       /* special function register declarations */
#include <stdio.h>       /* prototype declarations for I/O functions */

void func1(int d) {          /* function #1 */
  printf("In FUNC1(%d)\n", d);
}

void func2(int i) {          /* function #2 */
  printf("In FUNC2(%d)\n", i);
}

void main(void) {
  void (*f)(int i);          /* Declaration of a function pointer */
                             /* that takes one integer arguments */
                             /* and returns nothing */

  SCON  = 0x50;              /* SCON: mode 1, 8-bit UART, enable rcvr */
  TMOD |= 0x20;              /* TMOD: timer 1, mode 2, 8-bit reload */
  TH1   = 0xf3;              /* TH1:  reload value for 2400 baud */
  TR1   = 1;                 /* TR1:  timer 1 run */
  TI    = 1;                 /* TI:   set TI to send first char of UART */

  while( 1 ) {
    f = (void *)func1;       /* f points to function #1 */
    f(1);
    f = (void *)func2;       /* f points to function #2 */
    f(2);
  }
}
```

注意:

因为受 8051 堆栈空间的限制，连接器覆盖函数变量和参数。当使用一个函数指针，连接器不能正确的建立程序的调用树。因此，应该对覆盖的数据纠正调用树。用连接器的 ***OVERLAY*** *命令来纠正。参考 Ax51 宏汇编用户手册。*

术语表

A51
标准 8051 宏汇编。

AX51
扩展 8051 宏汇编。

ANSI
美国国家标准协会。负责定义 C 语言标准的组织。

argument
传递给一个宏或函数的值。

arithmetic types
整数，浮点数，或列举的数据类型。

array
一个元素的集合，有相同的数据类型。

ASCII
信息交换的美国标准。这是一个 256 个计算机代码的集合，代表数字，字符，标点，和别的特殊符号。低 128 个字符是标准的。余下的 128 个可定义。

batch file
一个 ASCII 文本文件，包含可从命令行调用的命令和程序。

Binary-Coded Decimal(BCD)
一个 BCD 是一个系统用二进制格式编码十进制数。一个数的每个十进制数用一个 4 位的的二进制值编码。一个字节可以保存 2 个 BCD 数字-一个在高 4 位，一个在低 4 位。

BL51
标准 8051 连接/定位器。

block
一个 C 声明的序列，包括定义和声明，包含在大括号内（{}）。

C51
传统 8051 和扩展 8051 器件的优化 C 编译器。

CX51
对 PHILIPS 80C51MX 结构和 DALLAS 80C390 的优化 C 编译器。

constant expression（常数表达式）
任何计算结果是常数值的表达式。常数可能包含字符和整数常数值。

control（控制）
命令行控制切换到编译器，汇编器或连接器。

declaration（声明）
一个 C 构造，把一个名称和变量，类型，或函数关联。

definition（定义）
一个 C 构造，指定一个函数的名称，正参，实体，和返回类型，或一个变量的初始化和存储分配。

directive（命令）
指令或控制切换到编译器，汇编器或连接器。

escape sequence（转义序列）
一个反斜杠（‘\’）字符后面跟单个字母或数字的组合，指定一个字符串和字符常数的特殊的字符值。

expression（表达式）
任何数目的操作数和操作符的组合，生成一个常数值。

formal parameters（正参）
接收传递给一个函数的参数值的变量。

function（函数）
一个组合声明，可以用名称调用，执行一个操作和/或返回一个值。

function body（函数体）
一个包含声明和函数实体的块。

function call（函数调用）
一个调用和可能传递参数给一个函数的表达式。

function declaration（函数声明）
一个函数的名称和返回类型，在程序的别处有明确的定义。

function difinition（函数定义）
一个函数的名称，正参，返回类型，声明，和实体。

function prototype（函数原型）
一个函数声明，包含函数名后括号中的正参的列表。

in-circuit emulator(ICE)（实时仿真器）
一个硬件设备，提供硬件级的单步，跟踪，和断点，帮助调试嵌入软件。一些 ICE 提供一个跟踪缓冲区保存最近的 CPU 事件。

include file（包含文件）
一个嵌入一个源文件的文本文件。

keyword（关键词）
一个保留的词，有一个对编译器或汇编器的预定义的意义。

L51
旧的 8051 连接/定位器版本。L51 用 **BL51** 连接/定位器替代。

LX51
扩展的 8051 连接/定位器。

LIB51，LIBX51
用库管理器利用库文件的命令。

library（库）
一个保存许多可能相关的目标模块的文件。连接器可以从库中提取模块来使用在一个目标文件中。

LSB
最低位或字节。

macro（宏）
代表一系列键击的标识符。

manifest constant（明显常数）
一个定义常数值的宏。

MCS®51
应用于 INTEL 系列 8051 兼容微处理器的通用名称。

memory model（存储模式）
指定函数参数和局部变量的存储区的任何模式。

mnemonic（助记符）
一个 ASCII 字符串，代表汇编语言指令的一个机器码操作符。

MON51
一个 8051 程序，可以载入目标 CPU，通过加快软件下载帮助调试和加快产品开发。

MSB
最高位或字节。

newline character（换行符）
一个字符用来标记一个文本文件中的一行的结束，或代表新行字符的转义序列（'\n'）。

null character
ASCII 字符，值为 0，代表转义序列（'\0'）。

null pointer（null 指针）
一个空指针。一个空指针是整数值 0。

object（目标）
一个可以运行的存储区。通常使用在和一个变量或函数关联的存储区。

object file（目标文件）
一个文件，由编译器建立，包含程序段信息和重定位机器码。

OH51，OHX51
转换绝对目标文件到 INTEL HEX 文件格式的命令。

opcode（操作码）
也指操作代码。一个操作码是一个机器代码的第一个字节，通常代表一个 2 位数十六进制数。操作码显示机器码指令的类型和操作平台的类型。

operand（操作数）
一个用在一个表达式中的变量或常数。

operator（操作符）
一个符号（例如，+，-，*，/），指定如何操作一个表达式的操作数。

parameter（参数）
传递给一个宏或函数的值。

PL/M-51
一个高级编程语言，由 INTEL 在 1980 引入。

pointer（指针）
一个变量，包含另一个变量，函数，或存储区的地址。

pragma
一个声明，在编译时传递一个指令给编译器。

preprocessor（预处理器）
编译器首先传递处理 C 文件内容的文本处理器。预处理器定义和扩展宏，读包含文件，和传递命令给编译器。

relocatable（重定位）
可以重定位和不在一个固定的地址的目标代码。

RTX51 Full
一个 8051 实时运行，提供一个多任务操作系统内核和所用的程序库。

RTX51 Tiny（微型 RTX51）
一个限制版的 RTX51。

scalar types（标量类型）
在 C 中，整数，列举，浮点，和指针类型。

scope（范围）
一个项（函数或变量）可以被引用的程序段。一个项的范围可以被限制到文件，函数，或块。

special function register（SFR）（特殊功能寄存器）
一个 SFR 是一个 8051 内部数据存储区的寄存器，用来读和写 8051 的硬件成员。这包括串口，计时器，计数器，I/O 口，和别的硬件控制寄存器。

source file（源文件）
一个文本文件，包含 C 程序或汇编程序代码。

stack（堆栈）
一个存储区，用一个堆栈指针间接访问，随着元素的推入和弹出动态的缩短和扩展。堆栈中的元素以 LIFO（后进先出）弹出。

static（静态）
一个存储类，当用在一个函数的变量声明中，使变量在声明的块或函数退出后仍保留他们的值。

stream functions（流函数）
库程序，用输入和输出流读和写字符。

string（字符串）
一个字符数组，用一个 NULL 字符（‘\0’）结束。

structure（结构）
一个元素的集合，可能有不同的类型，用一个名称组合在一起。

structure member（结构成员）
结构的一个元素。

token（标记）
一个基础符号，代表一个编程语言的一个名称或实体。

two's complement（二的补）
一个二元符号，代表正和负数。一个正值的所有位的补加 1 就是负值。

type（类型）
一个变量的值的范围的说明。例如，一个 int 类型可以使指定范围内的任何值（-32768 到+32767）。

type cast（类型转换）
一个操作，在括号内指定所需的类型，直接放在操作数前，把一个操作数的类型转换到别的类型。

µVision2
一个集成软件开发平台，支持 KEIL 软件开发工具。µVision2 包含计划管理，源代码编辑，和程序调试。

whitespace character（空白字符）
用来分隔 C 程序的字符，如空格，制表符，和换行。

wild card（匹配符）
一个字符（？或*），可以用在一个文件名中替代字符。

索引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X