原创翻译，转载注明
作者：小楼听雨
来源：http://www.xl7y.tk/

Keil C 编程教程：引言

►引言

用 c 语言为微控制器编程正在变的越来越普通，通常用汇编建立一个应用比用 c 语言要难的多，因此掌握嵌入式 c 编程是非常重要的。由于我们使用 Keil C51 编译器，也常常称之为 Keil C。

►关键字

Keil C 编译器添加的一些关键字：

_at_	far	sbit
alien	idata	sfr
bdata	interrupt	sfr16
bit	large	small
code	pdata	_task_
compact	_priority_	using
data	reentrant	xdata

data/idata:

描述：变量将被存储在控制器内部 ram 中。

example:

CODE:

```
unsigned char data x;
//or
unsigned char idata y;
```

bdata:

描述：变量被存储在可位寻址的内部 ram 中。

CODE:

```
unsigned char bdata x;
//变量 x 的每一位可按以下的方式存取
x ^ 1 = 1; //设置 x 的第一位
x ^ 0 = 0; //清除 x 的第零位
```

xdata:

描述：变量将被存储在控制器外部 ram 中。

example:

CODE:

```
unsigned char xdata x;
```

code:

描述：这个关键字是用来将常量存储在 rom 中。假设你有一个大的字符串，而且这个字符串在程序中不会再被改变，为这个字符串浪费 ram 是很愚蠢的事，因此我们要像下面的例子一样运用 code 关键字。

example:

CODE:

```
unsigned char code str="this is a constant string";
```

pdata:

描述：这个关键字将会使变量存储在分页寻址 ram 中，它运用不是很频繁。

example:

CODE:

```
unsigned char pdata x;
```

_at_:

描述：用来将变量存储在 ram 的指定位置。

example:

CODE:

```
unsigned char idata x _at_ 0x30;
//变量 x 将会存储在内部 ram 的 0x30 处
```

sbit:

描述：这个关键字用来定义 SFR（特殊功能寄存器）的某一位。

example:

CODE:

```
sbit Port0_0 = 0x80;
//地址 0x80 被定义为特殊位 Port0_0
```

sfr:

描述：sfr 被用来定义一个 8 位的特殊功能寄存器。

example:

CODE:

```
sfr Port1 = 0x90;
//地址 0x90 被定义为特殊功能寄存器 Port1
```

sfr16:

描述：用来定义两个连续的 8 位特殊功能寄存器。

example:

CODE:

```
sfr16 DPTR = 0x82;
//开始于 0x82 的 16 位特殊功能寄存器
```

using:

描述：这个关键字位某个函数定义寄存器组，用户可以指定 0-3 的寄存器组。

example:

CODE:

```
void function () using 2{
// code
}
//名为 function 的函数在执行代码的时候使用寄存器组 2
```

interrupt:

描述：这个关键字将会告诉编译器被描述的函数是一个终端服务程序。C51 编译器支持最多 32 个中断源（0-31），使用下面的中断向量地址来决定中断号。

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

example:

CODE:

```
void External_Int0() interrupt 0{
//code
}
```

►存储器模式

用户有三种类型的存储器模式变量：

1. **Small:** 所有变量都在内部 ram 中。
2. **Compact:** 参数及局部变量放入分页外内部存贮区（最大 256 bytes）。
3. **large:** 所有变量都在外部 ram 中，使用 DPTR 存取。

根据我们硬件的配置我们能够向下面这样指定存储器模式：

CODE:

```
//For Small Memory model
#pragma small
//For Compact memory model
#pragma compact
//For large memory model
#pragma large
```

原文： keil C Programming Tutorial: Introduction
http://www.8051projects.net/keil-c-programming-tutorial/introduction.php

Keil C 编程教程：指针

► Keil C 中的指针

Keil C 中的指针类似于标准 C，可以完成标准 C 中指针的所有运算。另外，Keil C 扩展了它的运算去满足 8051 微控制器架构，Keil C 提供两种不同类型的指针：

1. 一般指针
2. 存储器指针

►一般指针

一般指针同标准 C 中的指针声明是一样的

CODE:

```
char *ptr;      //字符型指针
int *num;       //整型指针
```

一般指针用三个字节存储。第一个存储存储器类型，第二个第三个分别是高位偏移和低位偏移。一般指针能够存取任何变量，无论该变量在什么位置。

►存储器指针

存储器指针在指针说明时就指定了存储类型，例如：

CODE:

```
char data *c;
//指向一个存储在 ram 中的字符

char xdata *c1;
//指向一个存储在外部 ram 中的字符

char code *c2;
//指向一个存储在 rom 中的字符
```

由于存储器指针在编译时已经指定存储类型，存储器指针存放时需要一个（idata, data, bdata 以及 pdata 指针）或两个字节（code 和 xdata 指针）。

Keil C 为存储器指针生成的代码比为一般指针生成的代码执行速度要快，这是因为存储器指针指向的内容类型在编译的时候是已知的而不是在运行的时候，编译器使用这些信息优化存取过程，因此如果你追求的是速度至上，那么推荐你用存储器指针。

一般指针和寄存器指针都可以在它们声明时指定存放区域。例如：

CODE:

```
//一般指针
char * idata ptr;
//字符型指针，该指针存储在内部 ram 中
int * xdata ptr1;
//整型指针，该指针存储在外部 ram 中

//存储器指针
```

```
char idata * xdata ptr2;
//指向内部ram的字符型指针，该指针存储在外部ram中。
int xdata * data ptr3;
//指向外部ram的字符型指针，该指针存储在ram中。
```

原文： keil C Programming Tutorial: pointers

http://www.8051projects.net/keil-c-programming-tutorial/pointers.php

原创翻译，转载注明
作者：小楼听雨
来源：http://www.xl7y.tk/

Keil C 编程教程：函数

► Keil C 中的函数

Keil C 编译器对标准 c 的函数声明进行的扩展如下：

- 为中断过程指定函数
- 选择要使用的寄存器组
- 选择存储器模式

►函数定义

[Return_type] Fucntion_name ([Arguments]) [Memory_model] [reentrant] [interrupt n] [using n]

Return_type: 函数返回值的类型，如果没有指定函数返回值类型，默认为 int 型。

Function_name: 函数名。

Arguments: 传递给函数的参数。

可选项：

这些选项你可以再函数定义的时候指定。

Memory_model:为函数指明存储器模式（**Large, Compact, Small**），例如：

CODE:

```
int add_number (int a, int b) Large
```

reentrant: 指明这个函数是可重入的还是递归的，关于这个选项稍后教程会进一步解释。

interrupt: 指明这个函数是一个中断服务程序，关于这个选项稍后教程会进一步解释。

using: 指定函数执行过程中使用的寄存器组，8051 架构有三个寄存器组，使用 using n 指定寄存器组 n。

CODE:

```
void function_name () using 2{ //函数使用寄存器组 2
        //函数代码
}
```

► 中断服务程序

可以用关键字 **interrupt** 和中断号将一个函数指定为中断服务程序，中断号指明终端服务程序所要处理的中断。

下面的表格描述了默认的中断：

Interrupt Number	Interrupt Description	Address
0	EXTERNAL INT 0	0003h
1	TIMER/COUNTER 0	000Bh
2	EXTERNAL INT 1	0013h
3	TIMER/COUNTER 1	001Bh
4	SERIAL PORT	0023h

由于 8051 生产商创造了新的器件，加入了更多的中断。Keil C51 支持 32 个中断（0-31），使用下表的中断向量入口地址去决定中断号。

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

中断函数可以像下面这样定义：

CODE:

```
void isr_name (void) interrupt 2 {
//中断程序代码
}
```

注意中断服务程序没有参数，返回类型只能是空。

►可重入函数

在 ANSI C 中有递归函数，在嵌入式C中为了满足同样的需求，我们有可重入函数，这些函数可以被递归地调用，被两个或两个以上的进程同时调用。
你现在可能在想，为什么要定义可重入函数？

你必须知道这些函数在递归调用的时候是怎样工作的，当一个函数在执行的时候，会有一些和它相关的动态数据，比如局部变量，当函数被递归地调用或者两个以上的进程同时调用时，CPU 必须保持局部变量的状态。

可重入函数定义如下：

CODE

```
void function_name (int argument) reentrant {
        //函数代码
}
```

每一个可重入函数都有相应的栈，由 startup.A51 文件定义。存储器模式决定了可重入函数的堆栈存储的区域。

- Small 模式可重入函数堆栈在内部 ram 中。
- Compant 模式可重入函数堆栈在分页寻址 ram 中。
- Large 模式可重入函数堆栈在外部 ram 中。

►实时任务

Keil C51 支持实时系统（RTOS）RTX51 Full 和 RTX51 Tiny。实时任务函数使用_task_和_priority_关键字定义，_task_定义一个实时任务函数，_priority_指明任务的优先级。

函数定义如下：

CODE:

```
void func (void) _task_ Number _priority_ Priority
{
        //code
}
```

Number:任务 ID，RTX51 Full 是 0 到 255，RTX51 Tiny 0 到 15。

Priority:任务优先级。

实时任务函数必须被定义为返回值为空且参数为空（没有参数传递给任务函数）。

原文： **keil C Programming Tutorial: Functions**

http://www.8051projects.net/keil-c-programming-tutorial/functions.php

►C 编程基础

正如我们前面讨论的那样，Keil C 与普通 C 语言编程并没有很大的区别。如果你了解汇编，写一个 C 程序更不会有问题，你唯一要记住的就是忘记控制器的通用寄存器和累加器，但是不要忘记端口和外围设备以及相关的寄存器。

在标准 C 中，所有的程序至少有一个 main 函数作为应用程序的入口点。与此相似，Keil C 中也有 main 函数，你所有的工作都在函数中完成，让我们进一步探究一下程序的工作流程。

当你在 PC 上运行 C 程序时，对于操作系统你的程序可以看做是一个子程序或进程，因此当程序退出后，将返回到操作系统。然而，在嵌入式 C 中没有操作系统，你必须确保你的程序一直在运行，从不退出。用 while(1) 或者 for(;;)进行无休止的循环即可实现。下面展示了一个基础 C 编程的轮廓：

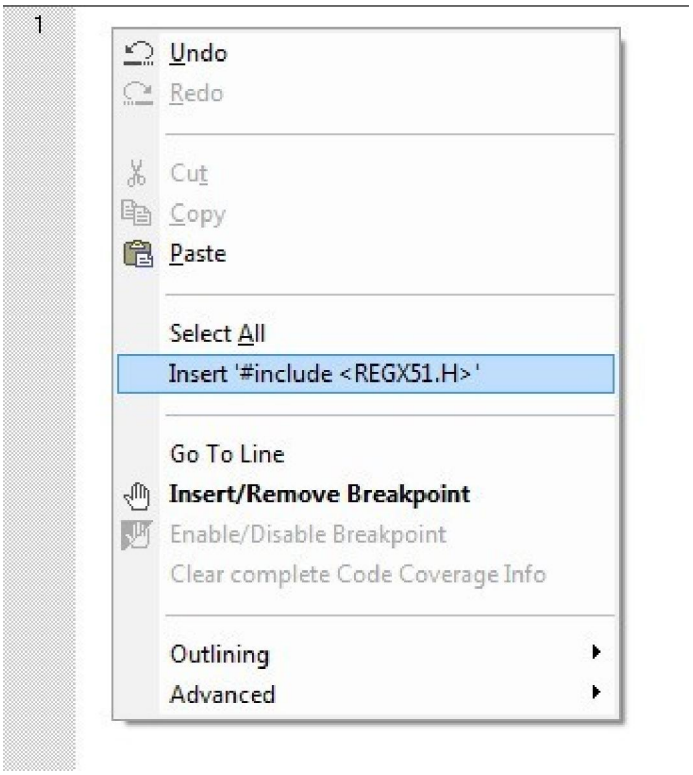
CODE:

```
void main(){
```

```
//初始化代码
        while(1){
                //while 1 循环
                  //这个循环中加入你需要无休止循环的代
码        }
}
```

当你为特定的控制器编写程序时，首先需要为该控制器添加头文件。项目建立后，为你的项目添加 C 文件，你唯一要做的就是在编辑窗口点击鼠标右键，它将会显示正确的头文件。

下图展示了添加头文件的窗口内容：



►编写特定的硬件代码

在编写硬件代码的时候，我们使用硬件设备如端口、定时器和串口等。不要忘记添加你使用的控制器的头文件，否则无法操作相关设备的寄存器。

我们编写一段简单的代码让端口 1 的引脚 1 上的 LED 闪烁：

CODE:

```
#include <REGx51.h> // 89C51 头文件
```

```
void main(){
        //main function starts
        unsigned int i;
        //Initializing Port1 pin1
        P1_1 = 0; //Make Pin1 o/p
        while(1){
                //Infinite loop main application
                //comes here
                for(i=0;i<1000;i++)
                        ; //delay loop
                P1_1 = ~P1_1;
                //complement Port1.1
                 //this will blink LED connected on
Port1.1
        }
}
```

现在你可以试验大量的程序了，“ 熟能生巧 “ 。

下一部分我们会学习 C 和汇编混合编程。

原文：keil C Programming Tutorial: Writing simple C program in Keil

http://www.8051projects.net/keil-c-programming-tutorial/writing-simple-c-program.php

Keil C 编程教程：C 与汇编混合编程

►混合编程

你很容易就能将你的程序和 8051 汇编代码连接起来，只要你遵循一些规则，你就能从 C 语言代码中调用汇编程序，反之亦然。汇编模块中声明的公共变量在 C 程序中依然可用。

有很多理由比如更快的执行速度或者使用汇编存取特殊功能寄存器等会让你选择汇编，在本部分教程我们将讨论如何让汇编程序带有 C 程序接口。

对于任何被 C 程序调用的汇编程序，你必须知道怎样传递参数给函数以及怎样获取函数的返回值。

►段命名

C51 编译器为每一个程序都产生了对象如程序代码程序数据和常变量，这些对象存储在存储器的代码单元和数据单元中，段名称在 C51 中是标准的，因此每一个汇编程序都需要遵循这些约定。

段名包括模块名字是对象所在源代码的名字，每一个段都有一个与段存储类型相匹配的前缀，前缀被前后连个问号包围。下面是一份标准的段命名前缀：

Segment Prefix	Memory Type	Description
?PR?	program	Executable program code
?CO?	code	Constant data in program memory
?BI?	bit	Bit data in internal data memory
?BA?	bdata	Bit-addressable data in internal data memory
?DT?	data	Internal data memory
?FD?	far	Far memory (RAM space)
?FC?	const far	Far memory (constant ROM space)
?ID?	idata	Indirectly-addressable internal data memory
?PD?	pdata	Paged data in external data memory
?XD?	xdata	Xdata memory (RAM space)
?XC?	const xdata	Xdata memory (constant ROM space)

►数据对象

数据对象就是那些你在 C 程序中声明的变量和常量，C51 编译器为每一类型变量的声明都生成了独立的段。下面的表列出了为不同类型变量产生的段名。

Segment Name	Description
?BA?*module_name*	Bit-addressable data objects
?BI?*module_name*	Bit objects
?CO?*module_name*	Constants (strings and initialized variables)
?DT?*module_name*	Objects declared in data
?FC?*module_name*	Objects declared in const far (requires OMF2 directive)
?FD?*module_name*	Objects declared in far (requires OMF2 directive)
?ID?*module_name*	Objects declared in idata
?PD?*module_name*	Objects declared in pdata
?XC?*module_name*	Objects declared in const xdata (requires OMF2 directive)
?XD?*module_name*	Objects declared in xdata

►程序对象

程序对象包含了 C51 编译器为 C 程序函数生成的代码。每一个函数在原模块中被分配了一个独立代码段，使用的命名规则为**?PR?*function_name*?*module_name*。**例如 uart.c 中叫做 send_char 的函数段名将会是 **?PR?SEND_CHAR?UART。**

C51 编译器创建独立的代码段为伴随着函数体而声明的变量。下表给出了不同存储器模型的段命名规则。

Small model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**data**	?DT?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

Compact model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**pdata**	?PD?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

Large model segment naming conventions		
Information	**Segment Type**	**Segment Name**
Program code	**code**	?PR?*function_name*?*module_name*
Local variables	**xdata**	?XD?*function_name*?*module_name*
Local bit variables	**bit**	?BI?*function_name*?*module_name*

段名随着函数类型的不同需轻微的修改（无参，有参和可重入函数）。下表解释了段名：

Declaration	Symbol	Description
void func (void) …	FUNC	Names of functions that have no arguments or whose arguments are not passed in registers are transferred to the object file without any changes. The function name is converted to uppercase.
void func1 (char) …	_FUNC1	For functions with arguments passed in registers, the underscore character ('_') is prefixed to the function name. This identifies those functions that transfer arguments in CPU registers.
void func2 (void) reentrant …	_?FUNC2	For functions that are reentrant, the string "_?" is prefixed to the function name. This is used to identify reentrant functions.

下一部分我们将学习 C 程序函数对汇编函的调用以及汇编函数对 C 函数的调用。

原文：keil C Programming Tutorial: C and Assembly together
http://www.8051projects.net/keil-c-programming-tutorial/mix-c-and-assembly.php

原创翻译，转载注明
作者：小楼听雨
来源：http://www.xl7y.tk/

Keil C 编程教程：连接 C 程序与汇编

►函数参数

C51 使用寄存器和存储器位置传递参数，缺省情况下多达三个参数可以在寄存器里传递，如果再多将会在存储器固定位置传递。看可以用

关键字 **NOREGPARMS** 禁止在寄存器里传递，被禁止后或者是参数太多，参数将会在存储器固定位置传递。

►寄存器里传递参数

C 程序会在寄存器或在存储器固定位置传递参数，下面的表格显示了参数传递时的寄存器使用情况。

Arg Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7 (MSB in R6, LSB in R7)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
2	R5	R4 & R5 (MSB in R4, LSB in R5)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
3	R3	R2 & R3 (MSB in R2, LSB in R3)		R1—R3 (Mem type in R3, MSB in R2, LSB in R1)

下面的例子更加清晰地阐释了参数传递的方法：

Declaration	Description
`func1 ( int a)`	The first and only argument, ***a***, is passed in registers R6 and R7.
`func2 ( int b, int c, int *d)`	The first argument, ***b***, is passed in registers R6 and R7. The second argument, ***c***, is passed in registers R4 and R5. The third argument, ***d***, is passed in registers R1, R2, and R3.
`func3 ( long e, long f)`	The first argument, ***e***, is passed in registers R4, R5, R6, and R7. The second argument, ***f***, cannot be located in registers since those available for a second parameter with a type of long are already used by the first argument. This parameter is passed using fixed memory locations.
`func4 ( float g, char h)`	The first argument, ***g***, passed in registers R4, R5, R6, and R7. The second parameter, ***h***, cannot be passed in registers and is passed in fixed memory locations.

►存储器固定位置传递

参数使用段名传递参数给汇编程序

?*function_name*?BYTE :除了 bit 型参数其他所有参数都定义在该段.

?*function_name*?BIT：Bit 型参数定义在该段

所有参数即使是使用寄存器传递的参数被分配到该空间，参数被按照在各自段中定义的顺序存储。

参数传递所使用的存储器固定位置可能在内部 ram 或是外部 ram 中，这取决于使用的存储器模式。**SMALL** 模式使用内部 ram，从而更加高效。**COMPACT** 和 **LARGE** 模式使用外部 ram。

►函数返回值

函数返回值总是使用 CPU 寄存器传递。下面的表格列出了返回值和寄存器使用类型的对应关系：

Return Type	Register	Description
Bit	Carry Flag	Single bit returned in the carry flag
char / unsigned char, 1-byte pointer	R7	Single byte typed returned in R7
int / unsigned int, 2-byte ptr	R6 & R7	MSB in R6, LSB in R7
long / unsigned long	R4-R7	MSB in R4, LSB in R7
Float	R4-R7	32-Bit IEEE format
generic pointer	R1-R3	Memory type in R3, MSB R2, LSB R1

►示例

下面的例子展示了这些段和函数声明在汇编中完成的。

CODE:

```
;和任意 C 程序兼容的汇编程序示例
;假设文件名为 asm_test.asm
```

```
name asm_test


;我们将要编写一个可以在C 语言中用
;unsigned long add(unsigned long, unsigned long);
;使用的函数
;由于我们需要传递参数给函数
;所以函数名前面需要加前缀'_'


;函数 "add"的代码段
?PR?_add?asm_test segment code
;函数"add"数据段
?DT?_add?asm_test segment data


;让其他的函数使用这块数据空间传递变量
public ?_add?BYTE
;是函数可以被访问
public _add


;定义函数add 的数据段
rseg ?DT?_add?asm_test
?_add?BYTE:
parm1:  DS 4    ;第一个参数
```

```
parm2:  ds 4    ;第二个参数

;你可以像下面展示的这样读取传递过来的参数或者直接使用寄存器传递值

rseg ?PR?_add?asm_test
_add:
;读取第一个参数
        mov parm1+3,r7
        mov parm1+2,r6
        mov parm1+1,r5
        mov parm1,r4
;param2 存储在存储器固定位置

;现在将两个变量相加
        mov a,parm2+3
        add a,parm1+3
;最低位相加完成后， 把它移入寄存器 r7
        mov r7,a
        mov a,parm2+2
        addc a,parm1+2
;存储第二最低有效位
```

```
        mov r6,a
        mov a,parm2+1
        addc a,parm1+1
;存储第二最高位
        mov r5,a
        mov a,parm2
        addc a,parm1
;存储最高有效位并返回结果

;keil will automatically store it to
;varable reading the resturn value
        mov r4,a
        ret

        end
```

现在我们用 C 语言调用上面的函数将会变得非常简单，我们下面像调用普通函数一样调用它：

CODE:

```
extern unsigned long add(unsigned long, unsigned
long);


void main(){
        unsigned long a;
        a = add(10,30);
        //a will have 40 after execution
        while(1);
}
```

原文：Keil C 编程教程：连接 C 程序与汇编

http://www.8051projects.net/keil-c-programming-tutorial/advanced-c-programming.php