第2章 C 语言的基本知识

这里不是教你如何使用C语言。而是怎样利用C语言来对凌阳的unSP内核的单片机进行编程。

以下，我们将复习关于C的一些概念，如结构联合和类型定义等。

2.1 数据与运算

2.1.1 数据与数据类型

在表 2.1 中列出了 unSP GCC 认可的基本数据类型及其值域。读者应该特别注意：表中所列的数据类型及其值域与一般机器使用的 GCC 数据类型之间有一些差别，比如 char 为 16 位等等。此外，unSP GCC 的 float 与 double 均存储为 32 位浮点数，而 unSP 汇编器的 float 与 double 分别是 32 位与 64 位浮点数。

表2.1 unSP GCC 的基本数据类型

数 据 类 型	数据长度（位数）	值　域
char	16	-32768~32767
short	16	-32768~32767
int	16	-32768~32767
long int	32	-2147483648~2147483647
unsigned char	16	0~65535
unsigned short	16	0~65535
unsigned int	16	0~65535
unsigned long int	32	0~4294967295
float	32	以 IEEE 格式表示的 32 位浮点数
double	32	以 IEEE 格式表示的 32 位浮点数

2.1.2 常量、变量、运算符与表达式

在程序运行过程中，其值不能改变的量称为常量。编程时可以用一个字符常量来代表一个看不出有什么意义的数字，比如：

```
#define   C_Fosc_49M        0x0080
```

这里定义了一个字符常量 C_Fosc_49M，它的值为 0x0080。在编程的时候，我们就可以用 C_Fosc_49M 来代替 0x0080。这样的好处是显而易见的，大家一眼就可以看出 Fosc 选择了 49MHz。

在程序运行过程中，其值可以改变的量称为变量。下面就有符号/无符号(signed/unsigned)问题作一些说明。在编写程序时，如果使用 signed 和 unsigned 两种数据类型，就得使用这两种格式类型的库函数，这将使占用的存储空间成倍增长。因此在编程时，如果只强调程序

的运算速度而又不进行负数运算时，最好采用 unsigned 格式。

unSP GCC 基本的算术运算符和 ANSI-C 是一样的，见表 2.2。

表2.2 unSP GCC 基本的算术运算符

算逻操作符	作用
+、-、*、/、%	加、减、乘、除、求余运算
&&、||	逻辑与、或
&、|、^、<<、>>	按位与、或、异或、左移、右移
>，>=，<，<=，= =，!=	大于、不小于、小于、不大于、等于、不等于
=	赋值运算符
？　：	条件运算符
,	逗号运算符
*、&	指针运算符
.	分量运算符
sizeof	求字节数运算符
[　]	下标运算符

下面举一个用使用位操作运算扫描识别键盘的例子。

如图 2.1 所示为 SPCE061A 与一个 4×4 键盘接口的电路图。

图 2.1 电路直接扫描一个矩阵键盘。在这些按规则进行扫描的键盘矩阵上，每次只有一行电平被拉低。在逐次扫描拉低这些行的同时，去读那些列的信息，在被拉低的行上被按下的按键所对应的列的位值为 0，其他列的位值为 1。如图 2.1 所示，SPCE061A 的 A 口低八位作为 4×4 键盘的接口，其中 IOA4～IOA7 作为行驱动线，IOA0～IOA3 作为列读入线。每隔约 20ms 行驱动线被逐次拉低，以避免键盘的抖动干扰。接下来程序所要做的工作就是测试输入的任何变化——新键的按下或旧键的释放。使用位操作运算，可以很容易地将这些变化识别出来。如表 2.3 所示，该表说明了如何使用位操作来识别键盘的变化过程。

表 2.3 适合于把 IOA4～IOA7 设为反相输出口的情况。设为反相输出口的意思是，我们在软件中送 1 到 IO 口，但 IO 口输出的是 0，两者正好是反相的。

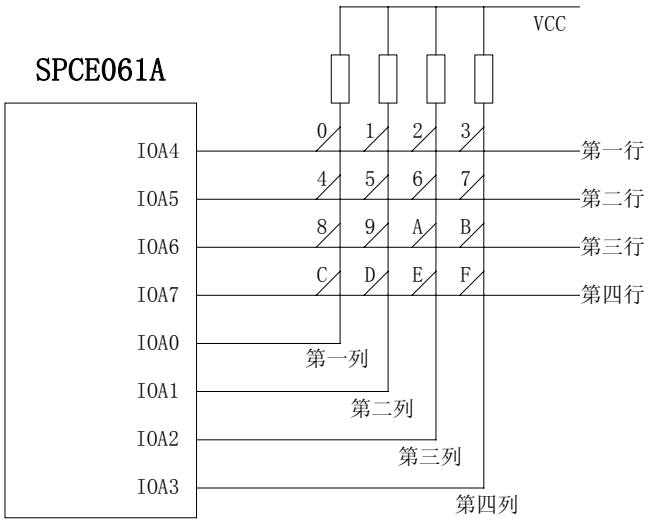


图2.1 键盘扫描原理图

表2.3 使用位操作检测键值变化

	IOA7	IOA6	IOA5	IOA4	IOA3	IOA2	IOA1	IOA0
原键值（old）	0	0	1	0	1	1	0	1

新键值（new）	0	1	0	0	1	1	1	0
中间变量(temp=old^ new)	0	1	1	0	0	0	1	1
新按键（temp&new）	0	1	0	0	0	0	0	1
释放键（temp&old）	0	0	1	0	0	0	1	0

从表 2.3 可以看出，A 口原读入值为 00101101B，由于行驱动线 IOA4～IOA7 为反相输出口，按 1 为低电平，0 为高电平的规定，此时硬件电路的第二行 IOA5 为低电平。由于 IOA0～IOA3 为列读入值，按 0 为低电平，1 为高电平的规定，则第二列的键被按下，其键值为 5。

根据同样的原理，对于新读入的值 01001110B，意味着，此时键值为 5 的键被释放，而第三行第一列键值为 8 的键被按下。下面所进行的逻辑位操作也正好说明了这一点。在表 2.3 中，IOA7～IOA4 的新值 0100B 和原值 0010B 不同说明扫描的行不同，而新按键栏和释放键栏中 IOA7～IOA4 为 1 的是扫描的行。而对于读回列信息的 IOA3～IOA0，新按键栏为 0001B 说明第一列有新按键被按下。而释放键栏为 0010B 说明第二列有键被释放。因而第四位的逻辑操作可以识别键的变化。

键扫描的程序如下：

```
**********************************************************************************
    unsigned int old,new,push,rel,temp,row;
    void key(void)
    {
        for(row=0x10;row<0x100;row<<1)              //扫描
        {
            *P_IOA_Data=*P_IOA_Data&row;
            new=(new<<4)| *P_IOA_Data&0x0f;         //读回列信息
        }
        if((temp=new^old)>0)                        //获取按下和释放的键值信息
        {
            push=temp&new;
            rel=temp&old;
            old=new;
        }
    }
**********************************************************************************
```

在程序中使用了 for 循环，详细内容将在下一节讨论。for 循环的内容将循环 4 次。变量 row 的初值为 0x10，左移 4 次后，变为 0x0100。

读回列信息部分，4 位一组，每行左移 4 位，采用逻辑操作，很容易判别是否有键变化。

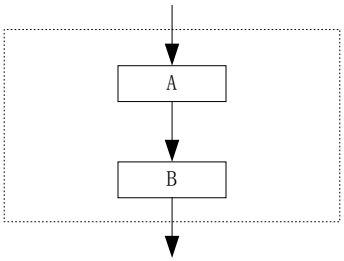
大家来看 if((temp=new^old)>0)这一行，这是一种“嵌入式赋值”，所以 if 测试中包含了对 temp 变量的赋值操作。请注意“＝”和“＝＝”的区别，“＝＝”号将只对等式进行测试，而不进行任何赋值操作。

2.2 流程控制语句

2.2.1　程序的基本结构及控制语句

归纳起来，C 语言有三种基本结构：

1、 顺序结构
2、 选择结构
3、 循环结构



顺序结构流程图

图2.2　　顺序结构流程图

顺序结构是一种最基本，最简单的编程结构。在这种结构中，程序由低地址向高地址顺序执行指令代码。如图 2.2 所示，程序先执行 A 操作，再执行 B 操作，两者是顺序执行的关系。

选择结构及其语句

在选择结构中程序首先对一个条件语句进行测试。当条件为真时，执行一个方向上的程序流程，当条件为假时，执行另一个方向上的程序流程。如图 2.3 所示，T 代表一个条件，当 T 条件成立时，执行 A 操作，否则执行 B 操作。常见的选择语句有：if，else if，switch/case 语句。

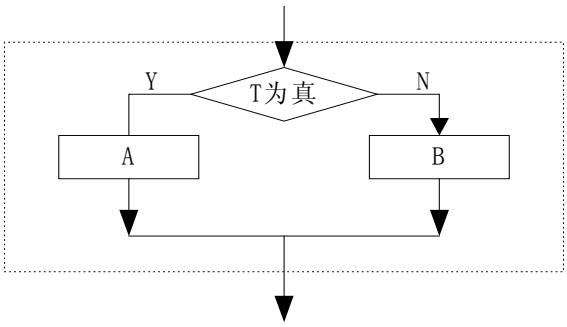


图2.3 选择结构流程图

C 语言的 if 语句有三种基本形式。

第一种形式为基本形式

if(表达式) 语句;

其语义是：如果表达式的值为真，则执行其后的语句， 否则不执行该语句。

第二种形式为 if-else 形式

```
if(表达式)
    语句 1;
else
    语句 2;
```

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2 。

第三种形式为 if-else-if 形式

前二种形式的 if 语句一般都用于两个分支的情况。当有多个分支选择时，可采用 if-else-if 语句，其一般形式为：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
```

…
else if(表达式 n)
　　语句 n；
else
　　语句 n+1；

其语义是：依次判断表达式的值，当出现某个值为真时， 则执行其对应的语句。然后跳到整个 if 语句之外继续执行程序。 如果所有的表达式均为假，则执行语句 n 。 然后继续执行后续程序。

由若干条 if，else if 语句嵌套可构成串行多分支结构。在 if 语句的嵌套中，请注意 if 和 else 的对应关系。else 总是与它上面最近的一个 if 语句相对应，如果 if 和 else 的数目不同时，可以用花括号将不对称的 if 括起来以确定它们之间的相应关系。

switch 语句的一般形式如下：

```
switch（表达式）
{
    case 常量表达式 1：语句 1；break；
    case 常量表达式 2：语句 2；break；
    ┇
    case 常量表达式 n：语句 n；break；
    default          ：语句 n+1；
}
```

说明：

每个 case 后的常量表达式只能是常量组成的表达式，当 switch 后的表达式的值与某一个常量表达式的值一致时。程序就转到此 case 后的语句开始执行，然后遇 break 就退出 switch 语句。如果没有一个常量表达式的值与 switch 后的值一致，就执行 default 后的语句。

各个 case 和 default 出现的次序不影响执行结果，一般情况下，尽量使用出现机率大的 case 放在前面。

循环结构及其语句

使用循环结构可以使分支流程重复地进行。

循环结构又分为“当”（while）型循环结构和“直到”（do－while）型循环结构两种。如图 2.4 所示。在“当”（while）型循环结构中，当判断条件 T 为真时，反复执行操作 A，当条件为假时才停止循环。在“直到”（do－while）型循环结构中，先执行操作 A，再判断条件 T，若 T 为真，则再执行 A，如此反复，直到 T 为假为止。

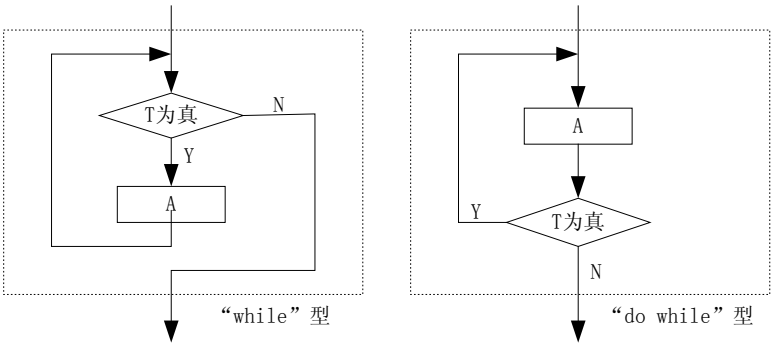


图2.4 循环结构流程图

构成循环结构的常见语句主要有：while，do while，for 等。

while 语句

while 语句的一般形式为： while(表达式)语句； 其中表达式是循环条件，语句为循环体。

while 语句的语义是：计算表达式的值，当值为真(非 0)时， 执行循环体语句。

使用 while 语句应注意以下几点：

1.while 语句中的表达式一般是关系表达或逻辑表达式，只要表达式的值为真(非 0)即可继续循环。

2.循环体如包括有一个以上的语句，则必须用{}括起来， 组成复合语句。

3.应注意循环条件的选择以避免死循环。

do－while 语句

do-while 语句的一般形式为：

```
do
    语句；
while(表达式)；
```

其中语句是循环体，表达式是循环条件。

do-while 语句的语义是：先执行循环体语句一次， 再判别表达式的值，若为真(非 0)则继续循环，否则终止循环。

do-while 语句的特点是在判断条件是否成立前，先执行循环体语句一次。这是与 while 语句的一个根本性的区别。

for 语句

for 语句的作用是用来表示循环次数已知的情况，它的使用格式为：

for(表达式一； 表达式二； 表达式三) 循环体语句

它的执行过程如下：

1>先求解表达式一。

2>求解表达式二，若其值为 0 则结束循环；若其值为非 0 则执行下面的第三步。

3>执行循环体语句，这个语句代表一条语句或一个复合语句。

4>求解表达式三。

5>转到第二步去执行。

说明：

表达式一和表达式三可以省略，看两个例子。

```
for (;i<100;i++)
for(i=1;i<100;)
for (;i<100;)
```

这三例子中，分号不能省略，另外在省略了表达式一和表达式三后，要注意使表达式二能够取到 0 值以避免形成死循环。

2.3 构造数据类型

前面介绍的字符型（char）、整型（int）和浮点型（float）等数据都属于基本数据类型。

C 语言还提供了一些构造数据类型，有：数组、指针、联合、枚举等。

2.3.1　数组

数组相当于是由若干类型相同的变量组成的一个有前后顺序的集合。利用数组可以通过一个统一的数组名称和一个序号，按照第一个、第二个……的方式来访问数组中的数据。

数组有一维数组，多维数组，字符数组三种类型，本书只介绍一维数组。

一维数组的定义

类型说明符　　数组名[常量表达式];

例如：

int b1[5];表示定义了一个含有五个整型元素、名称为 b1 的一维数组。

float x[10];定义了一个含有十个单精度型元素、名称为 x 的一维数组。

说明：

数组的引用

在已经定义了一个数组以后，怎么来使用数组中的元素呢？C 语言规定只能一个一个地引用数组元素而不能一次引用数组中的全部元素。

在 C 语言中，下标的取值范围是从[0,元素个数减 1]结束。

例如：int b1[3];　定义了一个含有 3 个整型元素的数组。我们在引用这 3 个元素时，下标只能取 0、1、2 三个值：b1[3]的这种使用方式是错误的，它表示数组中的第四个元素，而我们这个数组只定义它含有三个元素。

数组的初始化

在定义了数组后，对计算机来说只相当于定义了一个变量。这个变量的值是不确定的。所以，在使用数组时，最好对它进行初始化操作。

对数组的初始化操作可以采取以下方式：

static 类型说明符　数组名[N]＝｛值 1，值 2，……值 N｝；

说明：

1、对数组的初始化操作只能在定义数组时进行。

2、关键字 static 表示定义了一个静态变量。

我们来看一个用数组来处理 Fibonacci 数列的例子

$$Fib(n) = \begin{cases} 1 & n=0 \\ 1 & n=1 \\ Fib(n-2)+Fib(n-1) & n>1 \end{cases}$$

－－

```
#define NUM 20
main()
{
int i;
static int fib[NUM]={1,1};
for (i=2;i<NUM;i++)
    fib[i]=fib[i-2]+fib[i-1];
}
```

－－

程序定义了一个有 20 个元素的整型数组，并给前两个元素赋初值：fib[0]=1，f[1]=1。那其它元素的值是什么呢？答案为“0”。用关键字 static 定义的任何变量，如果没有赋初值的话，系统自动把其初值设为 0。

2.3.2　指针

指针是C语言中广泛使用的一种数据类型。运用指针编程是C语言最主要的风格之一。利用指针变量可以表示各种数据结构； 能很方便地使用数组和字符串； 并能象汇编语言一样处理内存地址，从而编出精练而高效的程序。

unSP GCC 编译器所认可的指针是 16 位的，这点大家编程时要注意。

指针变量的类型说明

其一般形式为： 类型说明符 *变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示本指针变量所指向的变量的数据类型。

例如： int *p1;表示 p1 是一个指针变量，它的值是某个整型变量的地址。 或者说 p1 指向一个整型变量。至于 p1 究竟指向哪一个整型变量， 应由向 p1 赋予的地址来决定。

指针变量的赋值

指针变量同普通变量一样，使用之前不仅要定义说明， 而且必须赋予具体的值。指针变量的赋值只能赋予地址， 决不能赋予任何其它数据，否则将引起错误。

指针变量的赋值运算有以下几种形式：

①指针变量初始化赋值，如：

```
int a;
int p=&a;      /*把整型变量 a 的地址赋予 p*/
```

②把一个变量的地址赋予指向相同数据类型的指针变量。例如：

```
int a,*pa;
pa=&a;         /*把整型变量 a 的地址赋予整型指针变量 pa*/
```

③把一个指针变量的值赋予指向相同类型变量的另一个指针变量。如：

```
int a,*pa,*pb;
pa=&a;
pb=pa;         /*把 a 的地址赋予指针变量 pb*/
```

由于 pa,pb 均为指向整型变量的指针变量，因此可以相互赋值。

④把数组的首地址赋予指向数组的指针变量。例如：

```
int a[5],*pa;
pa=a; (数组名表示数组的首地址，故可赋予指向数组的指针变量 pa)
```

⑤把字符串的首地址赋予指向字符类型的指针变量。例如：

```
char *pc="C Language";
```

这里应说明的是并不是把整个字符串装入指针变量，而是把存放该字符串的字符数组的首地址装入指针变量。

⑥把函数的入口地址赋予指向函数的指针变量。例如： int (*pf)();pf=f;　/*f 为函数名*/

关于指针应用的一个小例子

这个例子将 5 条河流的名称按字母顺序排列后输出。若采用普通的排序方法，逐个比较

之后交换字符串的位置。交换字符串的物理位置是通过字符串复制函数完成的。反复的交换将使程序执行的速度很慢，同时由于各字符串(河流名称) 的长度不同，又增加了存储管理的负担。 用指针数组能很好地解决这些问题。把所有的字符串存放在一个数组中， 把这些字符数组的首地址放在一个指针数组中，当需要交换两个字符串时， 只须交换指针数组相应两元素的内容(地址)即可，而不必交换字符串本身。程序中定义了两个函数（关于函数的调用，请看 2.4 节），一个名为 sort 完成排序，其形参为指针数组 name，即为待排序的各字符串数组的指针。形参 n 为字符串的个数。另一个函数名为 print，用于排序后字符串的输出，其形参与 sort 的形参相同。主函数 main 中，定义了指针数组 name 并作了初始化赋值。然后分别调用 sort 函数和 print 函数完成排序和输出。值得说明的是在 sort 函数中，对两个字符串比较，采用了 strcmp 函数，strcmp 函数允许参与比较的串以指针方式出现。Name[k]和 name[j]均为指针，因此是合法的。字符串比较后需要交换时， 只交换指针数组元素的值，而不交换具体的字符串， 这样将大大减少时间的开销，提高了运行效率。

现编程如下：

```
#include"string.h"
main()
{
    void sort(char *name[],int n);
    void print(char *name[],int n);
    static char *name[]={ "CHANGJIANG","YALUZANGBUJIANG","NUJIANG","YALUJI
ANG","DADUHE"};
    int n=5;
    sort(name,n);
    print(name,n);
}

void sort(char *name[],int n)
{
    char *pt;
    int i,j,k;
    for(i=0;i<n-1;i++)
    {
        k=i;
        for(j=i+1;j<n;j++)
            if(strcmp(name[k],name[j])>0) k=j;
        if(k!=i)
        {
            pt=name[i];
            name[i]=name[k];
            name[k]=pt;
        }
    }
}

void print(char *name[],int n)
```

```
{
    int  i;
    for  (i=0;i<n;i++)  printf("%s\n",name[i]);
}
```

2.3.3　结构体和共用体

结构体和共用体也是C语言中广泛使用的两种种构造数据类型。它们之间的区别在于在结构体中各成员有各自的内存空间，一个结构体变量的总长度是各成员长度之和。而在共用体中，各成员共享一段内存空间，一个共用体变量的长度等于各成员中最长的长度。应该说明的是，这里所谓的共享不是指把多个成员同时装入一个共用体变量内，而是指该共用体变量可被赋予任一成员值，但每次只能赋一种值，赋入新值则冲去旧值。

除了上述区别，结构体和共用体在很多地方是相似的。所以本小节只介绍结构体。

结构是一种“构造”而成的数据类型，在说明和使用之前必须先定义它，也就是构造它。如同在说明和调用函数之前要先定义函数一样。

定义结构的类型

定义一个结构的一般形式为：

```
struct  结构名
{
成员表列
};
```

成员表由若干个成员组成， 每个成员都是该结构的一个组成部分。对每个成员也必须作类型说明。例如：

```
struct date
{
unsigned int month;
unsigned int day
unsigned int year;
};
```

在这个结构定义中，结构名为 date，该结构包含由 3 个成员：month，day，year。这三个结构成员的数据类型都是整型变量。 应注意在括号后的分号是不可少的。结构定义之后，即可进行变量说明。凡说明为结构 date 的变量都由上述 3 个成员组成。由此可见，结构是一种复杂的数据类型，是数目固定，类型不同的若干有序变量的集合。

定义结构类型变量

定义一个结构变量有三种方法。限于篇幅，仅介绍一种。以上面定义的 date 为例来加以说明。

先定义结构，再说明结构变量。如：

```
struct date
{
unsigned int month;
unsigned int day
unsigned int year;
```

};
date date1，date2；

说明了两个变量 date 1 和 date 2 为 date 结构类型。

如果结构变量是全局变量或为静态变量， 则可对它作初始化赋值。对局部或自动结构变量不能作初始化赋值。

数组的元素也可以是结构类型的。 因此可以构成结构型数组。结构数组的每一个元素都是具有相同结构类型的下标结构变量。结构数组的定义方法和结构变量相似，只需说明它为数组类型即可。

结构指针变量的说明和使用一个指针变量当用来指向一个结构变量时， 称之为结构指针变量。

结构指针变量中的值是所指向的结构变量的首地址。 通过结构指针即可访问该结构变量， 这与数组指针和函数指针的情况是相同的。结构指针变量说明的一般形式为：

struct 结构名*结构指针变量名

例如，在前面定义了 date 这个结构， 如要说明一个指向 date 的指针变量 pdate，可写为： struct date *pdate;

结构类型变量的引用

我们对结构进行引用时，我们只能对结构体的成员进行赋值、存取和运算。

其访问的一般形式为： (*结构指针变量).成员名 或为：结构指针变量->成员名 。

例如： (*pdate).month 或者： pdate ->month

应该注意(*pdate)两侧的括号不可少， 因为成员符“.”的优先级高于“*”。如去掉括号写作*pdate.month 则等效于*(pdate.month)，这样，意义就完全不对了。

2.4 函数

在 C 语言中，程序是由一个主函数和其它若干函数构成的。在主函数中可以调用其它函数，其它函数之间也可以互相调用。

我们下面对函数进行一下初步的介绍。

1、一个 C 语言的程序（称作源文件）是由一个函数或多个函数组成的。

2、对于一个很大的任务，一般将它分解成若干源文件，分别编写和调试，这样可以提高开发效率。一个源文件可以被其它的 C 程序使用。

3、C 程序必须从 main()函数开始执行。

4、所有的函数在定义上讲都是互相独立的，即不存在嵌套定义。但是在调用时，可以互相调用或嵌套调用。

5、C 语言中有两类函数，系统提供的函数（库函数）和用户自定义的函数。

6、函数又分为：一、无参函数，只完成固定的功能。二、带参函数，这些参数规定了函数要执行什么样的操作。这两种函数都可以返回或不返回一个函数值。

函数的定义形式

类型说明符　函数名（形式参数表列）

形式参数类型说明

　　{变量定义部分

函数体语句}

解释：

1、**类型说明符**规定了这个函数的返回值类型，我们可以通过对函数返回值的判断，来了解函数的执行情况。

2、**函数名**规定了函数的名称，通过这个名称才能对某个函数进行调用。

3、**形式参数说明表列**规定了函数有什么样的参数（这部分和下面的形式参数类型说明部分可以省略，省略之后的函数被称作无参函数）。如 x,y,z 等用逗号分隔的变量。这些形式参数的作用是控制函数进行什么样的操作，它们的值由调用这个函数的程序给出。

4、**形式参数的类型说明**部分规定了形式参数的类型。形参与形参的类型可以放在一起。如可写成这样　fff(int x,int y,int z)。

5、**变量定义部分**规定了在函数内部要用到的变量和它们的类型。

6、**函数体语句**规定了函数中要执行的语句。函数体语句和变量定义部分用一对大括号包围起来。

当我们按如下方式定义一个函数时，称此函数为空函数。

类型说明符　函数名（）

{ }

如：

echoline() {}　这样就定义了一个空函数。

形式参数与实际参数

在 C 语言的程序中，大部分函数都是带参函数。所以在调用函数与定义函数时存在着数据传递。

在定义函数时函数名后括号内的变量就称为形式参数（形参）。而在调用函数时函数名后括号内的变量被称为实际参数（实参）。

关于形参与实参的说明：

1、注意形参是随函数的调用而产生，随函数调用的结束而消亡。

2、实参的作用是给对应的形参赋值，所以实参必须要能得到一个值，它可以是常量，变量或表达式。一般情况下实参与形参的类型应一致，这样可避免错误的产生。

3、在函数定义中，必须规定形参的类型。我们已经讲过可以有两种定义方式，如下：

```
int max(int x,int y,float z)
{…}
```

函数的返回值

如果想让函数向调用它的函数带回一个值，那么只能使用 return 语句。Return 语句的使用格式如下：

return(表达式);

它将表达式的值做为函数的返回值返回，结束本次函数调用并回到进行调用函数的语句。

关于函数返回值的说明：

1、return 只能返回一个值，而不能返回多个值。

2、return（表达式）语句中的表达式值的类型应与定义函数时函数的类型一致。如果不一致，以定义函数时规定的函数类型为准进行类型转换。

3、如果函数中没有 return 语句，返回值是一个不确定的数。

4、 为了明确规定函数没有返回值，可以用 void 关键字来定义函数，表示“无类型”。

函数调用格式

函数的调用格式为：函数名 （实参表列）

说明：

1、调用函数时要给出被调用的函数名和一对括号。如果这个函数是个带参函数，那么还要将实参变量用逗号隔开放到括号中。

2、在 C 语言中，我们可以将函数调用当成一个表达式。我们可以在任何能够放置表达式的地方来放置函数表达式。

3、函数实参的求值顺序在各 C 语言系统中是不一样的。在有的系统中实参的求值顺序是从左到右的，如下例：

i=3;printf("%d,%d",i,i++); 输出　3,3

unSP IDE 和 Turbo C 一样，实参的求值顺序是从右到左的，如下例：

i=3;printf("%d,%d",i,i++); 输出　4,3

函数调用规则

第一，被调用的函数必须是已经存在的函数，库函数或用户已定义过的函数。

第二、如果使用库函数，还要在使用库函数的源文件开头用＃include 命令声明库函数所在的头文件。

第三、如果使用用户自定义的函数，还要在主调函数中说明用户函数的返回值类型。注意，这时我们关心的只是函数的返回值类型，所以对函数的参数就没必要写出来了。

C 语言规定在以下几种情况中，可以不在主调函数中对被调用函数的返回值类型进行说明：

1、如果被调用函数的返回值为整型或字符型时，可不进行说明，系统自动将此被调函数的返回值看成整型。

2、如果被调用函数定义在主调函数之前，也可不对其进行说明。

3、如果在所有函数定义之前，对程序中的函数类型进行了说明，则在各个主调函数中不必再对被调用函数进行说明。

例如：

```
float max();
double min();
void equal();
main()
{…}
float max(float x,float y)
{…}
double min(double x,double y)
{…}
void echoline()
{…}
```

在这个例子中，我们在程序的开头对程序中的所有函数进行了说明。这样就不用在主调函数对其它函数进行说明了。

这种方法是我强烈推荐的！因为这种方法将所有的函数在程序开头进行了说明，不仅省去了在各函数中进行的函数说明，而且一看就知道这个程序中有哪些函数。

C 语言中不允许作嵌套的函数定义。因此各函数之间是平行的，不存在上一级函数和下一级函数的问题。 但是 C 语言允许在一个函数的定义中出现对另一个函数的调用。 这样就出现了函数的嵌套调用。即在被调函数中又调用其它函数。 这与其它语言的子程序嵌套的情形是类似的。

一个函数在它的函数体内调用它自身称为递归调用。这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身。每调用一次就进入新的一层。例如有函数 f 如下：

```
int f (int x)
{
    int y;
    z=f(y);
    return z;
}
```

这个函数是一个递归函数。 但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行， 必须在函数内有终止递归调用的手段。常用的办法是加条件判断， 满足某种条件后就不再作递归调用，然后逐层返回。

C 函数和汇编函数的相互调用

实际上，你不可能用 C 语言来写所有的程序。很多系统特别是实时时钟系统都是用 C 和汇编语言联合编程。unSP 单片机的开发仿真环境 IDE 同时提供了 C 语言和汇编语言的开发环境，C 函数和汇编函数可以方便地进行相互调用。

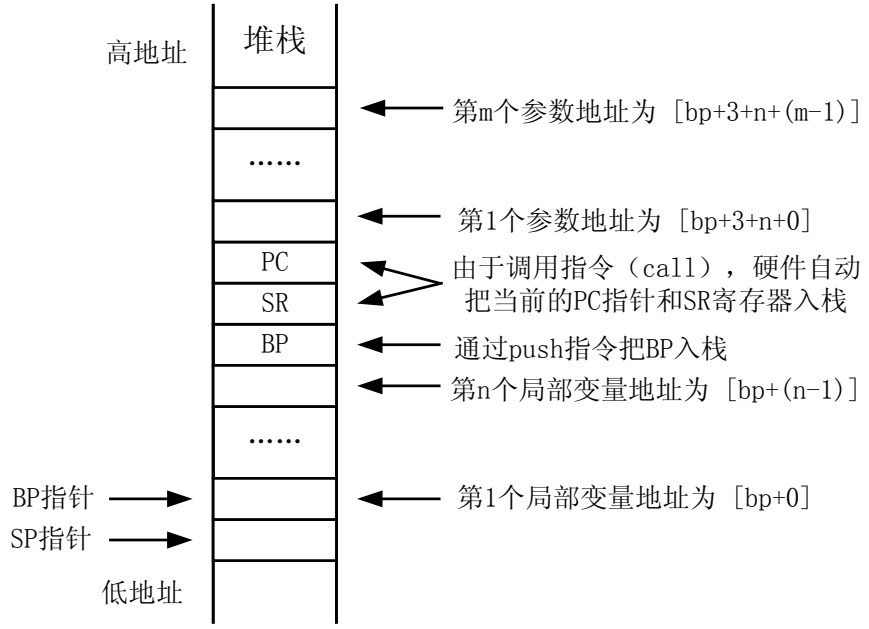


图2.5 参数传递的堆栈调用

由于 C 编译器产生的所有标号都以下划线“_”为前缀，而 C 程序在调用汇编函数时要求汇编函数名也以下划线“_”为前缀。

在进行参数传递时，参数以相反的顺序（从右到左）被压入栈中。

在汇编函数调用 C 函数时，函数调用者应切记在函数返回时将调用函数压入栈中的参数弹出。各参数和局部变量在堆栈中的排列如图 2.5 所示。

16 位的返回值存放在寄存器 R1 中。32 位的返回值存入寄存器对 R1、R2 中；其中低字在 R1 中，高字在 R2 中。若要返回结构则需在 R1 中存放一个指向结构的指针。

编译器会产生 prolog/epilog 过程动作来暂存或恢复 PC、SR 及 BP 寄存器。汇编器则通过‘CALL’指令可将 PC 和 SR 自动压入栈中，而通过‘RETF’或‘RETI’指令将其自动弹出栈来。

编译器所认可的指针是 16 位的。函数的指针实际上并非指向函数的入口地址，而是一个段地址向量_function_entry，在该向量里由 2 个连续的 word 的数据单元存放的值才是函数的入口地址。

下面以具体实例来说明 C 函数和汇编函数的相互调用。

在 C 中要调用一个汇编编写的函数，需要首先在 C 语言中声明此函数的函数原型。尽管不作声明也能通过编译并能执行代码，但是会带来很多的潜在的 bug。

我们看一个 C 调用汇编小例子。我们可以从这个例子中了解到 C 调用函数时是如何进行参数传递的，函数的返回值是怎样实现的。

主函数 main()为 C 函数，调用汇编函数 ADD()。主函数往 ADD()中传递 2 个参数，ADD()把这两个参数相加，所得的和返回给主函数。

```
extern int ADD();
int main(void)
{
    int i,j,SUM;
    i=2;
    j=3;
    SUM=ADD(i,j);
    return 0;
}
```

```
.CODE
.PUBLIC _ADD
_ADD:.PROC
    PUSH BP TO [SP];
    BP = SP + 1;
    R1 = [BP+3];   //取得第二个参数
    R2 = [BP+4];   //取得第一个参数
    R1 += R2;
    POP BP FROM [SP];
    RETF;
.ENDP
```

在汇编函数中要调用 C 语言的子函数，那么应该根据 C 的函数原型所要求的参数类型，分别把参数压入堆栈后，再调用 C 函数。调用结束后还需再进行弹栈，以恢复调用 C 函数前的堆栈指针。此过程很容易产生 bug，所以需要程序员细心处理。下面的例子给出了汇编调用 C 函数的过程。

主函数 main()为汇编函数，调用 C 函数 ADD()。主函数往 ADD()中传递 2 个参数，ADD()把这两个参数相加，所得的和返回给主函数。

```
.EXTERNAL _ADD;
.RAM
.VAR SUM
.CODE
.PUBLIC _main;
_main:
    R1 = 3;
    PUSH R1 TO [SP];          //第 2 个参数入栈
    R1 = 2;
    PUSH R1 TO [SP];          //第 1 个参数入栈
    CALL _ADD;
    [SUM]=R1;
    POP R1 FROM [SP];         //弹出参数恢复 SP 指针
.END
```

```
int ADD(int i,int j)
{
    i=i+j;
    return i;
}
```

2.5 unSP 的 C 语言嵌入式汇编

为了使 C 语言程序具有更高的效率和更多的功能，需在 C 语言程序里嵌入用汇编语言编写的子程序。一方面是为提高子程序的执行速度和效率；另一方面，可解决某些用 C 语言程序无法实现的机器语言操作。

通常，有两种方法可将汇编语言代码与 C 语言代码联合在一起。一种是把独立的汇编语言程序用 C 函数连接起来，通过 API (Application Program Interface) 的方式调用，这在上一章节就介绍了；另一种就是我们下面要讲的在线汇编方法，即将直接插入式汇编指令嵌入到 C 函数中。

采用 GCC 规定的在线汇编指令格式进行指令的输入，是 GCC 实现将 unSP 汇编指令嵌入 C 函数中的方法。GCC 在线汇编指令格式规定如下：

asm (“汇编指令模板”：输出参数：输入参数：clobbers 参数);

若无 clobber 参数，则在线汇编指令格式可简化为：

asm (“汇编指令模板”：输出参数：输入参数);

下面，将对在线汇编指令格式中的各种成分之内容进行介绍。

1)　汇编指令模板

模板是在线汇编指令中的主要成分，GCC 据此可在当前位置产生汇编指令输出。例如，下面一条在线汇编指令：

asm ("%0 += %1" : "+r" (foo) : "r" (bar));

此处，"%0 += %1"就是模板。其中，操作数"%0"、"%1"作为一种形式参数，分别会由第一个冒号后面实际的输出、输入参数取代。带百分号的数字表示的是第一个冒号后参数的序号。

如下例：

asm ("%0 = %1 + %2" : "=r" (foo) : "r" (bar), "i" (10));

"%0"会由参数 foo 取代，"%1"会由参数 bar 取代，而"%2"则会由数值 10 取代。

在汇编输出中，一个汇编指令模板里可以挂接多条汇编指令。其方法是用换行符'\n'来结束每一条指令，并用 Tab 键符'\t'将同一模板产生在汇编输出中的各条指令在换行显示时缩进到同一列，以使汇编指令显示清晰。如下例：

asm ("%0 += %1\n\t%0 += %1" : "+r" (foo) : "r" (bar));

2)　操作数

在线汇编指令格式中，第一冒号后的参数为输出操作数，第二冒号后的参数为输入操作数，第三冒号后跟着的则是 clobber 操作数。在各类操作数中，引号里的字符代表的是其存储类型约束符；括弧里面的字符串表示的是实际操作数。

如果输出参数有若干个，可用逗号“，”将每个参数隔开。同样，该法则适用于输入参数或 clobber 参数。

3)　操作数约束符

约束符的作用在于指示 GCC，使用在汇编指令模板中的操作数的存储类型。表 2.4 列出了一些约束符和它们分别代表的操作数不同的存储类型，也列出了用在操作数约束符之前的两个约束符前缀。

表2.4 操作数存储类型约束符及约束符前缀

约束符	操作数存储类型	约束符前缀及含义解释	
r	寄存器中的数值	=	+
m	存储器内的数值	为操作数赋值	操作数在被赋值前先要参加运算
i	立即数		
p	全局变量操作数		

4)　GCC 在线汇编指令举例

例 1：asm ("%0 = %1 + %2" : "=m" (foo) : "r" (bar), "i" (10));

操作数 foo 和 bar 都是局部变量。bar 的值会分配给寄存器（此例中寄存器为 R1），而 foo 的值会置入存储器中，其地址在此由 BP 寄存器指出。GCC 对此会产生如下代码：

```
[BP] = R1 + 10
```

注意，本在线汇编指令产生的汇编代码不能被正确汇编。正确的在线汇编指令应当是：

asm("%0=%1+%2":"=r"(foo):"r"(bar),"i"(10));

它产生如下的汇编代码：R1 = R4 + 10

例 2：　asm("%0 += %1" : "+r"(foo): "r"(bar));

操作数 foo 在被赋值前先要参加运算，故其约束符为"+r"，而非"=r"。

利用嵌入式汇编实现对端口寄存器的操作

在 C 的嵌入式汇编中，当使用端口寄存器名称时，需要在 C 文件中加入汇编的包含文件，如下所示：

asm(“.include hardware.inc”);

那么，我们就可以使用端口寄存器的名称，而不必去使用端口的实际的地址。

如果你不想包含那么大一个文件的话，也可以单独定义你用到的一些寄存器，如下所示：

asm(“.define　P_IOA_Dir　0x7002”);

5)　写端口寄存器

现举例说明：要设定 PortA 端口为输出端口，需要对 P_IOA_Dir 赋值 0xffff。那么在 C 中的嵌入式汇编的实现方式如下：

在 C 中有一个 int 型的变量 i，传送到 P_IOA_Dir 中，则嵌入汇编的实现方式如下：

```
asm(“.define　P_IOA_Dir　0x7002”);
int main(void)
{
    int i;
    asm("[P_IOA_Dir] = %0"　　: :"r"(i)); //只有输入参数,通过寄存器传递变量 i 的内容
}
```

如果需要对端口寄存器直接赋值一个立即数（比如对 P_IOA_Dir 赋值 0x1234），那么内嵌式汇编为：

```
asm("[P_IOA_Dir] = %0" : : "r"(0x1234)); //只有输入参数，通过寄存器传递立即数 0x1234
```

6)　读端口寄存器

对端口寄存器进行读操作的方法，与写类似，下面仍然以 P_IOA_Dir 为例。

如果要实现把端口的寄存器 P_IOA_Dir 的值读出并保存在 C 中的一个 int 变量 j 里，那么可以通过下面的方法来实现。

```
int j;
asm("%0 = [P_IOA_Dir]"    : "=r"(j));    //只有输出参数，而无输入参数
```

7)　利用 GCC 编程举例

下面是一段 GCC 的代码，实现对 A 口的初始化，设定 A 口为同向输出高电平。

```
asm("[P_IOA_Attrib] = %0\n\t"
    "[P_IOA_Data] = %0\n\t"
    "[P_IOA_Dir] = %0\n\t"
    :
    :
    "r"(0xffff)
    );
```

上面代码通过 GCC 编译后的代码为：

```
R1=(-1)                    // QImode move
                           // GCC inline ASM start
[P_IOA_Attrib] = R1
[P_IOA_Data] = R1
[P_IOA_Dir] = R1
                           // GCC inline ASM end
```

下面是一段 GCC 的代码，实现对 B 口的初始化：设定 B 口为具有上拉电阻的输入。

```
asm("[P_IOB_Attrib] = %0\n\t"
    "[P_IOB_Data] = %1\n\t"
    "[P_IOB_Dir] = %0\n\t"
    :
    :
    "r"(0),
    "r"(0xffff)
    );
```

上面一段代码通过 GCC 编译后的汇编代码为：

```
R2=(-1)                    // QImode move
R1=0                       // QImode move
                           // GCC inline ASM start
[P_IOB_Attrib] = R2
[P_IOB_Data] = R1
[P_IOB_Dir] = R2
                           // GCC inline ASM end
```

通过上述两段代码，使得 SPCE061A 的 B 口为输入，A 口为输出，如果我们要实现把 B 口得到的数据从 A 口输出，这样的 GCC 编程需要在 C 中先建立个 int 型的中间变量，通过

这个中间变量，写出两个 GCC 的代码来实现。

```
        …
int temp;
        …
asm("%0 = [P_IOB_Data]" : "=r"(temp));
asm("[P_IOA_Buffer] = %0"    : : "r"(temp)  );
```

通过 GCC 后的代码如下所示。这里将看不到 temp 的影子，GCC 会进行优化处理。

```
R1 = [P_IOB_Data]
[P_IOA_Buffer] = R1
```

通过上述方法的介绍，我们就可以在 C 语言中直接对 SPCE061A 的硬件进行操作。在对硬件读写语句较少的情况下，如果采用 C 调用汇编函数的方法显得有些臃肿，而使用嵌入式汇编会使得代码高效简洁！