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。