第13章 DSP程序设计

主要内容：

（1）DSP C语言程序设计

（2）C语言与汇编语言混合编程

（3）DSP程序烧写

13.1 DSP C语言程序设计

DSP支持使用ANCI C进行程序设计，并提供了相应的编译器和C优化编译工具，利用这些优化编译工具可以产生可与手工编写相比的汇编语言程序。

13.1.1 DSP C语言的特征

DSP C语言以ANSI C为基础，并对ANSI C进行了相应的限定和扩展。

以下是LF2407 C语言的一些不同于一般标准C的特征：

➢ 标识符和常量

- ❑ 所有标识符的前100个字符是有效的，区分大小写。
- ❑ 不允许多字节字符。
- ❑ 多字符的字符常数按序列中的最后一个字符来编码。

  例如：’abc’ == ’c’。

➢ 数据类型

- ❑ 整型、双精度型等数据类型长度与常见编译器中数据类型不同，所有的浮点型都是由MS320C2x/C2xx/C5x的32位的二进制浮点格式来表示。
- ❑ size_t（sizeof操作符的结果）定义为unsigned int。
- ❑ ptrdiff_t（指针加减的结果）定义为int。

- **类型转换**
  - 浮点数转换为整型数为向零取整转换。
  - 指针和整型数可以自由转换。
- **表达式**
  - 当两个有符号整型数相除，如果两个数中任一个为负数，则商为负数，并且余数的符号与被除数的符号相同。用斜线符号（/）可以得到商，用百分号（%）可以得到余数。

    例如：10 / -3 = -3； -10 / 3 = -3；

    10 % -3 = 1； -10 % 3 = -1；
  - 有符号型数的右移是算术移位，符号被保留。

➢声明

- ❑ 寄存器变量(register)声明对short,integer,pointer等所有类型的变量都有效。

➢预处理指令（#pragma）

- ❑ 预处理器会忽略所有不支持的预处理指令。
- ❑ 支持下列预处理指令：CODE_SECTION,DATA_SECTION和FUNC_EXT_CALLED。

13.1.2 数据类型

❖所有整数类型（char,short,int以及对应的无符号类型）都是相同的，都是由16位的二进制数来表示。

- 长整型（long）和无符号长整型（unsigned long）都是由32位的二进制数来表示。
- 有符号类型都是由基2的补码来表示。
- 字符型是有符号类型，等同于整型。
- 枚举（enum）类型的对象用16位数来表示；在表达上与整型相似。
- 所有浮点型（float,double和long double）相似，在TMS320C2x/C2xx/C5x中都是用32位浮点格式来表示。
- long和float类型以低有效字存储在低端的存储地址。

类型	长度	表示方法	范围	
			最小值	最大值
char signed char	16 位	ASCII	-32768	32767
unsigned char	16 位	ASCII	0	65535
short	16 位	基 2 补码	-32768	32767
unsigned short	16 位	二进制码	0	65535
int signed int	16 位	基 2 补码	-32768	32767
unsigned int	16 位	二进制码	0	65535
long signed long	32 位	基 2 补码	-2147483648	2147483647
unsigned long	32 位	二进制码	0	4294967295
enum	16 位	基 2 补码	-32768	32767
float	32 位	TMS320C2x/C2xx/C5x	1.19209290e-38	3.4028235e+38
double	32 位	TMS320C2x/C2xx/C5x	1.19209290e-38	3.4028235e+38
long double	32 位	TMS320C2x/C2xx/C5x	1.19209290e-38	3.4028235e+38
pointers	16 位	二进制码	0	0xFFFF

注：在TMS320C2x/C2xx/C5x C语言中，字节长度为16位，sizeof操作符返回的对象长度是以16位为字节长度的字节数。例如sizeof(int) = 1。

13.1.3 寄存器变量

C编译器在一个函数中最多只能用两个寄存器变量，而且必须在参数表或函数的开始处声明。在嵌套块中的寄存器变量定义被认为是一般的变量。

编译器用AR6和AR7作寄存器变量：

- AR6被指定为第一个寄存器变量。
- AR7被指定为第二个寄存器变量。

变量的地址放在指定的寄存器中，访问起来更加容易。16位的字节变量(char,short,int和pointer)可以用作寄存器变量。

在运行时，设置每一个寄存器变量需要四条指令。为了有效地利用这种方式，只有在一个变量被多次访问时，才使用寄存器变量。

程序优化编译器也会定义寄存器变量，但使用方式不同。编译器会自己决定哪些变量作为寄存器变量，程序中声明的寄存器变量会全部被忽略。

13.1.4 asm语句

TMS320C2x/C2xx/C5x的C编译器可以在编译器输出的汇编语言中直接嵌入汇编语言指令。这种能力是C语言的扩展——asm语句。asm语句能够实现一些C无法实现的功能。

```
/*************************************************************/
asm(" clrc    INTM");
/*************************************************************/
```

- 对于嵌入的汇编指令，编译器不会进行语法检查，编程者必须确认嵌入的指令合理有效。
- 使用asm指令的时候应小心不要破坏C语言的环境。如果C代码中插入跳转指令和标识符可能会引起不可预料的操作结果。能够改变块或其它影响汇编环境的指令也可能引起麻烦。
- 对带asm语句的代码使用优化器时要特别小心。尽管优化器不能删除asm指令，但它可以重新安排asm指令附近的代码顺序，这样就可能会引起不期望的结果。

13.1.5 访问I/O空间

➢ I/O空间地址声明

要在程序中访问io空间地址，必须首先用关键字“ioport”对要访问的地址进行定义。

语法：ioport *type* port*hex_num*

ioport　声明io空间端口变量的关键字；

type　变量类型，可以为char，short，int或unsigned int；

port*hex_num*　端口号，port后面接16进制数字。

```
/*************************************************/
ioport unsigned int port10;
/*************************************************/
```

注：声明io空间地址必须在C文件起始声明，不允许在函数中使用ioport声明io空间地址。

➢I/O空间地址访问

访问用ioport关键字声明的I/O端口变量和访问一般变量没有区别。

```
/*************************************************************/
ioport unsigned int port10; /* 访问I/O端口10h的变量 */
int func ()
{
    ...
    port10 = a;    /* 写 a到端口 10h */
    ...
    b = port10;   /* 读取端口10h的值到 b */
    ...
}
/*************************************************************/
```

I/O端口变量的使用不仅仅局限于赋值，和其他变量同样也可以应用于其它的表达式。

```
/*************************************************************/
call (port10);      /* read port 10h and pass to call */
a = port10 + b;     /* read port 10h, add b, assign to a */
port10 += a; /* read port 10h, add a, write to port 10h */
/*************************************************************/
```

~~~~~~

程序中访问的任何一个IO地址都必须在C语言程序起始处用ioport关键字声明！

~~~~~~

13.1.6 访问数据空间

访问数据空间不需要对要访问的单元预先声明，访问是通过指针的方法实现的。

```
/*************************************************************/
unsigned int org,cnt,block,offset,tmp,i;
org = *(unsigned int *) 0x8000;
cnt = *(unsigned int *) 0x8001;
block = *(unsigned int *) 0x8002;
offset = *(unsigned int *) 0x8003;
for (i=0; i<cnt; i++)
 {
     tmp = *(unsigned int *) (org + i);
     *(unsigned int *) (org + offset +i) = tmp;
}
/*************************************************************/
```

13.1.7 中断处理

（1）中断处理方法

- **查询法**

程序通过查询中断标志位来判断是否有中断发生，并进行相应的处理。

优点：流程易于控制，不会发生中断嵌套的问题，一般也不会发生丢失中断的问题。

缺点：中断实时性差。

➢ 回调法

为中断指定一个回调函数，即中断服务程序。将中断服务程序的入口地址放在中断向量处。

优点：中断实时性好，程序结构简洁，类似于windows操作系统下事件驱动的编程方式。

缺点：处理不好容易造成中断嵌套或丢失中断。

（2）回调法处理中断的一般性问题

- 中断服务函数可以和一般函数一样访问全局变量、分配局部变量和调用其它函数等。
- 进入中断服务函数，编译器将自动产生程序保护所有必要的寄存器，并在中断服务函数结束时恢复运行环境。
- c_int0是保留的复位中断处理函数，不会被调用，也不需要保护任何寄存器。
- 要将中断服务函数入口地址放在中断向量处以使中断服务函数可以被正确调用。
- 中断服务函数要尽量短小，避免中断嵌套等问题。

（3）用C编写中断服务函数

有两种方式定义中断服务函数：a)任何具有名为c_int*d* 的函数（*d*为0到9的数），都被假定为一个中断程序，c_int0函数留作系统复位中断用。

b)利用中断关键词interrupt进行定义。举例如下：

```
/*********************************************************/
void c_int1 ()
{
     ......
}
/*********************************************************/
interrupt void isr ()
{
     ......
}
/*********************************************************/
```

（4）C语言编写中断处理函数注意事项：

- 中断处理函数必须是void类型，而且不能有任何输入参数。
- 进入中断服务程序，编译器只保护与运行上下文相关的寄存器，而不是保护所有的寄存器。中断服务程序可以任意修改不被保护的寄存器，如外设控制寄存器等。
- 要注意IMR、INTM等中断控制量的设置。
- 中断处理函数可以被其他C程序调用，但是效率较差。
- 多个中断可以共用一个中断处理函数，除了c_int0。
- 使用中断处理函数和一些编译选项冲突，注意避免对包含中断处理函数的C程序采用这些编译选项。

13.2 C语言与汇编语言混合编程

C语言编写DSP程序对底层的了解要求较低，流程控制灵活，开发周期短。程序可读性、可移植性好，程序修改、升级方便。

某些硬件控制功能不如汇编语言灵活，程序实时性不理想，很多核心程序可能仍然需要利用汇编语言来实现。

13.2.1 C语言与汇编语言混合编程的方式

- C语言调用汇编语言编写的函数
- 使用内嵌汇编语句（asm语句）
- C语言访问汇编语言变量
- 手动修改C语言程序编译后生成的汇编代码

13.2.2 存储器模式

TMS320C2x/C2xx/C5x的C语言编译器将存储器分为两个线性的空间：

- 程序存储器　存储可执行代码
- 数据存储器　存储各种变量和堆栈

编译器将存储器以分段（section）的方法分配和管理，用户以不同的方式分配存储器，可以形成不同的系统配置。连接器将各个块连接在一起形成最终输出的存储器结构。

已初始化的段：

.text 包含所有可执行代码和浮点型常量 Page 0

.cinit 包含初始化变量和常量表 Page 0

.const 包含字符串常量，以及以const修饰的全局或静态变量的声明和初始化 Page 1

.switch 包含switch语句的分支跳转地址表 Page 0

未初始化的块：

.bss 为全局和静态变量保留空间 Page 1

.stack 为系统软件堆栈分配空间 Page 1

.system 为动态分配的内存保留空间，可以被calloc、malloc、realloc函数使用 Page 1

13.2.3 系统堆栈

系统堆栈分为硬件堆栈和软件堆栈。

DSP内部程序控制逻辑部分包含一定大小的堆栈，通常称为硬件堆栈，可以用来保存若干个分支、跳转、函数调用或中断服务程序的返回地址，也可以用来保存其它变量。

通过系统配置可以另外生成一定大小的软件堆栈，用来：

- 分配局部变量
- 传递函数参数
- 保存处理器状态
- 保存函数返回地址
- 保存临时结果
- 保存寄存器内容

堆栈从低端地址向高端地址生成。编译器利用两个辅助寄存器来管理堆栈：

- AR1　堆栈指针（SP, stack pointer），指向当前堆栈顶。
- AR0　帧指针（FP, frame pointer），指向当前帧的起始点，每一个函数都会在堆栈顶部建立一个新的帧，用来保存局部或临时变量。

C语言环境自动操作这两个寄存器。如果编写用到堆栈的汇编语言程序，一定要注意正确使用这两个寄存器。

用-stack连接选项可以指定软件堆栈的大小，用C编写DSP程序一定注意保留足够的堆栈空间！

注意：编译器不会检查堆栈溢出情况，堆栈溢出会破坏DSP运行环境，导致程序失败。编写DSP程序和配置DSP存储器资源要注意防止堆栈溢出的发生。

13.2.4 动态分配内存

TMS320C2x/C2xx/C5x C语言可调用malloc、calloc或realloc函数动态申请内存，申请的内存将分配在.system块。

动态分配的内存只能通过指针进行访问。将大数组通过这种方式来分配可以节省.bss块的空间。

通过连接器的-heap选项可以定义.system块。

```
/*************************************************************/
unsigned int *data;
data =(unsigned int *) malloc (100 * sizeof (unsigned
   int));
/*************************************************************/
```

13.2.5 寄存器规则

TMS320C2x/C2xx/C5x运行环境对寄存器的使用有严格的要求，如果编写涉及到寄存器的汇编程序，必须严格遵守这些规则，否则可能造成系统工作异常。

寄存器规则规定了编译器如何使用寄存器，和寄存器在函数调用的过程中如何进行保护。

寄存器按照保护方式分为两种：

- 调用保存（save on call），调用其它函数的函数负责保存这些寄存器的内容。
- 入口保存（save on entry），被调用的函数负责保存这些寄存器的内容。

注：无论是否使用优化编译，都必须遵守这些寄存器规则。

寄存器的使用和保护

寄存器	用途	调用时保护
AR0	帧指针（**FP**）	**Yes**
AR1	堆栈指针（**SP**）	**Yes**
AR2	局部变量指针（**LVP**）	**No**
AR3-AR5	表达式运算	**No**
AR6-AR7	寄存器变量	**Yes**
Accumulator	表达式运算 / 返回值	**No**

状态寄存器（ST0、ST1）单元

单元	名称	假定值
ARP	辅助寄存器指针	**1**
C	进位标志	**-**
DP	数据页	**-**
OV	溢出标值	**-**
OVM	溢出模式	**0**
PM	乘法移位模式	**0**
SXM	符号扩展模式	**-**
TC	测试模式	**-**

对于有假定值（为0或为1）的状态寄存器单元，在进行函数调用和函数返回时必须保证其值为假定值。

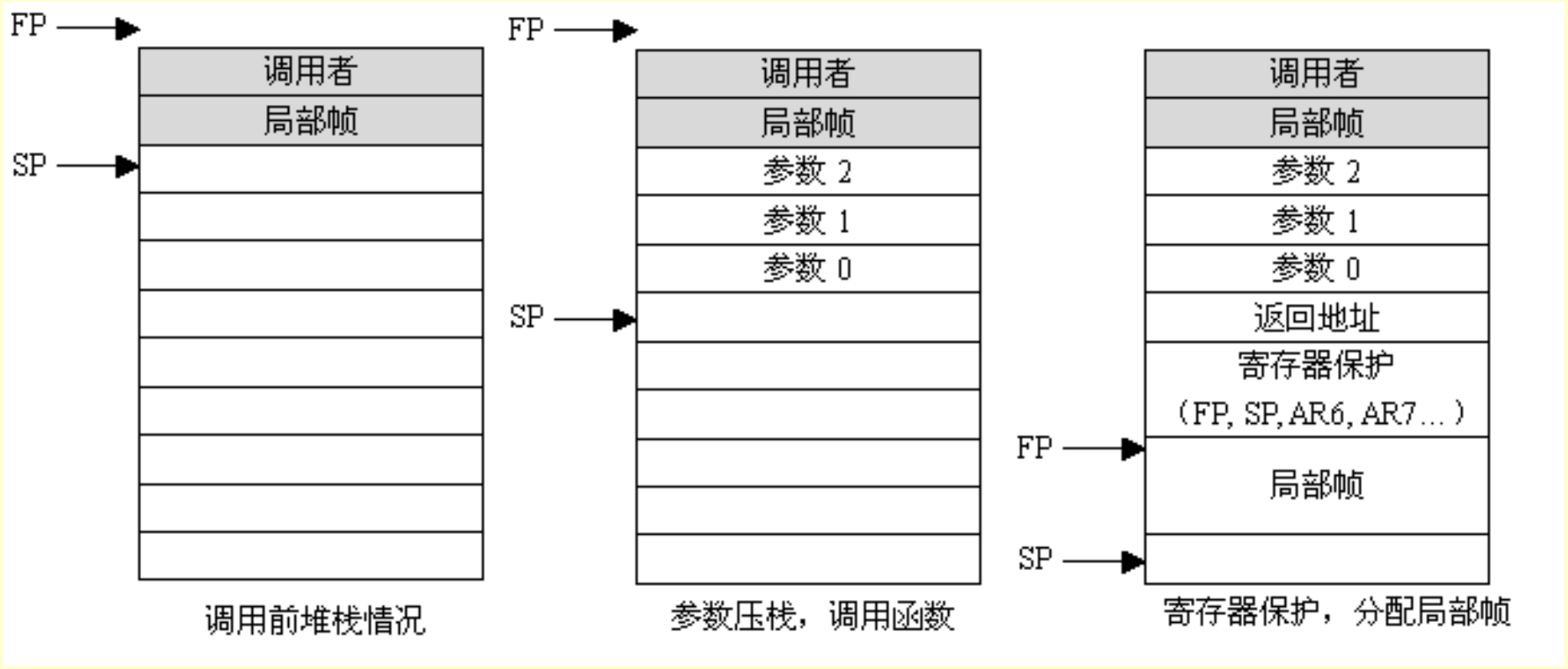
13.2.6 函数结构与调用规则

TMS320C2x/C2xx/C5x运行环境对函数调用有严格的要求，要调用C函数或要被C程序调用的汇编语言程序必须遵守这些规则，否则可能破坏C运行环境，造成程序失败。

- 当进行函数调用时，调用者要将传递参数压入系统堆栈传给被调用的函数，并将函数返回地址压栈。
- 被调用的函数要在函数运行结束时将返回值放在累加器里返回给调用者函数。

可见函数参数及返回地址等都是通过堆栈传递的，要编写汇编函数，必须明确函数调用的过程中堆栈的变化和应该进行的处理

进行函数调用过程中系统堆栈的变化



- 调用函数的工作：
  - 将参数反向压入堆栈（最右端的参数最先压栈，最左端的参数最后压栈），这样，当函数被调用时，最左端的参数会在堆栈的最顶部。
  - 调用被调用函数。
  - 当被调用函数返回时，调用者函数假定ARP已被设置为AR1。
  - 当被调用函数运行结束时，调用者函数要将压入堆栈的参数弹出以恢复堆栈状态。

注：如果用C程序调用汇编语言程序，C编译器会自动产生代码完成这些工作。

- 被调用函数的工作：
  - 进入函数时，函数假定ARP已经被设定为AR1。
  - 将返回地址从硬件堆栈弹出，压入软件堆栈。
  - 将FP（SP）压入软件堆栈。
  - 分配局部帧。
  - 如果函数中要修改AR6、AR7，将它们压入堆栈，其它寄存器不用进行保护就可以进行修改。
  - 实现函数功能。
  - 如果函数返回标量数据，将它放入累加器。
  - 将ARP设定为AR1。
  - 如果保护了AR6、AR7，恢复这两个寄存器。
  - 删除局部帧。

- 恢复FP（SP）。
- 将返回地址从软件堆栈中弹出，压入硬件堆栈。
- 返回。

❖ 一些特殊的情况：

- 返回一个结构：当函数的返回值为一个结构时，调用者函数负责分配存储空间，并将存储空间地址作为最后一个输入参数传递给被调用函数。被调用函数将要返回的结构拷贝到这个参数所指向的内存空间。
- 不将返回地址移动到软件堆栈：当被调用函数不再调用其它函数，或者确定调用深度不会超过8级，可以不用将返回地址移动到软件堆栈。
- 不分配局部帧：如果函数没有输入参数，不使用局部变量，就不需要修改AR0（FP），因此也不需要对其进行保护。

13.2.7 C程序调用汇编函数

C程序调用汇编函数必须要满足前面介绍的调用规则和寄存器规则，C程序可以访问汇编语言定义的变量或调用汇编语言函数，同样汇编语言也可以访问C程序定义的变量或调用C函数。

用C程序调用汇编函数有以下注意事项：

- 所有的函数（不论用C编写还是用汇编语言编写）都必须满足前面介绍的寄存器规则。
- 对于一些寄存器，如果函数要修改其内容，则必须事先对其进行保护。这些寄存器包括：
  - AR0(FP)
  - AR1(SP)
  - AR6
  - AR7

其它的寄存器可以不用保护自由使用。

- 如果函数改变了状态寄存器某些有假定值的位，则必须在函数结束前恢复其原有值。尤其要注意ARP必须为AR1。
- 中断服务程序必须保护所有其用到的寄存器。
- long型和float型变量在存储器中的存储方式为低有效位在低端地址。
- 函数返回值要通过累加器进行传递。
- 编译器会在所有对象的名称前面加下横线“_”，因此汇编语言模块定义对象名称时也要以下横线为前缀，才能使定义的对象可以被C代码访问。例如C语言对象x在汇编语言中就是_x。汇编语言可以使用任何不带下横线前缀的变量而不会和C语言对象冲突。
- 任何汇编语言定义的对象，如果要被C程序访问，则必须用.global修饰。同样任何C语言定义的对象，如果要被汇编语言访问，也必须以.global修饰。

(a) C program

```
extern int asmfunc(); /* declare external asm function  */
int gvar;             /* define global variable         */

main()
{
   int i;

   i = asmfunc(i);    /* call function normally         */
}
```

(b) Assembly language program

```
_asmfunc:
   POPD      *+         ; Move return address to C stack
   SAR  AR0, *+         ; Save FP
   SAR  AR1, *          ; Save SP
   LARK AR0, 1          ; Size of frame
   LAR  AR0, *0+, AR2   ; Set up FP and SP

   LDPK _gvar           ; Point to gvar
   SSXM                 ; Set sign extension
   LAC  _gvar           ; Load gvar
   LARK AR2, -3         ; Offset of argument
   MAR  *0+             ; Point to argument
   ADD  *, AR0          ; Add arg to gvar
   SACL _gvar           ; Save in gvar

   LARP AR1             ; Pop off frame
   SBRK 2
   LAR  AR0, *          ; Restore frame pointer
   PSHD *               ; Move return addr to C2x stack
   RET
```

C程序调用汇编语言函数实例

13.2.8 使用内嵌asm语句

在C程序中可以用asm语句插入单行汇编语句，用来实现用C语言很难实现的功能。但是如果使用这种语句，一定小心不要破坏C运行环境，因为C编译器不会对这种插入的汇编语句进行检查或分析。

- 不要插入跳转或标签，否则会破坏编译器的堆栈处理算法，造成无法预期的后果。
- 不要修改C变量，但可以任意读取C变量当前值。
- 不要在内嵌汇编语句中书写修饰性汇编代码（如.text、.data等），否则会破坏汇编环境。

内嵌汇编语句可以用来在编译输出结果中添加注释。

```
asm(“  ***** this is an assembly language comment”);
```

13.2.9 C程序访问汇编语言变量

有时候需要在C程序中访问汇编语言变量，这通常有两种方式：

- 访问.bss块中的变量：
  - 将要访问的变量定义在.bss块中。
  - 用.golbal修饰要访问的变量。
  - 在汇编语言中以下横线“_”为前缀声明要访问的变量。
  - 在C语言中将变量声明为外部变量（extern），就可以进行正常访问。

(a) C program

```
extern int var;        /* External variable          */
var = 1;               /* Use the variable           */
```

(b) Assembly language program

```
* Note the use of underscores in the following lines

    .bss        _var,1      ; Define the variable
    .global     _var        ; Declare it as external
```

❑ 访问非.bss块中的变量：

要访问非.bss块中的变量，通常的办法是在汇编语言中定义一个查找表，然后在C语言中通过指针来访问。

- 首先定义变量，而且最好放在独立的初始化块中。
- 定义一个全局的标识指向对象的起始点，这样对象可以分配在存储器空间的任何位置。
- 在C程序中将这个对象定义为外部对象（extern），并且对象名称不带下横线“_”前缀，就可以对其进行正常访问。

(a) C program

```
extern float sine[];    /* This is the object          */
f = sine[4];            /* Access sine as normal array*/
```

(b) Assembly language program

```
    .global   _sine      ; Declare variable as external
    .sect  "sine_tab"    ; Make a separate section
_sine:          ; The table starts here
    .float 0.0
    .float 0.015987
    .float 0.022145
```

13.2.10 修改编译器输出结果

程序设计者可以检查和修改C编译器输出的汇编语言程序，然后再对其进行汇编编译和连接。

在C语言可以使用内嵌汇编语句在编译输出结果中添加注释，以改善编译器输出汇编程序的可读性。

```
/*************************************************************/
asm(“  ***** this is an assembly language comment”);
/*************************************************************/
```

13.2.11 系统初始化

C程序开始运行时，必须首先初始化C运行环境，这是通过c_int0函数完成的，这个函数在运行支持库（rts, runtime-support library）中。连接器会将这个函数的入口地址放置在复位中断向量处，使其可以在初始化时被调用。

c_int0函数进行以下工作以建立C运行环境：

- 为系统堆栈产生.stack块，并初始化堆栈指针。
- 从.cinit块将初始化数据拷贝到.bss块中相应的变量。
- 调用main函数，开始运行C程序。

用户可以对c_int0函数进行修改，但修改后的函数必须完成以上任务。

13.3 DSP程序烧写

DSP程序编写完成，调试无误后，就可以将程序烧写到DSP的FLASH（EEPROM）中，使DSP可以脱离仿真器独立运行。

烧写DSP程序也要通过仿真器来进行，仿真器制造者会提供相应的烧写程序。要注意不同型号的DSP往往使用不同的烧写程序，不同类型的仿真器对烧写环境也会有不同的要求，使用前要详细阅读相应说明（readme）。

- MP/MC*模式置为0（微计算机模式）
- 仿真RAM不被PS*信号选中

DSP烧写步骤：

- 清除flash内容（bc0.bat, bc1.bat）

  将FLASH（EEPROM）中的所有位清零（set to 0）。

- 擦除flash内容（be0.bat, be1.bat）

  将FLASH（EEPROM）中的所有位置位（set to 1）。

- 把目标程序写进flash中（bp16k.bat, bp32k.bat）

  将FLASH（EEPROM）中的所有选中的位清零（set to 0）。

烧写程序需要注意：

- 电路元件初始化同步问题：由于外部器件初始化可能较慢，DSP初始化完成后要等一会儿再访问外部慢速器件 。
- 用仿真器执行速度比较慢，循环时间比较长，而烧写到DSP中可能时间比较短，要对决定循环时间的循环次数重新考虑。
- 用仿真器调试的时候，DSP运行的一些资源（如堆栈等）用的是仿真器中的资源，烧写到DSP中执行必须利用DSP本身的资源，烧写前必须对.cmd文件中定义的各种资源进行详细考虑。
- 连接仿真器的时候和不连接仿真器的时候电路板上负载状态不同，可能改变板上某些信号的抖动情况，若有某部分功能模块工作不正常，可能是由上述原因引起的干扰造成的。
- 浮点数运算的问题：考虑用全局变量，因为局部变量都是在堆栈里生成的，对堆栈要求太多。