Keil Cx51 编译器编程基本原则和代码的优化
——和复杂声明的理解

在方法论阐述前，先解决几个Keil固有的顽疾——号称标志性建筑物。

一、Keil的标题“礦ision3″
可以使用如下方法取消：
1 用UltraEdit打开uv3 目录下的uv3.exe文件。
2 在Edit-Gtreplace中的Find What栏中输入“B5 56 69 73 69 6F 6E”，这是“礦ision”的ASCII码串。
3 在Edit-Gtreplace中的Replace With栏中输入“75 56 69 73 69 6F 6E”，这是“uVision”的ASCII码串。
4 点Replace All。
5 保存退出。
P.S.UltraEdit可以在sina下载。不过这个是商业软件，下载的是试用版，破解只是找个序列号的事。如果你未能找到该编辑器，Q我即可：1275701567
序列号在此不能公布,高飞可不想进宫啊!

二、光标问题
Keil uv3 中会出现光标定位不准的问题，对程序员来说是个很大的困扰
修改方法：打开Keil文件夹下的Tool.ini文件做如下修改
NAME=″User″, ″w″
EMAIL=″a″
ANSI=1--在这个地方添加这条语句
BOOK0=″UV3RELEASE_NOTES.HTM″ (″uVision Release Notes″,GEN)
——副作用是字体看起来变大了。

三、汉字问题
输入的汉字也总觉得看着不爽，查看后发现是设置问题.如下改变后，即美观：
Edit-Configuration-Color&Fonts-51:Editor Asm Files(汇编语言编辑器，相应的选择其他项便可设置其他)-Text->点击font后的按钮,选择中文字体（如仿宋)－OK,这就可以了。

好,进入正题吧。资料是高飞的学习心得。资料一部分由自己总结，一部分是网络汇聚的精华。此资料可在发现错误的时候作为参考。欢迎提出疑问，Q：1275701567

这里是C51 的指导文献。Cx51 User's Guide(官网地址，点击之前请确认。)

一、51 单片机的存储器组织结构

刚学习单片机可能会犯糊涂。除了名词多，理解不便外，还有一点就是对单片机存储器结构的不清晰。而要使用好单片机就必须清除它的内部的组织结构，无论是在芯片的选择还是代码的编写。

51 单片机最早由Intel公司推出，它在一块超大规模集成电路芯片上同时集成了CPU、ROM、RAM以及TIMER/COUNTER，使用者只需外接少量的接口电路就可组成自己的专用微处理器系统。目前，市场上 51 单片机的硬件支持芯片以及软件应用程序的种类十分丰富，除了Intel公司之外，还有Philips、Siemens、AMD、Fujutsu、OKI、AMTEL等公司都推出了以 51 为核心的单片机。新一代的 51 单片机集成度更高，在片内集成了更多的功能部件，如A/D、PWM、PCA、WDT以及高速I/O口等，在工业测量控制领域内得到极为广泛的应用，一次，有人指出 51 单片机已成为事实上的工业标准。目前已有多个厂家生产不同型号的 51 单片机，它们各有特点，但基本内核相同，指令系统也完全兼容。

图 1 所示为 51 基本内核的机构框图，包括：

•中央处理器CPU，用于执行各种指令和运算处理器；
•内部数据存储器RAM，用于存放可以读写的数据；
•内部程序存储器ROM，用于存放程序指令或某些常数表格；
•4 个 8 位的并行I/O接口P0~P3。每个都可以用作输入输出口；
•2 个定时/计数器，用作外部事件计数器或内部定时；
•5 个中断源，2 个外部中断、2 个定时器中断、1 个串行口中断。采用 2 个优先级的嵌套中断结构，可以实现二级中断服务程序嵌套，每一个中断源都可以用程序设定为高优先级中断或低优先级中断；
•1 个串行接口电路，用作异步接收发送器；
•内部时钟电路，晶振和微调电容需要外接。

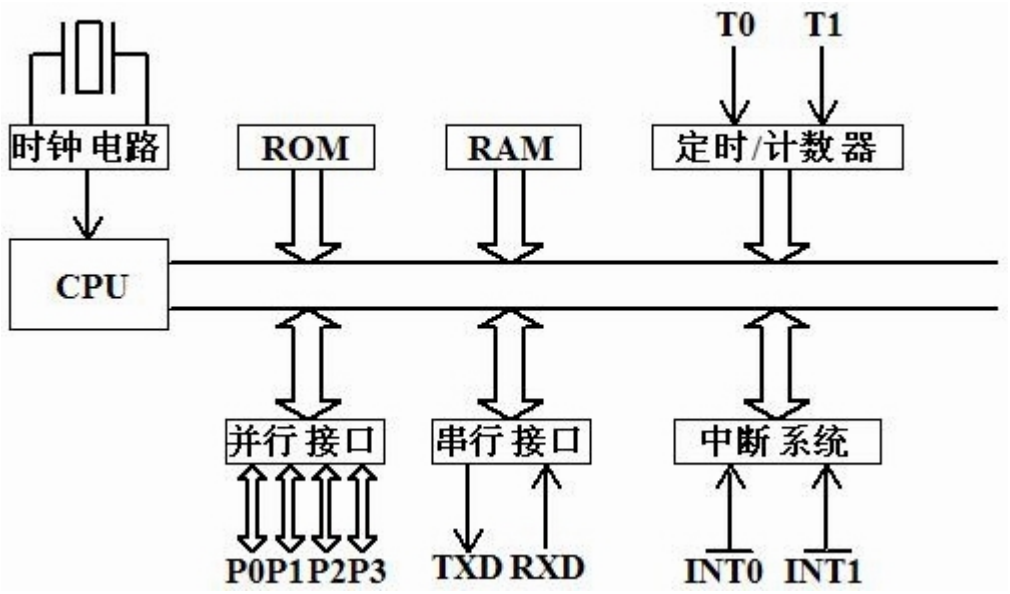

图 1　51 基本内核的结构框图

以上各部分通过内部总线相连接。在很多情况下，单片机还要和外部设备或外部存储器相连接，连接方式采用三总线(地址、数据、控制)方式，但在 51 单

片机中，没有单独的地址总线和数据总线，而是与并行I/O口中的P0 口及P2 公用。进行外部扩展时，P0 口分别作为低 8 位地址线和 8 位数据线；P2 口则作为高 8 位数据线，所以是 16 条地址线和 8 条数据新。但是明确一个概念，单片机进行外部扩展的地址和线和数据线都不是独立的总线，而是与并行I/O口共用的。这是 51 单片机结构上的一个特点。

对于采用高级语言Keil Cx51 的用户来说，了解和熟悉 51 单片机的存储器组织结构是十分必要的。这样在具体编程时可以合理安排各种变量，最大限度实现代码优化。从使用者的角度看，51 单片机有如下 3 个存储空间。

程序存储器ROM 对于普通 51 单片机，程序存储器ROM空间大小为 64KB，用与存放程序代码和一些表格常数，称为CODE空间。普通 51 可采用”代码分组”(CODE BANK)设计技术，将ROM空间扩展到 32x64KB，新型Philips 80C51Mx单片机的ROM空间最大可扩展到 16MB，称为ECODE和HCONST空间。51 单片机专门提供一个引脚“/EA”来区分片内ROM和片外ROM，/EA引脚接高电平时，单片机从内部ROM中读取指令，当地址超过片内ROM空间范围后，就自动转向片外ROM读取指令；/EA引脚接低电平时，所有的取指令操作均对片外ROM进行。程序存储器的某些地址单元是保留给系统使用的：0000H～0002H单元是所有执行程序的入口地址，复位后CPU总是从 0000H地址开始执行程序；0003H～002B单元均匀地分为 5 段，用于 5 个中断服务程序的入口，产生某个中断时，将自动进入其对应入口地址开始执行中断服务程序。一些新型 51 单片机增加了更多的中断源，它们的中断入口地址也相应增加。

片内数据存储器RAM 对于普通 51 单片机，片内数据存储器RAM空间最大为256B，用与存放程序执行过程中的各种变量及临时数据。片内RAM低 128 个字节可用直接寻址方式访问，也可用间接寻址方式访问，称为DATA区。其中 00H～1FH地址范围平均分为 4 组，每组都有 8 个工作寄存器R0～R7，称为工作寄存器区(Regeister Banks)。20H～2FH地址范围中，每个存储器单元的每一位都可以用位处理指令直接操作，该段地址范围称为位寻址区(BDATA)，其中每一位称为一个bit。对于 51 子系列单片机仅有上述低 128 个字节，对于 52 子系列单片机，增加了高 128 个字节的片内RAM，地址范围为 80H～FFH，该地址范围只能采用间接寻址方式访问，整个片内RAM地址范围 00H～FFH称为IDATA区。与IDATA空间高 128 个字节(地址范围 80H～FFH)重叠部分称为特殊功能寄存器区(SFR SPACE)，有些特殊功能寄存器是可以位寻址的，其可寻址位称为sbit。Philips公司推出的新型单片机 80C51Mx，其片内RAM最大可扩展到 64KB，称为EDATA区。

片外数据存储器RAM 对于普通 51 单片机，片外数据存储器RAM空间大小为64KB，称为XDATA区。在XDATA空间内进行分页寻址操作时，称为PDATA区。有些新型 51 单片机的扩充片内RAM，需要用专门的特殊功能寄存器“映像”(MAP)到XDATA地址空间；还有些新型 80C51 单片机可以将片外RAM最大扩展到 16MB，称为HDATA区。

图 2 所示为普通 51 单片机的存储器组织结构，其中，各部分空间说明及地址范围见表 1。

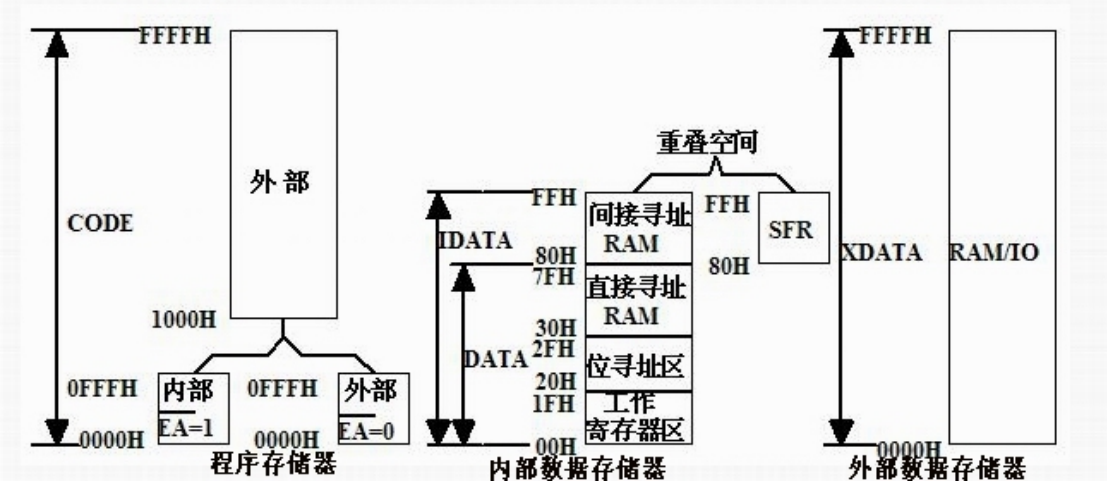

图 2 普通 51 单片机的存储器组织结构

空间名称	地址范围	说明
DATA	D:00H～D:7FH	片内 RAM 直接寻址
BDATA	D:20H～D:2FH	片内 RAM 位寻址
IDATA	I:00H～I:FFH	片内 RAM 间接寻址
XDATA	X:0000H～X:FFFFH	64KB 片外 RAM 数据区
CODE	C:0000H～C:FFFFH	64KB 片外 ROM 代码区
BANK0～BANK31	B0:0000～B0:FFFFH . . . B31:0000～B31:FFFFH	分组代码区，最大可扩展 32x64KB ROM

表 1 普通 51 单片机存储器空间分配表

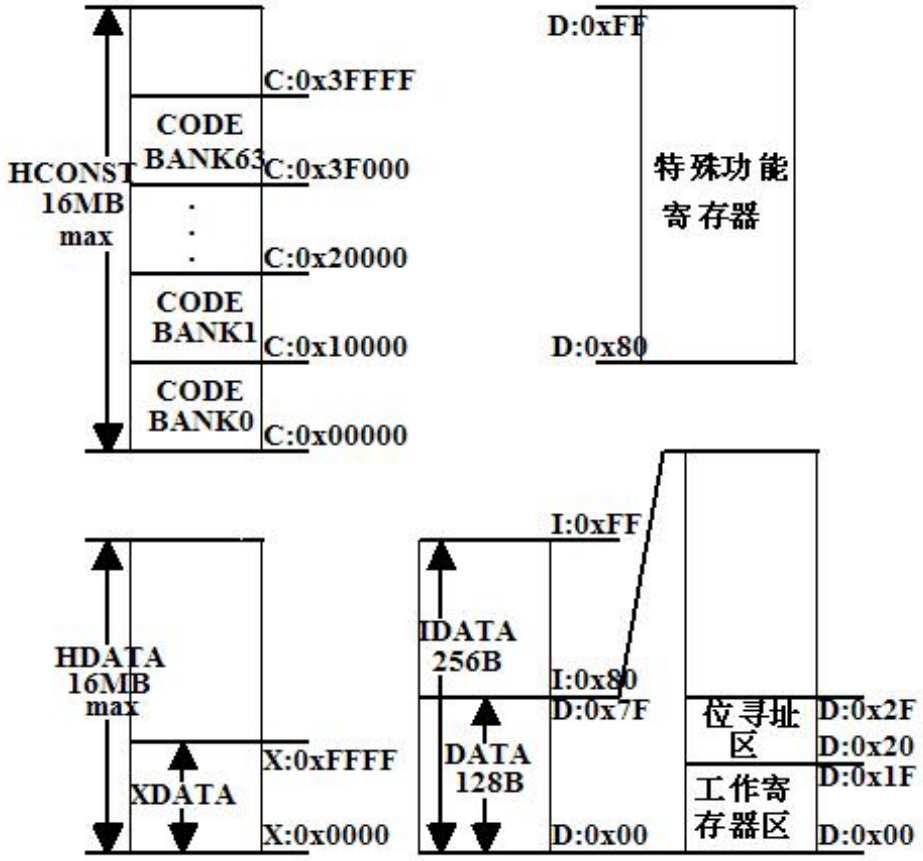

图 3 新型 51 单片机的扩展存储器组织结构

空间名称	地址范围	说明
DATA	D:00H～D:7FH	片内 RAM 直接寻址
BDATA	D:20H～D:2FH	片内 RAM 位寻址
IDATA	I:00H～I:FFH	片内 RAM 间接寻址
XDATA	X:0000H～X:FFFFH	64KB 片外 RAM 数据区
CODE	C:0000H～C:FFFFH	64KB 片外 ROM 代码区
HCONST(ECODE)	C:0000H～C:FFFFFFH	16MB 扩展片外 ROM 常数区(对 Dallas390 可用做代码区)
BANK0～BANK31	B0:0000～B0:FFFFH . . . B31:0000～B31:FFFFH	分组代码区，最大可扩展 32x64KB ROM

表 2 新型 80C51 单片机扩展存储器空间分配表

二、Cx51 应用程序编写的基本原则

C语言程序不要求具有固定的格式，但是在编写代码的时候还是应该遵循一定的规则。

1.首先要采用清晰的书写风格。在编写一个Cx51 程序时，对于while、for、do-while、if-else、switch-case等语句，或这些语句的嵌套组合，应采用“缩进”的书写方式。对于复合语句或函数，通常需要使用花括号”{}”，当语句嵌套较多时，容易产生花括号不匹配的情况。μVision的Edit下拉菜单中提供一个”Goto Matching Brace”选项，将光标放在某个花括号处，单击该选项，与之匹配括号中的内容将反白显示，适合用于检查各种括号的匹配情况。

2.对于一个表达式中各种运算执行的优先顺序不太明确或容易混淆的地方，应当采用圆括号”()”明确指定它们的优先顺序。对于程序中的函数，在使用之前应对函数的类型进行声明。对函数类型的声明必须保持与原来定义的函数类型一致，不一致时将导致编译出错(什么是声明，什么是定义？)。对于具有返回值的函数，使用return语句时，最好使用括号”()”将被返回的内容括起来，这样可使程序执行过程更清晰，便于理解和维护(标准C中提倡返回值void型的函数，也要使用return语句，这样使程序更”健壮”)。

3.一般情况下，对于普通的变量名或函数名采用小写字母表示，对于一些特殊变量名由预处理命令#define所定义的常数，则采用大写字母表示。为了帮助理解和记忆，变量或函数名中可带有下划线，例如，ext_int0、data_max等，但是以下划线”_”开头的变量或函数名通常保留为Cx51 编译系统说用。为了避免混淆，不要将下划线用做变量或函数的第一个字符。给变量或函数命名时，应按照”见名知义”的原则，例如”ext_int0”表示外部中断函数，”data_max”表示最大数据值等。

4.数组与指针语句具有十分密切的关系。对一个字符数组:char *name=”hello”，可以采用数组形式name[0]或指针形式*name来表示字符串的第一个字母h，两者在意义上是完全相同的。在实际程序设计中使用数组还是指针应视具体情况而定，一般来说，指针比较灵活简洁，而数组则比较直观，容易理解。

5.C语言是一种高级程序设计语言，与汇编语言不同，C语言提供了十分完备的规范化流程控制结构。因此在采用Cx51 设计单片机应用系统程序时，首先要注意尽可能采用结构化的程序设计方法，这样可使整个应用系统程序结构清晰，便于调试和维护。对于一个较大的应用程序，为了能够集中精力考虑各种具体问题，通常将整个程序按功能分成若干个模块，不同模块完成不同的功能。各个模块程序可以分别编写，甚至可以由多个人员分开编写。由于单个模块所完成的功能较简单，程序的设计和调试也相应要容易一些。对于一些常用的功能模块，可以作为一个应用程序库，以便以后直接调用。

6.在C语言中进行模块化程序设计是比较容易实现的，一个C语言函数可以认为是一个模块。所谓程序的模块化，不仅仅是要将整个程序划分成若干个功能模块，更重要的是还应当保持各个模块之间变量的相对独立性，即保持模块的独立性。在C语言的模块化编程过程中，如果过多地采用外部变量会减弱各个模块的独立性，因此为了保持整个程序具有较好的模块化结构，应尽量避免外部全局变量来传递信息，而应通过指定的参数来完成数据信息传递。对于不同的功能模块，可以分别制定相应的入口参数和出口参数，这样不会引起整个程序中变量管理的混乱。在μVision很容易实现模块化编程，只要将分别编写的各个程序模块

文件分别添加到项目中就可以了。强调一点，全局变量会削弱功能模块的独立性，也使得程序变得不安全。

7. 在程序设计过程中，对于经常使用的一些常数，如果将它们直接写到程序中去，一旦常数的数值发生变化，就必须逐个找出程序中的所有对应的常数，逐一进行修改，这样必然会降低程序的可维护性和可移植性。因此为了便于对整个程序进行修改维护，或为了帮助记忆，应当采用预处理命令的方式来定义常数。对于一些常用的常数，如π、e、EOF、TURE、FLASE以及不同型号 51 单片机中各种特殊功能寄存器和位地址等，应当集中起来放在一个头文件中进行定义，需要时再采用预处理命令#include将其包含到程序中去。这样不仅可以提高编译效率，而且还可以避免输入错误。

8. 一般来说，程序的执行效率主要取决于所采用算法的优劣和繁简。但对C语言而言，程序的执行效率在一定程度上还与程序的结构和设计方法有关。C语言具有十分丰富的运算符，合理地使用这些运算符可以设计出高效率的程序。例如，当条件表达式是由多个”&&”或”||”运算符连接在一起时，对于条件的判定总是从左至右逐个进行的，一旦条件满足时，就不再对后面其他条件进行判断。因此对于条件表达式的安排，应尽可能地将满足条件可能性较高的表达式放在整个条件式前面。合理使用中间变量往往也可以提高程序的执行效率。

三、Cx51 编译器的限制

由于历史的原因，Cx51 编译器具有如下限制：

- 最多只支持 19 级标准数据类型的修饰符，如数组描述符、间接操作符和函数操作符等。
- 名字最长为 255 个字符，但只有前 32 个字符有效。尽管C语言是对大小写敏感的，但同样的历史原因，目标文件中的名字是否大小写无关紧要。
- C语言程序中”case”语句变量的个数没有限制，仅由可用内存大小和函数的最大限制。
- 函数嵌套调用层最大为 10 层。
- 嵌套引用头文件最大数值为 9。
- 预处理器的条件编译指令最大嵌套深度为 20。
- 功能块({…})最大可嵌套 15 级。
- 宏最多可嵌套 8 级。
- 宏或函数调用中最多可传递 32 个参数。
- C语言语句或宏定义的一行中最多能写 2000 个字符。对于宏展开，其结果也不的超过 2000 个字符。

四、理解复杂的声明

随着学习C语言的深入，声明变得越来越复杂，理解的难度也随之增加。为了更好阅读和理解复杂的声明，这里介绍一条著名的**右-左法则**。使用这条法则时，从声明中间的标识符开始阅读，之后再轮流阅读该声明的右边和左边，直到阅读完毕完整的声明。

以下是例子：

1.考虑简单声明。

```
int x
```

这句话读作“x是#一个整数”

//井号(#)是一个占位符，不是需要考虑的实体，阅读时可忽略。

```
int     x      #
⊥       ⊥      ⊥
2       0      1
```

由于右边为空，直接阅读左边即可。

2.看一个指针声明的例子。这个例子读作“p是#一个指针，#是指向整数的”。//井号(#)是一个占位符，不是需要考虑的实体，阅读时可忽略。注意一下，即使右边是空的，也要依次从右往左看，直到看完左边所有的实体为止。

```
int     *      p      #      #
⊥       ⊥      ⊥      ⊥      ⊥
4       2      0      1      3
```


3.接下来的例子中，左右两边的实体个数一样多。

```
int    table     [4]
⊥       ⊥         ⊥
2       0         1
```

这句声明读作“table是一个数组，它含有 4 个整数”。

4.不管一个数组是几维的，该法则一概将数组看作一个实体。因此，下面这句多维数组的声明，应该读作“table是一个数组，它有 5x4 个整数”。

```
int    table     [4][5]
 ⊥      ⊥          ⊥
 2      0          1
```

5.下面的例子比较复杂，常常让人费解。这是一句关于整型指针数组的声明。

```
int     *     aryOfPtrs     [5]     #
⊥       ⊥         ⊥          ⊥      ⊥
4       2         0          1      3
```

这句读作“aryOfPtrs是一个指针数组，它含有 5 个指针，是指向整数的”。

6.在上一句声明中加上一个圆括号，就变成一句指向数组的指针声明。这样，这个指针指向整个数组，而不是指向数组中的某个元素。

```
int    ( * aryOfPtrs   # )     [5]
⊥      ⊥     ⊥       ⊥        ⊥
4      2     0       1        3
```

这句声明读作“aryOfPtrs是一个指针，它指向一个数组，该数组包含 5 个整数”。//也就是常说的行指针。

7. 下面看一个函数的声明。这是一句简单的函数原型的声明，该函数返回一个整数。

```
int    doIt     (…)
⊥      ⊥        ⊥
2      0        1
```

这句声明读作“doIt是一个函数，它返回一个整数”。

8. 再看一个函数声明，该函数返回一个整数型指针。

```
int    *    doIt    (int)     #
⊥      ⊥    ⊥       ⊥         ⊥
4      2    0       1         3
```

这个例子读作“doIt是一个函数，它返回一个整型指针”。

9. 最后，看一个指针与函数的另一种结合形式“函数指针”。

```
int    ( *   doIt   # )   (…)
⊥      ⊥     ⊥     ⊥      ⊥
4      2     0     1      3
```

这句声明应该读作“doIt是一个指针，它指向一个传递参数为(…)，返回值为整数的函数”。

通过以上例子可以看到，同样的符号组成不同的声明是跟结合性、优先级有很大关系的。

五、代码的优化

Cx51 编译器生成代码的效率依赖于以下几个方面。

1.存储器模式

对于代码大小和执行速度影响最大的是存储器模式。使用small模式对源程序进行编译可以生成长度最小和速度最快的代码。在small模式下，所有变量(除非在声明时指定了存储器类型)都将定位在 51 单片机的片内RAM中，而访问片内RAM的速度最快(一般为 1~2 个机器周期)，同时生成的代码要比采用compact或large模式时小得多。

2.变量定位

需要经常访问的数据对象应存放在片内RAM中，51 单片机访问片内RAM的速度要比访问外部XRAM快得多。片内RAM由工作寄存器组、位寻址区、堆栈区以及其他用户采用DATA类型定义的变量共享。由于片内RAM容量的限制(128～256 字节)，用户程序中的所有变量不可能全部放在片内RAM中，此时必须将某些放入其他存储器空间。有以下两种方法供参考。

一种方法是改变存储器模式让编译器自动完成变量定位，这种方法最简单，但改变存储器模式可能导致生成代码长度加大，执行效率低。

另一种方法是由用户根据需要决定哪些变量可以存放在外部XRAM中，并将这些变量定义为具有XDATA存储类型。通常字符串缓冲区和大型数组可以被定义为XDATA存储器类型，而不会对代码长度和执行效率造成太大的影响。

3.尽可能使用最小数据类型

51 系列单片机为 8 位CPU，它对单字节数据(如char、unsigned char类型)的操作要比多字节数据(如int、long类型)方便得多。因此建议尽可能使用最小数据类型。如果不是运算式的特殊要求，就不要进行int类型的转换。这一点可用乘法运算为例加以说明：两个char类型数据相乘恰好与 51 指令“MUL AB”相符，而如果采用long型数据相乘则需要调用Cx51 编译器的库函数。当然，这是在结果不溢出的情况下使用的。

4.尽可能使用unsigned数据类型

51 单片机不能直接支持带符号的运算，对于带符号的操作Cx51 编译器必须产生更多与之相关的代码，因此尽可能采用unsigned类型数据将使生成的代码体积更小。

5.尽可能使用局部变量

只要有可能，对于循环和临时变量运算应当采用局部变量。Cx51 编译器在进行优化处理时总是希望用工作寄存器来存放局部变量，而对工作寄存器的存取操作是最快的，通常采用unsigned char或unsingned int类型变量能获得更好的效果。

6.移位操作的使用

移位操作对于无符号数是安全的，无论是左移还是右移都是逻辑移位，被移走的位用 0 填充。这可以用来代替一些x2 或者/2 的乘除法运算。但对于有符号数来说，由于不同编译器采用的移位策略不一样，或是逻辑移位或是算数移位，因此不具有可移植性。关于移位的讨论网上有很多，可参考之。

六、Cx51 编译器常见警告与错误信息

小提示:按Ctrl+F键可以直接查找:

1. Warning 280:'i':unreferenced local variable
说明局部变量i 在函数中未作任何的存取操作。解决方法是删除函数中i 变量的声明。

2 Warning 206:'Music3':missing function-prototype
说明Music3( )函数未作声明或未作外部声明所以无法给其他函数调用。解决方法将定义void Music3(void)写在程序的最前端作声明。如果是其他文件的函数则要写成extern void Music3(void),即作外部声明。

3 Compling :C:\51\MANN.C
Error:318:can't open file 'beep.h'
说明在编译C:\51\MANN.C 程序过程中，由于main.c 用了指令#include "beep.h",但却找不到beep.h所致。解决方法编写一个beep.h 的包含档并存入到c:\51 的工作目录中。

4 Compling:C:\51\LED.C
Error 237:'LedOn':function already has a body
说明LedOn()函数名称重复定义，即有两个以上一样的函数名称。解决方法修改其中的一个函数名称使得函数名称都是独一无二的。

5. *** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
SEGMENT: PR _COMPARE TESTLCD
说明:程序中有些函数(或片段)以前(调试过程中)从未被调用过，或者根本没有调用它的语句。这条警告信息前应该还有一条信息指示出是哪个函数导致了这一问题。只要做点简单的调整就可以。不理它也没什么大不了的。
解决方法:去掉COMPARE()函数或利用条件编译#if …..#endif,可保留该函数并不被编译。

6 ***WARNING 6 :XDATA SPACE MEMORY OVERLAP FROM : 0025H TO: 0025H
说明外部存储器ROM的 0025H 重复定义地址。解决方法外部存储器ROM的定义如下：pdata unsigned char XFR_ADC _at_0x25，其中XFR_ADC变量的名称为 0x25,请检查是否有其它的变量名称也是定义在 0x25 处并改正它。

7 WARNING 206:'DelayX1ms': missing function-prototype
C:\51\INPUT.C
Error 267 :'DelayX1ms ':requires ANSI-style prototype C:\51\INPUT.C
说明程序中有调用DelayX1ms函数，但该函数没定义，即未编写程序内容或函数已定义但未作声明。解决方法编写DelayX1ms的内容。编写完后也要作声明或作外部声明。可在delay.h 的包含档声明成外部以便其它函数调用。

8 ***WARNING 1:UNRESOLVED EXTERNAL SYMBOL
SYMBOL:MUSIC3
MODULE:C:\51\MUSIC.OBJ(MUSIC)
***WARNING 2:REFERENCE MADE TO UNRESOLVED EXTERNAL
SYMBOL:MUSIC3
MODULE:C:\51\MUSIC.OBJ(MUSIC)
ADDRESS:0018H
说明程序中有调用MUSIC函数，但未将该函数的含扩文件C加入到工程文件Prj作编译和连接。解决方法：设MUSIC3 函数在MUSIC.C里，将MUSIC.C添加到工程文件中。

9 ***ERROR 107:ADDESS SPACE OVERFLOW
SPACE: DATA
SEGMENT: _DATA_GOUP_
LENGTH: 0018H
***ERROR 118: REFERENCE MADE TO ERRONEOUS EXTERNAL
SYMBOL: VOLUME
MODULE: C:\51\OSDM.OBJ (OSDM)
ADDRESS: 4036H
说明data 存储空间的地址范围为 0~0x7f,当全局变量数目和函数里的局部变量。如果存储模式设为SMALL，则局部变量先使用工作寄存器R2~R7 作暂存；当存储器不够用时则会以data型存储。别的空间作暂存的个数超过 0x7f 时就会出现地址不够的现象。解决方法将以data类型定义的全局变量，修改为idata类型的定义说明。(原文如此，实在不通。官网上也没有编号为 118 的错误类型)

10.***WARNING L15: MULTIPLE CALL TO SEGMENT
SEGMENT: PR _WRITE_GMVLX1_REG D_GMVLX1
CALLER1: PR VSYNC_INTERRUPT MAIN
CALLER2: C_C51STARTUP
***WARNING L15: MULTIPLE CALL TO SEGMENT
SEGMENT: PR _SPI_SEND_WORD D_SPI
CALLER1: PR VSYNC_INTERRUPT MAIN
CALLER2: C_C51STARTUP
***WARNING L15: MULTIPLE CALL TO SEGMENT
SEGMENT: PR SPI_RECEIVE_WORD D_SPI
CALLER1: PR VSYNC_INTERRUPT MAIN
CALLER2: C_C51STARTUP
该警告表示连接器发现有一个函数可能会被主函数和一个中断服务程序(或者调用中断服务程序的函数)同时调用，或者同时被多个中断服务程序调用。
出现这种问题的原因之一是这个函数是不可再入函数，当该函数运行时它可能会被一个中断打断，从而使得结果发生变化并可能会引起一些变量形式的冲突(即引起函数内一些数据的丢失，可重入性函数在任何时候都可以被ISR 打断，一段时间后又可以运行,但是相应数据不会丢失)。
原因之二是用于局部变量和变量(暂且这样翻译，arguments，[自变量，变元一数值，用于确定程序或子程序的值])的内存区被其他函数的内存区所覆盖，如果该函数被中断,则它的内存区就会被使用,这将导致其他函数的内存冲突。

例如，第一个警告中函数WRITE_GMVLX1_REG在D_GMVLX1.C或者D_GMVLX1.A51 被定义，它被一个中断服务程序或者一个调用了中断服务程序的函数调用了，调用它的函数是VSYNC_INTERRUPT,在MAIN.C 中。

解决方法:
如果你确定两个函数决不会在同一时间执行(该函数被主程序调用并且中断被禁止)，并且该函数不占用内存(假设只使用寄存器)，则你可以完全忽略这种警告。
如果该函数占用了内存,则应该使用连接器(linker)OVERLAY 指令将函数从覆盖分析(overlay analysis)中除去，例如:
OVERLAY (PR _WRITE_GMVLX1_REG D_GMVLX1 ! *)
上面的指令防止了该函数使用的内存区被其他函数覆盖。如果该函数中调用了其他函数，而这些被调用在程序中其他地方也被调用，你可能会需要也将这些函数排除在覆盖分析(overlay analysis)之外。这种OVERLAY指令能使编译器除去上述警告信息。

如果函数可以在其执行时被调用，则情况会变得更复杂一些。这时可以采用以下几种方法:
1.主程序调用该函数时禁止中断，可以在该函数被调用时用#pragma disable语句来实现禁止中断的目的。必须使用OVERLAY 指令将该函数从覆盖分析中除去。
2.复制两份该函数的代码,一份到主程序中,另一份复制到中断服务程序中。

3.将该函数设为可再入型。例如:

```
void myfunc(void) reentrant
{
...
}
```

这种设置将会产生一个可重入堆栈。该堆栈被被用于存储函数值和局部变量，用这种方法时重入堆栈必须在STARTUP.A51 文件中配置。这种方法消耗更多的RAM并会降低重入函数的执行速度。

七、Cx51 编译器错误信息中文翻译

Ambiguous operators need parentheses
不明确的运算需要用括号括起

Ambiguous symbol ``xxx``
不明确的符号

Argument list syntax error
参数表语法错误

Array bounds missing
丢失数组界限符

Array size toolarge
数组尺寸太大

Bad character in paramenters
参数中有不适当的字符

Bad file name format in include directive
包含命令中文件名格式不正确

Bad ifdef directive synatax
编译预处理ifdef 有语法错

Bad undef directive syntax
编译预处理undef 有语法错

Bit field too large
位字段太长

Call of non-function
调用未定义的函数

Call to function with no prototype
调用函数时没有函数的说明

Cannot modify a const object
不允许修改常量对象

Case outside of switch
漏掉了case 语句

Case syntax error
Case 语法错误

Code has no effect
代码不可述不可能执行到

Compound statement missing{
分程序漏掉"{"

Conflicting type modifiers
不明确的类型说明符

Constant expression required
要求常量表达式

Constant out of range in comparison
在比较中常量超出范围

Conversion may lose significant digits
转换时会丢失意义的数字

Conversion of near pointer not allowed
不允许转换近指针

Could not find file ``xxx``
找不到XXX 文件

Declaration missing ;
说明缺少";"

Declaration syntax error
说明中出现语法错误

Default outside of switch
Default 出现在switch 语句之外

Define directive needs an identifier
定义编译预处理需要标识符

Division by zero
用零作除数

Do statement must have while
Do-while 语句中缺少while 部分

Enum syntax error
枚举类型语法错误

Enumeration constant syntax error
枚举常数语法错误

Error directive :xxx
错误的编译预处理命令

Error writing output file
写输出文件错误

Expression syntax error
表达式语法错误

Extra parameter in call
调用时出现多余错误

File name too long
文件名太长

Function call missing)
函数调用缺少右括号

Fuction definition out of place
函数定义位置错误

Fuction should return a value
函数必需返回一个值

Goto statement missing label
Goto 语句没有标号

Hexadecimal or octal constant too large
16 进制或 8 进制常数太大

Illegal character ``x``
非法字符x

Illegal initialization
非法的初始化

Illegal octal digit
非法的 8 进制数字

Illegal pointer subtraction
非法的指针相减

Illegal structure operation
非法的结构体操作

Illegal use of floating point
非法的浮点运算

Illegal use of pointer
指针使用非法

Improper use of a typedefsymbol
类型定义符号使用不恰当

In-line assembly not allowed
不允许使用行间汇编

Incompatible storage class
存储类别不相容

Incompatible type conversion
不相容的类型转换

Incorrect number format
错误的数据格式

Incorrect use of default
Default 使用不当

Invalid indirection
无效的间接运算

Invalid pointer addition
指针相加无效

Irreducible expression tree
无法执行的表达式运算

Lvalue required
需要逻辑值 0 或非 0 值

Macro argument syntax error
宏参数语法错误

Macro expansion too long
宏的扩展以后太长

Mismatched number of parameters in definition
定义中参数个数不匹配

Misplaced break
此处不应出现break 语句

Misplaced continue
此处不应出现continue 语句

Misplaced decimal point
此处不应出现小数点

Misplaced elif directive
不应编译预处理elif

Misplaced else
此处不应出现else

Misplaced else directive
此处不应出现编译预处理else

Misplaced endif directive
此处不应出现编译预处理endif

Must be addressable
必须是可以编址的

Must take address of memory location
必须存储定位的地址

No declaration for function ``xxx``
没有函数xxx 的说明

No stack
缺少堆栈

No type information
没有类型信息

Non-portable pointer assignment
不可移动的指针(地址常数)赋值

Non-portable pointer comparison
不可移动的指针(地址常数)比较

Non-portable pointer conversion
不可移动的指针(地址常数)转换

Not a valid expression format type
不合法的表达式格式

Not an allowed type
不允许使用的类型

Numeric constant too large
数值常量太大

Out of memory
内存不够用

Parameter ``xxx`` is never used
能数xxx 没有用到

Pointer required on left side of ->
符号->的左边必须是指针

Possible use of ``xxx`` before definition
在定义之前就使用了xxx(警告)

Possibly incorrect assignment
赋值可能不正确

Redeclaration of ``xxx``
重复定义了xxx

Redefinition of ``xxx`` is not identical
xxx 的两次定义不一致

Register allocation failure
寄存器定址失败

Repeat count needs an lvalue
重复计数需要逻辑值

Size of structure or array not known
结构体或数给大小不确定

Statement missing ;
语句后缺少";"

Structure or union syntax error
结构体或联合体语法错误

Structure size too large
结构体尺寸太大

Sub scripting missing]
下标缺少右方括号

Superfluous & with function or array
函数或数组中有多余的"&"

Suspicious pointer conversion
可疑的指针转换

Symbol limit exceeded
符号超限

Too few parameters in call
函数调用时的实参少于函数的参数

Too many default cases Default
太多(switch 语句中一 个)

Too many error or warning messages
错误或警告信息太多

Too many type in declaration
说明中类型太多

Too much auto memory in function
函数用到的局部存储太多

Too much global data defined in file
文件中全局数据太多

Two consecutive dots
两个连续的句点

Type mismatch in parameter xxx
参数xxx 类型不匹配

Type mismatch in redeclaration of ``xxx``
xxx 重定义的类型不匹配

Unable to create output file ``xxx``
无法建立输出文件xxx

Unable to open include file ``xxx``
无法打开被包含的文件xxx

Unable to open input file ``xxx``
无法打开输入文件xxx

Undefined label ``xxx``
没有定义的标号xxx

Undefined structure ``xxx``
没有定义的结构xxx

Undefined symbol ``xxx``
没有定义的符号xxx

Unexpected end of file in comment started on line xxx
从xxx 行开始的注解尚未结束文件不能结束

Unexpected end of file in conditional started on line xxx
从xxx 开始的条件语句尚未结束文件不能结束

Unknown assemble instruction
未知的汇编结构

Unknown option
未知的操作

Unknown preprocessor directive: ``xxx``
不认识的预处理命令xxx

Unreachable code
无路可达的代码

Unterminated string or character constant
字符串缺少引号

User break
用户强行中断了程序

Void functions may not return a value
Void 类型的函数不应有返回值

Wrong number of arguments
调用函数的参数数目错

``xxx`` not an argument
xxx 不是参数

``xxx`` not part of structure
xxx 不是结构体的一部分

xxx statement missing (
xxx 语句缺少左括号

xxx statement missing)
xxx 语句缺少右括号

xxx statement missing ;
xxx 缺少分号

xxx`` declared but never used
声明了xxx 但没有使用

xxx`` is assigned a value which is never used
给xxx 赋了值但未用过

Zero length structure
结构体的长度为零