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。