嵌入式 C/C++语言精华文章集锦

C/C+语言 struct 深层探索

出处：PConline　作者：宋宝华

1.　struct 的巨大作用

面对一个人的大型 C/C++程序时，只看其对 struct 的使用情况我们就可以对其编写者的编程经验进行评估。因为一个大型的 C/C++程序，势必要涉及一些(甚至大量)进行数据组合的结构体，这些结构体可以将原本意义属于一个整体的数据组合在一起。从某种程度上来说，会不会用 struct，怎样用 struct 是区别一个开发人员是否具备丰富开发经历的标志。

在网络协议、通信控制、嵌入式系统的 C/C++编程中，我们经常要传送的不是简单的字节流（char 型数组），而是多种数据组合起来的一个整体，其表现形式是一个结构体。

经验不足的开发人员往往将所有需要传送的内容依顺序保存在 char 型数组中，通过指针偏移的方法传送网络报文等信息。这样做编程复杂，易出错，而且一旦控制方式及通信协议有所变化，程序就要进行非常细致的修改。

一个有经验的开发者则灵活运用结构体，举一个例子，假设网络或控制协议中需要传送三种报文，其格式分别为 packetA、packetB、packetC：

```
struct structA
{
 int a;
 char b;
};

struct structB
{
 char a;
 short b;
};

struct structC
{
 int a;
 char b;
 float c;
}
```

优秀的程序设计者这样设计传送的报文：

```
struct CommuPacket
{
```

```
    int iPacketType;     //报文类型标志
    union               //每次传送的是三种报文中的一种，使用 union
    {
        struct structA packetA; struct structB packetB;
        struct structC packetC;
    }
};
```

在进行报文传送时，直接传送 struct CommuPacket 一个整体。

假设发送函数的原形如下：

```
// pSendData：发送字节流的首地址，iLen：要发送的长度
Send(char * pSendData, unsigned int  iLen);
```

发送方可以直接进行如下调用发送 struct CommuPacket 的一个实例 sendCommuPacket：

```
Send( (char *)&sendCommuPacket , sizeof(CommuPacket) );
```

假设接收函数的原形如下：

```
// pRecvData：发送字节流的首地址，iLen：要接收的长度
//返回值：实际接收到的字节数
unsigned int Recv(char * pRecvData, unsigned int  iLen);
```

接收方可以直接进行如下调用将接收到的数据保存在 struct CommuPacket 的一个实例 recvCommuPacket 中：

```
Recv( (char *)&recvCommuPacket , sizeof(CommuPacket) );
```

接着判断报文类型进行相应处理：

```
switch(recvCommuPacket. iPacketType)
{
case PACKET_A:
…    //A 类报文处理
break;
case PACKET_B:
…    //B 类报文处理
break;
case PACKET_C:
…   //C 类报文处理
break;
}
```

以上程序中最值得注意的是

```
Send( (char *)&sendCommuPacket , sizeof(CommuPacket) );
Recv( (char *)&recvCommuPacket , sizeof(CommuPacket) );
```

中的强制类型转换：(char *)&sendCommuPacket、(char *)&recvCommuPacket，先取地址，再转化为 char 型指针，这样就可以直接利用处理字节流的函数。

利用这种强制类型转化，我们还可以方便程序的编写，例如要对 sendCommuPacket 所处内存初始化为 0，可以这样调用标准库函数 memset()：

```
memset((char *)&sendCommuPacket,0, sizeof(CommuPacket));
```

2. struct的成员对齐

Intel、微软等公司曾经出过一道类似的面试题：

```
#include <iostream.h>
```

```
#pragma pack(8)
struct example1
{
  short a;
   long b;
};
struct example2
{
    char c;
    example1 struct1;
    short e;
};
#pragma pack()
int main(int argc, char* argv[])
{
  example2 struct2;
  cout << sizeof(example1) << endl;
  cout << sizeof(example2) << endl;
  cout << (unsigned int)(&struct2.struct1) - (unsigned int)(&struct2) << endl;
  return 0;
}
```

问程序的输入结果是什么？

答案是：

8

16

4

不明白？还是不明白？下面一一道来：

2.1 自然对界

struct 是一种复合数据类型，其构成元素既可以是基本数据类型（如 int、long、float 等）的变量，也可以是一些复合数据类型（如 array、struct、union 等）的数据单元。对于结构体，编译器会自动进行成员变量的对齐，以提高运算效率。缺省情况下，编译器为结构体的每个成员按其自然对界（natural alignment）条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

自然对界(natural alignment)即默认对齐方式，是指按结构体的成员中 size 最大的成员对齐。

例如：

```
struct naturalalign
{
    char a;
    short b;
    char c;
};
```

在上述结构体中，size 最大的是 short，其长度为 2 字节，因而结构体中的 char 成员 a、c 都以 2 为单位对齐，sizeof(naturalalign)的结果等于 6；

如果改为：

```
struct naturalalign
```

```
{
    char a;
    int b;
    char c;
};
```

其结果显然为 12。

2.2 指定对界

一般地，可以通过下面的方法来改变缺省的对界条件：

- **使用伪指令#pragma pack (n)，编译器将按照 n 个字节对齐；**
- **使用伪指令#pragma pack ()，取消自定义字节对齐方式。**

注意：如果#pragma pack (n)中指定的 n 大于结构体中最大成员的 size，则其不起作用，结构体仍然按照 size 最大的成员进行对界。

例如：

```
#pragma pack (n)
struct naturalalign
{
   char a;
   int b;
   char c;
};
#pragma pack ()
```

当 n 为 4、8、16 时，其对齐方式均一样，sizeof(naturalalign)的结果都等于 12。而当 n 为 2 时，其发挥了作用，使得 sizeof(naturalalign)的结果为 6。

在 VC++ 6.0 编译器中，我们可以指定其对界方式（见图 1），其操作方式为依次选择 projetct > setting > C/C++菜单，在 struct member alignment 中指定你要的对界方式。

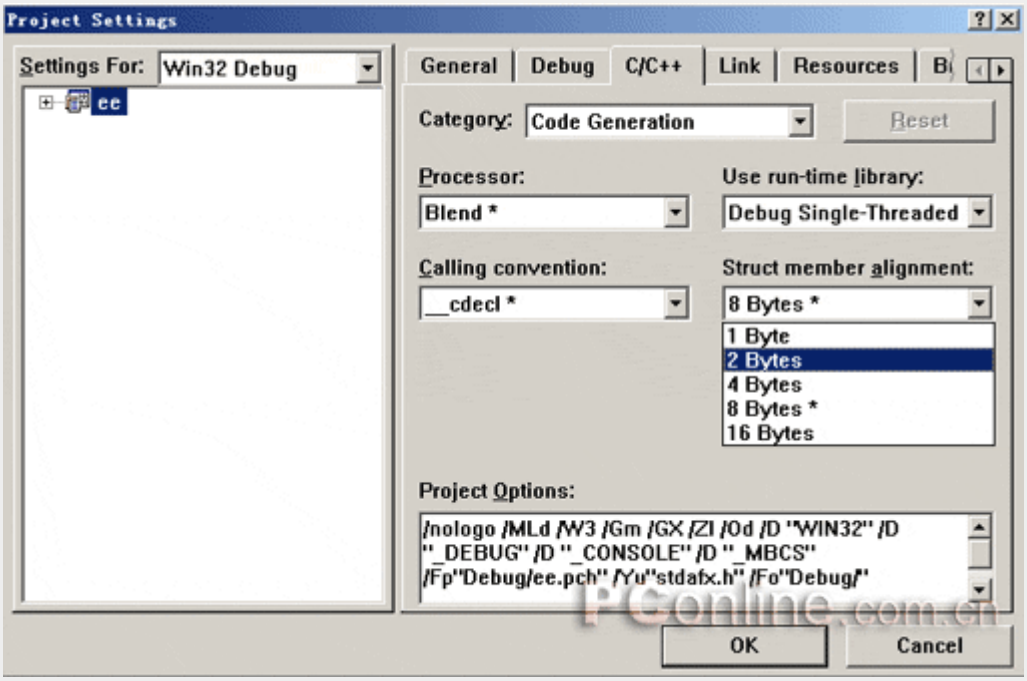


图 1　在 VC++ 6.0 中指定对界方式

另外，通过__attribute((aligned (n)))也可以让所作用的结构体成员对齐在 n 字节边界上，但是它较少被使用，因而不作详细讲解。

2.3 面试题的解答

至此，我们可以对 Intel、微软的面试题进行全面的解答。

程序中第 2 行#pragma pack (8)虽然指定了对界为 8，但是由于 struct example1 中的成员最大 size 为 4（long 变量 size 为 4），故 struct example1 仍然按 4 字节对界，struct example1 的 size 为 8，即第 18 行的输出结果；

struct example2 中包含了 struct example1,其本身包含的简单数据成员的最大 size 为 2(short 变量 e)，但是因为其包含了 struct example1，而 struct example1 中的最大成员 size 为 4，struct example2 也应以 4 对界，#pragma pack (8)中指定的对界对 struct example2 也不起作用，故 19 行的输出结果为 16；

由于 struct example2 中的成员以 4 为单位对界，故其 char 变量 c 后应补充 3 个空，其后才是成员 struct1 的内存空间，20 行的输出结果为 4。

3. C 和 C++间 struct 的深层区别

在 C++语言中 struct 具有了“类” 的功能，其与关键字 class 的区别在于 struct 中成员变量和函数的默认访问权限为 public，而 class 的为 private。

例如，定义 struct 类和 class 类：

```
struct structA
{
 char a;
 …
}
class classB
{
 char a;
 …
}
```

则：

```
structA a;
a.a = 'a';     //访问 public 成员，合法
classB b;
b.a = 'a';     //访问 private 成员，不合法
```

许多文献写到这里就认为已经给出了 C++中 struct 和 class 的全部区别，实则不然，另外一点需要注意的是：

C++中的 struct 保持了对 C 中 struct 的全面兼容（这符合 C++的初衷——“a better c”），因而，下面的操作是合法的：

```
//定义 struct
struct structA
{
char a;
char b;
int c;
};
```

```
structA a = {'a', 'a',1};    //  定义时直接赋初值
```

即 struct 可以在定义的时候直接以{ }对其成员变量赋初值，而 class 则不能，在经典书目《thinking C++ 2nd edition》中作者对此点进行了强调。

4. struct 编程注意事项

看看下面的程序：

```
1. #include <iostream.h>
2. struct structA
3. {
4.     int iMember;
5.     char *cMember;
6. };
7. int main(int argc, char* argv[])
8.{
9.     structA instant1,instant2;
10.    char c = 'a';
11.    instant1.iMember = 1;
12.    instant1.cMember = &c;
13.    instant2 = instant1;
14.    cout << *(instant1.cMember) << endl;
15.    *(instant2.cMember) = 'b';
16.    cout << *(instant1.cMember) << endl;
17.    return 0;
   }
```

14 行的输出结果是：a

16 行的输出结果是：b

Why?我们在 15 行对 instant2 的修改改变了 instant1 中成员的值！

原因在于 13 行的 instant2 = instant1 赋值语句采用的是变量逐个拷贝，这使得 instant1 和 instant2 中的 cMember 指向了同一片内存，因而对 instant2 的修改也是对 instant1 的修改。

在 C 语言中，当结构体中存在指针型成员时，一定要注意在采用赋值语句时是否将 2 个实例中的指针型成员指向了同一片内存。

在 C++语言中，当结构体中存在指针型成员时，我们需要重写 struct 的拷贝构造函数并进行“=”操作符重载。

C++中 extern "C"含义深层探索

作者： 宋宝华　**e-mail:** 21cnbao@21cn.com　**出处：** 太平洋电脑网

1.引言

C++语言的创建初衷是“a better C”，但是这并不意味着 C++中类似 C 语言的全局变量和函数所采用的编译和连接方式与 C 语言完全相同。作为一种欲与 C 兼容的语言，C++保留了一部分过程式语言的特点（被世人称为“不彻底地面向对象”），因而它可以定义不属于任何类的全局变量和函数。

但是，C++毕竟是一种面向对象的程序设计语言，为了支持函数的重载，C++对全局函数的处理方式与 C 有明显的不同。

2.从标准头文件说起

某企业曾经给出如下的一道面试题：

面试题

为什么标准头文件都有类似以下的结构？

```
#ifndef __INCvxWorksh
#define __INCvxWorksh
#ifdef __cplusplus
extern "C" {
#endif
/*...*/
#ifdef __cplusplus
}
#endif
#endif /* __INCvxWorksh */
```

分析

显然，头文件中的编译宏“#ifndef __INCvxWorksh、#define __INCvxWorksh、#endif” 的作用是防止该头文件被重复引用。

那么

```
#ifdef __cplusplus
extern "C" {
#endif
#ifdef __cplusplus
}
#endif
```

的作用又是什么呢？我们将在下文一一道来。

3.深层揭密 extern "C"

extern "C" 包含双重含义，从字面上即可得到：首先，被它修饰的目标是“extern”的；其次，被它修饰的目标是“C”的。让我们来详细解读这两重含义。

（1）被 extern "C"限定的函数或变量是 extern 类型的；

extern 是 C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。记住，下列语句：

```
extern int a;
```

仅仅是一个变量的声明，其并不是在定义变量 a，并未为 a 分配内存空间。变量 a 在所有模块中作为一种全局变量只能被定义一次，否则会出现连接错误。

通常，在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字 extern 声明。例如，如果模块 B 欲引用该模块 A 中定义的全局变量和函数时只需包含模块 A 的头文件即可。这样，模块 B 中调用模块 A 中的函数时，在编译阶段，模块 B 虽然找不到该函数，但是并不会报错；它会在连接阶段中从模块 A 编译生成的目标代码中找到此函数。

与 extern 对应的关键字是 static，被它修饰的全局变量和函数只能在本模块中使用。因此，一个函数或变量只可能被本模块使用时，其不可能被 extern “C”修饰。

（2）被 extern "C"修饰的变量和函数是按照 C 语言方式编译和连接的；

未加 extern “C”声明时的编译方式

首先看看 C++中对类似 C 的函数是怎样编译的。

作为一种面向对象的语言，C++支持函数重载，而过程式语言 C 则不支持。函数被 C++编译后在符号库中的名字与 C 语言的不同。例如，假设某个函数的原型为：

```
void foo( int x, int y );
```

该函数被 C 编译器编译后在符号库中的名字为_foo，而 C++编译器则会产生像_foo_int_int 之类的名字（不同的编译器可能生成的名字不同，但是都采用了相同的机制，生成的新名字称为“mangled name”）。_foo_int_int 这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。例如，在 C++中，函数 void foo(int x, int y)与 void foo(int x, float y)编译生成的符号是不相同的，后者为_foo_int_float。

同样地，C++中的变量除支持局部变量外，还支持类成员变量和全局变量。用户所编写程序的类成员变量可能与全局变量同名，我们以"."来区分。而本质上，编译器在进行编译时，与函数的处理相似，也为类中的变量取了一个独一无二的名字，这个名字与用户程序中同名的全局变量名字不同。

未加 extern "C"声明时的连接方式

假设在 C++中，模块 A 的头文件如下：

```
// 模块 A 头文件  moduleA.h
#ifndef MODULE_A_H
   #define MODULE_A_H
    int foo( int x, int y );
#endif
```

在模块 B 中引用该函数：

```
// 模块 B 实现文件  moduleB.cpp
#include "moduleA.h"
foo(2,3);
```

实际上，在连接阶段，连接器会从模块 A 生成的目标文件 moduleA.obj 中寻找_foo_int_int 这样的符号！

加 extern "C"声明后的编译和连接方式

加 extern "C"声明后，模块 A 的头文件变为：

```
// 模块 A 头文件  moduleA.h
#ifndef MODULE_A_H
   #define MODULE_A_H
    extern "C" int foo( int x, int y );
#endif
```

在模块 B 的实现文件中仍然调用 foo(2,3)，其结果是：

（1）模块 A 编译生成 foo 的目标代码时，没有对其名字进行特殊处理，采用了 C 语言的方式；

（2）连接器在为模块 B 的目标代码寻找 foo(2,3)调用时，寻找的是未经修改的符号名_foo。

如果在模块 A 中函数声明了 foo 为 extern "C"类型，而模块 B 中包含的是 extern int foo(int x, int y ），则模块 B 找不到模块 A 中的函数；反之亦然。

所以，可以用一句话概括 extern “C”这个声明的真实目的（任何语言中的任何语法特性的诞生都不是随意而为的，来源于真实世界的需求驱动。我们在思考问题时，不能只停留在这个语言是怎么做的，还要问一问它为什么要这么做，动机是什么，这样我们可以更深入地理解许多问题）：

实现 C++与 C 及其它语言的混合编程。

明白了 C++中 extern "C"的设立动机，我们下面来具体分析 extern "C"通常的使用技巧。

4.extern "C"的惯用法

（1）在C++中引用C语言中的函数和变量，在包含C语言头文件（假设为cExample.h）时，需进行下列处理：

```
extern "C"
{
#include "cExample.h"
}
```

而在C语言的头文件中，对其外部函数只能指定为extern类型，C语言中不支持extern "C"声明，在.c文件中包含了extern "C"时会出现编译语法错误。

笔者编写的C++引用C函数例子工程中包含的三个文件的源代码如下：

```
/* c语言头文件：cExample.h */
#ifndef C_EXAMPLE_H
#define C_EXAMPLE_H
extern int add(int x,int y);
#endif
/* c语言实现文件：cExample.c */
#include "cExample.h"
int add( int x, int y )
{
   return x + y;
}
// c++实现文件，调用add：cppFile.cpp
extern "C"
{
#include "cExample.h"
}
int main(int argc, char* argv[])
{
   add(2,3);
   return 0;
}
```

如果C++调用一个C语言编写的.DLL时，当包括.DLL的头文件或声明接口函数时，应加extern "C" { }。

（2）在C中引用C++语言中的函数和变量时，C++的头文件需添加extern "C"，但是在C语言中不能直接引用声明了extern "C"的该头文件，应该仅将C文件中将C++中定义的extern "C"函数声明为extern类型。

笔者编写的C引用C++函数例子工程中包含的三个文件的源代码如下：

```
//C++头文件 cppExample.h
#ifndef CPP_EXAMPLE_H
#define CPP_EXAMPLE_H
extern "C" int add( int x, int y );
#endif
//C++实现文件 cppExample.cpp
#include "cppExample.h"
int add( int x, int y )
```

```
{
    return x + y;
}
/* C 实现文件 cFile.c
/* 这样会编译出错：#include "cExample.h" */
extern int add( int x, int y );
int main( int argc, char* argv[] )
{
    add( 2, 3 );
    return 0;
}
```

如果深入理解了第 3 节中所阐述的 extern "C"在编译和连接阶段发挥的作用，就能真正理解本节所阐述的从 C++引用 C 函数和 C 引用 C++函数的惯用法。对第 4 节给出的示例代码，需要特别留意各个细节。

C 语言高效编程的几招

编写高效简洁的 C 语言代码，是许多软件工程师追求的目标。本文就工作中的一些体会和经验做相关的阐述，不对的地方请

各位指教。

第 1 招：以空间换时间

计算机程序中最大的矛盾是空间和时间的矛盾，那么，从这个角度出发逆向思维来考虑程序的效率问题，我们就有了解决问题

的第 1 招--以空间换时间。

例如：字符串的赋值。

方法 A，通常的办法：	方法 B:
#define LEN 32	const char string2[LEN]="This is an example!"
char string1 [LEN];	char*cp;
memset (string1,0,LEN);	cp=string2;
strcpy (string1,"This is an example!!"	(使用的时候可以直接用指针来操作。)

从上面的例子可以看出，A 和 B 的效率是不能比的。在同样的存储空间下，B 直接使用指针就可以操作了，而 A 需要调用两个字符函数才能完成。B 的缺点在于灵活性没有 A 好。在需要频繁更改一个字符串内容的时候，A 具有更好的灵活性；如果采用方法 B，则需要预存许多字符串，虽然占用了 大量的内存，但是获得了程序执行的高效率。

如果系统的实时性要求很高，内存还有一些，那我推荐你使用该招数。

该招数的边招--使用宏函数而不是函数。举例如下：

方法 C:

```
#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

int BIT_MASK (int_bf)

{

return ((IU<<(bw##_bf))-1)<<(bs##_bf);

}

void SET_BITS(int_dst,int_bf,int_val)

{

_dst=((_dst) & ~ (BIT_MASK(_bf)))|\
(((_val)<<<(bs##_bf))&(BIT_MASK(_bf)))

}

SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumb
er);
```

方法 D:

```
#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

#define bmMCDR2_ADDRESS BIT_MASK
(MCDR2_ADDRESS)

#define BIT_MASK(_bf)(((1U<<(bw##_bf))-1)<<
(bs##_bf)

#define SET_BITS(_dst,_bf,_val)\
((_dst)=((_dst)&~(BIT_MASK(_bf)))|
(((_val)<<(bs##_bf))&(BIT_MASK(_bf))))

SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumb
er);
```

函数和宏函数的区别就在于，宏函数占用了大量的空间，而函数占用了时间。大家要知道的是，函数调用是要使用系统的栈来保存数据的，如果编译器里有栈检查选项，一般在函数的头会嵌入一些汇编语句对当前栈进行检查；同时，CPU 也要在函数调用时保存和恢复当前的现场，进行压栈和弹栈操作，所以，函数调用需要一些 CPU 时间。而宏函数不存在这个问题。宏函数仅仅作为预先写好的代码嵌入到当前程序，不会产生函数调用，所以仅仅是占用了空间，在频繁调用同一个宏函数的时候，该现象尤其突出。

D 方法是我看到的最好的置位操作函数，是 ARM 公司源码的一部分，在短短的三行内实现了很多功能，几乎涵盖了所有的位操作功能。C 方法是其变体，其中滋味还需大家仔细体会。

第 2 招：数学方法解决问题

现在我们演绎高效 C 语言编写的第二招--采用数学方法来解决问题。

数学是计算机之母，没有数学的依据和基础，就没有计算机的发展，所以在编写程序的时候，采用一些数学方法会对程序的执行效率有数量级的提高。

举例如下，求 1~100 的和。

方法 E

```
int I,j;
```

方法 F

```
int I;
```

for (I=1; I<=100; I++){ *j+=I;* }	I=(100*(1+100))/2

这个例子是我印象最深的一个数学用例，是我的饿计算机启蒙老师考我的。当时我只有小学三年级，可惜我当时不知道用公式 Nx(N+1)/2 来解决这个问题。方法 E 循环了 100 次才解决问题，也就是说最少用了 100 个赋值、100 个判断、200 个加法(I 和 j)；而方法 F 仅仅用了 1 个加法、1 个乘法、1 次除法。效果自然不言而喻。所以，现在我在编程序的时候，更多的是动脑筋找规律，最大限度地发挥数学的威力来提高程序运行的效率。

第 3 招：使用位操作

实现高效的 C 语言编写的第三招--使用位操作，减少除法和取模的运算。

在计算机程序中，数据的位是可以操作的最小数据单位，理论上可以用"位运算"来完成所有的运算和操作。一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以有效地提高程序运行的效率。举例台如下：

方法 G	方法 H
int I,J; I=257/8; J=456%32;	int I,J; I=257>>3; J=456-(456>>4<<4);

在字面上好象 H 比 G 麻烦了好多，但是，仔细查看产生的汇编代码就会明白，方法 G 调用了基本的取模函数和除法函数，既有函数调用，还有很多汇编代码和寄存器参与运算；而方法 H 则仅仅是几句相关的汇编，代码更简洁、效率更高。当然，由于编译器的不同，可能效率的差距不大，但是，以我目前遇到的 MS C,ARM C 来看，效率的差距还是不小。相关汇编代码就不在这里列举了。

运用这招需要注意的是，因为 CPU 的不同而产生的问题。比如说，在 PC 上用这招编写的程序，并在 PC 上调试通过，在移植到一个 16 位机平台上的时候，可能会产生代码隐患。所以只有在一定技术进阶的基础下才可以使用这招。

第 4 招：汇编嵌入

高效 C 语言编程的必杀技，第四招--嵌入汇编。

"在熟悉汇编语言的人眼里，C 语言编写的程序都是垃圾"。这种说法虽然偏激了一些，但是却有它的道理。汇编语言是效率最高的计算机语言，但是，不可能靠着它来写一个操作系统吧？所以，为了获得程序的高效率，我们只好采用变通的方法--嵌入汇编、混合编程。

举例如下，将数组一赋值给数组二，要求每一个字节都相符。char string1[1024], string2[1024];

方法 I int I; for (I=0; I<1024; I++) *(string2+I)=*(string1+I)	方法 J #int I; for(I=0; I<1024; I++) *(string2+I)=*(string1+I); #else #ifdef_ARM_ _asm { MOV R0,string1 MOV R1,string2 MOV R2,#0 loop: LDMIA R0!,[R3-R11] STMIA R1!,[R3-R11] ADD R2,R2,#8 CMP R2, #400 BNE loop } #endif

方法 I 是最常见的方法，使用了 1024 次循环；方法 J 则根据平台不同做了区分，在 ARM 平台下，用嵌入汇编仅用 128 次循环就完成了同样的操作。这里有朋友会说，为什么不用标准的内存拷贝函数呢？这是因为在源数据里可能含有数据为 0 的字节，这样的话，标准库函数会提前结束而不会完成我们要求的操作。这个例程典型应用于 LCD 数据的拷贝过程。根据不同的 CPU，熟练使用相应的嵌入汇编，可以大大提高程序执行的效率。

虽然是必杀技，但是如果轻易使用会付出惨重的代价。这是因为，使用了嵌入汇编，便限制了程序的可移植性，使程序在不同平台移植的过程中，卧虎藏龙、险象环生！同时该招数也与现代软件工程的思想相违背，只有在迫不得已的情况下才可以采用。切记。

使用 C 语言进行高效率编程，我的体会仅此而已。在此已本文抛砖引玉，还请各位高手共同切磋。希望各位能给出更好的方法，大家一起提高我们的编程技巧。

摘自《单片机与嵌入式系统应用》2003.9

想成为嵌入式程序员应知道的 0x10 个基本问题

-|endeaver 发表于 2006-3-8 16:16:00

C 语言测试是招聘嵌入式系统程序员过程中必须而且有效的方法。这些年，我既参加也组织了许多这种测试，在这过程中我意识到这些测试能为带面试者和被面试者提供许多有用信息，此外，撇开面试的压力不谈，这种测试也是相当有趣的。

从被面试者的角度来讲，你能了解许多关于出题者或监考者的情况。这个测试只是出题者为显示其对 ANSI 标准细节的知识而不是技术技巧而设计吗？这个愚蠢的问题吗？如要你答出某个字符的 ASCII 值。这些问题着重考察你的系统调用和内存分配策略方面的能力吗？这标志着出题者也许花时间在微机上而不上在嵌入式系统上。如果上述任何问题的答案是"是"的话，那么我知道我得认真考虑我是否应该去做这份工作。

从面试者的角度来讲，一个测试也许能从多方面揭示应试者的素质：最基本的，你能了解应试者 C 语言的水平。不管怎么样，看一下这人如何回答他不会的问题也是满有趣。应试者是以好的直觉做出明智的选择，还是只是瞎蒙呢？当应试者在某个问题上卡住时是找借口呢，还是表现出对问题的真正的好奇心，把这看成学习的机会呢？我发现这些信息与他们的测试成绩一样有用。

有了这些想法，我决定出一些真正针对嵌入式系统的考题，希望这些令人头痛的考题能给正在找工作的人一点帮住。这些问题都是我这些年实际碰到的。其中有些题很难，但它们应该都能给你一点启迪。

这个测试适于不同水平的应试者，大多数初级水平的应试者的成绩会很差，经验丰富的程序员应该有很好的成绩。为了让你能自己决定某些问题的偏好，每个问题没有分配分数，如果选择这些考题为你所用，请自行按你的意思分配分数。

预处理器（Preprocessor）

1．用预处理指令#define 声明一个常数，用以表明 1 年中有多少秒（忽略闰年问题）

```
#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

我在这想看到几件事情：

•; #define 语法的基本知识（例如：不能以分号结束，括号的使用，等等）

•; 懂得预处理器将为你计算常数表达式的值，因此，直接写出你是如何计算一年中有多少秒而不是计算出实际的值，是更清晰而没有代价的。

•; 意识到这个表达式将使一个 16 位机的整型数溢出-因此要用到长整型符号 L,告诉编译器这个常数是的长整型数。

•; 如果你在你的表达式中用到 UL（表示无符号长整型），那么你有了一个好的起点。记住，第一印象很重要。

2．写一个"标准"宏 MIN ，这个宏输入两个参数并返回较小的一个。

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

这个测试是为下面的目的而设的：

•; 标识#define 在宏中应用的基本知识。这是很重要的，因为直到嵌入(inline)操作符变为标准 C 的一部分，宏是方便产生嵌入代码的唯一方法，对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

•; 三重条件操作符的知识。这个操作符存在 C 语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码，了解这个用法是很重要的。

•; 懂得在宏中小心地把参数用括号括起来

•; 我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

```
least = MIN(*p++, b);
```

3. 预处理器标识#error 的目的是什么？

如果你不知道答案，请看参考文献 1。这问题对区分一个正常的伙计和一个书呆子是很有用的。只有书呆子才会读 C 语言课本的附录去找出象这种问题的答案。当然如果你不是在找一个书呆子，那么应试者最好希望自己不要知道答案。

死循环（Infinite loops）

4. 嵌入式系统中经常要用到无限循环，你怎么样用 C 编写死循环呢？

这个问题用几个解决方案。我首选的方案是：

```
while(1)
{
?}
```

一些程序员更喜欢如下方案：

```
for(;;)
{
?}
```

这个实现方式让我为难，因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这个作为方案，我将用这个作为一个机会去探究他们这样做的基本原理。如果他们的基本答案是："我被教着这样做，但从没有想到过为什么。"这会给我留下一个坏印象。

第三个方案是用 goto

```
Loop:
...
goto Loop;
```

应试者如给出上面的方案，这说明或者他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。

数据声明（Data declarations）

5. 用变量 a 给出下面的定义

a) 一个整型数（An integer）

b)一个指向整型数的指针（ A pointer to an integer）

c)一个指向指针的的指针，它指向的指针是指向一个整型数（ A pointer to a pointer to an intege）r

d)一个有 10 个整型数的数组（ An array of 10 integers）

e) 一个有 10 个指针的数组，该指针是指向一个整型数的。（An array of 10 pointers to integers）

f) 一个指向有 10 个整型数数组的指针（ A pointer to an array of 10 integers）

g) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer）

h)一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数（ An array of ten pointers to functions that take an integer argument and return an integer ）

答案是：

```
a) int a; // An integer
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer
```

人们经常声称这里有几个问题是那种要翻一下书才能回答的问题，我同意这种说法。当我写这篇文章时，为了确定语法的正确性，我的确查了一下书。但是当我被面试的时候，我期望被问到这个问题（或者相近的问题）。因为在被面试的这段时间里，我确定我知道这个问题的答案。应试者如果不知道所有的答案（或至少大部分答案），那么也就没有为这次面试做准备，如果该面试者没有为这次面试做准备，那么他又能为什么出准备呢？

Static

6. 关键字 static 的作用是什么？

这个简单的问题很少有人能回答完全。在 C 语言中，关键字 static 有三个明显的作用：

•; 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。

•; 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。

•; 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

大多数应试者能正确回答第一部分，一部分能正确回答第二部分，同是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数据和代码范围的好处和重要性。

Const

7. 关键字 const 有什么含意？

我只要一听到被面试者说："const 意味着常数"，我就知道我正在和一个业余者打交道。去年 Dan Saks 已经在他的文章里完全概括了 const 的所有用法，因此 ESP(译者：Embedded Systems Programming)的每一位读者应该非常熟悉 const 能做什么和不能做什么.如果你从没有读到那篇文章，只要能说出 const 意味着"只读"就可以了。尽管这个答案不是完全的答案，但我接受它作为一个正确的答案。（如果你想知道更详细的答案，仔细读一下 Saks 的文章吧。）

如果应试者能正确回答这个问题，我将问他一个附加的问题：

下面的声明都是什么意思？

```
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
```

/******/

前两个的作用是一样，a 是一个常整型数。第三个意味着 a 是一个指向常整型数的指针（也就是，整型数是不可修改的，但指针可以）。第四个意思 a 是一个指向整型数的常指针（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的）。最后一个意味着 a 是一个指向常整型数的常指针（也就是说，指针指向的整型数是不可修改的，同时指针也是不可修改的）。如果应试者能正确回答这些问题，那么他就给我留下了一个好印象。顺带提一句，也许你可能会问，即使不用关键字 const，也还是能很容易写出功能正确的程序，那么我为什么还要如此看重关键字 const 呢？我也如下的几下理由：

•; 关键字 const 的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。（当然，懂得用 const 的程序员很少会留下的垃圾让别人来清理的。）

•; 通过给优化器一些附加的信息，使用关键字 const 也许能产生更紧凑的代码。

•; 合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少 bug 的出现。

Volatile

8. 关键字 volatile 有什么含意?并给出三个不同的例子。

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：

•; 并行设备的硬件寄存器（如：状态寄存器）
•; 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)
•; 多线程应用中被几个任务共享的变量

回答不出这个问题的人是不会被雇佣的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等等打交道，所有这些都要求用到 volatile 变量。不懂得 volatile 的内容将会带来灾难。
假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，看一下这家伙是不是直正懂得 volatile 完全的重要性。
•; 一个参数既可以是 const 还可以是 volatile 吗？解释为什么。
•; 一个指针可以是 volatile 吗？解释为什么。
•; 下面的函数有什么错误：

```
int square(volatile int *ptr)
{
return *ptr * *ptr;
}
```

下面是答案：
•; 是的。一个例子是只读的状态寄存器。它是 volatile 因为它可能被意想不到地改变。它是 const 因为程序不应该试图去修改它。
•; 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个 buffer 的指针时。
•; 这段代码有点变态。这段代码的目的是用来返指针*ptr 指向值的平方，但是，由于*ptr 指向一个 volatile 型参数，编译器将产生类似下面的代码：

```
int square(volatile int *ptr)
{
int a,b;
a = *ptr;
b = *ptr;
return a * b;
}
```

由于*ptr 的值可能被意想不到地该变，因此 a 和 b 可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

```
long square(volatile int *ptr)
{
int a;
a = *ptr;
return a * a;
}
```

位操作（Bit manipulation）

9. 嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清除 a 的 bit 3。在以上两个操作中，要保持其它位不变。

对这个问题有三种基本的反应

•; 不知道如何下手。该被面者从没做过任何嵌入式系统的工作。

•; 用 bit fields。Bit fields 是被扔到 C 语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了的你的代码是不可重用的。我最近不幸看到 Infineon 为其较复杂的通信芯片写的驱动程序，它用到了 bit fields 因此完全对我无用，因为我的编译器用其它的方式来实现 bit fields 的。从道德讲：永远不要让一个非嵌入式的家伙粘实际硬件的边。

•; 用 #defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。最佳的解决方案如下：

```
#define BIT3 (0x1 << 3)
static int a;

void set_bit3(void) {
a |= BIT3;
}
void clear_bit3(void) {
a &= ~BIT3;
}
```

一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数，这也是可以接受的。我希望看到几个要点：说明常数、|=和&=~操作。

访问固定的内存位置（Accessing fixed memory locations）

10. 嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为 0x67a9 的整型变量的值为 0xaa66。编译器是一个纯粹的 ANSI 编译器。写代码去完成这一任务。

这一问题测试你是否知道为了访问一绝对地址把一个整型数强制转换（typecast）为一指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下：

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

A more obscure approach is:

一个较晦涩的方法是：

```
*(int * const)(0x67a9) = 0xaa55;
```

即使你的品味更接近第二种方案，但我建议你在面试时使用第一种方案。

中断（Interrupts）

11. 中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展—让标准 C 支持中断。具代表事实是，产生了一个新的关键字__interrupt。下面的代码就使用了__interrupt 关键字去定义了一个中断服务子程序(ISR)，请评论一下这段代码的。

```
__interrupt double compute_area (double radius)
{
```

```
double area = PI * radius * radius;
printf("\nArea = %f", area);
return area;
}
```

这个函数有太多的错误了，以至让人不知从何说起了：

•; ISR 不能返回一个值。如果你不懂这个，那么你不会被雇用的。

•; ISR 不能传递参数。如果你没有看到这一点，你被雇用的机会等同第一项。

•; 在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈，有些处理器/编译器就是不允许在 ISR 中做浮点运算。此外，ISR 应该是短而有效率的，在 ISR 中做浮点运算是不明智的。

•; 与第三点一脉相承，printf()经常有重入和性能上的问题。如果你丢掉了第三和第四点，我不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。

*****

代码例子（Code examples）

12 . 下面的代码输出是什么，为什么？

```
void foo(void)
{
unsigned int a = 6;
int b = -20;
(a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

这个问题测试你是否懂得 C 语言中的整数自动转换原则，我发现有些开发者懂得极少这些东西。不管如何，这无符号整型问题的答案是输出是">6"。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。 因此-20 变成了一个非常大的正整数，所以该表达式计算出的结果大于 6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是丰常重要的。如果你答错了这个问题，你也就到了得不到这份工作的边缘。

13. 评价下面的代码片断：

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

对于一个 int 型不是 16 位的处理器为说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ~0;
```

这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里，好的嵌入式程序员非常准确地明白硬件的细节和它的局限，然而 PC 机程序往往把硬件作为一个无法避免的烦恼。

到了这个阶段，应试者或者完全垂头丧气了或者信心满满志在必得。如果显然应试者不是很好，那么这个测试就在这里结束了。但如果显然应试者做得不错，那么我就扔出下面的追加问题，这些问题是比较难的，我想仅仅非常优秀的应试者能做得不错。提出这些问题，我希望更多看到应试者应付问题的方法，而不是答案。不管如何，你就当是这个娱乐吧...

动态内存分配（Dynamic memory allocation）

14. 尽管不像非嵌入式计算机那么常见，嵌入式系统还是有从堆（heap）中动态分配内存的过程的。那么嵌入式系统中，动态分配内存可能发生的问题是什么？

这里，我期望应试者能提到内存碎片，碎片收集的问题，变量的持行时间等等。这个主题已经在 ESP 杂志中被广泛地讨论过了（主要是 P.J. Plauger, 他的解释远远超过我这里能提到的任何解释），所有回过头看一下这些杂志吧！让应试者进入一种虚假的安全感觉后，我拿出这么一个小节目：

下面的代码片段的输出是什么，为什么？

```
char *ptr;
if ((ptr = (char *)malloc(0)) ==
NULL)
else
puts("Got a null pointer");
puts("Got a valid pointer");
```

这是一个有趣的问题。最近在我的一个同事不经意把 0 值传给了函数 malloc，得到了一个合法的指针之后，我才想到这个问题。这就是上面的代码，该代码的输出是"Got a valid pointer"。我用这个来开始讨论这样的一问题，看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要，但解决问题的方法和你做决定的基本原理更重要些。

Typedef

:

15 Typedef 在 C 语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如，思考一下下面的例子：

```
#define dPS struct s *
typedef struct s * tPS;
```

以上两种情况的意图都是要定义 dPS 和 tPS 作为一个指向结构 s 指针。哪种方法更好呢？（如果有的话）为什么？

这是一个非常微妙的问题，任何人答对这个问题（正当的原因）是应当被恭喜的。答案是：typedef 更好。思考下面的例子：

```
dPS p1,p2;
tPS p3,p4;
```

第一个扩展为

```
struct s * p1, p2;
```

.

上面的代码定义 p1 为一个指向结构的指，p2 为一个实际的结构，这也许不是你想要的。第二个例子正确地定义了 p3 和 p4 两个指针。

晦涩的语法

16 . C 语言同意一些令人震惊的结构,下面的结构是合法的吗，如果是它做些什么？

```
int a = 5, b = 7, c;
c = a+++b;
```

这个问题将做为这个测验的一个愉快的结尾。不管你相不相信，上面的例子是完全合乎语法的。问题是编译器如何处理它？水平不高的编译作者实际上会争论这个问题，根据最处理原则，编译器应当能处理尽可能所有合法的用法。因此，上面的代码被处理成：

```
c = a++ + b;
```

因此，这段代码持行后 a = 6, b = 7, c = 12。
如果你知道答案，或猜出正确答案，做得好。如果你不知道答案，我也不把这个当作问题。我发现这个问题的最大好处是这是一个关于代码编写风格，代码的可读性，代码的可修改性的好的话题。
好了，伙计们，你现在已经做完所有的测试了。这就是我出的 C 语言测试题，我怀着愉快的心情写完它，希望你以同样的心情读完它。如果是认为这是一个好的测试，那么尽量都用到你的找工作的过程中去吧。天知道也许过个一两年，我就不做现在的工作，也需要找一个。
Nigel Jones 是一个顾问，现在住在 Maryland，当他不在水下时，你能在多个范围的嵌入项目中找到他。 他很高兴能收到读者的来信，他的 email 地址是: NAJones@compuserve.com 。
References

•; Jones, Nigel, "In Praise of the #error directive," Embedded Systems Programming, September 1999, p. 114.
•; Jones, Nigel, " Efficient C Code for Eight-bit MCUs ," Embedded Systems Programming, November 1998, p. 66

C 语言嵌入式系统编程修炼

C 语言嵌入式系统编程修炼之一:背景篇

作者:宋宝华　　更新日期:2005-08-30
来源:yesky.com

不同于一般形式的软件编程，嵌入式系统编程建立在特定的硬件平台上，势必要求其编程语言具备较强的硬件直接操作能力。无疑，汇编语言具备这样的特质。但是，归因于汇编语言开发过程的复杂性，它并不是嵌入式系统开发的一般选择。而与之相比，C 语言--一种"高级的低级"语言，则成为嵌入式系统开发的最佳选择。笔者在嵌入式系统项目的开发过程中，一次又一次感受到 C 语言的精妙，沉醉于 C 语言给嵌入式开发带来的便利。

图 1 给出了本文的讨论所基于的硬件平台，实际上，这也是大多数嵌入式系统的硬件平台。它包括两部分：

（1） 以通用处理器为中心的协议处理模块，用于网络控制协议的处理；

（2） 以数字信号处理器（DSP）为中心的信号处理模块，用于调制、解调和数/模信号转换。

本文的讨论主要围绕以通用处理器为中心的协议处理模块进行，因为它更多地牵涉到具体的 C 语言编程技巧。而 DSP 编程则重点关注具体的数字信号处理算法，主要涉及通信领域的知识，不是本文的讨论重点。

着眼于讨论普遍的嵌入式系统 C 编程技巧，系统的协议处理模块没有选择特别的 CPU，而是选择了众所周知的 CPU 芯片--80186，每一位学习过《微机原理》的读者都应该对此芯片有一个基本的认识，且对其指令集比较熟悉。80186 的字长是 16 位，可以寻址到的内存空间为 1MB，只有实地址模式。C 语言编译生成的指针为 32 位（双字），高 16 位为段地址，低 16 位为段内偏移，一段最多 64KB。

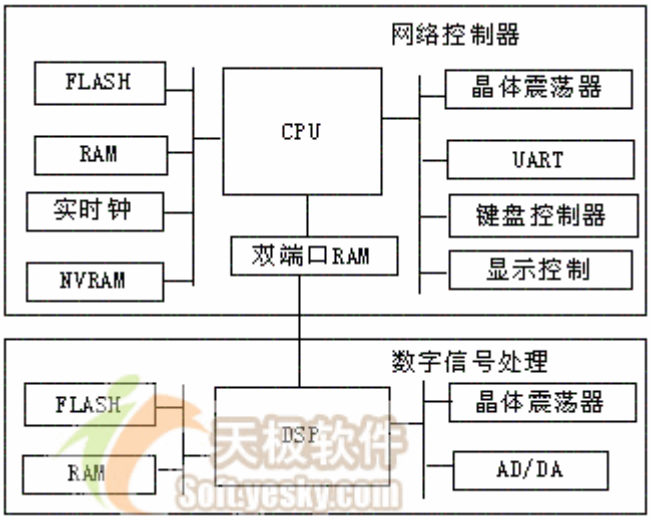


图 1 系统硬件架构

协议处理模块中的 FLASH 和 RAM 几乎是每个嵌入式系统的必备设备，前者用于存储程序，后者则是程序运行时指令及数据的存放位置。系统所选择的 FLASH 和 RAM 的位宽都为 16 位，与 CPU 一致。

实时钟芯片可以为系统定时，给出当前的年、月、日及具体时间（小时、分、秒及毫秒），可以设定其经过一段时间即向 CPU 提出中断或设定报警时间到来时向 CPU 提出中断（类似闹钟功能）。

NVRAM（非易失去性 RAM）具有掉电不丢失数据的特性，可以用于保存系统的设置信息，譬如网络协议参数等。在系统掉电或重新启动后，仍然可以读取先前的设置信息。其位宽为 8 位，比 CPU 字长小。文章特意选择一个与 CPU 字长不一致的存储芯片，为后文中一节的讨论创造条件。

UART 则完成 CPU 并行数据传输与 RS-232 串行数据传输的转换，它可以在接收到[1~MAX_BUFFER]字节后向 CPU 提出中断，MAX_BUFFER 为 UART 芯片存储接收到字节的最大缓冲区。

键盘控制器和显示控制器则完成系统人机界面的控制。

以上提供的是一个较完备的嵌入式系统硬件架构，实际的系统可能包含更少的外设。之所以选择一个完备的系统，是为了后文更全面的讨论嵌入式系统 C 语言编程技巧的方方面面，所有设备都会成为后文的分析目标。

嵌入式系统需要良好的软件开发环境的支持，由于嵌入式系统的目标机资源受限，不可能在其上建立庞大、复杂的开发环境，因而其开发环境和目标运行环境相互分离。因此，嵌入式应用软件的开发方式一般是，在宿主机(Host)上建立开发环境，进行应用程序编码和交叉编译，然后宿主机同目标机(Target)建立连接，将应用程序下载到目标机上进行交叉调试，经过调试和优化，最后将应用程序固化到目标机中实际运行。

CAD-UL 是适用于 x86 处理器的嵌入式应用软件开发环境，它运行在 Windows 操作系统之上，可生成 x86 处理器的目标

代码并通过 PC 机的 COM 口（RS-232 串口）或以太网口下载到目标机上运行，如图 2。其驻留于目标机 FLASH 存储器中的 monitor 程序可以监控宿主机 Windows 调试平台上的用户调试指令，获取 CPU 寄存器的值及目标机存储空间、I/O 空间的内容。

图 2 交叉开发环境

后续章节将从软件架构、内存操作、屏幕操作、键盘操作、性能优化等多方面阐述 C 语言嵌入式系统的编程技巧。软件架构是一个宏观概念，与具体硬件的联系不大；内存操作主要涉及系统中的 FLASH、RAM 和 NVRAM 芯片；屏幕操作则涉及显示控制器和实时钟；键盘操作主要涉及键盘控制器；性能优化则给出一些具体的减小程序时间、空间消耗的技巧。

在我们的修炼旅途中将经过 25 个关口，这些关口主分为两类，一类是技巧型，有很强的适用性；一类则是常识型，在理论上有些意义。.

C 语言嵌入式系统编程修炼之二:软件架构篇

作者:宋宝华　　更新日期:2005-07-22

模块划分

模块划分的"划"是规划的意思，意指怎样合理的将一个很大的软件划分为一系列功能独立的部分合作完成系统的需求。C 语言作为一种结构化的程序设计语言，在模块的划分上主要依据功能（依功能进行划分在面向对象设计中成为一个错误，牛顿定律遇到了>相对论），C 语言模块化程序设计需理解如下概念：

（1） 模块即是一个.c 文件和一个.h 文件的结合，头文件(.h)中是对于该模块接口的声明；

（2） 某模块提供给其它模块调用的外部函数及数据需在.h 中文件中冠以 extern 关键字声明；

（3） 模块内的函数和全局变量需在.c 文件开头冠以 static 关键字声明；

（4） 永远不要在.h 文件中定义变量！定义变量和声明变量的区别在于定义会产生内存分配的操作，是汇编阶段的概念；而声明则只是告诉包含该声明的模块在连接阶段从其它模块寻找外部函数和变量。如：

```
/*module1.h*/
int a = 5; /* 在模块 1 的.h 文件中定义 int a */

/*module1 .c*/
#include "module1.h" /* 在模块 1 中包含模块 1 的.h 文件 */
```

```
/*module2 .c*/
#include "module1.h" /* 在模块 2 中包含模块 1 的.h 文件 */

/*module3 .c*/
#include "module1.h" /* 在模块 3 中包含模块 1 的.h 文件 */
```

以上程序的结果是在模块 1、2、3 中都定义了整型变量 a，a 在不同的模块中对应不同的地址单元，这个世界上从来不需要这样的程序。正确的做法是：

```
/*module1.h*/
extern int a; /* 在模块 1 的.h 文件中声明 int a */

/*module1 .c*/
#include "module1.h" /* 在模块 1 中包含模块 1 的.h 文件 */
int a = 5; /* 在模块 1 的.c 文件中定义 int a */

/*module2 .c*/
#include "module1.h" /* 在模块 2 中包含模块 1 的.h 文件 */

/*module3 .c*/
#include "module1.h" /* 在模块 3 中包含模块 1 的.h 文件 */
```

这样如果模块 1、2、3 操作 a 的话，对应的是同一片内存单元。

一个嵌入式系统通常包括两类模块：

（1）硬件驱动模块，一种特定硬件对应一个模块；

（2）软件功能模块，其模块的划分应满足低偶合、高内聚的要求。

多任务还是单任务

所谓"单任务系统"是指该系统不能支持多任务并发操作，宏观串行地执行一个任务。而多任务系统则可以宏观并行（微观上可能串行）地"同时"执行多个任务。

多任务的并发执行通常依赖于一个多任务操作系统(OS)，多任务 OS 的核心是系统调度器，它使用任务控制块(TCB)来管理任务调度功能。TCB 包括任务的当前状态、优先级、要等待的事件或资源、任务程序码的起始地址、初始堆栈指针等信息。调度器在任务被激活时，要用到这些信息。此外，TCB 还被用来存放任务的"上下文"(context)。任务的上下文就是当一个执行中的任务被停止时，所要保存的所有信息。通常，上下文就是计算机当前的状态，也即各个寄存器的内容。当发生任务切换时，当前运行的任务的上下文被存入 TCB，并将要被执行的任务的上下文从它的 TCB 中取出，放入各个寄存器中。

嵌入式多任务 OS 的典型例子有 Vxworks、ucLinux 等。嵌入式 OS 并非遥不可及的神坛之物，我们可以用不到 1000

行代码实现一个针对 80186 处理器的功能最简单的 OS 内核，作者正准备进行此项工作，希望能将心得贡献给大家。

究竟选择多任务还是单任务方式，依赖于软件的体系是否庞大。例如，绝大多数手机程序都是多任务的，但也有一些小灵通的协议栈是单任务的，没有操作系统，它们的主程序轮流调用各个软件模块的处理程序，模拟多任务环境。

单任务程序典型架构

（1）从 CPU 复位时的指定地址开始执行；

（2）跳转至汇编代码 startup 处执行；

（3）跳转至用户主程序 main 执行，在 main 中完成：

a.初试化各硬件设备；

b.初始化各软件模块；

c.进入死循环（无限循环），调用各模块的处理函数

用户主程序和各模块的处理函数都以 C 语言完成。用户主程序最后都进入了一个死循环，其首选方案是：

```
while(1)
{
}
```

有的程序员这样写：

```
for(;;)
{
}
```

这个语法没有确切表达代码的含义，我们从 for(;;)看不出什么，只有弄明白 for(;;)在 C 语言中意味着无条件循环才明白其意。

下面是几个"著名"的死循环：

（1）操作系统是死循环；

（2）WIN32 程序是死循环；

（3）嵌入式系统软件是死循环；

（4）多线程程序的线程处理函数是死循环。

你可能会辩驳，大声说："凡事都不是绝对的，2、3、4 都可以不是死循环"。Yes，you are right，但是你得不到鲜

花和掌声。实际上，这是一个没有太大意义的牛角尖，因为这个世界从来不需要一个处理完几个消息就喊着要 OS 杀死它的 WIN32 程序，不需要一个刚开始 RUN 就自行了断的嵌入式系统，不需要莫名其妙启动一个做一点事就干掉自己的线程。有时候，过于严谨制造的不是便利而是麻烦。君不见，五层的 TCP/IP 协议栈超越严谨的 ISO/OSI 七层协议栈大行其道成为事实上的标准？

经常有网友讨论：

```
printf("%d,%d",++i,i++); /* 输出是什么？ */
c = a+++b; /* c=? */
```

等类似问题。面对这些问题，我们只能发出由衷的感慨：世界上还有很多有意义的事情等着我们去消化摄入的食物。

实际上，嵌入式系统要运行到世界末日。

中断服务程序

中断是嵌入式系统中重要的组成部分，但是在标准 C 中不包含中断。许多编译开发商在标准 C 上增加了对中断的支持，提供新的关键字用于标示中断服务程序（ISR），类似于__interrupt、#program interrupt 等。当一个函数被定义为 ISR 的时候，编译器会自动为该函数增加中断服务程序所需要的中断现场入栈和出栈代码。

中断服务程序需要满足如下要求：

(1)不能返回值；

(2)不能向 ISR 传递参数；

(3) ISR 应该尽可能的短小精悍；

(4) printf(char * lpFormatString,…)函数会带来重入和性能问题，不能在 ISR 中采用。

在某项目的开发中，我们设计了一个队列，在中断服务程序中，只是将中断类型添加入该队列中，在主程序的死循环中不断扫描中断队列是否有中断，有则取出队列中的第一个中断类型，进行相应处理。

```
/* 存放中断的队列 */
typedef struct tagIntQueue
{
  int intType; /* 中断类型 */
  struct tagIntQueue *next;
}IntQueue;

IntQueue lpIntQueueHead;

__interrupt ISRexample ()
{
```

```
  int intType;
  intType = GetSystemType();
  QueueAddTail(lpIntQueueHead, intType); /* 在队列尾加入新的中断 */
}
```

在主程序循环中判断是否有中断：

```
While(1)
{
  If( !IsIntQueueEmpty() )
  {
    intType = GetFirstInt();
    switch(intType) /* 是不是很象 WIN32 程序的消息解析函数? */
    {
      /* 对，我们的中断类型解析很类似于消息驱动 */
      case xxx: /* 我们称其为"中断驱动"吧？ */
        …
        break;
      case xxx:
        …
        break;
      …
    }
  }
}
```

按上述方法设计的中断服务程序很小，实际的工作都交由主程序执行了。

硬件驱动模块

一个硬件驱动模块通常应包括如下函数：

（1）中断服务程序 ISR

（2）硬件初始化

a.修改寄存器，设置硬件参数（如 UART 应设置其波特率，AD/DA 设备应设置其采样速率等）；

b.将中断服务程序入口地址写入中断向量表：

```
/* 设置中断向量表 */
m_myPtr = make_far_pointer(01); /* 返回 void far 型指针 void far * */
m_myPtr += ITYPE_UART; /* ITYPE_UART： uart 中断服务程序 */
/* 相对于中断向量表首地址的偏移 */
```

```
*m_myPtr = &UART _Isr; /* UART _Isr：UART 的中断服务程序 */
```

（3）设置 CPU 针对该硬件的控制线

a. 如果控制线可作 PIO（可编程 I/O）和控制信号用，则设置 CPU 内部对应寄存器使其作为控制信号；

b. 设置 CPU 内部的针对该设备的中断屏蔽位，设置中断方式（电平触发还是边缘触发）。

（4）提供一系列针对该设备的操作接口函数。例如，对于 LCD，其驱动模块应提供绘制像素、画线、绘制矩阵、显示字符点阵等函数；而对于实时钟，其驱动模块则需提供获取时间、设置时间等函数。

C 的面向对象化

在面向对象的语言里面，出现了类的概念。类是对特定数据的特定操作的集合体。类包含了两个范畴：数据和操作。而 C 语言中的 struct 仅仅是数据的集合，我们可以利用函数指针将 struct 模拟为一个包含数据和操作的"类"。下面的 C 程序模拟了一个最简单的"类"：

```
#ifndef C_Class
#define C_Class struct
#endif
C_Class A
{
  C_Class A *A_this; /* this 指针 */
  void (*Foo)(C_Class A *A_this); /* 行为：函数指针 */
  int a; /* 数据 */
  int b;
};
```

我们可以利用 C 语言模拟出面向对象的三个特性：封装、继承和多态，但是更多的时候，我们只是需要将数据与行为封装以解决软件结构混乱的问题。C 模拟面向对象思想的目的不在于模拟行为本身，而在于解决某些情况下使用 C 语言编程时程序整体框架结构分散、数据和函数脱节的问题。我们在后续章节会看到这样的例子。

总结

本篇介绍了嵌入式系统编程软件架构方面的知识，主要包括模块划分、多任务还是单任务选取、单任务程序典型架构、中断服务程序、硬件驱动模块设计等，从宏观上给出了一个嵌入式系统软件所包含的主要元素。

请记住：软件结构是软件的灵魂！结构混乱的程序面目可憎，调试、测试、维护、升级都极度困难。.

小力力力 2005-09-21 17:29

C 语言嵌入式系统编程修炼之三:内存操作

作者:宋宝华　　更新日期:2005-07-22

数据指针

在嵌入式系统的编程中，常常要求在特定的内存单元读写内容，汇编有对应的 MOV 指令，而除 C/C++以外的其它编程语言基本没有直接访问绝对地址的能力。在嵌入式系统的实际调试中，多借助 C 语言指针所具有的对绝对地址单元内容的读写能力。**以指针直接操作内存多发生在如下几种情况：**

(1) 某 I/O 芯片被定位在 CPU 的存储空间而非 I/O 空间，而且寄存器对应于某特定地址；

(2) 两个 CPU 之间以双端口 RAM 通信，CPU 需要在双端口 RAM 的特定单元（称为 mail box）书写内容以在对方 CPU 产生中断；

(3) 读取在 ROM 或 FLASH 的特定单元所烧录的汉字和英文字模。

譬如：

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0xF0000+0xFF00(80186 使用 16 位段地址和 16 位偏移地址)写入 11。

在使用绝对地址指针时，要注意指针自增自减操作的结果取决于指针指向的数据类别。上例中 p++后的结果是 p=0xF000FF01，若 p 指向 int，即：

```
int *p = (int *)0xF000FF00;
```

p++(或++p)的结果等同于：p = p+sizeof(int)，而 p-(或-p)的结果是 p = p-sizeof(int)。

同理，若执行：

```
long int *p = (long int *)0xF000FF00;
```

则 p++(或++p)的结果等同于：p = p+sizeof(long int) ，而 p-(或-p)的结果是 p = p-sizeof(long int)。

记住：**CPU 以字节为单位编址，而 C 语言指针以指向的数据类型长度作自增和自减**。理解这一点对于以指针直接操作内存是相当重要的。

函数指针

首先要理解以下三个问题：

（1）C 语言中函数名直接对应于函数生成的指令代码在内存中的地址，因此函数名可以直接赋给指向函数的指针；

（2）调用函数实际上等同于"调转指令＋参数传递处理＋回归位置入栈"，本质上最核心的操作是将函数生成的目标代码的首地址赋给 CPU 的 PC 寄存器；

（3）因为函数调用的本质是跳转到某一个地址单元的 code 去执行，所以可以"调用"一个根本就不存在的函数实体，晕？请往下看：

请拿出你可以获得的任何一本大学《微型计算机原理》教材，书中讲到，186 CPU 启动后跳转至绝对地址 0xFFFF0（对应 C 语言指针是 0xF000FFF0，0xF000 为段地址，0xFFF0 为段内偏移）执行，请看下面的代码：

```
typedef void (*lpFunction) ( ); /* 定义一个无参数、无返回类型的函数指针类型*/
/* 定义一个函数指针，指向 CPU 启动后所执行第一条指令的位置*/
lpFunction lpReset = (lpFunction)0xF000FFF0;
lpReset(); /* 调用函数 */
```

在以上的程序中，我们根本没有看到任何一个函数实体，但是我们却执行了这样的函数调用：lpReset()，它实际上起到了"软重启"的作用，跳转到 CPU 启动后第一条要执行的指令的位置。

记住：**函数无它，唯指令集合耳；你可以调用一个没有函数体的函数，本质上只是换一个地址开始执行指令！**

数组 vs.动态申请

在嵌入式系统中动态内存申请存在比一般系统编程时更严格的要求，这是因为嵌入式系统的内存空间往往是十分有限的，不经意的内存泄露会很快导致系统的崩溃。

所以一定要保证你的 malloc 和 free 成对出现，如果你写出这样的一段程序：

```
char * function(void)
{
  char *p;
  p = (char *)malloc(…);
  if(p==NULL)
    …;
    … /* 一系列针对 p 的操作 */
  return p;
}
```

在某处调用 function()，用完 function 中动态申请的内存后将其 free，如下：

```
char *q = function();
…
free(q);
```

上述代码明显是不合理的，因为违反了 malloc 和 free 成对出现的原则，即"谁申请，就由谁释放"原则。不满足这个原则，会导致代码的耦合度增大，因为用户在调用 function 函数时需要知道其内部细节！

正确的做法是在调用处申请内存，并传入 function 函数，如下：

```
char *p=malloc(…);
if(p==NULL)
…;
function(p);
…
free(p);
p=NULL;
```

而函数 function 则接收参数 p，如下：

```
void function(char *p)
{
  … /* 一系列针对 p 的操作 */
}
```

基本上，动态申请内存方式可以用较大的数组替换。对于编程新手，笔者推荐你尽量采用数组！嵌入式系统可以以博大的胸襟接收瑕疵，而无法"海纳"错误。毕竟，以最笨的方式苦练神功的郭靖胜过机智聪明却范政治错误走反革命道路的杨康。

给出原则：

（1）尽可能的选用数组，数组不能越界访问（真理越过一步就是谬误，数组越过界限就光荣地成全了一个混乱的嵌入式系统）；

（2）如果使用动态申请，则申请后一定要判断是否申请成功了，并且 malloc 和 free 应成对出现！

关键字 const

const 意味着"只读"。区别如下代码的功能非常重要，也是老生长叹，如果你还不知道它们的区别，而且已经在程序界摸爬滚打多年，那只能说这是一个悲哀：

```
const int a;
int const a;
const int *a;
int * const a;
```

```
int const * a const;
```

（1）关键字 const 的作用是为给读你代码的人传达非常有用的信息。例如，在函数的形参前添加 const 关键字意味着这个参数在函数体内不会被修改，属于"输入参数"。在有多个形参的时候，函数的调用者可以凭借参数前是否有 const 关键字，清晰的辨别哪些是输入参数，哪些是可能的输出参数。

（2）合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改，这样可以减少 bug 的出现。

const 在 C++语言中则包含了更丰富的含义，而在 C 语言中仅意味着："只能读的普通变量"，可以称其为"不能改变的变量"（这个说法似乎很拗口，但却最准确的表达了 C 语言中 const 的本质），在编译阶段需要的常数仍然只能以#define 宏定义！故在 C 语言中如下程序是非法的：

```
const int SIZE = 10;
char a[SIZE]; /* 非法：编译阶段不能用到变量 */
```

关键字 volatile

C 语言编译器会对用户书写的代码进行优化，譬如如下代码：

```
int a,b,c;
a = inWord(0x100); /*读取 I/O 空间 0x100 端口的内容存入 a 变量*/
b = a;
a = inWord (0x100); /*再次读取 I/O 空间 0x100 端口的内容存入 a 变量*/
c = a;
```

很可能被编译器优化为：

```
int a,b,c;
a = inWord(0x100); /*读取 I/O 空间 0x100 端口的内容存入 a 变量*/
b = a;
c = a;
```

但是这样的优化结果可能导致错误，如果 I/O 空间 0x100 端口的内容在执行第一次读操作后被其它程序写入新值，则其实第 2 次读操作读出的内容与第一次不同，b 和 c 的值应该不同。在变量 a 的定义前加上 volatile 关键字可以防止编译器的类似优化，正确的做法是：

```
volatile int a;
```

volatile 变量可能用于如下几种情况：

(1) 并行设备的硬件寄存器（如：状态寄存器，例中的代码属于此类）；

(2) 一个中断服务子程序中会访问到的非自动变量(也就是全局变量)；

(3) 多线程应用中被几个任务共享的变量。

CPU 字长与存储器位宽不一致处理

在背景篇中提到，本文特意选择了一个与 CPU 字长不一致的存储芯片，就是为了进行本节的讨论，解决 CPU 字长与存储器位宽不一致的情况。80186 的字长为 16，而 NVRAM 的位宽为 8，在这种情况下，我们需要为 NVRAM 提供读写字节、字的接口，如下：

```
typedef unsigned char BYTE;
typedef unsigned int WORD;
/* 函数功能：读 NVRAM 中字节
* 参数：wOffset，读取位置相对 NVRAM 基地址的偏移
* 返回：读取到的字节值
*/
extern BYTE ReadByteNVRAM(WORD wOffset)
{
  LPBYTE lpAddr = (BYTE*)(NVRAM + wOffset * 2); /* 为什么偏移要×2? */

  return *lpAddr;
}

/* 函数功能：读 NVRAM 中字
* 参数：wOffset，读取位置相对 NVRAM 基地址的偏移
* 返回：读取到的字
*/
extern WORD ReadWordNVRAM(WORD wOffset)
{
  WORD wTmp = 0;
  LPBYTE lpAddr;
  /* 读取高位字节 */
  lpAddr = (BYTE*)(NVRAM + wOffset * 2); /* 为什么偏移要×2? */
  wTmp += (*lpAddr)*256;
  /* 读取低位字节 */
  lpAddr = (BYTE*)(NVRAM + (wOffset +1) * 2); /* 为什么偏移要×2? */
  wTmp += *lpAddr;
  return wTmp;
}

/* 函数功能：向 NVRAM 中写一个字节
*参数：wOffset，写入位置相对 NVRAM 基地址的偏移
* byData，欲写入的字节
*/
```

```
extern void WriteByteNVRAM(WORD wOffset, BYTE byData)
{
  …
}

/* 函数功能：向 NVRAM 中写一个字 */
*参数：wOffset，写入位置相对 NVRAM 基地址的偏移
* wData，欲写入的字
*/
extern void WriteWordNVRAM(WORD wOffset, WORD wData)
{
  …
}
```

子贡问曰：Why 偏移要乘以 2?

子曰：请看图 1，16 位 80186 与 8 位 NVRAM 之间互连只能以地址线 A1 对其 A0,CPU 本身的 A0 与 NVRAM 不连接。因此，NVRAM 的地址只能是偶数地址，故每次以 0x10 为单位前进！

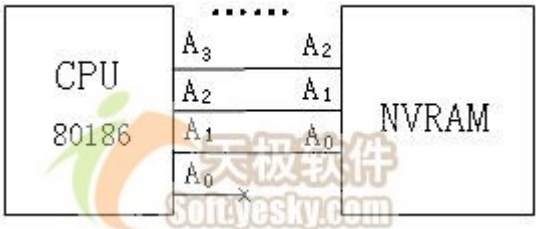


图 1 CPU 与 NVRAM 地址线连接

子贡再问：So why 80186 的地址线 A0 不与 NVRAM 的 A0 连接?

子曰：请看《IT 论语》之《微机原理篇》，那里面讲述了关于计算机组成的圣人之道。

总结

本篇主要讲述了嵌入式系统 C 编程中内存操作的相关技巧。掌握并深入理解关于数据指针、函数指针、动态申请内存、const 及 volatile 关键字等的相关知识，是一个优秀的 C 语言程序设计师的基本要求。当我们已经牢固掌握了上述技巧后，我们就已经学会了 C 语言的 99%，因为 C 语言最精华的内涵皆在内存操作中体现。

我们之所以在嵌入式系统中使用 C 语言进行程序设计，99%是因为其强大的内存操作能力！

如果你爱编程，请你爱 C 语言；

如果你爱 C 语言，请你爱指针；

如果你爱指针，请你爱指针的指针！.

C 语言嵌入式系统编程修炼之四:屏幕操作

作者:宋宝华　　更新日期:2005-07-22

汉字处理

现在要解决的问题是，嵌入式系统中经常要使用的并非是完整的汉字库，往往只是需要提供数量有限的汉字供必要的显示功能。例如，一个微波炉的 LCD 上没有必要提供显示"电子邮件"的功能；一个提供汉字显示功能的空调的 LCD 上不需要显示一条"短消息"，诸如此类。但是一部手机、小灵通则通常需要包括较完整的汉字库。

如果包括的汉字库较完整，那么，由内码计算出汉字字模在库中的偏移是十分简单的：汉字库是按照区位的顺序排列的，前一个字节为该汉字的区号，后一个字节为该字的位号。每一个区记录 94 个汉字，位号则为该字在该区中的位置。因此，汉字在汉字库中的具体位置计算公式为：94*(区号-1)+位号-1。减 1 是因为数组是以 0 为开始而区号位号是以 1 为开始的。只需乘上一个汉字字模占用的字节数即可，即：(94*(区号-1)+位号-1)*一个汉字字模占用字节数，以 16*16 点阵字库为例，计算公式则为：(94*(区号-1)+(位号-1))*32。汉字库中从该位置起的 32 字节信息记录了该字的字模信息。

对于包含较完整汉字库的系统而言，我们可以以上述规则计算字模的位置。但是如果仅仅是提供少量汉字呢？譬如几十至几百个？最好的做法是：

定义宏：

```
# define EX_FONT_CHAR(value)
# define EX_FONT_UNICODE_VAL(value) (value),
# define EX_FONT_ANSI_VAL(value) (value),
```

定义结构体：

```
typedef struct _wide_unicode_font16x16
{
  WORD value; /* 内码 */
  BYTE data[32]; /* 字模点阵 */
}Unicode;
#define CHINESE_CHAR_NUM … /* 汉字数量 */
```

字模的存储用数组：

```
Unicode chinese[CHINESE_CHAR_NUM] =
{
{
EX_FONT_CHAR("业")
EX_FONT_UNICODE_VAL(0x4e1a)
```

```
{0x04, 0x40, 0x04, 0x40, 0x04, 0x40, 0x04, 0x44, 0x44, 0x46, 0x24, 0x4c, 0x24, 0x48, 0x14, 0x50,
0x1c, 0x50, 0x14, 0x60, 0x04, 0x40, 0x04, 0x40, 0x04, 0x44, 0xff, 0xfe, 0x00, 0x00, 0x00, 0x00}
},
{
EX_FONT_CHAR("中")
EX_FONT_UNICODE_VAL(0x4e2d)
{0x01, 0x00, 0x01, 0x00, 0x21, 0x08, 0x3f, 0xfc, 0x21, 0x08, 0x21, 0x08, 0x21, 0x08, 0x21, 0x08,
0x21, 0x08,
0x3f, 0xf8, 0x21, 0x08, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00}
},
{
EX_FONT_CHAR("云")
EX_FONT_UNICODE_VAL(0x4e91)
{0x00, 0x00, 0x00, 0x30, 0x3f, 0xf8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0c, 0xff, 0xfe, 0x03, 0x00,
0x07, 0x00,

0x06, 0x40, 0x0c, 0x20, 0x18, 0x10, 0x31, 0xf8, 0x7f, 0x0c, 0x20, 0x08, 0x00, 0x00}
},
{
EX_FONT_CHAR("件")
EX_FONT_UNICODE_VAL(0x4ef6)
{0x10, 0x40, 0x1a, 0x40, 0x13, 0x40, 0x32, 0x40, 0x23, 0xfc, 0x64, 0x40, 0xa4, 0x40, 0x28, 0x40,
0x2f, 0xfe,

0x20, 0x40, 0x20, 0x40, 0x20, 0x40, 0x20, 0x40, 0x20, 0x40, 0x20, 0x40, 0x20, 0x40}
}
}
```

要显示特定汉字的时候，只需要从数组中查找内码与要求汉字内码相同的即可获得字模。如果前面的汉字在数组中以内码大小顺序排列，那么可以以二分查找法更高效的查找到汉字的字模。

这是一种很有效的组织小汉字库的方法，它可以保证程序有很好的结构。

系统时间显示

从 NVRAM 中可以读取系统的时间，系统一般借助 NVRAM 产生的秒中断每秒读取一次当前时间并在 LCD 上显示。关于时间的显示，有一个效率问题。因为时间有其特殊性，那就是 60 秒才有一次分钟的变化，60 分钟才有一次小时变化，如果我们每次都将读取的时间在屏幕上完全重新刷新一次，则浪费了大量的系统时间。

一个较好的办法是我们在时间显示函数中以静态变量分别存储小时、分钟、秒，只有在其内容发生变化的时候才更新其显示。

```
extern void DisplayTime(…)
{
```

```
  static BYTE byHour,byMinute,bySecond;
  BYTE byNewHour, byNewMinute, byNewSecond;
  byNewHour = GetSysHour();
  byNewMinute = GetSysMinute();
  byNewSecond = GetSysSecond();

  if(byNewHour!= byHour)
  {
    … /* 显示小时 */
    byHour = byNewHour;
  }
  if(byNewMinute!= byMinute)
  {
    … /* 显示分钟 */
    byMinute = byNewMinute;
  }
  if(byNewSecond!= bySecond)
  {
    … /* 显示秒钟 */
    bySecond = byNewSecond;
  }
}
```

这个例子也可以顺便作为 C 语言中 static 关键字强大威力的证明。当然，在 C++语言里，static 具有了更加强大的威力，它使得某些数据和函数脱离"对象"而成为"类"的一部分，正是它的这一特点，成就了软件的无数优秀设计。

动画显示

动画是无所谓有，无所谓无的，静止的画面走的路多了，也就成了动画。随着时间的变更，在屏幕上显示不同的静止画面，即是动画之本质。所以，在一个嵌入式系统的 LCD 上欲显示动画，必须借助定时器。没有硬件或软件定时器的世界是无法想像的：

（1） 没有定时器，一个操作系统将无法进行时间片的轮转，于是无法进行多任务的调度，于是便不再成其为一个多任务操作系统；

（2） 没有定时器，一个多媒体播放软件将无法运作，因为它不知道何时应该切换到下一帧画面；

（3） 没有定时器，一个网络协议将无法运转，因为其无法获知何时包传输超时并重传之，无法在特定的时间完成特定的任务。

因此，没有定时器将意味着没有操作系统、没有网络、没有多媒体，这将是怎样的黑暗？所以，合理并灵活地使用各种定时器，是对一个软件人的最基本需求！

在 80186 为主芯片的嵌入式系统中，我们需要借助硬件定时器的中断来作为软件定时器，在中断发生后变更画面的显示内容。在时间显示"xx:xx"中让冒号交替有无，每次秒中断发生后，需调用 ShowDot：

```
void ShowDot()
{
  static BOOL bShowDot = TRUE; /* 再一次领略 static 关键字的威力 */
  if(bShowDot)
  {
    showChar(' :' ,xPos,yPos);
  }
  else
  {
    showChar('  ' ,xPos,yPos);
  }
  bShowDot = ! bShowDot;
}
```

菜单操作

无数人为之绞尽脑汁的问题终于出现了，在这一节里，我们将看到，在 C 语言中哪怕用到一丁点的面向对象思想，软件结构将会有何等的改观！

笔者曾经是个笨蛋，被菜单搞晕了，给出这样的一个系统：

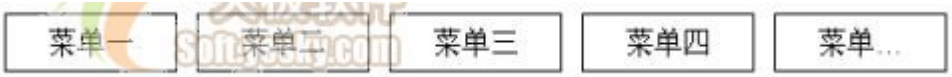


图 1 菜单范例

要求以键盘上的"← →"键切换菜单焦点，当用户在焦点处于某菜单时，若敲击键盘上的 OK、CANCEL 键则调用该焦点菜单对应之处理函数。我曾经傻傻地这样做着：

```
/* 按下 OK 键 */
void onOkKey()
{
  /* 判断在什么焦点菜单上按下 Ok 键，调用相应处理函数 */
  Switch(currentFocus)
  {
    case MENU1:
      menu1OnOk();
      break;
    case MENU2:
      menu2OnOk();
      break;
    …
  }
}
```

```
/* 按下 Cancel 键 */
void onCancelKey()
{
  /* 判断在什么焦点菜单上按下 Cancel 键，调用相应处理函数 */
  Switch(currentFocus)
  {
    case MENU1:
      menu1OnCancel();
      break;
    case MENU2:
      menu2OnCancel();
      break;
    …
  }
}
```

终于有一天，我这样做了：

```
/* 将菜单的属性和操作"封装"在一起 */
typedef struct tagSysMenu
{
  char *text; /* 菜单的文本 */
  BYTE xPos; /* 菜单在 LCD 上的 x 坐标 */
  BYTE yPos; /* 菜单在 LCD 上的 y 坐标 */
  void (*onOkFun)(); /* 在该菜单上按下 ok 键的处理函数指针 */
  void (*onCancelFun)(); /* 在该菜单上按下 cancel 键的处理函数指针 */
}SysMenu, *LPSysMenu;
```

当我定义菜单时，只需要这样：

```
static SysMenu menu[MENU_NUM] =
{
  {
    "menu1", 0, 48, menu1On0k, menu1OnCancel
  }
  ,
  {
    " menu2", 7, 48, menu2On0k, menu2OnCancel
  }
  ,
  {
    " menu3", 7, 48, menu3On0k, menu3OnCancel
  }
```

```
  ,
  {
    " menu4", 7, 48, menu4OnOk, menu4OnCancel
  }
  …
};
```

OK 键和 CANCEL 键的处理变成：

```
/* 按下 OK 键 */
void onOkKey()
{
  menu[currentFocusMenu].onOkFun();
}
/* 按下 Cancel 键 */
void onCancelKey()
{
  menu[currentFocusMenu].onCancelFun();
}
```

程序被大大简化了，也开始具有很好的可扩展性！我们仅仅利用了面向对象中的封装思想，就让程序结构清晰，其结果是几乎可以在无需修改程序的情况下在系统中添加更多的菜单，而系统的按键处理函数保持不变。

面向对象，真神了！

模拟 MessageBox 函数

MessageBox 函数，这个 Windows 编程中的超级猛料，不知道是多少入门者第一次用到的函数。还记得我们第一次在 Windows 中利用 MessageBox 输出 "Hello,World!"对话框时新奇的感觉吗？无法统计，这个世界上究竟有多少程序员学习 Windows 编程是从 MessageBox ("Hello,World!",…)开始的。在我本科的学校，广泛流传着一个词汇，叫做"’Hello,World’级程序员"，意指入门级程序员，但似乎"’Hello,World’级"这个说法更搞笑而形象。

图 2 经典的 Hello,World!

图 2 给出了两种永恒经典的 Hello,World 对话框，一种只具有"确定"，一种则包含"确定"、"取消"。是的，MessageBox 的确有，而且也应该有两类！这完全是由特定的应用需求决定的。

嵌入式系统中没有给我们提供 MessageBox，但是鉴于其功能强大，我们需要模拟之，一个模拟的 MessageBox 函数为：

```
/*****************************************
/* 函数名称: MessageBox
/* 功能说明: 弹出式对话框,显示提醒用户的信息
/* 参数说明: lpStr --- 提醒用户的字符串输出信息
/* TYPE --- 输出格式(ID_OK = 0, ID_OKCANCEL = 1)
/* 返回值: 返回对话框接收的键值,只有两种 KEY_OK, KEY_CANCEL
/*****************************************
typedef enum TYPE { ID_OK,ID_OKCANCEL }MSG_TYPE;
extern BYTE MessageBox(LPBYTE lpStr, BYTE TYPE)
{
  BYTE keyValue = -1;

  ClearScreen(); /* 清除屏幕 */
  DisplayString(xPos,yPos,lpStr,TRUE); /* 显示字符串 */
  /* 根据对话框类型决定是否显示确定、取消 */
  switch (TYPE)
  {
    case ID_OK:
      DisplayString(13,yPos+High+1, " 确定 ", 0);
      break;
    case ID_OKCANCEL:
      DisplayString(8, yPos+High+1, " 确定 ", 0);
      DisplayString(17,yPos+High+1, " 取消 ", 0);
      break;
    default:
      break;
  }
  DrawRect(0, 0, 239, yPos+High+16+4); /* 绘制外框 */
  /* MessageBox 是模式对话框，阻塞运行，等待按键 */
  while( (keyValue != KEY_OK) || (keyValue != KEY_CANCEL) )
  {
    keyValue = getSysKey();
  }
  /* 返回按键类型 */
  if(keyValue== KEY_OK)
  {
    return ID_OK;
  }
  else
  {
    return ID_CANCEL;
  }
}
```

上述函数与我们平素在 VC++等中使用的 MessageBox 是何等的神似啊？实现这个函数，你会看到它在嵌入式系统中的妙用是无穷的。

总结

本篇是本系列文章中技巧性最深的一篇，它提供了嵌入式系统屏幕显示方面一些很巧妙的处理方法，灵活使用它们，我们将不再被 LCD 上凌乱不堪的显示内容所困扰。

屏幕乃嵌入式系统生存之重要辅助，面目可憎之显示将另用户逃之夭夭。屏幕编程若处理不好，将是软件中最不系统、最混乱的部分，笔者曾深受其害。.

C 语言嵌入式系统编程修炼之五:键盘操作

作者:宋宝华　　更新日期:2005-07-22

处理功能键

功能键的问题在于，用户界面并非固定的，用户功能键的选择将使屏幕画面处于不同的显示状态下。例如，主画面如图 1：

图 1 主画面

当用户在设置 XX 上按下 Enter 键之后，画面就切换到了设置 XX 的界面，如图 2：

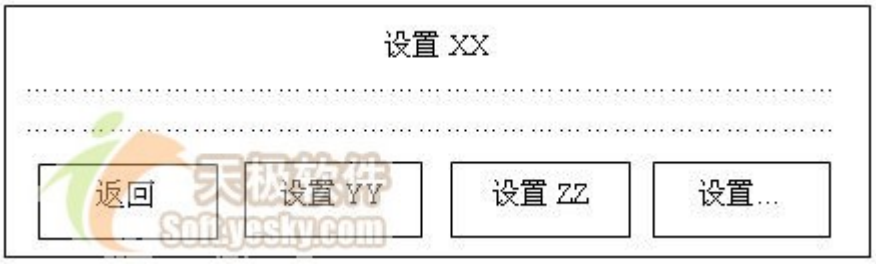


图 2 切换到设置 XX 画面

程序如何判断用户处于哪一画面，并在该画面的程序状态下调用对应的功能键处理函数，而且保证良好的结构，是一个值得思考的问题。

让我们来看看 WIN32 编程中用到的"窗口"概念，当消息（message）被发送给不同窗口的时候，该窗口的消息处理函数

（是一个 callback 函数）最终被调用，而在该窗口的消息处理函数中，又根据消息的类型调用了该窗口中的对应处理函数。通过这种方式，WIN32 有效的组织了不同的窗口，并处理不同窗口情况下的消息。

我们从中学习到的就是：

（1）将不同的画面类比为 WIN32 中不同的窗口，将窗口中的各种元素（菜单、按钮等）包含在窗口之中；

（2）给各个画面提供一个功能键"消息"处理函数，该函数接收按键信息为参数；

（3）在各画面的功能键"消息"处理函数中，判断按键类型和当前焦点元素，并调用对应元素的按键处理函数。

```
/* 将窗口元素、消息处理函数封装在窗口中 */
struct windows
{
  BYTE currentFocus;
  ELEMENT element[ELEMENT_NUM];
  void (*messageFun) (BYTE keyValue);
  …
};
/* 消息处理函数 */
void messageFunction(BYTE keyValue)
{
  BYTE i = 0;
  /* 获得焦点元素 */
  while ( (element [i].ID!= currentFocus)&& (i < ELEMENT_NUM) )
  {
    i++;
  }
  /* "消息映射" */
  if(i < ELEMENT_NUM)
  {
    switch(keyValue)
    {
      case OK:
        element[i].OnOk();
        break;
      …
    }
  }
}
```

在窗口的消息处理函数中调用相应元素按键函数的过程类似于"消息映射"，这是我们从 WIN32 编程中学习到的。编程到了一个境界，很多东西都是相通的了。其它地方的思想可以拿过来为我所用，是为编程中的"拿来主义"。

在这个例子中，如果我们还想玩得更大一点，我们可以借鉴 MFC 中处理 MESSAGE_MAP 的方法，我们也可以学习 MFC 定义几个精妙的宏来实现"消息映射"。

处理数字键

用户输入数字时是一位一位输入的，每一位的输入都对应着屏幕上的一个显示位置（x 坐标，y 坐标）。此外，程序还需要记录该位置输入的值，所以有效组织用户数字输入的最佳方式是定义一个结构体，将坐标和数值捆绑在一起：

```
/* 用户数字输入结构体 */
typedef struct tagInputNum
{
  BYTE byNum; /* 接收用户输入赋值 */
  BYTE xPos; /* 数字输入在屏幕上的显示位置 x 坐标 */
  BYTE yPos; /* 数字输入在屏幕上的显示位置 y 坐标 */
}InputNum, *LPInputNum;
```

那么接收用户输入就可以定义一个结构体数组，用数组中的各位组成一个完整的数字：

```
InputNum inputElement[NUM_LENGTH]; /* 接收用户数字输入的数组 */
/* 数字按键处理函数 */
extern void onNumKey(BYTE num)
{
if(num==0|| num==1) /* 只接收二进制输入 */
{
  /* 在屏幕上显示用户输入 */
                                    DrawText(inputElement[currentElementInputPlace].xPos,
inputElement[currentElementInputPlace].yPos, "%1d", num);
  /* 将输入赋值给数组元素 */
  inputElement[currentElementInputPlace].byNum = num;
  /* 焦点及光标右移 */
  moveToRight();
}
}
```

将数字每一位输入的坐标和输入值捆绑后，在数字键处理函数中就可以较有结构的组织程序，使程序显得很紧凑。

整理用户输入

继续第 2 节的例子，在第 2 节的 onNumKey 函数中，只是获取了数字的每一位，因而我们需要将其转化为有效数据，譬如要转化为有效的 XXX 数据，其方法是：

```
/* 从 2 进制数据位转化为有效数据：XXX */
void convertToXXX()
{
```

```
  BYTE i;
  XXX = 0;
  for (i = 0; i < NUM_LENGTH; i++)
  {
    XXX += inputElement[i].byNum*power(2, NUM_LENGTH - i - 1);
  }
}
```

反之，我们也可能需要在屏幕上显示那些有效的数据位，因为我们也需要能够反向转化：

```
/* 从有效数据转化为 2 进制数据位：XXX */
void convertFromXXX()
{
  BYTE i;
  XXX = 0;
  for (i = 0; i < NUM_LENGTH; i++)
  {
    inputElement[i].byNum = XXX / power(2, NUM_LENGTH - i - 1) % 2;
  }
}
```

当然在上面的例子中，因为数据是 2 进制的，用 power 函数不是很好的选择，直接用"<< >>"移位操作效率更高，我们仅是为了说明问题的方便。试想，如果用户输入是十进制的，power 函数或许是唯一的选择了。

总结

本篇给出了键盘操作所涉及的各个方面：功能键处理、数字键处理及用户输入整理，基本上提供了一个全套的按键处理方案。对于功能键处理方法，将 LCD 屏幕与 Windows 窗口进行类比，提出了较新颖地解决屏幕、键盘繁杂交互问题的方案。

计算机学的许多知识都具有相通性，因而，不断追赶时髦技术而忽略基本功的做法是徒劳无意的。我们最多需要"精通"三种语言（精通，一个在如今的求职简历里泛滥成灾的词语），最佳拍档是汇编、C、C++（或 JAVA），很显然，如果你"精通"了这三种语言，其它语言你应该是可以很快"熟悉"的，否则你就没有"精通"它们..

C 语言嵌入式系统编程修炼之六:性能优化

作者:宋宝华　　更新日期:2005-07-22

使用宏定义

在 C 语言中，宏是产生内嵌代码的唯一方法。对于嵌入式系统而言，为了能达到性能要求，宏是一种很好的代替函数的方法。

写一个"标准"宏 MIN ，这个宏输入两个参数并返回较小的一个：

错误做法：

```
#define MIN(A,B)    ( A <= B ? A : B )
```

正确做法：

```
#define MIN(A,B)  ((A) <= (B) ? (A) : (B) )
```

对于宏，我们需要知道三点：

(1)宏定义"像"函数；

(2)宏定义不是函数，因而需要括上所有"参数"；

(3)宏定义可能产生副作用。

下面的代码：

```
least = MIN(*p++, b);
```

将被替换为：

```
( (*p++) <= (b) ?(*p++):(b) )
```

发生的事情无法预料。

因而不要给宏定义传入有副作用的"参数"。

使用寄存器变量

当对一个变量频繁被读写时，需要反复访问内存，从而花费大量的存取时间。为此，C 语言提供了一种变量，即寄存器变量。这种变量存放在 CPU 的寄存器中，使用时，不需要访问内存，而直接从寄存器中读写，从而提高效率。寄存器变量的说明符是 register。对于循环次数较多的循环控制变量及循环体内反复使用的变量均可定义为寄存器变量，而循环计数是应用寄存器变量的最好候选者。

(1) 只有局部自动变量和形参才可以定义为寄存器变量。因为寄存器变量属于动态存储方式，凡需要采用静态存储方式的量都不能定义为寄存器变量，包括：模块间全局变量、模块内全局变量、局部 static 变量；

(2) register 是一个"建议"型关键字，意指程序建议该变量放在寄存器中，但最终该变量可能因为条件不满足并未成为寄存器变量，而是被放在了存储器中，但编译器中并不报错（在 C++语言中有另一个"建议"型关键字：inline）。

下面是一个采用寄存器变量的例子：

```
/* 求 1+2+3+….+n 的值 */
WORD Addition(BYTE n)
{
  register i,s=0;
  for(i=1;i<=n;i++)
  {
    s=s+i;
  }
  return s;
}
```

本程序循环 n 次，i 和 s 都被频繁使用，因此可定义为寄存器变量。

内嵌汇编

程序中对时间要求苛刻的部分可以用内嵌汇编来重写，以带来速度上的显著提高。但是，开发和测试汇编代码是一件辛苦的工作，它将花费更长的时间，因而要慎重选择要用汇编的部分。

在程序中，存在一个 80-20 原则，即 20%的程序消耗了 80%的运行时间，因而我们要改进效率，最主要是考虑改进那 20%的代码。

嵌入式 C 程序中主要使用在线汇编，即在 C 程序中直接插入_asm{ }内嵌汇编语句：

```
/* 把两个输入参数的值相加，结果存放到另外一个全局变量中 */
int result;
void Add(long a, long *b)
{
  _asm
  {
    MOV AX, a
    MOV BX, b
    ADD AX, [BX]
    MOV result, AX
  }
}
```

利用硬件特性

首先要明白 CPU 对各种存储器的访问速度，基本上是：

CPU 内部 RAM > 外部同步 RAM > 外部异步 RAM > FLASH/ROM

对于程序代码，已经被烧录在 FLASH 或 ROM 中，我们可以让 CPU 直接从其中读取代码执行，但通常这不是一个好办法，我们最好在系统启动后将 FLASH 或 ROM 中的目标代码拷贝入 RAM 中后再执行以提高取指令速度；

对于 UART 等设备，其内部有一定容量的接收 BUFFER，我们应尽量在 BUFFER 被占满后再向 CPU 提出中断。例如计算机终端在向目标机通过 RS-232 传递数据时，不宜设置 UART 只接收到一个 BYTE 就向 CPU 提中断，从而无谓浪费中断处理时间；

如果对某设备能采取 DMA 方式读取，就采用 DMA 读取，DMA 读取方式在读取目标中包含的存储信息较大时效率较高，其数据传输的基本单位是块，而所传输的数据是从设备直接送入内存的（或者相反）。DMA 方式较之中断驱动方式，减少了 CPU 对外设的干预，进一步提高了 CPU 与外设的并行操作程度。

活用位操作

使用 C 语言的位操作可以减少除法和取模的运算。在计算机程序中数据的位是可以操作的最小数据单位，理论上可以用"位运算"来完成所有的运算和操作，因而，灵活的位操作可以有效地提高程序运行的效率。举例如下：

```
/* 方法 1 */
int i,j;
i = 879 / 16;
j = 562 % 32;
/* 方法 2 */
int i,j;
i = 879 >> 4;
j = 562 - (562 >> 5 << 5);
```

对于以 2 的指数次方为"*"、"/"或"%"因子的数学运算，转化为移位运算"<< >>"通常可以提高算法效率。因为乘除运算指令周期通常比移位运算大。

C 语言位运算除了可以提高运算效率外，在嵌入式系统的编程中，它的另一个最典型的应用，而且十分广泛地正在被使用着的是位间的与（&）、或（|）、非（~）操作，这跟嵌入式系统的编程特点有很大关系。我们通常要对硬件寄存器进行位设置，譬如，我们通过将 AM186ER 型 80186 处理器的中断屏蔽控制寄存器的第低 6 位设置为 0（开中断 2），最通用的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp &~INT_I2_MASK);
```

而将该位设置为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp | INT_I2_MASK);
```

判断该位是否为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
if(wTemp & INT_I2_MASK)
{
… /* 该位为 1 */
}
```

上述方法在嵌入式系统的编程中是非常常见的，我们需要牢固掌握。

总结

在性能优化方面永远注意 80-20 准备，不要优化程序中开销不大的那 80%，这是劳而无功的。

宏定义是 C 语言中实现类似函数功能而又不具函数调用和返回开销的较好方法，但宏在本质上不是函数，因而要防止宏展开后出现不可预料的结果，对宏的定义和使用要慎而处之。很遗憾，标准 C 至今没有包括 C++中 inline 函数的功能，inline 函数兼具无调用开销和安全的优点。

使用寄存器变量、内嵌汇编和活用位操作也是提高程序效率的有效方法。

除了编程上的技巧外，为提高系统的运行效率，我们通常也需要最大可能地利用各种硬件设备自身的特点来减小其运转开销，例如减小中断次数、利用 DMA 传输方式等。.

C/C++语言 void 及 void 指针深层探索

1.概述

许多初学者对 C/C++语言中的 void 及 void 指针类型不甚理解，因此在使用上出现了一些错误。本文将对 void 关键字的深刻含义进行解说，并详述 void 及 void 指针类型的使用方法与技巧。

2.void 的含义

void 的字面意思是“无类型”，void *则为“无类型指针”，void *可以指向任何类型的数据。

void 几乎只有“注释”和限制程序的作用，因为从来没有人会定义一个 void 变量，让我们试着来定义：

```
void a;
```

这行语句编译时会出错，提示“illegal use of type ‘void‘”。不过，即使 void a 的编译不会出错，它也没有任何实际意义。

void 真正发挥的作用在于：

（1） 对函数返回的限定；

（2） 对函数参数的限定。

我们将在第三节对以上二点进行具体说明。

众所周知，如果指针 p1 和 p2 的类型相同，那么我们可以直接在 p1 和 p2 间互相赋值；如果 p1 和 p2 指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。

例如：

```
float *p1;
int *p2;
p1 = p2;
```

其中 p1 = p2 语句会编译出错，提示“ ‘= ‘ : cannot convert from ‘int * ‘ to ‘float * ‘”，必须改为：

p1 = (float *)p2;

而 void *则不同，任何类型的指针都可以直接赋值给它，无需进行强制类型转换：

```
void *p1;
int *p2;
p1 = p2;
```

但这并不意味着，void *也可以无需强制类型转换地赋给其它类型的指针。因为“无类型”可以包容“有类型”，而“有类型”则不能包容“无类型”。道理很简单，我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”。下面的语句编译出错：

```
void *p1;
int *p2;
p2 = p1;
```

提示“ ‘= ‘ : cannot convert from ‘void * ‘ to ‘int * ‘”。

3. void 的使用

下面给出 void 关键字的使用规则：

规则一 如果函数没有返回值，那么应声明为 void 类型

在 C 语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为 void 类型。例如：

```
add ( int a, int b )
{
return a + b;
}
int main(int argc, char* argv[])
{
printf ( "2 + 3 = %d", add ( 2, 3) );
}
```

程序运行的结果为输出：

2 + 3 = 5

这说明不加返回值说明的函数的确为 int 函数。

林锐博士《高质量 C/C++编程》中提到：“C++语言有很严格的类型安全检查，不允许上述情况（指函数不加类型声明）发生”。可是编译器并不一定这么认定，譬如在 Visual C++6.0 中上述 add 函数的编译无错也无警告且运行正确，所以不能寄希望于编译器会做严格的类型检查。

因此，为了避免混乱，我们在编写 C/C++程序时，对于任何函数都必须一个不漏地指定其类型。如果函数没有返回值，一定要声明为 void 类型。这既是程序良好可读性的需要，也是编程规范性的要求。另外，加上 void 类型声明后，也可以发挥代码的“自注释”作用。代码的“自注释”即代码能自己注释自己。

规则二 如果函数无参数，那么应声明其参数为 void

在 C++语言中声明一个这样的函数：

```
int function(void)
{
return 1;
}
```

则进行下面的调用是不合法的：

```
function(2);
```

因为在 C++中，函数参数为 void 的意思是这个函数不接受任何参数。

我们在 Turbo C 2.0 中编译：

```
#include "stdio.h"
fun()
{
return 1;
}
main()
{
printf("%d",fun(2));
getchar();
}
```

编译正确且输出 1，这说明，在 C 语言中，可以给无参数的函数传送任意类型的参数，但是在 C++编译器中编译同样的代码则会出错。在 C++中，不能向无参数的函数传送任何参数，出错提示“‘fun‘ : function does not take 1 parameters”。

所以，无论在 C 还是 C++中，若函数不接受任何参数，一定要指明参数为 void。

规则三 小心使用 void 指针类型

按照 ANSI(American National Standards Institute)标准，不能对 void 指针进行算法操作，即下列操作都是不合法的：

```
void * pvoid;
pvoid++; //ANSI：错误
pvoid += 1; //ANSI：错误
//ANSI 标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。
//例如：
int *pint;
pint++; //ANSI：正确
```

pint++的结果是使其增大 sizeof(int)。

但是大名鼎鼎的 GNU(GNU‘s Not Unix 的缩写)则不这么认定，它指定 void *的算法操作与 char *一致。
因此下列语句在 GNU 编译器中皆正确：

```
pvoid++; //GNU：正确
pvoid += 1; //GNU：正确
```

pvoid++的执行结果是其增大了 1。

在实际的程序设计中，为迎合 ANSI 标准，并提高程序的可移植性，我们可以这样编写实现同样功能的代码：

```
void * pvoid;
(char *)pvoid++; //ANSI：正确；GNU：正确
(char *)pvoid += 1; //ANSI：错误；GNU：正确
```

GNU 和 ANSI 还有一些区别，总体而言，GNU 较 ANSI 更“开放”，提供了对更多语法的支持。但是我们在真实设计时，还是应该尽可能地迎合 ANSI 标准。

规则四 如果函数的参数可以是任意类型指针，那么应声明其参数为 void *

典型的如内存操作函数 memcpy 和 memset 的函数原型分别为：

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

这样，任何类型的指针都可以传入 memcpy 和 memset 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果 memcpy 和 memset 的参数类型不是 void *，而是 char *，那才叫真的奇怪了！这样的 memcpy 和 memset 明显不是一个“纯粹的，脱离低级趣味的”函数！

下面的代码执行正确：

```
//示例：memset 接受任意类型指针
int intarray[100];
memset ( intarray, 0, 100*sizeof(int) ); //将 intarray 清 0

//示例：memcpy 接受任意类型指针
int intarray1[100], intarray2[100];
memcpy ( intarray1, intarray2, 100*sizeof(int) ); //将 intarray2 拷贝给 intarray1
```

有趣的是，memcpy 和 memset 函数返回的也是 void *类型，标准库函数的编写者是多么地富有学问啊！

规则五 void 不能代表一个真实的变量

下面代码都企图让 void 代表一个真实的变量，因此都是错误的代码：

```
void a; //错误
function(void a); //错误
```

void 体现了一种抽象，这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人（还有人妖？）。

void 的出现只是为了一种抽象的需要，如果你正确地理解了面向对象中“抽象基类”的概念，也很容易理解 void 数据类型。正如不能给抽象基类定义一个实例，我们也不能定义一个 void（让我们类比的称 void 为“抽象数据类型”）变量。

4.总结

小小的 void 蕴藏着很丰富的设计哲学，作为一名程序设计人员，对问题进行深一个层次的思考必然使我们受益匪浅。

C/C++语言可变参数表深层探索

作者: 宋宝华 **e-mail:**21cnbao@21cn.com

1.引言

C/C++语言有一个不同于其它语言的特性，即其支持可变参数，典型的函数如 printf、scanf 等可以接受数量不定的参数。如：

```
printf ( "I love you" );
printf ( "%d", a );
printf ( "%d,%d", a, b );
```

第一、二、三个 printf 分别接受 1、2、3 个参数，让我们看看 printf 函数的原型：

```
int printf ( const char *format, ... );
```

从函数原型可以看出，其除了接收一个固定的参数 format 以外，后面的参数用“…”表示。在 C/C++语言中，“…”表示可以接受不定数量的参数，理论上来讲，可以是 0 或 0 以上的 n 个参数。

本文将对 C/C++可变参数表的使用方法及 C/C++支持可变参数表的深层机理进行探索。

2.可变参数表的用法

2.1 相关宏

标准 C/C++包含头文件 stdarg.h，该头文件中定义了如下三个宏：

```
void va_start ( va_list arg_ptr, prev_param ); /* ANSI version */
type va_arg ( va_list arg_ptr, type );
void va_end ( va_list arg_ptr );
```

在这些宏中，va 就是 variable argument(可变参数)的意思；arg_ptr 是指向可变参数表的指针；prev_param 则指可变参数表的前一个固定参数；type 为可变参数的类型。va_list 也是一个宏，其定义为 typedef char * va_list，实质上是一 char 型指针。char 型指针的特点是++、--操作对其作用的结果是增 1 和减 1（因为 sizeof(char)为 1），与之不同的是 int 等其它类型指针的++、--操作对其作用的结果是增 sizeof(type)或减 sizeof(type)，而且 sizeof(type)大于 1。

通过 va_start 宏我们可以取得可变参数表的首指针，这个宏的定义为：

```
#define va_start ( ap, v ) ( ap = (va_list)&v + _INTSIZEOF(v) )
```

显而易见，其含义为将最后那个固定参数的地址加上可变参数对其的偏移后赋值给 ap，这样 ap 就是可变参数表的首地址。其中的_INTSIZEOF 宏定义为：

```
#define _INTSIZEOF(n) ((sizeof ( n ) + sizeof ( int ) - 1 ) & ~( sizeof( int ) - 1 ) )
```

va_arg 宏的意思则指取出当前 arg_ptr 所指的可变参数并将 ap 指针指向下一可变参数,其原型为：

```
#define va_arg(list, mode) ((mode *)(list =\
 (char *) ((((int)list + (__builtin_alignof(mode)<=4?3:7)) &\
 (__builtin_alignof(mode)<=4?-4:-8))+sizeof(mode))))[-1]
```

对这个宏的具体含义我们将在第 3 节深入讨论。

而 va_end 宏被用来结束可变参数的获取，其定义为：

```
#define va_end ( list )
```

可以看出，va_end (list)实际上被定义为空，没有任何真实对应的代码，用于代码对称，与 va_start 对应；另外，它还可能发挥代码的“自注释”作用。所谓代码的“自注释”，指的是代码能自己注释自己。

下面我们以具体的例子来说明以上三个宏的使用方法。

2.2 一个简单的例子

```
#include <stdarg.h>
/*  函数名：max
 *  功能：返回 n 个整数中的最大值
 *  参数：num：整数的个数 ...：num 个输入的整数
 *  返回值：求得的最大整数
 */
int max ( int num, ... )
{
    int m = -0x7FFFFFFF;  /* 32 系统中最小的整数 */
    va_list ap;
    va_start ( ap, num );
    for ( int i= 0; i< num; i++ )
    {
        int t = va_arg (ap, int);
        if ( t > m )
        {
            m = t;
    }
    }
    va_end (ap);
    return m;
}

/* 主函数调用 max */
int main ( int argc, char* argv[] )
{
    int n = max ( 5, 5, 6 ,3 ,8 ,5);  /* 求 5 个整数中的最大值 */
    cout << n;
    return 0;
}
```

函数 max 中首先定义了可变参数表指针 ap，而后通过 va_start (ap, num)取得了参数表首地址（赋给了 ap），其后的 for 循环则用来遍历可变参数表。这种遍历方式与我们在数据结构教材中经常看到的遍历方式是类似的。

函数 max 看起来简洁明了，但是实际上 printf 的实现却远比这复杂。max 函数之所以看起来简单，是因为：

(1) max 函数可变参数表的长度是已知的，通过 num 参数传入；

(2) max 函数可变参数表中参数的类型是已知的，都为 int 型。

而 printf 函数则没有这么幸运。首先，printf 函数可变参数的个数不能轻易的得到，而可变参数的类型也不是固定的，需由格式字符串进行识别（由%f、%d、%s 等确定），因此则涉及到可变参数表的更复杂应用。

下面我们以实例来分析可变参数表的高级应用。

2.3 高级应用

下面这个程序是我们为某嵌入式系统（该系统中 CPU 的字长为 16 位）编写的在屏幕上显示格式字符串

的函数 DrawText，它的用法类似于 int printf (const char *format, ...)函数，但其输出的目标为嵌入式系统的液晶显示屏幕（LED）。

```
//////////////////////////////////////////////////////////////////////////////
//  函数名称: DrawText
//  功能说明: 在显示屏上绘制文字
//  参数说明: xPos ---横坐标的位置  [0 .. 30]
//            yPos ---纵坐标的位置  [0 .. 64]
//            ...  可以同数字一起显示，需设置标志(%d、%l、%x、%s)
//////////////////////////////////////////////////////////////////////////////
extern void DrawText ( BYTE xPos, BYTE yPos, LPBYTE lpStr, ... )
{
    BYTE   lpData[100];  //缓冲区
    BYTE   byIndex;
    BYTE   byLen;
    DWORD  dwTemp;
    WORD   wTemp;
    int    i;
    va_list lpParam;

    memset( lpData, 0, 100);
    byLen  = strlen( lpStr );
    byIndex = 0;
    va_start ( lpParam, lpStr );

    for ( i = 0; i < byLen; i++ )
    {
        if( lpStr[i] != '%' )    //不是格式符开始
        {
            lpData[byIndex++] = lpStr[i];
        }
        else
        {
            switch (lpStr[i+1])
            {
            //整型
            case   'd':
            case   'D':
                wTemp  = va_arg ( lpParam, int );
                byIndex += IntToStr( lpData+byIndex, (DWORD)wTemp );
                i++;
                break;
            //长整型
            case   'l':
            case   'L':
```

```
                dwTemp = va_arg ( lpParam, long );
                byIndex += IntToStr ( lpData+byIndex, (DWORD)dwTemp );
                i++;
                break;
            //16 进制（长整型）
            case   'x':
            case   'X':
                dwTemp = va_arg ( lpParam, long );
                byIndex += HexToStr ( lpData+byIndex, (DWORD)dwTemp );
                i++;
                break;
            default:
               lpData[byIndex++] = lpStr[i];
                break;
            }
        }
    }
    va_end ( lpParam );
    lpData[byIndex]   = '\0';
    DisplayString ( xPos, yPos, lpData, TRUE);  //在屏幕上显示字符串 lpData
}
```

在这个函数中，需通过对传入的格式字符串（首地址为 lpStr）进行识别来获知可变参数个数及各个可变参数的类型，具体实现体现在 for 循环中。譬如，在识别为%d 后，做的是 va_arg (lpParam, int)，而获知为%l 和%x 后则进行的是 va_arg (lpParam, long)。格式字符串识别完成后，可变参数也就处理完了。

在项目的最初，我们一直苦于不能找到一个好的办法来混合输出字符串和数字，我们采用了分别显示数字和字符串的方法，并分别指定坐标，程序条理被破坏。而且，在混合显示的时候，要给各类数据分别人工计算坐标，我们感觉头疼不已。以前的函数为：

```
//显示字符串
showString ( BYTE xPos, BYTE yPos, LPBYTE lpStr )
//显示数字
showNum ( BYTE xPos, BYTE yPos, int num )
//以 16 进制方式显示数字
showHexNum ( BYTE xPos, BYTE yPos, int num )
```

最终，我们用 DrawText (BYTE xPos, BYTE yPos, LPBYTE lpStr, ...)函数代替了原先所有的输出函数，程序得到了简化。就这样，兄弟们用得爽翻了。

3. 运行机制探索

通过第 2 节我们学会了可变参数表的使用方法，相信喜欢抛根问底的读者还不甘心，必然想知道如下问题：

（1）为什么按照第 2 节的做法就可以获得可变参数并对其进行操作？

（2）C/C++在底层究竟是依靠什么来对这一语法进行支持的，为什么其它语言就不能提供可变参数表呢？

我们带着这些疑问来一步步进行摸索。

3.1 调用机制反汇编

反汇编是研究语法深层特性的终极良策，先来看看 2.2 节例子中主函数进行 max（ 5, 5, 6 ,3 ,8 ,5)调用时的反汇编：

```
1. 004010C8   push        5
2. 004010CA   push        8
3. 004010CC   push        3
4. 004010CE   push        6
5. 004010D0   push        5
6. 004010D2   push        5
7. 004010D4   call        @ILT+5(max) (0040100a)
```

从上述反汇编代码中我们可以看出，C/C++函数调用的过程中：

第一步：将参数从右向左入栈（第 1～6 行）；

第二步：调用 call 指令进行跳转（第 7 行）。

这两步包含了深刻的含义，它说明 C/C++默认的调用方式为由调用者管理参数入栈的操作，且入栈的顺序为从右至左，这种调用方式称为_cdecl 调用。x86 系统的入栈方向为从高地址到低地址，故第 1 至 n 个参数被放在了地址递增的堆栈内。在被调用函数内部，读取这些堆栈的内容就可获得各个参数的值，让我们反汇编到 max 函数的内部：

```
int max ( int num, ...)
{
1. 00401020   push        ebp
2. 00401021   mov         ebp,esp
3. 00401023   sub         esp,50h
4. 00401026   push        ebx
5. 00401027   push        esi
6. 00401028   push        edi
7. 00401029   lea         edi,[ebp-50h]
8. 0040102C   mov         ecx,14h
9. 00401031   mov         eax,0CCCCCCCCh
10. 00401036   rep stos     dword ptr [edi]
      va_list ap;
      int m = -0x7FFFFFFF; /* 32 系统中最小的整数 */
11. 00401038   mov         dword ptr [ebp-8],80000001h
      va_start ( ap, num );
12. 0040103F   lea         eax,[ebp+0Ch]
13. 00401042   mov         dword ptr [ebp-4],eax
      for ( int i= 0; i< num; i++ )
14. 00401045   mov         dword ptr [ebp-0Ch],0
15. 0040104C   jmp         max+37h (00401057)
16. 0040104E   mov         ecx,dword ptr [ebp-0Ch]
17. 00401051   add         ecx,1
18. 00401054   mov         dword ptr [ebp-0Ch],ecx
19. 00401057   mov         edx,dword ptr [ebp-0Ch]
20. 0040105A   cmp         edx,dword ptr [ebp+8]
21. 0040105D   jge         max+61h (00401081)
```

```
        {
            int t= va_arg (ap, int);
22. 0040105F    mov         eax,dword ptr [ebp-4]
23. 00401062    add         eax,4
24. 00401065    mov         dword ptr [ebp-4],eax
25. 00401068    mov         ecx,dword ptr [ebp-4]
26. 0040106B    mov         edx,dword ptr [ecx-4]
27. 0040106E    mov         dword ptr [t],edx
            if ( t > m )
28. 00401071    mov         eax,dword ptr [t]
29. 00401074    cmp         eax,dword ptr [ebp-8]
30. 00401077    jle         max+5Fh (0040107f)
                m = t;
31. 00401079    mov         ecx,dword ptr [t]
32. 0040107C    mov         dword ptr [ebp-8],ecx
        }
33. 0040107F    jmp         max+2Eh (0040104e)
        va_end (ap);
34. 00401081    mov         dword ptr [ebp-4],0
        return m;
35. 00401088    mov         eax,dword ptr [ebp-8]
    }
36. 0040108B    pop         edi
37. 0040108C    pop         esi
38. 0040108D    pop         ebx
39. 0040108E    mov         esp,ebp
40. 00401090    pop         ebp
41. 00401091    ret
```

分析上述反汇编代码，对于一个真正的程序员而言，将是一种很大的享受；而对于初学者，也将使其受益良多。所以请一定要赖着头皮认真研究，千万不要被吓倒！

行 1～10 进行执行函数内代码的准备工作，保存现场。第 2 行对堆栈进行移动；第 3 行则意味着 max 函数为其内部局部变量准备的堆栈空间为 50h 字节；第 11 行表示把变量 n 的内存空间安排在了函数内部局部栈底减 8 的位置（占用 4 个字节）。

第 12～13 行非常关键，对应着 va_start (ap, num)，这两行将第一个可变参数的地址赋值给了指针 ap。另外，从第 12 行可以看出 num 的地址为 ebp+0Ch；从第 13 行可以看出 ap 被分配在函数内部局部栈底减 4 的位置上（占用 4 个字节）。

第 22～27 行最为关键，对应着 va_arg (ap, int)。其中，22~24 行的作用为将 ap 指向下一可变参数（可变参数的地址间隔为 4 个字节，从 add eax,4 可以看出）；25~27 行则取当前可变参数的值赋给变量 t。这段反汇编很奇怪，它先移动可变参数指针，再在赋值指令里面回过头来取先前的参数值赋给 t（从 mov edx,dword ptr [ecx-4]语句可以看出）。Visual C++同学玩得有意思，不知道碰见同样的情况 Visual Basic 等其它同学怎么玩？

第 36～41 行恢复现场和堆栈地址，执行函数返回操作。

痛苦的反汇编之旅差不多结束了，看了这段反汇编我们总算弄明白了可变参数的存放位置以及它们被读取的方式，顿觉全省轻松！

3.2 特殊的调用约定

除此之外，我们需要了解 C/C++函数调用对参数占用空间的一些特殊约定，因为在_cdecl 调用协议中，有些变量类型是按照其它变量的尺寸入栈的。

例如，字符型变量将被自动扩展为一个字的空间，因为入栈操作针对的是一个字。

参数 n 实际占用的空间为((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))，这就是第 2.1 节_INTSIZEOF(v)宏的来历！

既然如此，2.1 节给出的 va_arg (list, mode)宏为什么玩这么大的飞机就很清楚了。这个问题就留个读者您来分析。

C/C++数组名与指针区别深层探索

作者： 宋宝华 **e-mail:**21cnbao@21cn.com

1. 引言

指针是 C/C++语言的特色，而数组名与指针有太多的相似，甚至很多时候，数组名可以作为指针使用。于是乎，很多程序设计者就被搞糊涂了。而许多的大学老师，他们在 C 语言的教学过程中也错误得给学生讲解：“数组名就是指针”。很幸运，我的大学老师就是其中之一。时至今日，我日复一日地进行着 C/C++项目的开发，而身边还一直充满这样的程序员，他们保留着“数组名就是指针”的误解。

想必这种误解的根源在于国内某著名的 C 程序设计教程。如果这篇文章能够纠正许多中国程序员对数组名和指针的误解，笔者就不甚欣慰了。借此文，笔者站在无数对知识如饥似渴的中国程序员之中，深深寄希望于国内的计算机图书编写者们，能以“深入探索”的思维方式和精益求精的认真态度来对待图书编写工作，但愿市面上多一些融入作者思考结晶的心血之作！

2. 魔幻数组名

请看程序（本文程序在 WIN32 平台下编译）：

```
1. #include <iostream.h>
2. int main(int argc, char* argv[])
3. {
4.   char str[10];
5.   char *pStr = str;
6.   cout << sizeof(str) << endl;
7.   cout << sizeof(pStr) << endl;
8.   return 0;
9. }
```

2.1 数组名不是指针

我们先来推翻“数组名就是指针”的说法，用反证法。

证明 数组名不是指针

假设：数组名是指针；

则：pStr 和 str 都是指针；

因为：在 WIN32 平台下，指针长度为 4；

所以：第 6 行和第 7 行的输出都应该为 4；

实际情况是：第 6 行输出 10，第 7 行输出 4；

所以：假设不成立，数组名不是指针

2.2 数组名神似指针

上面我们已经证明了数组名的确不是指针，但是我们再看看程序的第 5 行。该行程序将数组名直接赋值给指针，这显

得数组名又的确是个指针！
我们还可以发现数组名显得像指针的例子：

```
1. #include <string.h>
2. #include <iostream.h>
3. int main(int argc, char* argv[])
4. {
5.  char str1[10] = "I Love U";
6.  char str2[10];
7.  strcpy(str2,str1);
8.  cout << "string array 1: " << str1 << endl;
9.  cout << "string array 2: " << str2 << endl;
10. return 0;
11. }
```

标准 C 库函数 strcpy 的函数原形中能接纳的两个参数都为 char 型指针，而我们在调用中传给它的却是两个数组名！
函数输出：

```
string array 1: I Love U
string array 2: I Love U
```

数组名再一次显得像指针！
既然数组名不是指针，而为什么到处都把数组名当指针用？于是乎，许多程序员得出这样的结论：数组名（主）是（谓）不是指针的指针（宾）。
整个一魔鬼。
3. 数组名大揭密
那么，是揭露数组名本质的时候了，先给出三个结论：
(1)数组名的内涵在于其指代实体是一种数据结构，这种数据结构就是数组；
(2)数组名的外延在于其可以转换为指向其指代实体的指针，而且是一个指针常量；
(3)指向数组的指针则是另外一种变量类型（在 WIN32 平台下，长度为 4），仅仅意味着数组的存放地址！
3.1 数组名指代一种数据结构：数组
现在可以解释为什么第 1 个程序第 6 行的输出为 10 的问题，根据结论 1，数组名 str 的内涵为一种数据结构，即一个长度为 10 的 char 型数组，所以 sizeof(str)的结果为这个数据结构占据的内存大小：10 字节。
再看：

```
1. int intArray[10];
2. cout << sizeof(intArray) ;
```

第 2 行的输出结果为 40（整型数组占据的内存空间大小）。
如果 C/C++程序可以这样写：

```
1. int[10] intArray;
2. cout << sizeof(intArray) ;
```

我们就都明白了，intArray 定义为 int[10]这种数据结构的一个实例，可惜啊，C/C++目前并不支持这种定义方式。
3.2 数组名可作为指针常量
根据结论 2，数组名可以转换为指向其指代实体的指针，所以程序 1 中的第 5 行数组名直接赋值给指针，程序 2 第 7 行直接将数组名作为指针形参都可成立。
下面的程序成立吗？

```
1. int intArray[10];
2. intArray++;
```

读者可以编译之，发现编译出错。原因在于，虽然数组名可以转换为指向其指代实体的指针，但是它只能被看作一个

指针常量，不能被修改。
而指针，不管是指向结构体、数组还是基本数据类型的指针，都不包含原始数据结构的内涵，在 WIN32 平台下，sizeof 操作的结果都是 4。
顺便纠正一下许多程序员的另一个误解。**许多程序员以为 sizeof 是一个函数，而实际上，它是一个操作符**，不过其使用方式看起来的确太像一个函数了。语句 sizeof(int)就可以说明 sizeof 的确不是一个函数，因为函数接纳形参（一个变量），世界上没有一个 C/C++函数接纳一个数据类型（如 int）为“形参”。

3.3 数据名可能失去其数据结构内涵

到这里似乎数组名魔幻问题已经宣告圆满解决，但是平静的湖面上却再次掀起波浪。请看下面一段程序：

```
1. #include <iostream.h>
2. void arrayTest(char str[])
3. {
4.       cout << sizeof(str) << endl;
5. }
6. int main(int argc, char* argv[])
7. {
8.       char str1[10] = "I Love U";
9.       arrayTest(str1);
10.      return 0;
11. }
```

程序的输出结果为 4。不可能吧？
4，一个可怕的数字，前面已经提到其为指针的长度！
结论 1 指出，数据名内涵为数组这种数据结构，在 arrayTest 函数体内，str 是数组名，那为什么 sizeof 的结果却是指针的长度？这是因为：
(1)数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；
(2)很遗憾，在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。
所以，数据名作为函数形参时，其全面沦落为一个普通指针！它的贵族身份被剥夺，成了一个地地道道的只拥有 4 个字节的平民。
以上就是结论 4。

4. 结论

本文以打破沙锅问到底的探索精神用数段程序实例论证了数据名和指针的区别。
最后，笔者再次表达深深的希望，愿我和我的同道中人能够真正以谨慎的研究态度来认真思考开发中的问题，这样才能在我们中间产生大师级的程序员，顶级的开发书籍。每次拿着美国鬼子的开发书籍，我们不免发出这样的感慨：我们落后太远了。

C/C++程序员应聘常见面试题深入剖析(1)

作者: 宋宝华 **e-mail:**21cnbao@21cn.com **出处:** 软件报

1.引言

本文的写作目的并不在于提供 C/C++程序员求职面试指导，而旨在从技术上分析面试题的内涵。文中的大多数面试题来自各大论坛，部分试题解答也参考了网友的意见。
许多面试题看似简单，却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 strcpy 函数都可看出面试者在技术上究竟达到了怎样的程度，我们能真正写好一个 strcpy 函数吗？我们都觉得自己能，可是我们写出的 strcpy 很可能只能拿到 10 分中的 2 分。读者可从本文看到 strcpy

函数从 2 分到 10 分解答的例子，看看自己属于什么样的层次。此外，还有一些面试题考查面试者敏捷的思维能力。
分析这些面试题，本身包含很强的趣味性；而作为一名研发人员，通过对这些面试题的深入剖析则可进一步增强自身的内功。

2. 找错题

试题 1：

```
void test1()
{
   char string[10];
   char* str1 = "0123456789";
    strcpy( string, str1 );
}
```

试题 2：

```
void test2()
{
   char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

试题 3：

```
void test3(char* str1)
{
   char string[10];
   if( strlen( str1 ) <= 10 )
    {
       strcpy( string, str1 );
    }
}
```

解答：

试题 1 字符串 str1 需要 11 个字节才能存放下（包括末尾的'\0'），而 string 只有 10 个字节的空间，strcpy 会导致数组越界；
对试题 2，如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string, str1)调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；
对试题 3，if(strlen(str1) <= 10)应改为 if(strlen(str1) < 10)，因为 strlen 的结果未统计'\0'所占用的 1 个字节。

剖析：

考查对基本功的掌握：
（1）字符串以'\0'结尾；

（2）对数组越界把握的敏感度；
（3）库函数 strcpy 的工作方式，如果编写一个标准 strcpy 函数的总分值为 10，下面给出几个不同得分的答案：

2 分

```
void strcpy( char *strDest, char *strSrc )
{
   while( (*strDest++ = * strSrc++) != ‘\0’ );
}
```

4 分

```
void strcpy( char *strDest, const char *strSrc )
//将源字符串加 const，表明其为输入参数，加 2 分
{
   while( (*strDest++ = * strSrc++) != ‘\0’ );
}
```

7 分

```
void strcpy(char *strDest, const char *strSrc)
{
//对源地址和目的地址加非 0 断言，加 3 分
  assert( (strDest != NULL) && (strSrc != NULL) );
  while( (*strDest++ = * strSrc++)  !=  ‘\0’ );
}
```

10 分

```
//为了实现链式操作，将目的地址返回，加 3 分！
char * strcpy( char *strDest, const char *strSrc )
{
  assert( (strDest != NULL) && (strSrc != NULL) );
  char *address = strDest;
  while( (*strDest++ = * strSrc++) != ‘\0’ );
  return address;
}
```

从 2 分到 10 分的几个答案我们可以清楚的看到，小小的 strcpy 竟然暗藏着这么多玄机，真不是盖的！需要多么扎实的基本功才能写一个完美的 strcpy 啊！

（4）对 strlen 的掌握，它没有包括字符串末尾的'\0'。

读者看了不同分值的 strcpy 版本，应该也可以写出一个 10 分的 strlen 函数了，完美的版本为：

```
int strlen( const char *str )    //输入参数 const
{
   assert( strt != NULL );    //断言字符串地址非 0
   int len;
   while( (*str++) != '\0' )
   {
      len++;
   }
   return len;
}
```

试题 4：

```
void GetMemory( char *p )
{
p = (char *) malloc( 100 );
}
void Test( void )
{
char *str = NULL;
GetMemory( str );
strcpy( str, "hello world" );
printf( str );
}
```

试题 5：

```
char *GetMemory( void )
{
    char p[] = "hello world";
    return p;
}
void Test( void )
{
    char *str = NULL;
    str = GetMemory();
    printf( str );
}
```

试题 6：

```
void GetMemory( char **p, int num )
{
    *p = (char *) malloc( num );
}
void Test( void )
{
    char *str = NULL;
    GetMemory( &str, 100 );
    strcpy( str, "hello" );
    printf( str );
}
```

试题 7：

```
void Test( void )
{
    char *str = (char *) malloc( 100 );
    strcpy( str, "hello" );
    free( str );
    ...  //省略的其它语句
}
```

解答：

试题 4 传入中 GetMemory(char *p)函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的值，执行完

```
char *str = NULL;
GetMemory( str );
```

后的 str 仍然为 NULL；

试题 5 中

```
    char p[] = "hello world";
    return p;
```

的 p[]数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

试题 6 的 GetMemory 避免了试题 4 的问题，传入 GetMemory 的参数为字符串指针的指针，但是在 GetMemory 中执行申请内存及赋值语句

```
*p = (char *) malloc( num );
```

后未判断内存是否申请成功，应加上：

```
if ( *p == NULL )
 {
   ...//进行申请内存失败处理
  }
```

试题 7 存在与试题 6 同样的问题，在执行

```
char *str = (char *) malloc(100);
```

后未进行内存是否申请成功的判断；另外，在 free(str)后未置 str 为空，导致可能变成一个“野”指针，应加上：

```
str = NULL;
```

试题 6 的 Test 函数中也未对 malloc 的内存进行释放。

剖析：

试题 4～7 考查面试者对内存操作的理解程度，基本功扎实的面试者一般都能正确的回答其中 50~60 的错误。但是要完全解答正确，却也绝非易事。

对内存操作的考查主要集中在：

（1）指针的理解；

（2）变量的生存期及作用范围；

（3）良好的动态内存申请和释放习惯。

在看看下面的一段程序有什么错误：

```
swap( int* p1,int* p2 )
{
    int *p;
    *p = *p1;
    *p1 = *p2;
    *p2 = *p;
}
```

在 swap 函数中，p 是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在 VC++中 DEBUG 运行时提示错误“Access Violation”。该程序应该改为：

```
swap( int* p1,int* p2 )
{
```

```
    int p;
    p = *p1;
    *p1 = *p2;
    *p2 = p;
}
```

C/C++程序员应聘常见面试题深入剖析(2)

作者：宋宝华　**e-mail**:21cnbao@21cn.com　　**出处**：软件报

3.内功题

试题 1：分别给出 BOOL，int，float，指针变量 与“零值”比较的 if 语句（假设变量名为 var）

解答：

BOOL 型变量：if(!var)

int 型变量：　if(var==0)

float 型变量：

```
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON)
```

指针变量：　if(var==NULL)

剖析：

考查对 0 值判断的“内功”，BOOL 型变量的 0 判断完全可以写成 if(var==0)，而 int 型变量也可以写成 if(!var)，指针变量的判断也可以写成 if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。

一般的，如果想让 if 判断一个变量的“真”、“假”，应直接使用 if(var)、if(!var)，表明其为“逻辑”判断；如果用 if 判断一个数值型变量(short、int、long 等)，应该用 if(var==0)，表明是与 0 进行“数值”上的比较；而判断指针则适宜用 if(var==NULL)，这是一种很好的编程习惯。

浮点型变量并不精确，所以不可将 float 变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。如果写成 if (x == 0.0)，则判为错，得 0 分。

试题 2：以下为 Windows NT 下的 32 位 C++程序，请计算 sizeof 的值

```
void Func ( char str[100] )
{
    sizeof( str ) = ?
}
void *p = malloc( 100 );
sizeof ( p ) = ?
```

解答：

```
sizeof( str ) = 4
sizeof ( p ) = 4
```

剖析：

Func (char str[100])函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

数组名的本质如下：

（1）数组名指代一种数据结构，这种数据结构就是数组；
例如：
```
char str[10];
cout << sizeof(str) << endl;
```
输出结果为 10，str 指代数据结构 char[10]。
（2）数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；
```
char str[10];
str++;   //编译出错，提示 str 不是左值
```
（3）数组名作为函数形参时，沦为普通指针。
Windows NT 32 位平台下，指针的长度（占用内存的大小）为 4 字节，故 sizeof(str) 、sizeof (p) 都为 4。
试题 3：写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事？
```
least = MIN(*p++, b);
```
解答：
```
#define MIN(A,B)    ((A) <=  (B) ? (A) : (B))
```
MIN(*p++, b)会产生宏的副作用
剖析：
这个面试题主要考查面试者对宏定义的使用，宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。
程序员对宏定义的使用要非常小心，特别要注意两个问题：
（1）谨慎地将宏定义中的“参数”和整个宏用用括弧括起来。所以，严格地讲，下述解答：
```
#define MIN(A,B)    (A) <=  (B) ? (A) : (B)
#define MIN(A,B)    (A <=  B ? A : B )
```
都应判 0 分；
（2）防止宏的副作用。
宏定义#define MIN(A,B)   ((A) <= (B) ? (A) : (B))对 MIN(*p++, b)的作用结果是：
```
((*p++) <=  (b) ? (*p++) : (b))
```
这个表达式会产生副作用，指针 p 会作两次++自增操作。
除此之外，另一个应该判 0 分的解答是：
```
#define MIN(A,B)    ((A) <=  (B) ? (A) : (B));
```
这个解答在宏定义的后面加“;”，显示编写者对宏的概念模糊不清，只能被无情地判 0 分并被面试官淘汰。
试题 4：为什么标准头文件都有类似以下的结构？
```
  #ifndef __INCvxWorksh
  #define __INCvxWorksh
  #ifdef __cplusplus
  extern "C" {
  #endif
  /*...*/
  #ifdef __cplusplus
  }
```

```
#endif
#endif /* __INCvxWorksh */
```

解答：

头文件中的编译宏

```
#ifndef  __INCvxWorksh
#define  __INCvxWorksh
#endif
```

的作用是防止被重复引用。

作为一种面向对象的语言，C++支持函数重载，而过程式语言 C 则不支持。函数被 C++编译后在 symbol 库中的名字与 C 语言的不同。例如，假设某个函数的原型为：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后在 symbol 库中的名字为_foo，而 C++编译器则会产生像_foo_int_int 之类的名字。_foo_int_int 这样的名字包含了函数名和函数参数数量及类型信息，C++就是考这种机制来实现函数重载的。

为了实现 C 和 C++的混合编程，C++提供了 C 连接交换指定符号 extern "C"来解决名字匹配问题，函数声明前加上 extern "C"后，则编译器就会按照 C 语言的方式将该函数编译为_foo，这样 C 语言中就可以调用 C++的函数了。

试题 5：编写一个函数，作用是把一个 char 组成的字符串循环右移 n 个。比如原来是“abcdefghi”如果 n=2，移位后应该是“hiabcdefgh”

函数头是这样的：

```
//pStr 是指向以'\0'结尾的字符串的指针
//steps 是要求移动的 n
void LoopMove ( char * pStr, int steps )
{
//请填充...
}
```

解答：

正确解答 1：

```
void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    strcpy ( tmp, pStr + n );
    strcpy ( tmp + steps, pStr);
    *( tmp + strlen ( pStr ) ) = '\0';
    strcpy( pStr, tmp );
}
```

正确解答 2：

```
void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    memcpy( tmp, pStr + n, steps );
    memcpy(pStr + steps, pStr, n );
```

```
    memcpy(pStr, tmp, steps );
}
```

剖析：

这个试题主要考查面试者对标准库函数的熟练程度，在需要的时候引用库函数可以很大程度上简化程序编写的工作量。

最频繁被使用的库函数包括：

（1）strcpy

（2）memcpy

（3）memset

试题 6：已知 WAV 文件格式如下表，打开一个 WAV 文件，以适当的数据结构组织 WAV 文件头并解析 WAV 格式的各项信息。

WAVE 文件格式说明表

	偏移地址	字节数	数据类型	内　　容
文件头	00H	4	Char	"RIFF"标志
	04H	4	int32	文件长度
	08H	4	Char	"WAVE"标志
	0CH	4	Char	"fmt"标志
	10H	4		过渡字节（不定）
	14H	2	int16	格式类别
	16H	2	int16	通道数
	18H	2	int16	采样率（每秒样本数），表示每个通道的播放速度
	1CH	4	int32	波形音频数据传送速率
	20H	2	int16	数据块的调整数（按字节算的）
	22H	2		每样本的数据位数
	24H	4	Char	数据标记符 " data "
	28H	4	int32	语音数据的长度

解答：

将 WAV 文件格式定义为结构体 WAVEFORMAT：

```
typedef struct tagWaveFormat
{
        char cRiffFlag[4];
        UIN32 nFileLen;
        char cWaveFlag[4];
        char cFmtFlag[4];
        char cTransition[4];
        UIN16 nFormatTag ;
        UIN16 nChannels;
        UIN16 nSamplesPerSec;
        UIN32 nAvgBytesperSec;
        UIN16 nBlockAlign;
        UIN16 nBitNumPerSample;
        char cDataFlag[4];
```

```
        UIN16 nAudioLength;
} WAVEFORMAT;
```
假设 WAV 文件内容读出后存放在指针 buffer 开始的内存单元内，则分析文件格式的代码很简单，为：
```
WAVEFORMAT  waveFormat;
memcpy( &waveFormat, buffer,sizeof( WAVEFORMAT ) );
```
直接通过访问 waveFormat 的成员，就可以获得特定 WAV 文件的各项格式信息。

剖析：

试题 6 考查面试者组织数据结构的能力，有经验的程序设计者将属于一个整体的数据成员组织为一个结构体，利用指针类型转换，可以将 memcpy、memset 等函数直接用于结构体地址，进行结构体的整体操作。

透过这个题可以看出面试者的程序设计经验是否丰富。

试题 7：编写类 String 的构造函数、析构函数和赋值函数，已知类 String 的原型为：
```
class String
{
public:
    String(const char *str = NULL); // 普通构造函数
    String(const String &other); // 拷贝构造函数
    ~ String(void); // 析构函数
    String & operate =(const String &other); // 赋值函数
private:
    char *m_data; // 用于保存字符串
};
```
解答：
```
//普通构造函数
String::String(const char *str)
{
        if(str==NULL)
        {
            m_data = new char[1]; // 得分点：对空字符串自动申请存放结束标志'\0'的空
                                  //加分点：对 m_data 加 NULL 判断
            *m_data = '\0';
    }
        else
        {
        int length = strlen(str);
        m_data = new char[length+1]; // 若能加 NULL 判断则更好
        strcpy(m_data, str);
        }
}
// String 的析构函数
String::~String(void)
{
    delete [] m_data; // 或 delete m_data;
}
```

```
//拷贝构造函数
String::String(const String &other)        // 得分点：输入参数为 const 型
{
        int length = strlen(other.m_data);
        m_data = new char[length+1];          //加分点：对 m_data 加 NULL 判断
        strcpy(m_data, other.m_data);
}
//赋值函数
String & String::operate =(const String &other) // 得分点：输入参数为 const 型
{
        if(this == &other)                //得分点：检查自赋值
            return *this;
        delete [] m_data;              //得分点：释放原有的内存资源
        int length = strlen( other.m_data );
        m_data = new char[length+1];    //加分点：对 m_data 加 NULL 判断
        strcpy( m_data, other.m_data );
        return *this;                   //得分点：返回本对象的引用
}
```

剖析：

能够准确无误地编写出 String 类的构造函数、拷贝构造函数、赋值函数和析构函数的面试者至少已经具备了 C++基本功的 60%以上！

在这个类中包括了指针类成员变量 m_data，当类中包括指针类成员变量时，一定要重载其拷贝构造函数、赋值函数和析构函数，这既是对 C++程序员的基本要求，也是《Effective C++》中特别强调的条款。

仔细学习这个类，特别注意加注释的得分点和加分点的意义，这样就具备了 60%以上的 C++基本功！

试题 8：请说出 static 和 const 关键字尽可能多的作用

解答：

static 关键字至少有下列 n 个作用：

（1）函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；

（2）在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

（3）在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；

（4）在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

（5）在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

const 关键字至少有下列 n 个作用：

（1）欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；

（2）对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const；

（3）在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；

（4）对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量；

（5）对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。例

如：

```
const classA operator*(const classA& a1,const classA& a2);
```

operator*的返回结果必须是一个 const 对象。如果不是，这样的变态代码也不会编译出错：

```
classA a, b, c;
(a * b) = c; // 对 a*b 的结果赋值
```

操作(a * b) = c 显然不符合编程者的初衷，也没有任何意义。

剖析：

惊讶吗？小小的 static 和 const 居然有这么多功能，我们能回答几个？如果只能回答 1~2 个，那还真得闭关再好好修炼修炼。

这个题可以考查面试者对程序设计知识的掌握程度是初级、中级还是比较深入，没有一定的知识广度和深度，不可能对这个问题给出全面的解答。大多数人只能回答出 static 和 const 关键字的部分功能。

4. 技巧题

试题 1：请写一个 C 函数，若处理器是 Big_endian 的，则返回 0；若是 Little_endian 的，则返回 1

解答：

```
int checkCPU()
{
    {
        union w
        {
            int  a;
            char b;
        } c;
        c.a = 1;
        return (c.b == 1);
    }
}
```

剖析：

嵌入式系统开发者应该对 Little-endian 和 Big-endian 模式非常了解。采用 Little-endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big-endian 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001
存放内容	0x12	0x34

32bit 宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式(假设从地址 0x4000 开始存放)为：

内存地	0x4000	0x4001	0x4002	0x4003

址				
存放内容	0x78	0x56	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

联合体 union 的存放顺序是所有成员都从低地址开始存放，面试者的解答利用该特性，轻松地获得了 CPU 对内存采用 Little-endian 还是 Big-endian 模式读写。如果谁能当场给出这个解答，那简直就是一个天才的程序员。

试题 2：写一个函数返回 1+2+3+…+n 的值（假定结果不会超过长整型变量的范围）

解答：

```
int Sum( int n )
{
    return  ( (long)1 + n) * n / 2;    //或 return  (1l + n) * n / 2;
}
```

剖析：

对于这个题，只能说，也许最简单的答案就是最好的答案。下面的解答，或者基于下面的解答思路去优化，不管怎么“折腾”，其效率也不可能与直接 return (1l + n) * n / 2 相比！

```
int Sum( int n )
{
  long sum = 0;
  for( int i=1; i<=n; i++ )
  {
     sum += i;
  }
  return sum;
}
```

所以程序员们需要敏感地将数学等知识用在程序设计中。

一道著名外企面试题的抽丝剥茧

宋宝华 21cnbao@21cn.com 软件报

问题：对于一个字节（8bit）的数据，求其中“1”的个数，要求算法的执行效率尽可能地高。

分析：作为一道著名外企的面试题，看似简单，实则可以看出一个程序员的基本功底的扎实程度。你或许已经想到很多方法，譬如除、余操作，位操作等，但都不是最快的。本文一步步分析，直到最后给出一个最快的方法，相信你看到本文最后的那个最快的方法时会有惊诧的感觉。

解答：

首先，很自然的，你想到除法和求余运算，并给出了如下的答案：

方法 1：使用除、余操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  for (i = 0; i < 8; i++)
  {
    if (a % 2 == 1)
    {
      num++;
    }
    a = a / 2;
  }
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

很遗憾，众所周知，除法操作的运算速率实在是很低的，这个答案只能意味着面试者被淘汰！

好，精明的面试者想到了以位操作代替除法和求余操作，并给出如下答案：

方法 2：使用位操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  for (i = 0; i < 8; i++)
  {
    num += (a >> i) &0x01;
  }
  /*或者这样计算 1 的个数：*/
  /* for(i=0;i<8;i++)
  {
  if((a>>i)&0x01)
  num++;
  }
```

```
   */
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

方法二中 num += (a >> i) &0x01;操作的执行效率明显高于方法一中的

```
if (a % 2 == 1)
    {
      num++;
    }
a = a / 2;
```

到这个时候，面试者有被录用的可能性了，但是，难道最快的就是这个方法了吗？没有更快的了吗？方法二真的高山仰止了吗？

能不能不用做除法、位操作就直接得出答案的呢？于是你想到把 0~255 的情况都罗列出来，并使用分支操作，给出如下答案：

方法 3：使用分支操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  switch (a)
  {
    case 0x0:
      num = 0;
      break;
    case 0x1:
    case 0x2:
    case 0x4:
    case 0x8:
    case 0x10:
    case 0x20:
    case 0x40:
    case 0x80:
      num = 1;
      break;
    case 0x3:
    case 0x6:
    case 0xc:
    case 0x18:
```

```
    case 0x30:
    case 0x60:
    case 0xc0:
      num = 2;
      break;
      //...
  }
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

方法三看似很直接，实际执行效率可能还会小于方法二，因为分支语句的执行情况要看具体字节的值，如果 a=0，那自然在第 1 个 case 就得出了答案，但是如果 a=255，则要在最后一个 case 才得出答案，即在进行了 255 次比较操作之后！

看来方法三不可取！但是方法三提供了一个思路，就是罗列并直接给出值，离最后的方法四只有一步之遥。眼看着就要被这家著名外企录用，此时此刻，绝不对放弃寻找更快的方法。

终于，灵感一现，得到方法四，一个令你心潮澎湃的答案，快地令人咋舌，算法中不需要进行任何的运算。你有足够的信心了，把下面的答案递给面试官：

方法 4：直接得到结果

```
#include
#define BYTE unsigned char
/* 定义查找表 */
BYTE numTable[256] =
{
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3,
    3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3,
    4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4,
    3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3,
    4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6,
    6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4,
    5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3,
    4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4,
    4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
    7, 6, 7, 7, 8
};
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a = 0;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  /* 用 BYTE 直接作为数组的下标取出 1 的个数，妙哉！ */
```

```
  printf("\nthe num of 1 in the BYTE is %d", checknum[a]);
  return 0;
}
```

这是个典型的空间换时间算法，把 0~255 中 1 的个数直接存储在数组中，字节 a 作为数组的下标，checknum[a] 直接就是 a 中“1”的个数！算法的复杂度如下：
时间复杂度：O(1)
空间复杂度：O(2n)
恭喜你，你已经被这家著名的外企录用！老总向你伸出手，说：“Welcome to our company”。

C/C++结构体的一个高级特性－－指定成员的位数

宋宝华 21cnbao@21cn.com sweek
在大多数情况下，我们一般这样定义结构体：

```
struct student
{
    unsigned int sex;
    unsigned int age;
};
```

对于一般的应用，这已经能很充分地实现数据了的“封装”。
但是，在实际工程中，往往碰到这样的情况：那就是要用一个基本类型变量中的不同的位表示不同的含义。譬如一个 cpu 内部的标志寄存器，假设为 16 bit，而每个 bit 都可以表达不同的含义，有的表示结果是否为 0，有的表示是否越界等等。这个时候我们用什么数据结构来表达这个寄存器呢？
答案还是结构体！
为达到此目的，我们要用到结构体的高级特性，那就是在基本成员变量的后面添加：

```
: 数据位数
```

组成新的结构体：

```
struct xxx
{
    成员 1 类型成员 1 : 成员 1 位数;
    成员 2 类型成员 2 : 成员 2 位数;
    成员 3 类型成员 3 : 成员 3 位数;
};
```

基本的成员变量就会被拆分！这个语法在初级编程中很少用到，但是在高级程序设计中不断地被用到！
例如：

```
struct student
{
    unsigned int sex : 1;
    unsigned int age : 15;
};
```

上述结构体中的两个成员 sex 和 age 加起来只占用了一个 unsigned int 的空间（假设 unsigned int 为 16 位）。
基本成员变量被拆分后，访问的方法仍然和访问没有拆分的情况是一样的，例如：

```
struct student sweek;
sweek.sex = MALE;
sweek.age = 20;
```
虽然拆分基本成员变量在语法上是得到支持的，但是并不等于我们想怎么分就怎么分，例如下面的拆分显然是不合理的：
```
struct student
{
    unsigned int sex : 1;
    unsigned int age : 12;
};
```
这是因为 1+12 = 13，不能再组合成一个基本成员，不能组合成 char、int 或任何类型，这显然是不能"自圆其说"的。

在拆分基本成员变量的情况下，我们要特别注意数据的存放顺序，这还与 CPU 是 Big endian 还是 Little endian 来决定。Little endian 和 Big endian 是 CPU 存放数据的两种不同顺序。对于整型、长整型等数据类型，Big endian 认为第一个字节是最高位字节（按照从低地址到高地址的顺序存放数据的高位字节到低位字节）；而 Little endian 则相反，它认为第一个字节是最低位字节（按照从低地址到高地址的顺序存放数据的低位字节到高位字节）。

我们定义 IP 包头结构体为：
```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
            version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
          ihl:4;
#else
#error    "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __u16     tot_len;
    __u16     id;
    __u16     frag_off;
    __u8    ttl;
    __u8    protocol;
    __u16     check;
    __u32     saddr;
    __u32     daddr;
    /*The options start here. */
};
```
在 Little endian 模式下，iphdr 中定义：
```
    __u8    ihl:4,
            version:4;
```
其存放方式为：

第 1 字节低 4 位 ihl

第 1 字节高 4 位 version （IP 的版本号）
若在 Big endian 模式下还这样定义，则存放方式为：
第 1 字节低 4 位 version （IP 的版本号）
第 1 字节高 4 位 ihl
这与实际的 IP 协议是不匹配的，所以在 Linux 内核源代码中，IP 包头结构体的定义利用了宏：

```
#if defined(__LITTLE_ENDIAN_BITFIELD)
...
#elif defined (__BIG_ENDIAN_BITFIELD)
...
#endif
```

来区分两种不同的情况。
由此我们总结全文的主要观点：
（1）C/C++语言的结构体支持对其中的基本成员变量按位拆分；
（2）拆分的位数应该是合乎逻辑的，应仍然可以组合为基本成员变量；
要特别注意拆分后的数据的存放顺序，这一点要结合具体的 CPU 的结构。

C/C++中的近指令、远指针和巨指针

宋宝华 email:21cnbao@21cn.com sweek
在我们的 C/C++学习生涯中、在我们大脑的印象里，通常只有指针的概念，很少听说指针还有远、近、巨之分的，从没听说过什么近指针、远指针和巨指针。
可以，某年某月的某一天，你突然看到这样的语句：

```
char near *p; /*定义一个字符型"近"指针*/
char far *p; /*定义一个字符型"远"指针*/
char huge *p; /*定义一个字符型"巨"指针*/
```

实在不知道语句中的"near"、"far"、"huge"是从哪里冒出来的，是个什么概念！本文试图对此进行解答，解除许多人的困惑。

这一点首先要从 8086 处理器体系结构和汇编渊源讲起。大家知道，8086 是一个 16 位处理器，它设定了四个段寄存器，专门用来保存段地址：CS（Code Segment）：代码段寄存器；DS（Data Segment）：数据段寄存器；SS（Stack Segment）：堆栈段寄存器；ES（Extra Segment）：附加段寄存器。8086 采用段式访问，访问本段（64K 范围内）的数据或指令时，不需要变更段地址（意味着段地址寄存器不需修改），而访问本段范围以外的数据或指令时，则需要变更段地址（意味着段地址寄存器需要修改）。

因此，在 16 位处理器环境下，如果访问本段内地址的值，用一个 16 位的指针（表示段内偏移）就可以访问到；而要访问本段以外地址的值，则需要用 16 位的段内偏移+16 位的段地址，总共 32 位的指针。

这样，我们就知道了远、近指针的区别：

- 近指针是只能访问本段、只包含本段偏移的、位宽为 16 位的指针；
- 远指针是能访问非本段、包含段偏移和段地址的、位宽为 32 位的指针。

近指针只能对 64k 字节数据段内的地址进行存取，如：

```
char near *p;
p=(char near *)0xffff;
```

远指针是 32 位指针，它表示段地址：偏移地址，远指针可以进行跨段寻址，可以访问整个内存的地址。如定

义远程指针 p 指向 0x1000 段的 0x2 号地址，即 1000:0002，则可写作：

```
char far *p;
p=(char far *)0x10000002;
```

除了远指针和近指针外，还有一个巨指针的概念。

和远指针一样，巨指针也是 32 位的指针，指针也表示为 16 位段：16 位偏移，也可以寻址任何地址。它和远指针的区别在于进行了规格化处理。远指针没有规格化，可能存在两个远指针实际指向同一个物理地址，但是它们的段地址和偏移地址不一样，如 23B0:0004 和 23A1:00F4 都指向同一个物理地址 23604！巨指针通过特定的例程保证：每次操作完成后其偏移量均小于 10h，即只有最低 4 位有数值，其余数值都被进位到段地址上去了，这样就可以避免 Far 指针在 64K 边界时出乎意料的回绕的行为。当然，一次操作必须小于 64K。下面的函数可以将远指针转换为巨指针：

```
void normalize(void far ** p)
{
    *p=(void far *)(((long)*p&0xffff000f)+(((long)*p&0x0000fff00<<12));
}
```

从上面的函数中我们再一次看到了指针之指针的使用，这个函数要修改指针的值，因此必须传给它的指针的指针作为参数。

讲到这里，笔者要强调的是：**近指针、远指针、巨指针是段寻址的 16bit 处理器的产物**（如果处理器是 16 位的，但是不采用段寻址的话，也不存在近指针、远指针、巨指针的概念），当前普通 PC 所使用的 32bit 处理器（80386 以上）一般运行在保护模式下的，指针都是 32 位的，可平滑地址，已经不分远、近指针了。但是在嵌入式系统领域下，8086 的处理器仍然有比较广泛的市场，如 AMD 公司的 AM186ED、AM186ER 等处理器，开发这些系统的程序时，我们还是有必要弄清楚指针的寻址范围。

如果读者还想更透彻地理解本文讲解的内容，不妨再温习一下微机原理、8086 汇编，并参考 C/C++高级编程书籍的相关内容。

从两道经典试题谈 C/C++中联合体（union）的使用

宋宝华 21cnbao sweek@21cn.com

试题一：编写一段程序判断系统中的 CPU 是 Little endian 还是 Big endian 模式？

分析：

作为一个计算机相关专业的人，我们应该在计算机组成中都学习过什么叫 Little endian 和 Big endian。Little endian 和 Big endian 是 CPU 存放数据的两种不同顺序。对于整型、长整型等数据类型，Big endian 认为第一个字节是最高位字节（按照从低地址到高地址的顺序存放数据的高位字节到低位字节）；而 Little endian 则相反，它认为第一个字节是最低位字节（按照从低地址到高地址的顺序存放数据的低位字节到高位字节）。

例如，假设从内存地址 0x0000 开始有以下数据：

0x0000	0x0001	0x0002	0x0003
0x12	0x34	0xab	0xcd

如果我们去读取一个地址为 0x0000 的四个字节变量，若字节序为 big-endian，则读出结果为 0x1234abcd；若字节序位 little-endian，则读出结果为 0xcdab3412。如果我们将 0x1234abcd 写入到以 0x0000 开始的内存中，则 Little endian 和 Big endian 模式的存放结果如下：

地址	0x0000	0x0001	0x0002	0x0003

big-endian	0x12	0x34	0xab	0xcd
little-endian	0xcd	0xab	0x34	0x12

一般来说，x86 系列 CPU 都是 little-endian 的字节序，PowerPC 通常是 Big endian，还有的 CPU 能通过跳线来设置 CPU 工作于 Little endian 还是 Big endian 模式。

解答：

显然，解答这个问题的方法只能是将一个字节（CHAR/BYTE 类型）的数据和一个整型数据存放于同样的内存开始地址，通过读取整型数据，分析 CHAR/BYTE 数据在整型数据的高位还是低位来判断 CPU 工作于 Little endian 还是 Big endian 模式。得出如下的答案：

```
typedef unsigned char BYTE;
int main(int argc, char* argv[])
{
    unsigned int num,*p;
    p = &num;
    num = 0;
  *(BYTE *)p = 0xff;

    if(num == 0xff)
    {
        printf("The endian of cpu is little\n");
    }
    else    //num == 0xff000000
    {
        printf("The endian of cpu is big\n");
    }
    return 0;
}
```

除了上述方法(通过指针类型强制转换并对整型数据首字节赋值，判断该赋值赋给了高位还是低位)外，还有没有更好的办法呢？我们知道，union 的成员本身就被存放在相同的内存空间（共享内存，正是 union 发挥作用、做贡献的去处），因此，我们可以将一个 CHAR/BYTE 数据和一个整型数据同时作为一个 union 的成员，得出如下答案：

```
int checkCPU()
{
  {
    union w
    {
      int a;
      char b;
    } c;
    c.a = 1;
    return (c.b == 1);
  }
}
```

实现同样的功能，我们来看看 Linux 操作系统中相关的源代码是怎么做的：

```
static union { char c[4]; unsigned long l; } endian_test = { { 'l', '?', '?', 'b' } };
```

#define ENDIANNESS ((char)endian_test.l)

Linux 的内核作者们仅仅用一个 union 变量和一个简单的宏定义就实现了一大段代码同样的功能！由以上一段代码我们可以深刻领会到 Linux 源代码的精妙之处！(如果 **ENDIANNESS='l'表示系统为 little endian, 为'b'表示 big endian**)

试题二：假设网络节点 A 和网络节点 B 中的通信协议涉及四类报文，报文格式为"报文类型字段+报文内容的结构体"，四个报文内容的结构体类型分别为 STRUCTTYPE1~ STRUCTTYPE4，请编写程序以最简单的方式组织一个统一的报文数据结构。

分析：

报文的格式为"报文类型+报文内容的结构体"，在真实的通信中，每次只能发四类报文中的一种，我们可以将四类报文的结构体组织为一个 union（共享一段内存，但每次有效的只是一种），然后和报文类型字段统一组织成一个报文数据结构。

解答：

根据上述分析，我们很自然地得出如下答案：

```
typedef unsigned char BYTE;

//报文内容联合体
typedef union tagPacketContent
{
    STRUCTTYPE1 pkt1;
    STRUCTTYPE2 pkt2;
    STRUCTTYPE3 pkt1;
    STRUCTTYPE4 pkt2;
}PacketContent;

//统一的报文数据结构
typedef struct tagPacket
{
    BYTE pktType;
    PacketContent pktContent;
}Packet;
```

总结

在 C/C++程序的编写中，当多个基本数据类型或复合数据结构要占用同一片内存时，我们要使用联合体（试题一是这样的例证）；当多种类型，多个对象，多个事物只取其一时（我们姑且通俗地称其为"n 选 1"），我们也可以使用联合体来发挥其长处（试题二是这样的例证）。

基于 ARM 的嵌入式 Linux 移植真实体验

基于 ARM 的嵌入式 Linux 移植真实体验（1）——基本概念

宋宝华 21cnbao@21cn.com 出处：dev.yesky.com

1.引言

ARM 是 Advanced RISC Machines（高级精简指令系统处理器）的缩写，是 ARM 公司提供的一种微处理器知识产权（IP）核。

ARM 的应用已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场。基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75％以上的市场份额。揭开你的手机、MP3、PDA，嘿嘿，里面多半藏着一个基于 ARM 的微处理器！

ARM 内核的数个系列（ARM7、ARM9、ARM9E、ARM10E、SecurCore、Xscale、StrongARM），各自满足不同应用领域的需求，无孔不入的渗入嵌入式系统各个角落的应用。这是一个 ARM 的时代！

下面的图片显示了 ARM 的随处可见：

有人的地方就有江湖（《武林外传》），有嵌入式系统的地方就有 ARM。

构建一个复杂的嵌入式系统，仅有硬件是不够的，我们还需要进行操作系统的移植。我们通常在 ARM 平台上构建 Windows CE、Linux、Palm OS 等操作系统，其中 Linux 具有开放源代码的优点。

下图显示了基于 ARM 嵌入式系统中软件与硬件的关系：

日前，笔者作为某嵌入式 ARM（硬件）/Linux（软件）系统的项目负责人，带领项目组成员进行了下述

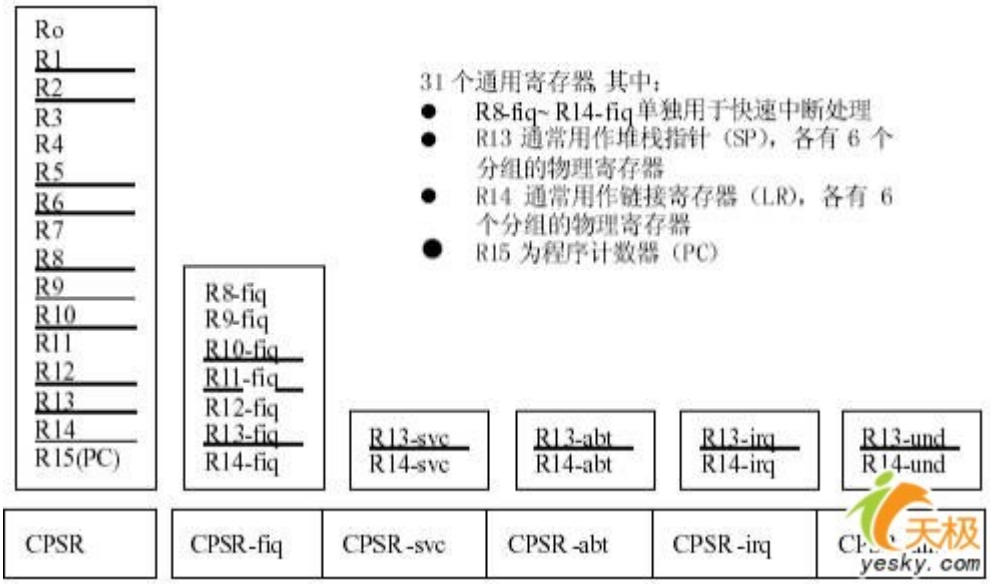
工作：
(1)基于 ARM920T 内核 S3C2410A CPU 的电路板设计；
(2)ARM 处理下底层软件平台搭建：
a.Bootloader 的移植；
b.嵌入式 Linux 操作系统内核的移植；
c.嵌入式 Linux 操作系统根文件系统的创建；
d.电路板上外设 Linux 驱动程序的编写。
本文将真实地再现本项目开发过程中作者的心得，以便与广大读者共勉。第一章将简单地介绍本 ARM 开发板的硬件设计，第二章分析 Bootloader 的移植方法，第三章叙述嵌入式 Linux 的移植及文件系统的构建方法，第四章讲解外设的驱动程序设计，第五章给出一个已构建好的软硬件平台上应用开发的实例。
如果您有良好的嵌入式系统开发基础，您将非常容易领会本文讲解地内容。即便是您从来没有嵌入式系统的开发经历，本文也力求让您读起来不觉得生涩。您可以通过如下 email 与作者联系：21cnbao@21cn.com。

2.ARM 体系结构

作为一种 RISC 体系结构的微处理器，ARM 微处理器具有 RISC 体系结构的典型特征。还具有如下增强特点：
(1)在每条数据处理指令当中，都控制算术逻辑单元(ALU)和移位器，以使 ALU 和移位器获得最大的利用率；
(2)自动递增和自动递减的寻址模式，以优化程序中的循环；
(3)同时 Load 和 Store 多条指令，以增加数据吞吐量；
(4)所有指令都条件执行，以增大执行吞吐量。
ARM 体系结构的字长为 32 位，它们都支持 Byte(8 位)、Halfword(16 位)和 Word(32 位)3 种数据类型。ARM 处理器支持 7 种处理器模式，如下表：

处理器模式	描述
User	普通程序执行的模式
FRQ	用于高速数据传输或者通道处理
IRQ	用于通用中断处理
Supervisor	操作系统的保护模式
Abort	用于实现虚存或者存储保护
Undefined	支持软件模拟或者硬件协处理
System	运行特权操作系统任务

大部分应用程序都在 User 模式下运行。当处理器处于 User 模式下时，执行的程序无法访问一些被保护的系统资源，也不能改变模式，否则就会导致一次异常。对系统资源的使用由操作系统来控制。User 模式之外的其它几种模式也称为特权模式，它们可以完全访问系统资源，可以自由地改变模式。其中的 FIQ、IRQ、supervisor、Abort 和 undefined 5 种模式也被称为异常模式。在处理特定的异常时，系统进入这几种模式。这 5 种异常模式都有各自的额外的寄存器，用于避免在发生异常的时候与用户模式下的程序发生冲突。
还有一种模式是 system 模式，任何异常都不会导致进入这一模式，而且它使用的寄存器和 User 模式下基本相同。它是一种特权模式，用于有访问系统资源请求而又需要避免使用额外的寄存器的操作系统任务。
程序员可见的 ARM 寄存器共有 37 个：31 个通用寄存器以及 6 个针对 ARM 处理器的不同工作模式所设立的专用状态寄存器，如下图：



ARM9 采用 5 级流水线操作：指令预取、译码、执行、数据缓冲、写回。ARM9 设置了 16 个字的数据缓冲和 4 个字的地址缓冲。这 5 级流水已被很多的 RISC 处理器所采用，被看作 RISC 结构的“经典”。

3. 硬件设计

3.1 S3C2410A 微控制器

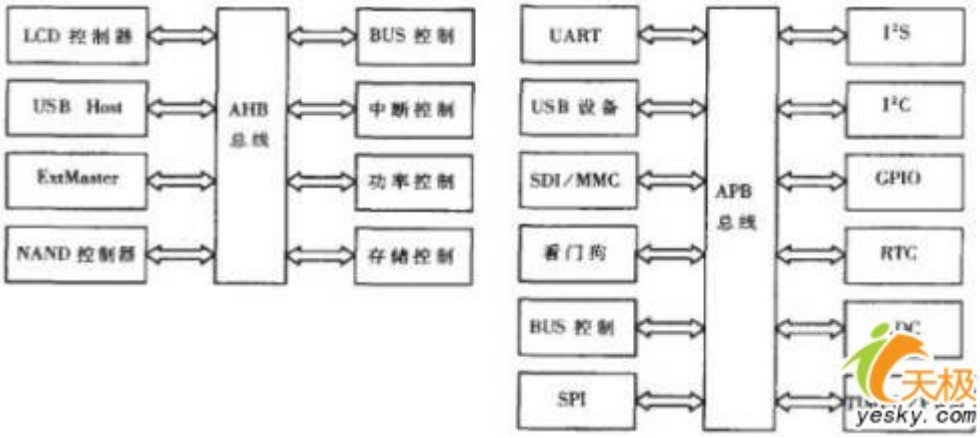
电路板上的 ARM 微控制器 S3C2410A 采用了 ARM920T 核，它由 ARM9TDMI、存储管理单元 MMU 和高速缓存三部分组成。其中，MMU 可以管理虚拟内存，高速缓存由独立的 16KB 地址和 16KB 数据高速 Cache 组成。ARM920T 有两个内部协处理器：CP14 和 CP15。CP14 用于调试控制，CP15 用于存储系统控制以及测试控制。

S3C2410A 集成了大量的内部电路和外围接口：

- LCD 控制器(支持 STN 和 TFT 带有触摸屏的液晶显示屏)
- SDRAM 控制器
- 3 个通道的 UART
- 4 个通道的 DMA
- 4 个具有 PWM 功能的计时器和一个内部时钟
- 8 通道的 10 位 ADC
- 触摸屏接口
- I^2C 总线接口
- I^2S 总线接口
- 两个 USB 主机接口
- 一个 USB 设备接口
- 两个 SPI 接口
- SD 接口
- MMC 卡接口

S3C2410A 集成了一个具有日历功能的 RTC 和具有 PLL(MPLL 和 UPLL)的芯片时钟发生器。MPLL 产生主时钟，能够使处理器工作频率最高达到 203MHz。这个工作频率能够使处理器轻松运行 WinCE、Linux 等操作系统以及进行较为复杂的信息处理。UPLL 则产生实现 USB 模块的时钟。

下图显示了 S3C2410A 的集成资源和外围接口：



我们需要对上图中的 AHB 总线和 APB 总线的概念进行一番解释。ARM 核开发的目的，是使其作为复杂片上系统的一个处理单元来应用的，所以还必须提供一个 ARM 与其它片上宏单元通信的接口。为了减少不必要的设计资源的浪费，ARM 公司定义了 AMBA(Advanced Microcontroller Bus Architecture)总线规范，它是一组针对基于 ARM 核的、片上系统之间通信而设计的、标准的、开放协议。

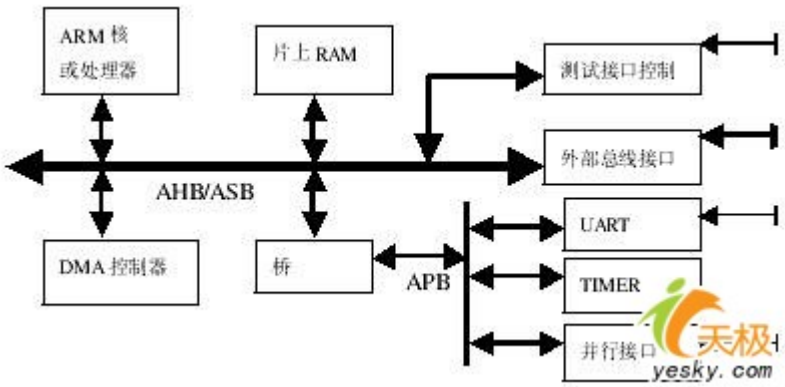
在 AMBA 总线规范中，定义了 3 种总线：

(1)AHB—Advanced High Performace Bus，用于高性能系统模块的连接，支持突发模式数据传输和事务分割；

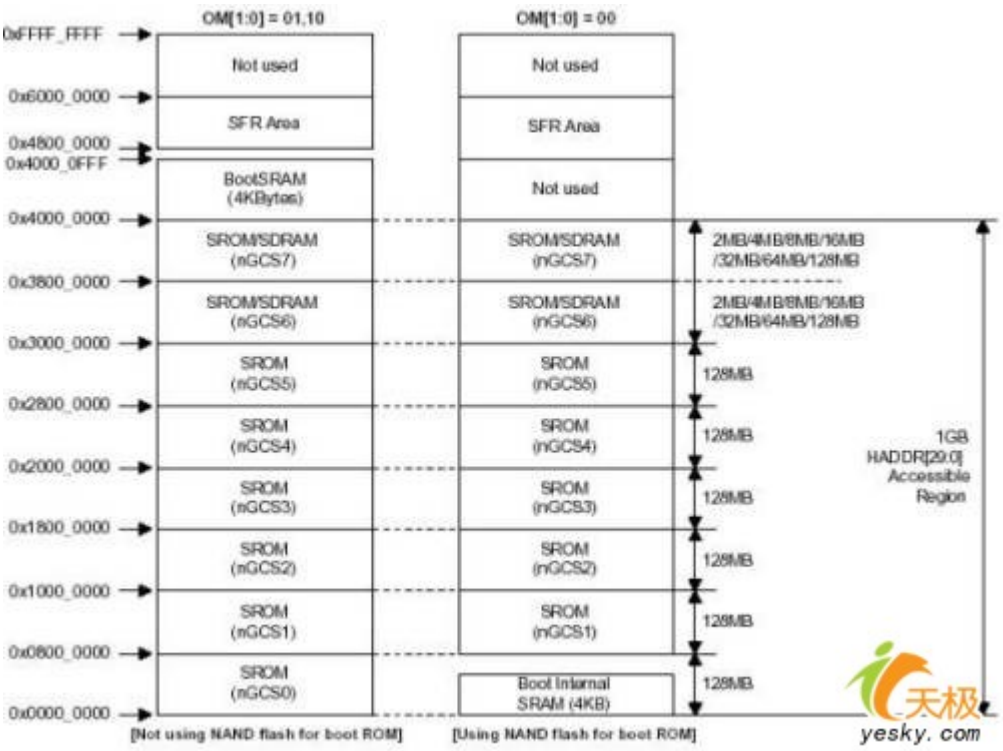
(2)ASB—Advanced System Bus，也用于高性能系统模块的连接，支持突发模式数据传输，这是较老的系统总线格式，后来由 AHB 总线替代；

(3)APB—Advanced PeriPheral Bus，用于较低性能外设的简单连接，一般是接在 AHB 或 ASB 系统总线上的第二级总线。

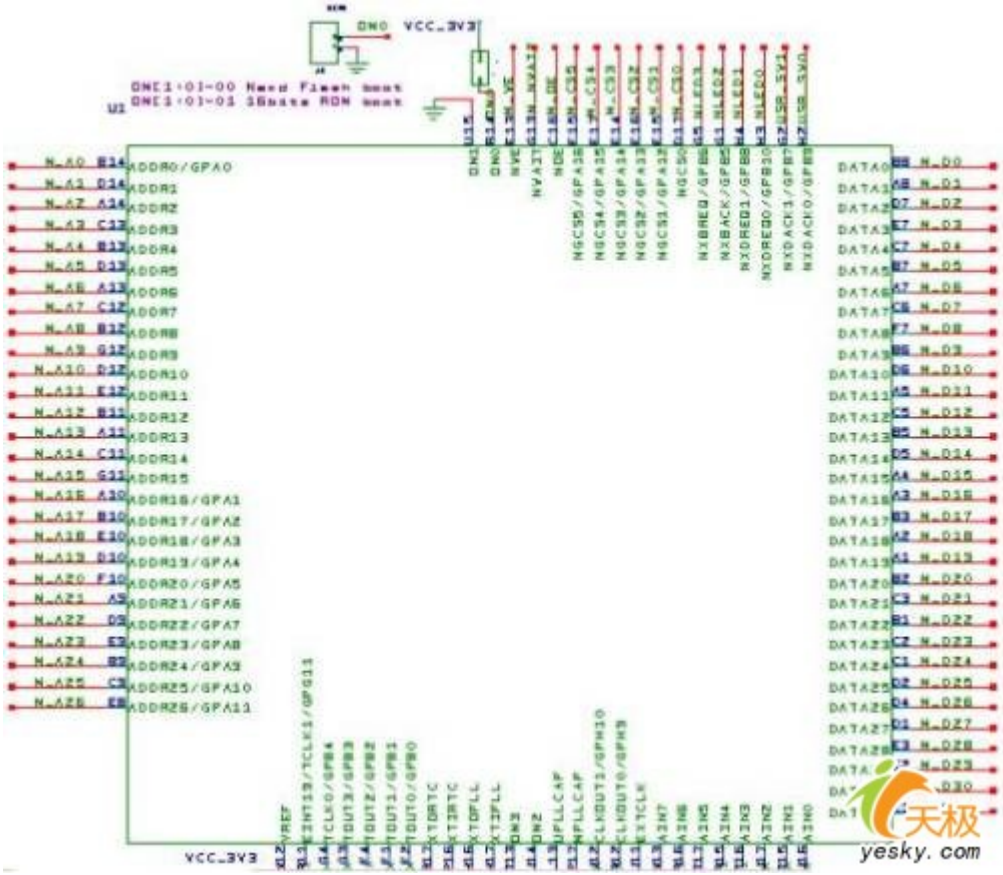
典型的 AMBA 总线系统如下图：



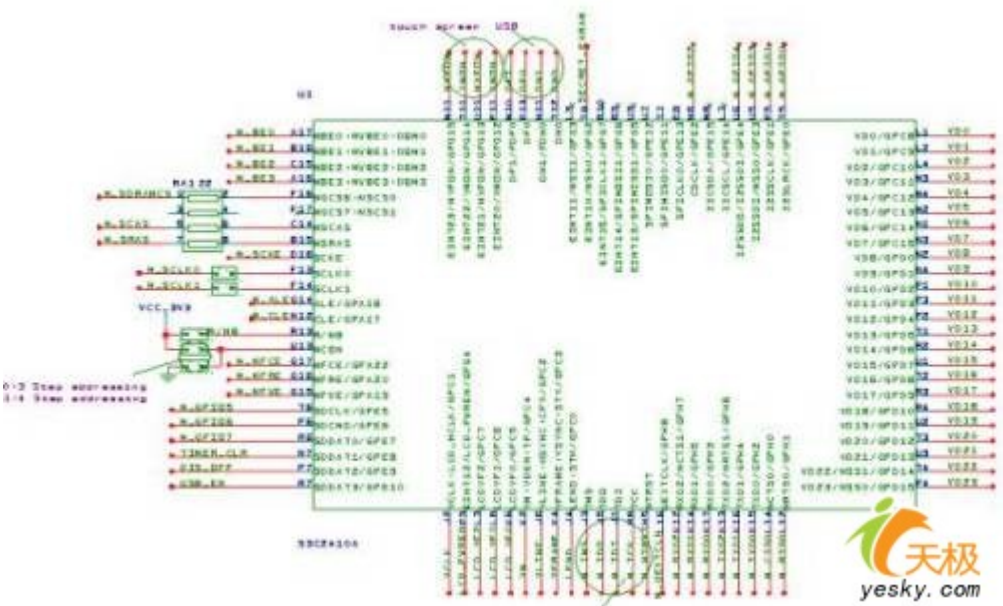
S3C2410A 将系统的存储空间分成 8 个 bank，每个 bank 的大小是 128M 字节，共 1G 字节。Bank0 到 bank5 的开始地址是固定的，用于 ROM 或 SRAM。bank6 和 bank7 可用于 ROM、SRAM 或 SDRAM。所有内存块的访问周期都可编程，外部 Wait 也能扩展访问周期。下图给出了 S3C2410A 的内存组织：



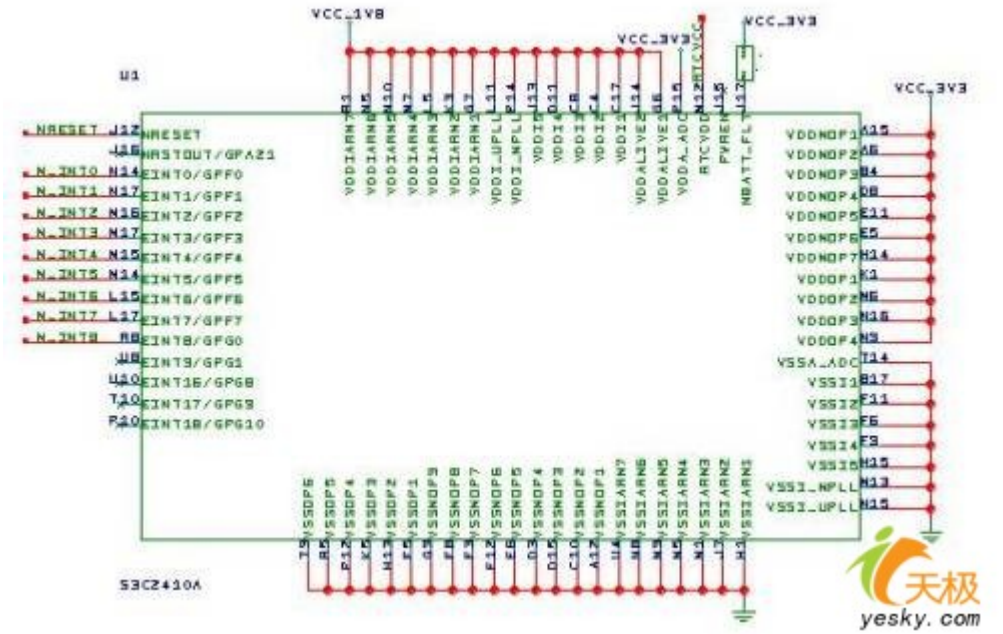
下图给出了 S3C2410A 的数据总线、地址总线和片选电路：



SDRAM 控制信号、集成 USB 接口电路：



内核与存储单元供电电路(S3C2410A 对于片内的各个部件采用了独立的电源供给,内核采用 1.8V 供电,存储单元采用 3.3V 独立供电):

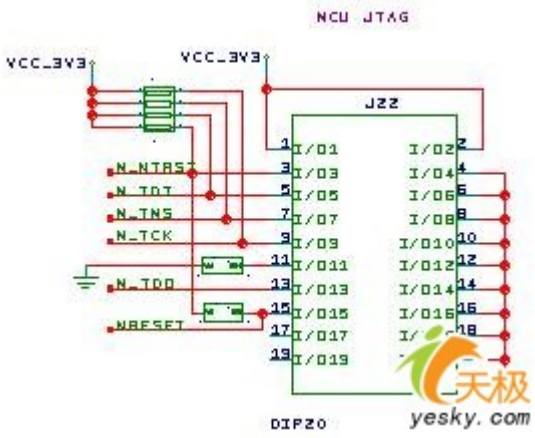


JTAG 标准通过边界扫描技术提供了对电路板上每一元件的功能、互联及相互间影响进行测试的方法,极大地方便了系统电路的调试。

测试接入端口 TAP 的管脚定义如下:

- TCK:专用的逻辑测试时钟,时钟上升沿按串行方式对测试指令、数据及控制信号进行移位操作,下降沿用于对输出信号移位操作;
- TMS:测试模式选择,在 TCK 上升沿有效的逻辑测试控制信号;
- TDI:测试数据输入,用于接收测试数据与测试指令;
- TDO:测试数据输出,用于测试数据的输出。

S3C2410A 调试用 JTAG 接口电路:



3.2 SDRAM 存储器

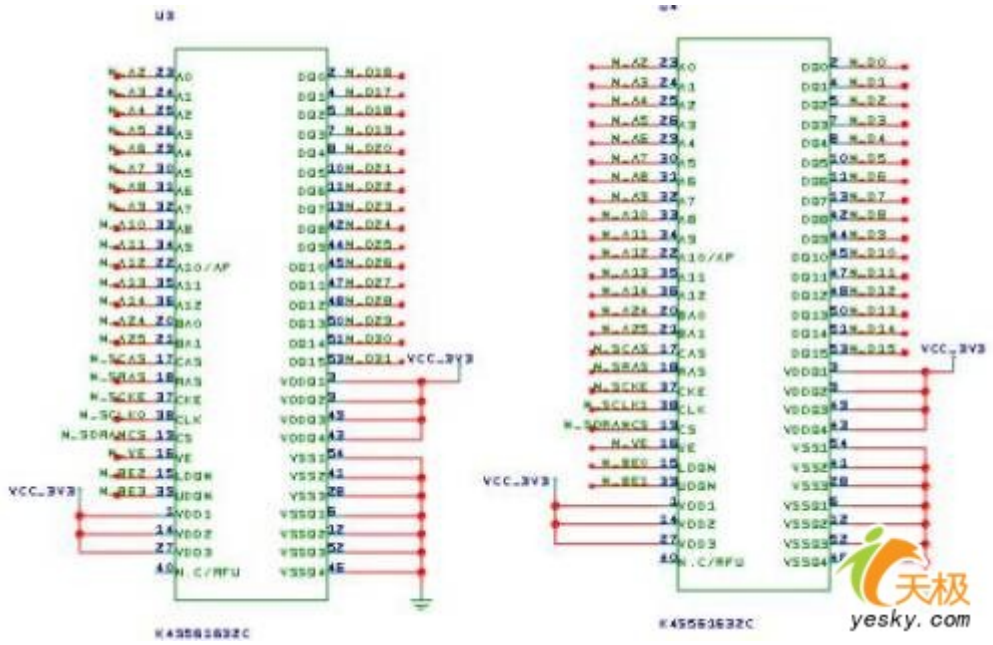
SDRAM 被用来存放操作系统（从 FLASH 解压缩拷入）以及存放各类动态数据，采用 SAMSUNG 公司的 K4S561632，它是 4Mx16bitx4bank 的同步 DRAM，容量为 32MB。用 2 片 K4S561632 实现位扩展，使数据总线宽度达到 32bit，总容量达到 64MB，将其地址空间映射在 S3C2410A 的 bank6。

SDRAM 所有的输入和输出都与系统时钟 CL K 上升沿同步，由输入信号 RA S、CA S、WE 组合产生 SDRAM 控制命令，其基本的控制命令如下：

命令	RAS	CAS	WE	编码
空操作(NOP)	H	H	H	000
激活操作(ACT)	L	H	H	001
读操作(RD)	H	L	H	010
写操作(WR)	H	L	L	011
突发中断(BT)	H	H	L	100
预充电(PCH)	L	H	L	01
自动刷新(ARF)	L	L	H	
模式寄存器设置(MRS)	L	L	L	

SDRAM 在具体操作之前首先必须通过 MRS 命令设置模式寄存器，以便确定 SDRAM 的列地址延迟、突发类型、突发长度等工作模式；再通过 ACT 命令激活对应地址的组，同时输入行地址；然后通过 RD 或 WR 命令输入列地址，将相应数据读出或写入对应的地址；操作完成后用 PCH 命令或 BT 命令中止读或写操作。在没有操作的时候，每隔一段时间必须用 ARF 命令刷新数据，防止数据丢失。

下图给出了 SDRAM 的连接电路：



3.3 FLASH 存储器

NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。

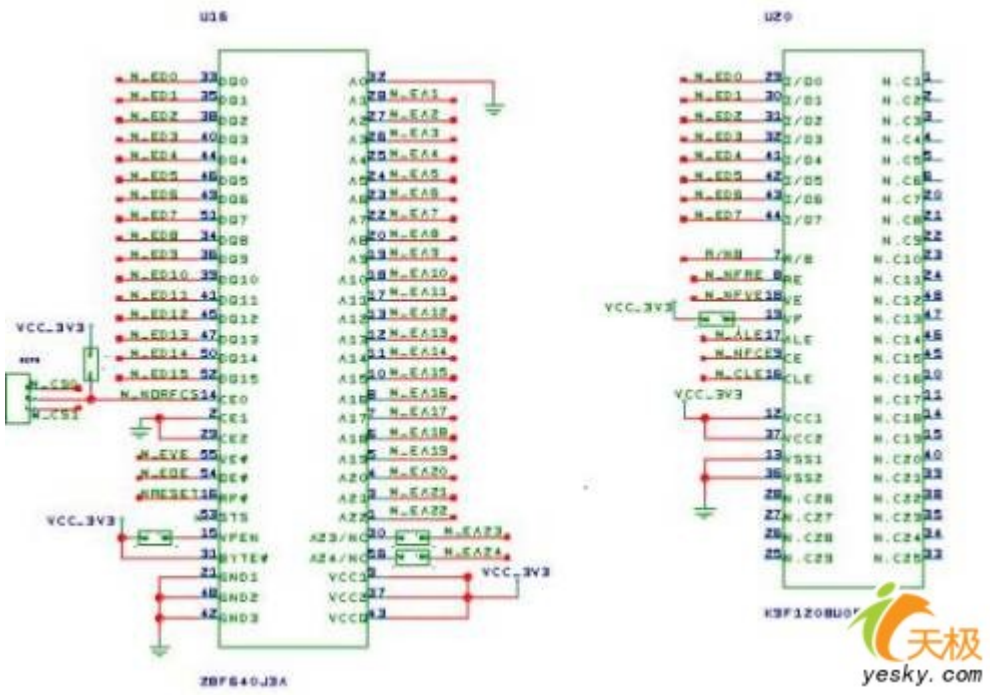
NOR 的特点是芯片内执行(XIP，Execute In Place)，即应用程序可直接在 Flash 闪存内运行，不必把代码读到系统 RAM 中。NOR 的传输效率很高，在 1~4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 Flash 的管理和需要特殊的系统接口，S3C2410A 内嵌了 NAND FLASH 控制器。

S3C2410A 支持从 GCS0 上的 NOR FLASH 启动（16 位或 32 位）或从 NAND FLASH 启动，需要通过 OM0 和 OM1 上电时的上下拉来设置：

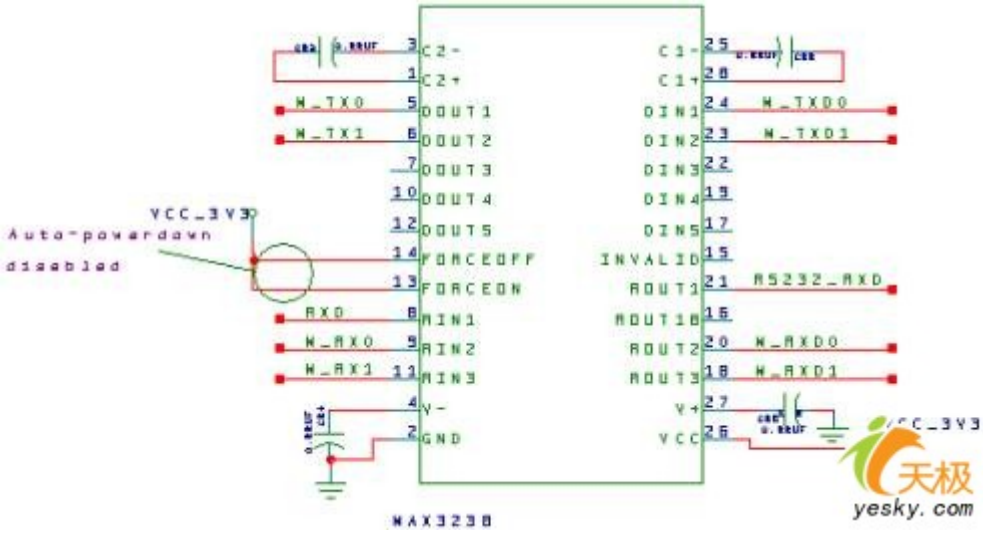
OM1	OM0	S10-3	S10-2	Boot 模式
0	0	ON	ON	由 NAND flash 启动
0	1	ON	OFF	由 16 位 NOR flash 启动
1	0	OFF	ON	由 32 位 NOR flash 启动
1	1	OFF	OFF	CPU 测试模式

在系统中分别采用了一片 NOR FLASH(28F640)和 NAND FLASH(K9S1208)，电路如下图：



3.4 串口

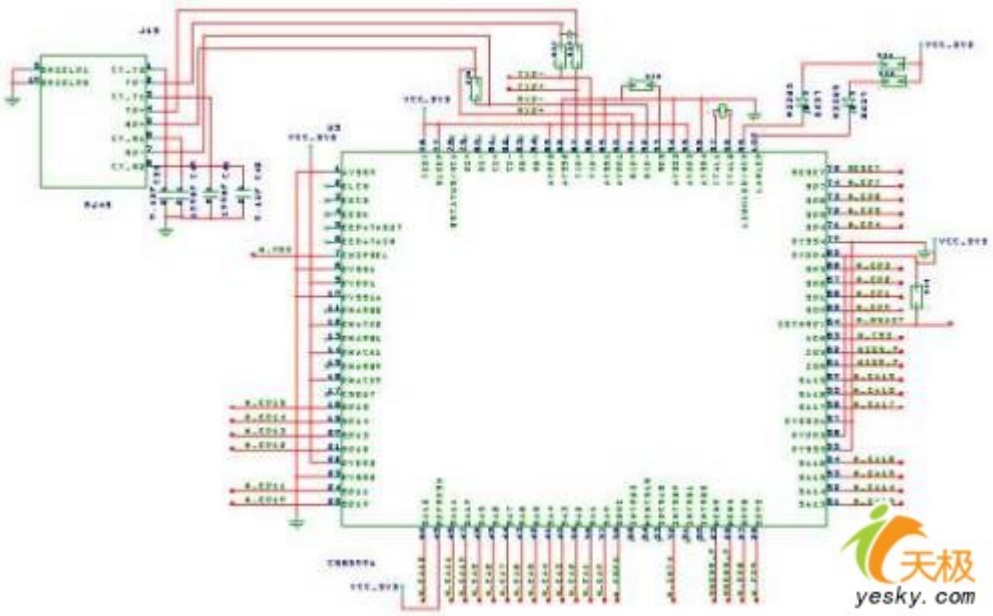
S3C2410 内部集成了 UART 控制器,实现了并串转换。外部还需提供 CMOS/TTL 电平与 RS232 之间的转换：



3.5 以太网

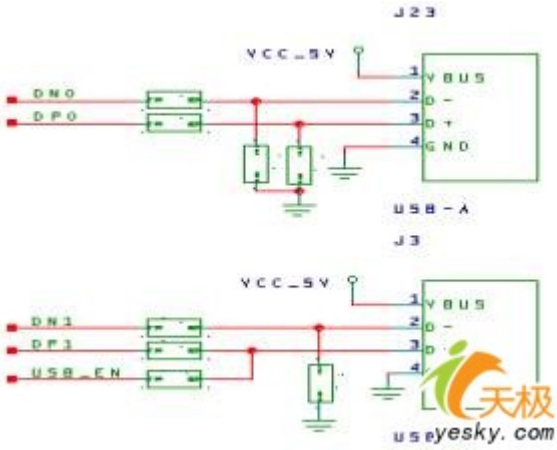
以太网控制芯片采用 CIRRUS LOGIC 公司生产的 CS8900A，其突出特点是使用灵活，其物理层接口、数据传输模式和工作模式等都能根据需要而动态调整，通过内部寄存器的设置来适应不同的应用环境。它符合 IEEE803.3 以太网标准，带有传送、接收低通滤波的 10Base-T 连接端口，支持 10Base2，10Base5 和 10Base-F 的 AUI 接口，并能自动生成报头，自动进行 CRC 检验，在冲突后自动重发。

CS8900A 支持的传输模式有 I/O 和 Memory 模式。当 CS8900A 有硬件复位或软件复位时,它将默认成为 8 位工作模式。因此，要使 CS8900A 工作于 16 位模式，系统必须在访问之前提供给总线高位使能管脚(/SBHE)一个由高到低、再由低到高变化的电平。



3.6 USB 接口

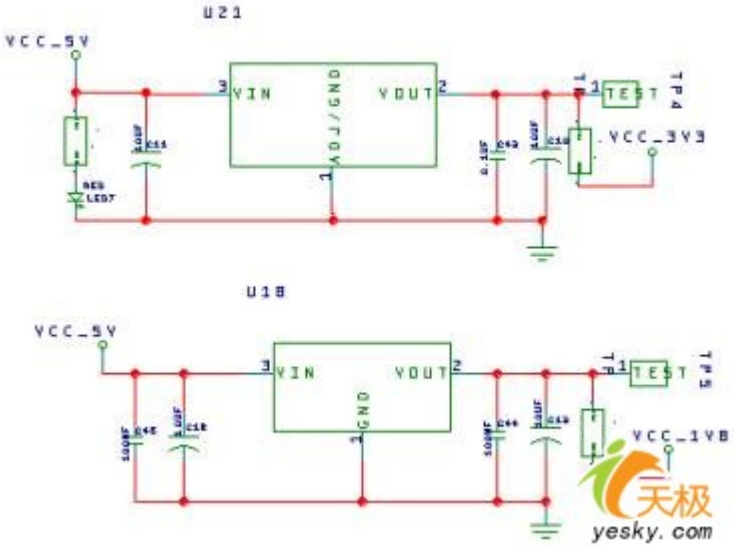
USB 系统由 USB 主机（USB Host）、USB 集线器（USB Hub）和 USB 设备（USB Device）组成。USB 和主机系统的接口称作主机控制器（Host Controller），它是由硬件和软件结合实现的。根集线器是综合于主机系统内部的，用以提供 USB 的连接点。USB 的设备包括集线器（Hub）和功能器件（Function）。S3C2410A 集成了 USB host 和 USB device，外部连接电路如下图：



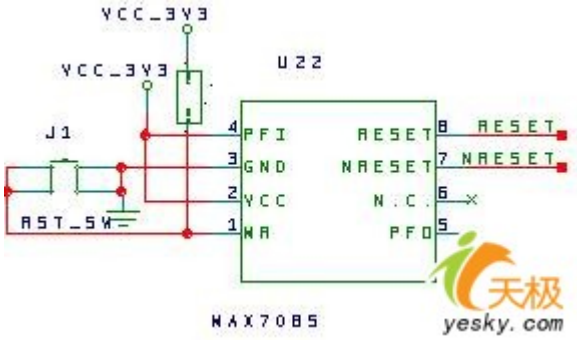
3.7 电源

LDO(Low Dropout)属于 DC/DC 变换器中的降压变换器，它具有低成本、低噪声、低功耗等突出优点，另外它所需要的外围器件也很少，通常只有 1~2 个旁路电容。

在电路板上我们分别用两个 LDO 来实现 5V 向 3.3V（存储接口电平）和 1.8V（ARM 内核电平）的转换。

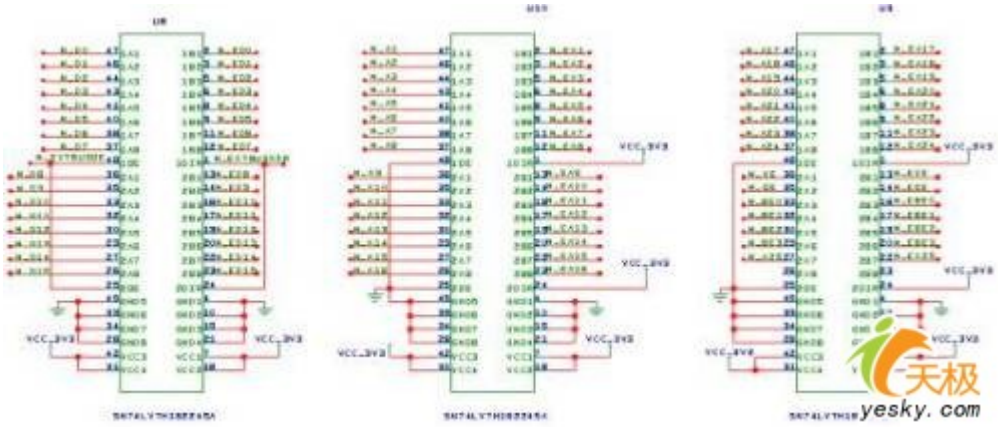


up 监控电路采用 MAX708 芯片，提供上电、掉电以及降压情况下的复位输出及低电平有效的人工复位输出：

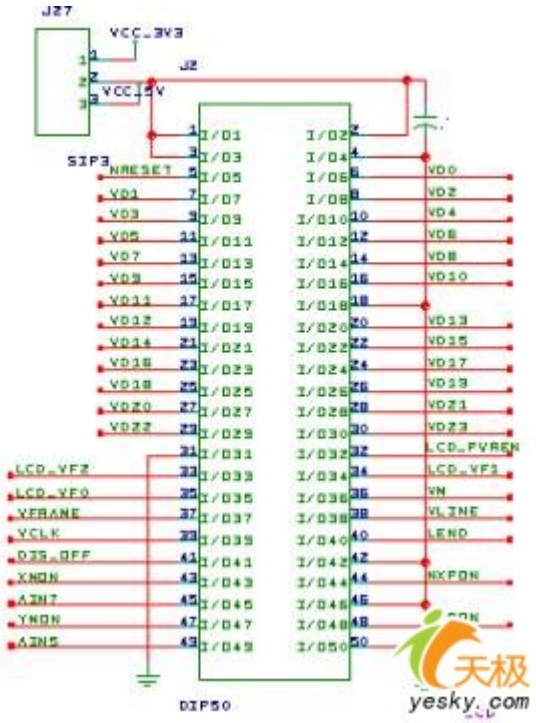


3.8 其它

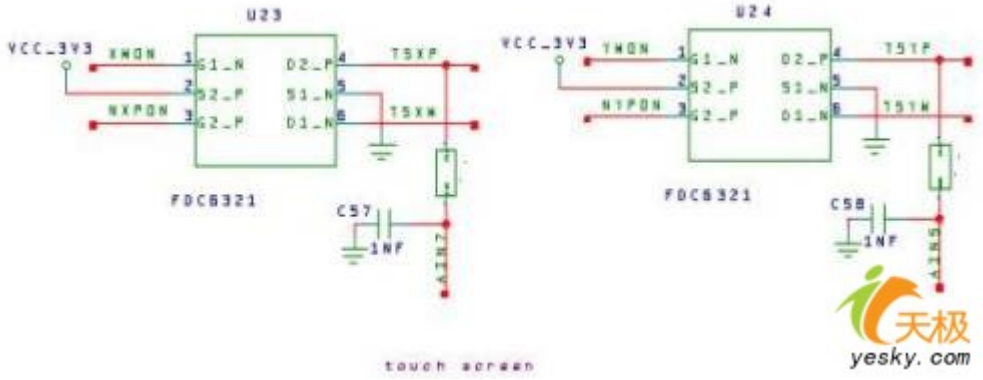
SN74LVTH62245A 提供总线驱动和缓冲能力：



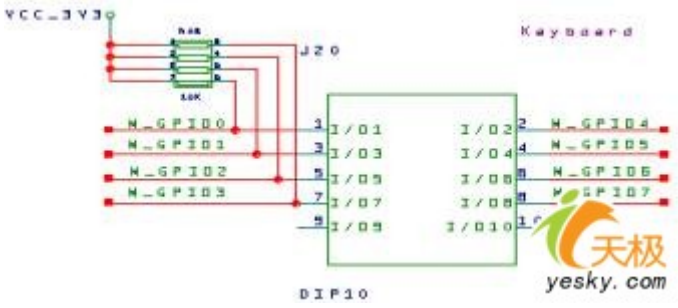
S3C2410A 集成 LCD 液晶显示器控制电路，外部引出接口：



触摸屏有电阻式、电容式等，其本质是一种将手指在屏幕上的触点位置转化为电信号的传感器。手指触到屏幕，引起触点位置电阻或电容的变化，再通过检测这一电性变化，从而获得手指的坐标位置。通过 S3C2410A 集成的 AD 功能，完成电信号向屏幕坐标的转化，触摸屏接口如下：

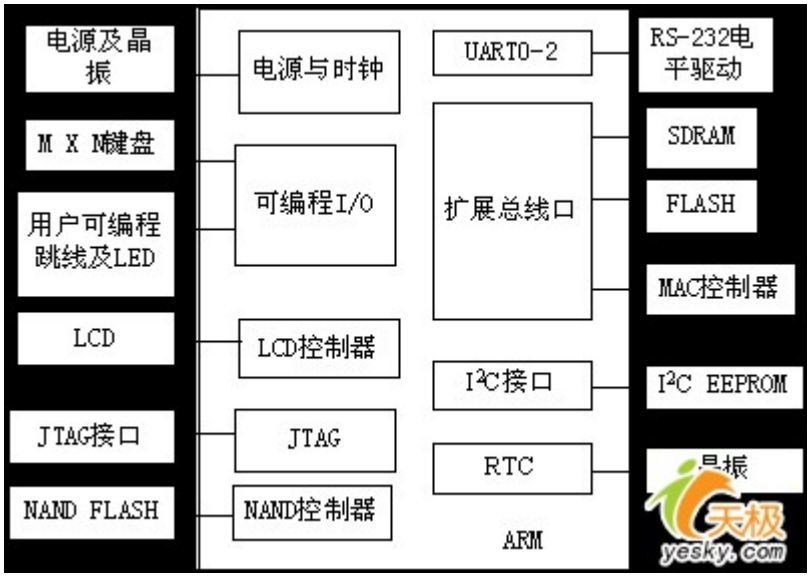


键盘则直接利用 CPU 的可编程 I/O 口，若连接 mxn 键盘，则需要 m+n 个可编程 I/O 口，由软件实现键盘扫描，识别按键：



3.9 整体架构

下图呈现了 ARM 处理器及外围电路的整体设计框架：



4.小结

本章讲解了基于 S3C2410A ARM 处理器电路板硬件设计的基本组成，为后续各章提供了总体性的准备工作。

基于 ARM 的嵌入式 Linux 移植真实体验（2）——BootLoader

宋宝华 21cnbao@21cn.com 出处:dev.yesky.com

BootLoader 指系统启动后，在操作系统内核运行之前运行的一段小程序。通过 BootLoader，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。通常，BootLoader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 BootLoader 几乎是不可能的。尽管如此，我们仍然可以对 BootLoader 归纳出一些通用的概念来，以指导用户特定的 BootLoader 设计与实现。

BootLoader 的实现依赖于 CPU 的体系结构，因此大多数 BootLoader 都分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。而 stage2 则通常用 C 语言来实现，这样可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。

BootLoader 的 stage1 通常包括以下步骤：

Ø 硬件设备初始化；

Ø 为加载 Boot Loader 的 stage2 准备 RAM 空间；

Ø 拷贝 Boot Loader 的 stage2 到 RAM 空间中；

Ø 设置好堆栈；

Ø 跳转到 stage2 的 C 入口点。

Boot Loader 的 stage2 通常包括以下步骤：

Ø 初始化本阶段要使用到的硬件设备；

Ø 检测系统内存映射(memory map)；

Ø 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中；

Ø 为内核设置启动参数；

Ø 调用内核。

本系统中的 BootLoader 参照韩国 mizi 公司的 vivi 进行修改。

1.开发环境

我们购买了武汉创维特信息技术有限公司开发的具有自主知识产权的应用于嵌入式软件开发的集成软、硬件开发平台 ADT（ARM Development Tools）它为基于 ARM 核的嵌入式应用提供了一整套完备的开发方案，包括程序编辑、工程管理和设置、程序编译、程序调试等。

ADT 嵌入式开发环境由 ADT Emulator for ARM 和 ADT IDE for ARM 组成。ADT Emulator for ARM 通过 JTAG 实现主机和目标机之间的调试支持功能。它无需目标存储器，不占用目标系统的任何端口资源。目标程序直接在目标板上运行，通过 ARM 芯片的 JTAG 边界扫描口进行调试，属于完全非插入式调试，其仿真效果接近真实系统。

ADT IDE for ARM 为用户提供高效明晰的图形化嵌入式应用软件开发环境，包括一整套完备的面向嵌入式系统的开发和调试工具：源码编辑器、工程管理器、工程编译器（编译器、汇编器和连接器）、集成调试环境、ADT Emulator for ARM 调试接口等。其界面同 Microsoft Visual Studio 环境相似，用户可以在 ADT IDE for ARM 集成开发环境中创建工程、打开工程，建立、打开和编辑文件，编译、连接、设置、运行、调试嵌入式应用程序。

ADT 嵌入式软件开发环境采用主机－目标机交叉开发模型。ADT IDE for ARM 运行于主机端，而 ADT Emulator for ARM 实现 ADT IDE for ARM 与目标机之间的连接。开发时，首先由 ADT IDE for ARM 编译连接生成目标代码，然后建立与 ADT Emulator for ARM 之间的调试通道，调试通道建立成功后，就可以在 ADT IDE for ARM 中通过 ADT Emulator for ARM 控制目标板实现目标程序的调试，包括将目标代码下载到目标机中，控制程序运行，调试信息观察等等。

2.ARM 汇编

ARM 本身属于 RISC 指令系统，指令条数就很少，而其编程又以 C 等高级语言为主，我们仅需要在 Bootloader 的第一阶段用到少量汇编指令：

（1）+-运算

```
ADD     r0,   r1,   r2
```

—— r0 := r1 + r2
SUB r0, r1, r2
—— r0 := r1 - r2
其中的第二个操作数可以是一个立即数：
ADD r3, r3, #1
—— r3 := r3 + 1
第二个操作数还可以是位移操作后的结果：
ADD r3, r2, r1, LSL #3
—— r3 := r2 + 8.r1
（2）位运算
AND r0, r1, r2
—— r0 := r1 and r2
ORR r0, r1, r2
—— r0 := r1 or r2
EOR r0, r1, r2
—— r0 := r1 xor r2
BIC r0, r1, r2
—— r0 := r1 and not r2
（3）寄存器搬移
MOV r0, r2
—— r0 := r2
MVN r0, r2
—— r0 := not r2
（4）比较
CMP r1, r2
—— set cc on r1 - r2
CMN r1, r2
—— set cc on r1 + r2
TST r1, r2
—— set cc on r1 and r2
TEQ r1, r2
—— set cc on r1 or r2
这些指令影响 CPSR 寄存器中的 (N, Z, C, V) 位
（5）内存操作
LDR r0, [r1]
—— r0 := mem [r1]
STR r0, [r1]
— mem [r1] := r0
LDR r0, [r1, #4]
—— r0 := mem [r1+4]
LDR r0, [r1, #4] !
—— r0 := mem [r1+4] r1 := r1 + 4
LDR r0, [r1], #4
—— r0 := mem [r1] r1 := r1 +4

```
LDRB r0 , [r1]
——  r0 := mem8 [r1]
LDMIA r1, {r0, r2, r5}
——  r0 := mem [r1]    r2 := mem [r1+4]    r5 := mem [r1+8]
```

{..} 可以包括 r0~r15 中的所有寄存器，若包括 r15 (PC)将导致程序的跳转。

（6）控制流

例 1:

```
MOV r0, #0 ; initialize counter
LOOP:
   ADD r0, r0, #1 ; increment counter
   CMP r0, #10 ; compare with limit
   BNE LOOP ; repeat if not equal
```

例 2:

```
CMP r0, #5
ADDNE r1, r1, r0
SUBNE r1, r1, r2
——
if (r0 != 5) {
r1 := r1 + r0 - r2
}
```

3.BootLoader 第一阶段

3.1 硬件设备初始化

基本的硬件初始化工作包括：

Ø 屏蔽所有的中断；

Ø 设置 CPU 的速度和时钟频率；

Ø RAM 初始化；

Ø 初始化 LED

ARM 的中断向量表设置在 0 地址开始的 8 个字空间中，如下表：

每当其中的某个异常发生后即将 PC 值置到相应的中断向量处，每个中断向量处放置一个跳转指令到相应的中断服务程序去进行处理，中断向量表的程序如下：

```
@ 0x00: Reset
       b     Reset
@ 0x04: Undefined instruction exception
UndefEntryPoint:
       b     HandleUndef
@ 0x08: Software interrupt exception
SWIEntryPoint:
       b     HandleSWI
@ 0x0c: Prefetch Abort (Instruction Fetch Memory Abort)
PrefetchAbortEnteryPoint:
       b     HandlePrefetchAbort
@ 0x10: Data Access Memory Abort
DataAbortEntryPoint:
       b     HandleDataAbort
```

```
@ 0x14: Not used
NotUsedEntryPoint:
       b     HandleNotUsed
@ 0x18: IRQ(Interrupt Request) exception
IRQEntryPoint:
       b     HandleIRQ
@ 0x1c: FIQ(Fast Interrupt Request) exception
FIQEntryPoint:
       b     HandleFIQ
```

复位时关闭看门狗定时器、屏蔽所有中断：

```
Reset:
       @ disable watch dog timer
       mov r1, #0x53000000
       mov r2, #0x0
       str   r2, [r1]
       @ disable all interrupts
       mov r1, #INT_CTL_BASE
       mov r2, #0xffffffff
       str   r2, [r1, #oINTMSK]
       ldr   r2, =0x7ff
       str   r2, [r1, #oINTSUBMSK]
```

设置系统时钟：

```
    @init clk
    @ 1:2:4
    mov r1, #CLK_CTL_BASE
    mov r2, #0x3
    str   r2, [r1, #oCLKDIVN]
    mrc p15, 0, r1, c1, c0, 0            @ read ctrl register
    orr   r1, r1, #0xc0000000            @ Asynchronous
    mcr p15, 0, r1, c1, c0, 0            @ write ctrl register
    @ now, CPU clock is 200 Mhz
    mov r1, #CLK_CTL_BASE
    ldr   r2, mpll_200mhz
    str   r2, [r1, #oMPLLCON]
```

点亮所有的用户 LED：

```
       @ All LED on
       mov r1, #GPIO_CTL_BASE
       add  r1, r1, #oGPIO_F
       ldr   r2,=0x55aa
       str   r2, [r1, #oGPIO_CON]
       mov r2, #0xff
       str   r2, [r1, #oGPIO_UP]
       mov r2, #0x00
       str   r2, [r1, #oGPIO_DAT]
```

设置（初始化）内存映射：

```
ENTRY(memsetup)
       @ initialise the static memory
       @ set memory control registers
       mov r1, #MEM_CTL_BASE
       adrl  r2, mem_cfg_val
       add  r3, r1, #52
1:     ldr   r4, [r2], #4
       str   r4, [r1], #4
       cmp r1, r3
     bne  1b
       mov pc, lr
```

设置（初始化）UART：

```
       @ set GPIO for UART
      mov r1, #GPIO_CTL_BASE
       add  r1, r1, #oGPIO_H
       ldr   r2, gpio_con_uart
       str   r2, [r1, #oGPIO_CON]
       ldr   r2, gpio_up_uart
       str   r2, [r1, #oGPIO_UP]
       bl     InitUART
@ Initialize UART
@
@ r0 = number of UART port
InitUART:
       ldr   r1, SerBase
       mov r2, #0x0
       str   r2, [r1, #oUFCON]
       str   r2, [r1, #oUMCON]
       mov r2, #0x3
       str   r2, [r1, #oULCON]
       ldr   r2, =0x245
       str   r2, [r1, #oUCON]
#define UART_BRD ((50000000 / (UART_BAUD_RATE * 16)) - 1)
       mov r2, #UART_BR
       str   r2, [r1, #oUBRDIV]
       mov r3, #100
       mov r2, #0x0
1:     sub  r3, r3, #0x1
       tst    r2, r3
       bne  1b
#if 0
       mov r2, #'U'
       str   r2, [r1, #oUTXHL]
```

```
1:      ldr   r3, [r1, #oUTRSTAT]
        and  r3, r3, #UTRSTAT_TX_EMPTY
        tst    r3, #UTRSTAT_TX_EMPTY
        bne  1b
        mov r2, #'0'
        str   r2, [r1, #oUTXHL]
1:      ldr   r3, [r1, #oUTRSTAT]
        and  r3, r3, #UTRSTAT_TX_EMPTY
        tst    r3, #UTRSTAT_TX_EMPTY
        bne  1b
#endif
        mov pc, lr
```

此外，vivi 还提供了几个汇编情况下通过串口打印字符的函数 PrintChar、PrintWord 和 PrintHexWord：

```
@ PrintChar : prints the character in R0
@   r0 contains the character
@   r1 contains base of serial port
@   writes ro with XXX, modifies r0,r1,r2
@   TODO : write ro with XXX reg to error handling
PrintChar:
TXBusy:
        ldr   r2, [r1, #oUTRSTAT]
        and  r2, r2, #UTRSTAT_TX_EMPTY
        tst    r2, #UTRSTAT_TX_EMPTY
        beq  TXBusy
        str   r0, [r1, #oUTXHL]
        mov pc, lr

@ PrintWord : prints the 4 characters in R0
@   r0 contains the binary word
@   r1 contains the base of the serial port
@   writes ro with XXX, modifies r0,r1,r2
@   TODO : write ro with XXX reg to error handling
PrintWord:
        mov r3, r0
        mov r4, lr
        bl     PrintChar
        mov r0, r3, LSR #8              /* shift word right 8 bits */
        bl     PrintChar
        mov r0, r3, LSR #16             /* shift word right 16 bits */
        bl     PrintChar
        mov r0, r3, LSR #24             /* shift word right 24 bits */
        bl     PrintChar
        mov r0, #'\r'
        bl     PrintChar
```

```
        mov r0, #'\n'
       bl     PrintChar
      mov pc, r4
@ PrintHexWord : prints the 4 bytes in R0 as 8 hex ascii characters
@   followed by a newline
@   r0 contains the binary word
@   r1 contains the base of the serial port
@   writes ro with XXX, modifies r0,r1,r2
@   TODO : write ro with XXX reg to error handling
PrintHexWord:
       mov r4, lr
       mov r3, r0
       mov r0, r3, LSR #28
       bl     PrintHexNibble
       mov r0, r3, LSR #24
       bl     PrintHexNibble
       mov r0, r3, LSR #20
       bl     PrintHexNibble
       mov r0, r3, LSR #16
      bl     PrintHexNibble
       mov r0, r3, LSR #12
       bl     PrintHexNibble
       mov r0, r3, LSR #8
       bl     PrintHexNibble
       mov r0, r3, LSR #4
       bl     PrintHexNibble
       mov r0, r3
       bl     PrintHexNibble
       mov r0, #'\r'
       bl     PrintChar
       mov r0, #'\n'
       bl     PrintChar
       mov pc, r4
```

3.2Bootloader 拷贝

配置为从 NAND FLASH 启动，需要将 NAND FLASH 中的 vivi 代码 copy 到 RAM 中：

```
#ifdef CONFIG_S3C2410_NAND_BOOT
       bl     copy_myself
       @ jump to ram
       ldr   r1, =on_the_ram
       add  pc, r1, #0
       nop
       nop
1:     b     1b           @ infinite loop
#ifdef CONFIG_S3C2410_NAND_BOOT
```

```
@
@ copy_myself: copy vivi to ram
@
copy_myself:
       mov r10, lr
       @ reset NAND
       mov r1, #NAND_CTL_BASE
       ldr   r2, =0xf830           @ initial value
       str   r2, [r1, #oNFCONF]
       ldr   r2, [r1, #oNFCONF]
       bic   r2, r2, #0x800        @ enable chip
       str   r2, [r1, #oNFCONF]
       mov r2, #0xff        @ RESET command
       strb  r2, [r1, #oNFCMD]
       mov r3, #0                    @ wait
1:     add  r3, r3, #0x1
       cmp r3, #0xa
       blt    1b
2:     ldr   r2, [r1, #oNFSTAT]       @ wait ready
       tst    r2, #0x1
       beq  2b
       ldr   r2, [r1, #oNFCONF]
       orr   r2, r2, #0x800        @ disable chip
       str   r2, [r1, #oNFCONF]
       @ get read to call C functions (for nand_read())
       ldr   sp, DW_STACK_START      @ setup stack pointer
       mov fp, #0                    @ no previous frame, so fp=0
       @ copy vivi to RAM
       ldr   r0, =VIVI_RAM_BASE
       mov     r1, #0x0
      mov r2, #0x20000
       bl     nand_read_ll
       tst    r0, #0x0
       beq  ok_nand_read
#ifdef CONFIG_DEBUG_LL
bad_nand_read:
       ldr   r0, STR_FAIL
       ldr   r1, SerBase
       bl     PrintWord
1:     b     1b           @ infinite loop
#endif
ok_nand_read:
#ifdef CONFIG_DEBUG_LL
       ldr   r0, STR_OK
```

```
        ldr   r1, SerBase
        bl     PrintWord
#endif
        @ verify
        mov r0, #0
        ldr   r1, =0x33f00000
        mov r2, #0x400      @ 4 bytes * 1024 = 4K-bytes
go_next:
        ldr   r3, [r0], #4
        ldr   r4, [r1], #4
        teq   r3, r4
        bne  notmatch
        subs r2, r2, #4
        beq  done_nand_read
        bne  go_next
notmatch:
#ifdef CONFIG_DEBUG_LL
        sub  r0, r0, #4
        ldr   r1, SerBase
        bl     PrintHexWord
        ldr   r0, STR_FAIL
        ldr   r1, SerBase
        bl     PrintWord
#endif
1:     b     1b
done_nand_read:
#ifdef CONFIG_DEBUG_LL
        ldr   r0, STR_OK
        ldr   r1, SerBase
        bl     PrintWord
#endif
        mov pc, r10
@ clear memory
@ r0: start address
@ r1: length
mem_clear:
        mov r2, #0
        mov r3, r2
        mov r4, r2
        mov r5, r2
        mov r6, r2
        mov r7, r2
        mov r8, r2
        mov r9, r2
```

```
clear_loop:
       stmia       r0!, {r2-r9}
       subs r1, r1, #(8 * 4)
       bne  clear_loop
       mov pc, lr
#endif @ CONFIG_S3C2410_NAND_BOOT
```

3.3 进入 C 代码

首先要设置堆栈指针 sp，堆栈指针的设置是为了执行 C 语言代码作好准备。设置好堆栈后，调用 C 语言的 main 函数：

```
@ get read to call C functions
ldr   sp, DW_STACK_START      @ setup stack pointer
mov fp, #0                    @ no previous frame, so fp=0
mov a2, #0                   @ set argv to NULL
bl     main                     @ call main
mov pc, #FLASH_BASE              @ otherwise, reboot
```

4. BootLoader 第二阶段

vivi Bootloader 的第二阶段又分成了八个小阶段，在 main 函数中分别调用这几个小阶段的相关函数：

```
int main(int argc, char *argv[])
{
       int ret;
       /*
        * Step 1:
        */
      putstr("\r\n");
       putstr(vivi_banner);
        reset_handler();
       /*
        * Step 2:
        */
       ret = board_init();
       if (ret) {
              putstr("Failed a board_init() procedure\r\n");
              error();
       }
       /*
        * Step 3:
        */
       mem_map_init();
       mmu_init();
       putstr("Succeed memory mapping.\r\n");
       /*
        * Now, vivi is running on the ram. MMU is enabled.
        */
       /*
        * Step 4:
```

```
         */
        /* initialize the heap area*/
        ret = heap_init();
        if (ret) {
                putstr("Failed initailizing heap region\r\n");
                error();
        }
        /* Step 5:
         */
        ret = mtd_dev_init();
        /* Step 6:
         */
        init_priv_data();
        /* Step 7:
         */
        misc();
        init_builtin_cmds();
        /* Step 8:
         */
        boot_or_vivi();
        return 0;
}
```

STEP1 的 putstr(vivi_banner)语句在串口输出一段字符说明 vivi 的版本、作者等信息，vivi_banner 定义为：

```
const char *vivi_banner =
                        "VIVI version " VIVI_RELEASE " (" VIVI_COMPILE_BY "@"
                        VIVI_COMPILE_HOST ") (" VIVI_COMPILER ") " UTS_VERSION "\r\n";
```

reset_handler 进行相应的复位处理：

```
void
reset_handler(void)
{
        int pressed;
        pressed = is_pressed_pw_btn();
        if (pressed == PWBT_PRESS_LEVEL) {
                DPRINTK("HARD RESET\r\n");
                hard_reset_handle();
        } else {
                DPRINTK("SOFT RESET\r\n");
                soft_reset_handle();
        }
}
```

hard_reset_handle 会 clear 内存，而软件复位处理则什么都不做：

```
static void
hard_reset_handle(void)
{
```

```
        clear_mem((unsigned long)USER_RAM_BASE, (unsigned long)USER_RAM_SIZE);
}
```

STEP2 进行板初始化，设置时间和可编程 I/O 口：

```
int board_init(void)
{
        init_time();
        set_gpios();
        return 0;
}
```

STEP3 进行内存映射及 MMU 初始化：

```
void mem_map_init(void)
{
#ifdef CONFIG_S3C2410_NAND_BOOT
        mem_map_nand_boot();
#else
        mem_map_nor();
#endif
        cache_clean_invalidate();
        tlb_invalidate();
}
```

S3C2410A 的 MMU 初始化只需要调用通用的 arm920 MMU 初始化函数：

```
static inline void arm920_setup(void)
{
        unsigned long ttb = MMU_TABLE_BASE;
__asm__(
        /* Invalidate caches */
        "mov       r0, #0\n"
        "mcr       p15, 0, r0, c7, c7, 0\n"  /* invalidate I,D caches on v4 */
        "mcr       p15, 0, r0, c7, c10, 4\n" /* drain write buffer on v4 */
        "mcr       p15, 0, r0, c8, c7, 0\n"  /* invalidate I,D TLBs on v4 */
        /* Load page table pointer */
        "mov       r4, %0\n"
        "mcr       p15, 0, r4, c2, c0, 0\n"  /* load page table pointer */
        /* Write domain id (cp15_r3) */
        "mvn       r0, #0\n"                 /* Domains 0, 1 = client */
        "mcr       p15, 0, r0, c3, c0, 0\n"  /* load domain access register */
        /* Set control register v4 */
        "mrc       p15, 0, r0, c1, c0, 0\n"  /* get control register v4 */
        /*   Clear   out   'unwanted'   bits   (then   put   them   in   if   we   need   them)   */
/* .RVI ..RS B... .CAM */
        "bic  r0, r0, #0x3000\n"         /* ..11 .... .... .... */
        "bic  r0, r0, #0x0300\n"         /* .... ..11 .... .... */
        "bic  r0, r0, #0x0087\n"         /* .... .... 1... .111 */
        /* Turn on what we want */
```

```
        /* Fault checking enabled */
        "orr  r0, r0, #0x0002\n"          /* .... .... .... ..1. */
#ifdef CONFIG_CPU_D_CACHE_ON
        "orr  r0, r0, #0x0004\n"          /* .... .... .... .1.. */
#endif
#ifdef CONFIG_CPU_I_CACHE_ON
        "orr  r0, r0, #0x1000\n"          /* ...1 .... .... .... */
#endif
        /* MMU enabled */
        "orr  r0, r0, #0x0001\n"          /* .... .... .... ...1 */
        "mcr       p15, 0, r0, c1, c0, 0\n"  /* write control register */
        : /* no outputs */
        : "r" (ttb) );
}
```

STEP4 设置堆栈；STEP5 进行 mtd 设备的初始化，记录 MTD 分区信息；STEP6 设置私有数据；STEP7 初始化内建命令。STEP8 启动一个 SHELL，等待用户输出命令并进行相应处理。在 SHELL 退出的情况下，启动操作系统：

```
#define DEFAULT_BOOT_DELAY       0x30000000
void boot_or_vivi(void)
{
        char c;
        int ret;
        ulong boot_delay;
        boot_delay = get_param_value("boot_delay", &ret);
        if (ret) boot_delay = DEFAULT_BOOT_DELAY;
        /* If a value of boot_delay is zero,
         * unconditionally call vivi shell */
        if (boot_delay == 0) vivi_shell();
        /*
        * wait for a keystroke (or a button press if you want.)
         */
        printk("Press Return to start the LINUX now, any other key for vivi\n");
        c = awaitkey(boot_delay, NULL);
        if (((c != '\r') && (c != '\n') && (c != '\0'))) {
                printk("type \"help\" for help.\n");
                vivi_shell();
        }
        run_autoboot();
        return;

}
```

SHELL 中读取用户从串口输出的命令字符串，执行该命令：

```
void
vivi_shell(void)
```

```
{
#ifdef CONFIG_SERIAL_TERM
       serial_term();
#else
#error there is no terminal.
#endif
}
void serial_term(void)
{
       char cmd_buf[MAX_CMDBUF_SIZE];
       for (;;) {
             printk("%s> ", prompt);
              getcmd(cmd_buf, MAX_CMDBUF_SIZE);
              /* execute a user command */
              if (cmd_buf[0])
                     exec_string(cmd_buf);
       }
}
```

5.电路板调试

在电路板的调试过程中，我们首先要在 ADT 新建的工程中添加第一阶段的汇编代码 head.S 文件，修改 Link 脚本，将代码和数据映射到 S3C2410A 自带的 0x40000000 开始的 4KB 内存空间内：

```
SECTIONS
{
       . = 0x40000000;
       .text : { *(.text) }
       Image_RO_Limit = .;
       Image_RW_Base = .;
       .data : { *(.data) }
       .rodata : { *(.rodata) }
       Image_ZI_Base = .;
       .bss : { *(.bss) }
       Image_ZI_Limit = .;
       __bss_start__ = .;
       __bss_end__ = .;
       __EH_FRAME_BEGIN__ = .;
       __EH_FRAME_END__ = .;
PROVIDE (__stack = .);
       end = .;
       _end = .;
       .debug_info     0 : { *(.debug_info)  }
      .debug_line            0 : { *(.debug_line)  }
     .debug_abbrev   0 : { *(.debug_abbrev)}
     .debug_frame    0 : { *(.debug_frame) }
```

}

借助万用表、示波器等仪器仪表，调通 SDRAM，并将 vivi 中自带的串口、NAND FLASH 驱动添加到工程中，调试通过板上的串口和 FLASH。如果板电路的原理与三星公司 DEMO 板有差距，则 vivi 中硬件的操作要进行相应的修改。全部调试通过后，修改 vivi 源代码，重新编译 vivi，将其烧录入 NAND FLASH 就可以在复位后启动这个 Bootloader 了。

调试板上的新增硬件时，宜在 ADT 中添加相应的代码，在不加载操作系统的情况下，单纯地操作这些硬件。如果电路板设计有误，要进行飞线和割线等处理。

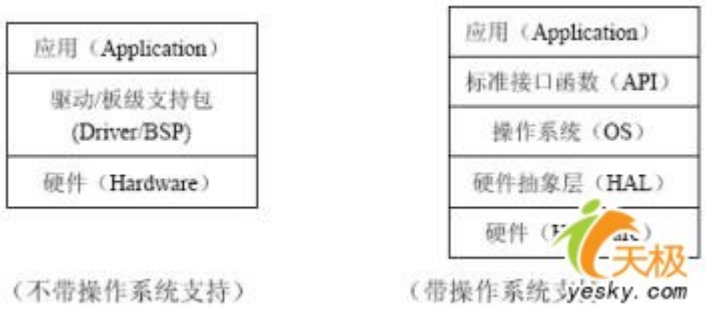
6. 小结

本章讲解了 ARM 汇编、Bootloader 的功能，Bootloader 的调试环境及 ARM 电路板的调试方法。

基于 ARM 的嵌入式 Linux 移植真实体验（3）——操作系统

宋宝华 21cnbao@21cn.com 出处:dev.yesky.com

在笔者撰写的《C 语言嵌入式系统编程修炼之道》一文中，主要陈诉的软件架构是单任务无操作系统平台的，而本文的侧重点则在于讲述操作系统嵌入的软件架构，二者的区别如下图：



嵌入式操作系统并不总是必须的，因为程序完全可以在裸板上运行。尽管如此，但对于复杂的系统，为使其具有任务管理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理和中断处理的能力，提供多任务处理，更好的分配系统资源的功能，很有必要针对特定的硬件平台和实际应用移植操作系统。鉴于 Linux 的源代码开放性，它成为嵌入式操作系统领域的很好选择。国内外许多知名大学、公司、研究机构都加入了嵌入式 Linux 的研究行列，推出了一些著名的版本：

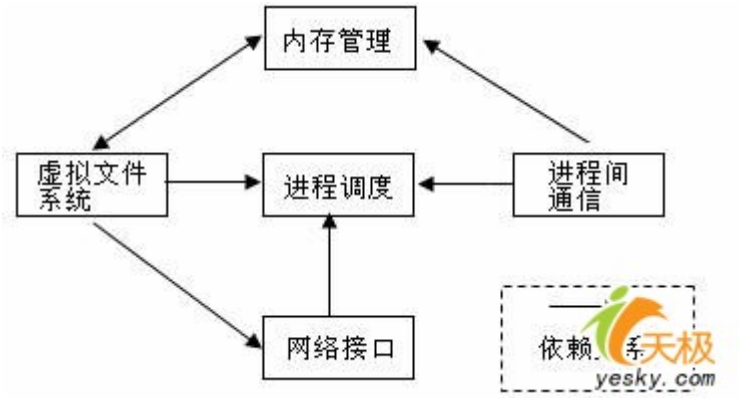
Ø RT-Linux 提供了一个精巧的实时内核，把标准的 Linux 核心作为实时核心的一个进程同用户的实时进程一起调度。RT-Linux 已成功地应用于航天飞机的空间数据采集、科学仪器测控和电影特技图像处理等广泛的应用领域。如 NASA(美国国家宇航局)将装有 RT-Linux 的设备放在飞机上，以测量 Georage 咫风的风速；

Ø uCLinux（Micro-Control-Linux，u 表示 Micro，C 表示 Control）去掉了 MMU（内存管理）功能，应用于没有虚拟内存管理的微处理器/微控制器，它已经被成功地移植到了很多平台上。

本章涉及的 mizi-linux 由韩国 mizi 公司根据 Linux 2.4 内核移植而来，支持 S3C2410A 处理器。

1.Linux 内核要点

和其他操作系统一样，Linux 包含进程调度与进程间通信(IPC)、内存管理(MMU)、虚拟文件系统(VFS)、网络接口等，下图给出了 Linux 的组成及其关系：



Linux 内核源代码包括多个目录：

（1）arch：包括硬件特定的内核代码，如 arm、mips、i386 等；

（2）drivers：包含硬件驱动代码，如 char、cdrom、scsi、mtd 等；

（3）include：通用头文件及针对不同平台特定的头文件，如 asm-i386、asm-arm 等；

（4）init：内核初始化代码；

（5）ipc：进程间通信代码；

（6）kernel：内核核心代码；

（7）mm：内存管理代码；

（8）net：与网络协议栈相关的代码，如 ipv4、ipv6、ethernet 等；

（9）fs：文件系统相关代码，如 nfs、vfat 等；

（10）lib：库文件，与平台无关的 strlen、strcpy 等，如在 string.c 中包含：

```
char * strcpy(char * dest,const char *src)
{
	char *tmp = dest;

	while ((*dest++ = *src++) != '\0')
		/* nothing */;
	return tmp;
}
```

（11）Documentation：文档。

在 Linux 内核的实现中，有一些数据结构使用非常频繁，对研读内核的人来说至为关键，它们是：

1.task_struct

Linux 内核利用 task_struct 数据结构代表一个进程，用 task_struct 指针形成一个 task 数组。当建立新进程的时候，Linux 为新的进程分配一个 task_struct 结构，然后将指针保存在 task 数组中。调度程序维护 current 指针，它指向当前正在运行的进程。

2.mm_struct

每个进程的虚拟内存由 mm_struct 结构代表。该结构中包含了一组指向 vm-area_struct 结构的指针，vm-area_struct 结构描述了虚拟内存的一个区域。

3.inode

Linux 虚拟文件系统中的文件、目录等均由对应的索引节点(inode)代表。

2.Linux 移植项目

mizi-linux 已经根据 Linux 2.4 内核针对 S3C2410A 这一芯片进行了有针对性的移植工作，包括：

（1）修改根目录下的 Makefile 文件

a.指定目标平台为 ARM：

```
#ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/)
ARCH := arm
```

b.指定交叉编译器：

```
CROSS_COMPILE      = arm-linux-
```

（2）修改 arch 目录中的文件

根据本章第一节可知，Linux 的 arch 目录存放硬件相关的内核代码，因此，在 Linux 内核中增加对 S3C2410 的支持，最主要就是要修改 arch 目录中的文件。

a.在 arch/arm/Makefile 文件中加入：

```
ifeq ($(CONFIG_ARCH_S3C2410),y)
TEXTADDR    = 0xC0008000
MACHINE           = s3c2410
Endif
```

b.在 arch\arm\config.in 文件中加入：

```
if [ "$CONFIG_ARCH_S3C2410" = "y" ]; then
comment 'S3C2410 Implementation'
dep_bool ' SMDK (MERI TECH BOARD)' CONFIG_S3C2410_SMDK $CONFIG_ARCH_S3C2410
dep_bool '    change AIJI' CONFIG_SMDK_AIJI
dep_tristate 'S3C2410 USB function support' CONFIG_S3C2410_USB $CONFIG_ARCH_S3C2100
dep_tristate ' Support for S3C2410 USB character device emulation' CONFIG_S3C2410_USB_CHAR $CONFIG_S3C2410_USB
fi     # /* CONFIG_ARCH_S3C2410 */
```

arch\arm\config.in 文件还有几处针对 S3C2410 的修改。

c.在 arch/arm/boot/Makefile 文件中加入：

```
ifeq ($(CONFIG_ARCH_S3C2410),y)
ZTEXTADDR = 0x30008000
ZRELADDR    = 0x30008000
endif
```

d.在 linux/arch/arm/boot/compressed/Makefile 文件中加入：

```
ifeq ($(CONFIG_ARCH_S3C2410),y)
OBJS           += head-s3c2410.o
endif
```

加入的结果是 head-s3c2410.S 文件被编译为 head-s3c2410.o。

e.加入 arch\arm\boot\compressed\ head-s3c2410.S 文件

```
#include <linux/config.h>
#include <linux/linkage.h>
#include <asm/mach-types.h>

      .section    ".start", #alloc, #execinstr

__S3C2410_start:

      @ Preserve r8/r7 i.e. kernel entry values
```

```
        @ What is it?
        @ Nandy

        @ Data cache, Intstruction cache, MMU might be active.
        @ Be sure to flush kernel binary out of the cache,
        @ whatever state it is, before it is turned off.
        @ This is done by fetching through currently executed
        @ memory to be sure we hit the same cache

        bic    r2, pc, #0x1f
        add r3, r2, #0x4000              @ 16 kb is quite enough...
1:      ldr r0, [r2], #32
        teq    r2, r3
        bne 1b
        mcr p15, 0, r0, c7, c10, 4      @ drain WB
        mcr p15, 0, r0, c7, c7, 0       @ flush I & D caches

#if 0
        @ disabling MMU and caches
        mrc p15, 0, r0, c1, c0, 0       @ read control register
        bic    r0, r0, #0x05                @ disable D cache and MMU
        bic r0, r0, #1000                   @ disable I cache
        mcr p15, 0, r0, c1, c0, 0
#endif

        /*
         * Pause for a short time so that we give enough time
         * for the host to start a terminal up.
         */
        mov r0, #0x00200000
1:      subs r0, r0, #1
        bne 1b
```

该文件中的汇编代码完成 S3C2410 特定硬件相关的初始化。

f.在 arch\arm\def-configs 目录中增加配置文件

g.在 arch\arm\kernel\Makefile 中增加对 S3C2410 的支持

```
no-irq-arch      := $(CONFIG_ARCH_INTEGRATOR) $(CONFIG_ARCH_CLPS711X) \
                $(CONFIG_FOOTBRIDGE) $(CONFIG_ARCH_EBSA110) \
                $(CONFIG_ARCH_SA1100) $(CONFIG_ARCH_CAMELOT) \
                $(CONFIG_ARCH_S3C2400) $(CONFIG_ARCH_S3C2410) \
                $(CONFIG_ARCH_MX1ADS) $(CONFIG_ARCH_PXA)
obj-$(CONFIG_MIZI)    += event.o
obj-$(CONFIG_APM)    += apm2.o
```

h.修改 arch/arm/kernel/debug-armv.S 文件，在适当的位置增加如下关于 S3C2410 的代码：

```
#elif defined(CONFIG_ARCH_S3C2410)
```

```
            .macro addruart,rx
            mrc     p15, 0, \rx, c1, c0
            tst     \rx, #1         @ MMU enabled ?
            moveq   \rx, #0x50000000        @ physical base address
            movne   \rx, #0xf0000000        @ virtual address
            .endm

            .macro senduart,rd,rx
            str     \rd, [\rx, #0x20]       @ UTXH
            .endm

            .macro waituart,rd,rx
            .endm

            .macro busyuart,rd,rx
1001: ldr     \rd, [\rx, #0x10]       @ read UTRSTAT
            tst     \rd, #1 << 2            @ TX_EMPTY ?
            beq     1001b
            .endm
```

i.修改 arch/arm/kernel/setup.c 文件

此文件中的 setup_arch 非常关键，用来完成与体系结构相关的初始化：

```
void __init setup_arch(char **cmdline_p)
{
      struct tag *tags = NULL;
      struct machine_desc *mdesc;
      char *from = default_command_line;

      ROOT_DEV = MKDEV(0, 255);

      setup_processor();
      mdesc = setup_machine(machine_arch_type);
      machine_name = mdesc->name;

      if (mdesc->soft_reboot)
            reboot_setup("s");

      if (mdesc->param_offset)
            tags = phys_to_virt(mdesc->param_offset);

      /*
       * Do the machine-specific fixups before we parse the
       * parameters or tags.
       */
```

```
        if (mdesc->fixup)
                mdesc->fixup(mdesc, (struct param_struct *)tags,
                             &from, &meminfo);

        /*
         * If we have the old style parameters, convert them to
         * a tag list before.
         */
        if (tags && tags->hdr.tag != ATAG_CORE)
                convert_to_tag_list((struct param_struct *)tags,
                                    meminfo.nr_banks == 0);

        if (tags && tags->hdr.tag == ATAG_CORE)
                parse_tags(tags);

        if (meminfo.nr_banks == 0) {
                meminfo.nr_banks      = 1;
                meminfo.bank[0].start = PHYS_OFFSET;
                meminfo.bank[0].size  = MEM_SIZE;
        }

        init_mm.start_code = (unsigned long) &_text;
        init_mm.end_code   = (unsigned long) &_etext;
        init_mm.end_data   = (unsigned long) &_edata;
        init_mm.brk        = (unsigned long) &_end;

        memcpy(saved_command_line, from, COMMAND_LINE_SIZE);
        saved_command_line[COMMAND_LINE_SIZE-1] = '\0';
        parse_cmdline(&meminfo, cmdline_p, from);
        bootmem_init(&meminfo);
        paging_init(&meminfo, mdesc);
        request_standard_resources(&meminfo, mdesc);

        /*
         * Set up various architecture-specific pointers
         */
        init_arch_irq = mdesc->init_irq;

#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
        conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
        conswitchp = &dummy_con;
#endif
```

```
#endif
}
```
j.修改 arch/arm/mm/mm-armv.c 文件（arch/arm/mm/目录中的文件完成与 ARM 相关的 MMU 处理）修改

```
init_maps->bufferable = 0;
```
为

```
init_maps->bufferable = 1;
```
要轻而易举地进行上述马拉松式的内核移植工作并非一件轻松的事情，需要对 Linux 内核有很好的掌握，同时掌握硬件特定的知识和相关的汇编。幸而 mizi 公司的开发者们已经合力为我们完成了上述工作，这使得小弟们在将 mizi-linux 移植到自身开发的电路板的过程中只需要关心如下几点：

（1）内核初始化：Linux 内核的入口点是 start_kernel()函数。它初始化内核的其他部分，包括捕获，IRQ 通道，调度，设备驱动，标定延迟循环，最重要的是能够 fork“init”进程，以启动整个多任务环境。

我们可以在 init 中加上一些特定的内容。

(2)设备驱动：设备驱动占据了 Linux 内核很大部分。同其他操作系统一样，设备驱动为它们所控制的硬件设备和操作系统提供接口。

本文第四章将单独讲解驱动程序的编写方法。

(3)文件系统：Linux 最重要的特性之一就是对多种文件系统的支持。这种特性使得 Linux 很容易地同其他操作系统共存。文件系统的概念使得用户能够查看存储设备上的文件和路径而无须考虑实际物理设备的文件系统类型。Linux 透明的支持许多不同的文件系统，将各种安装的文件和文件系统以一个完整的虚拟文件系统的形式呈现给用户。

我们可以在 K9S1208 NAND FLASH 上移植 cramfs、jfss2、yaffs 等 FLASH 文件系统。

3. init 进程

在 init 函数中“加料”，可以使得 Linux 启动的时候做点什么，例如广州友善之臂公司的 demo 板在其中加入了公司信息：

```
static int init(void * unused)
{
	lock_kernel();
	do_basic_setup();

	prepare_namespace();

	/*
	 * Ok, we have completed the initial bootup, and
	 * we're essentially up and running. Get rid of the
	 * initmem segments and start the user-mode stuff..
	 */
	free_initmem();
	unlock_kernel();

	if (open("/dev/console", O_RDWR, 0) < 0)
		printk("Warning: unable to open an initial console.\n");

	(void) dup(0);
```

```
        (void) dup(0);

        /*
         * We try each of these until one succeeds.
         *
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */

        printk("=========================================\n");
        printk("=      Friendly-ARM Tech. Ltd.          =\n");
        printk("=       http://www.arm9.net             =\n");
        printk("=      http://www.arm9.com.cn           =\n");
        printk("=========================================\n");

        if (execute_command)
                execve(execute_command,argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_init,envp_init);
        panic("No init found. Try passing init= option to kernel.");
}
```

这样在 Linux 的启动过程中，会额外地输出：

```
=========================================
=          Friendly-ARM Tech. Ltd.      =
=          http://www.arm9.net          =
=          http://www.arm9.com.cn       =
=========================================
```

4.文件系统移植

文件系统是基于被划分的存储设备上的逻辑上单位上的一种定义文件的命名、存储、组织及取出的方法。如果一个 Linux 没有根文件系统，它是不能被正确的启动的。因此，我们需要为 Linux 创建根文件系统，我们将其创建在 K9S1208 NAND FLASH 上。

Linux 的根文件系统可能包括如下目录（或更多的目录）：

（1）/bin (binary)：包含着所有的标准命令和应用程序；

（2）/dev (device)：包含外设的文件接口，在 Linux 下，文件和设备采用同种地方法访问的，系统上的每个设备都在/dev 里有一个对应的设备文件；

（3）/etc (etcetera)：这个目录包含着系统设置文件和其他的系统文件，例如/etc/fstab(file system table)记录了启动时要 mount 的 filesystem；

（4）/home：存放用户主目录；

（5）/lib(library)：存放系统最基本的库文件；

（6）/mnt：用户临时挂载文件系统的地方；

（7）/proc：linux 提供的一个虚拟系统，系统启动时在内存中产生，用户可以直接通过访问这些文件来获得

系统信息；
（8）/root：超级用户主目录；
（9）/sbin：这个目录存放着系统管理程序，如 fsck、mount 等；
（10）/tmp(temporary)：存放不同的程序执行时产生的临时文件；
（11）/usr(user)：存放用户应用程序和文件。
采用 BusyBox 是缩小根文件系统的好办法，因为其中提供了系统的许多基本指令但是其体积很小。众所周知，瑞士军刀以其小巧轻便、功能众多而闻名世界，成为各国军人的必备工具，并广泛应用于民间，而 BusyBox 也被称为嵌入式 Linux 领域的"瑞士军刀"。
此地址可以下载 BusyBox：http://www.busybox.net，当前最新版本为 1.1.3。编译好 busybox 后，将其放入/bin 目录，若要使用其中的命令，只需要建立 link，如：

```
ln -s ./busybox ls
ln -s ./busybox mkdir
```

4.1 cramfs

在根文件系统中，为保护系统的基本设置不被更改，可以采用 cramfs 格式，它是一种只读的闪存文件系统。制作 cramfs 文件系统的方法为：建立一个目录，将需要放到文件系统的文件 copy 到这个目录，运行"mkcramfs 目录名 image 名"就可以生成一个 cramfs 文件系统的 image 文件。例如如果目录名为 rootfs，则正确的命令为：

```
mkcramfs rootfs rootfs.ramfs
```

我们使用下面的命令可以 mount 生成的 rootfs.ramfs 文件，并查看其中的内容：

```
mount -o loop -t cramfs rootfs.ramfs /mount/point
```

此地址可以下载 mkcramfs 工具：http://sourceforge.net/projects/cramfs/。

4.2 jfss2

对于 cramfs 闪存文件系统，如果没有 ramfs 的支持则只能读，而采用 jfss2（The Journalling Flash File System version 2）文件系统则可以直接在闪存中读、写数据。jfss2 是一个日志结构(log-structured)的文件系统，包含数据和原数据(meta-data)的节点在闪存上顺序地存储。jfss2 记录了每个擦写块的擦写次数，当闪存上各个擦写块的擦写次数的差距超过某个预定的阀值，开始进行磨损平衡的调整。调整的策略是，在垃圾回收时将擦写次数小的擦写块上的数据迁移到擦写次数大的擦写块上以达到磨损平衡的目的。
与 mkcramfs 类似，同样有一个 mkfs.jffs2 工具可以将一个目录制作为 jffs2 文件系统。假设把/bin 目录制作为 jffs2 文件系统，需要运行的命令为：

```
mkfs.jffs2 -d /bin -o jffs2.img
```

4.3 yaffs

yaffs 是一种专门为嵌入式系统中常用的闪存设备设计的一种可读写的文件系统，它比 jffs2 文件系统具有更快的启动速度，对闪存使用寿命有更好的保护机制。为使 Linux 支持 yaffs 文件系统，我们需要将其对应的驱动加入到内核中 fs/yaffs/，并修改内核配置文件。使用我们使用 mkyaffs 工具可以将 NAND FLASH 中的分区格式化为 yaffs 格式（如/bin/mkyaffs /dev/mtdblock/0 命令可以将第 1 个 MTD 块设备分区格式化为 yaffs），而使用 mkyaffsimage（类似于 mkcramfs、mkfs.jffs2）则可以将某目录生成为 yaffs 文件系统镜像。
嵌入式 Linux 还可以使用 NFS（网络文件系统）通过以太网挂接根文件系统，这是一种经常用来作为调试使用的文件系统启动方式。通过网络挂接的根文件系统，可以在主机上生成 ARM 交叉编译版本的目标文件或二进制可执行文件，然后就可以直接装载或执行它，而不用频繁地写入 flash。
采用不同的文件系统启动时，要注意通过第二章介绍的 BootLoader 修改启动参数，如广州友善之臂的 demo 提供如下三种启动方式：
（1）从 cramfs 挂接根文件系统：root=/dev/bon/2()；
（2）从移植的 yaffs 挂接根文件系统：root=/dev/mtdblock/0；

（3）从以太网挂接根文件系统：root=/dev/nfs。

5.小结

本章介绍了嵌入式 Linux 的背景、移植项目、init 进程修改和文件系统移植，通过这些步骤，我们可以在嵌入式系统上启动一个基本的 Linux。

基于 ARM 的嵌入式 Linux 移植真实体验（4）——设备驱动

宋宝华 21cnbao@21cn.com 出处:dev.yesky.com

设备驱动程序是操作系统内核和机器硬件之间的接口，它为应用程序屏蔽硬件的细节，一般来说，Linux 的设备驱动程序需要完成如下功能：

Ø 设备初始化、释放；
Ø 提供各类设备服务；
Ø 负责内核和设备之间的数据交换；
Ø 检测和处理设备工作过程中出现的错误。

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得 Linux 的设备操作犹如文件一般。在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作，如 open ()、close ()、read ()、write () 等。

Linux 主要将设备分为二类：字符设备和块设备。字符设备是指设备发送和接收数据以字符的形式进行；而块设备则以整个数据缓冲区的形式进行。在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了；而块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备主要针对磁盘等慢速设备。

1.内存分配

由于Linux驱动程序在内核中运行,因此在设备驱动程序需要申请/释放内存时,不能使用用户级的malloc/free函数，而需由内核级的函数 kmalloc/kfree () 来实现，kmalloc()函数的原型为：

```
void kmalloc (size_t size ,int priority);
```

参数 size 为申请分配内存的字节数，kmalloc 最多只能开辟 128k 的内存；参数 priority 说明若 kmalloc()不能马上分配内存时用户进程要采用的动作：GFP_KERNEL 表示等待，即等 kmalloc()函数将一些内存安排到交换区来满足你的内存需要，GFP_ATOMIC 表示不等待，如不能立即分配到内存则返回 0 值；函数的返回值指向已分配内存的起始地址，出错时，返回 0。

kmalloc ()分配的内存需用 kfree()函数来释放，kfree ()被定义为：

```
# define kfree (n) kfree_s( (n) ,0)
```

其中 kfree_s () 函数原型为：

```
void kfree_s (void * ptr ,int size);
```

参数 ptr 为 kmalloc()返回的已分配内存的指针，size 是要释放内存的字节数，若为 0 时，由内核自动确定内存的大小。

2.中断

许多设备涉及到中断操作，因此，在这样的设备的驱动程序中需要对硬件产生的中断请求提供中断服务程序。与注册基本入口点一样，驱动程序也要请求内核将特定的中断请求和中断服务程序联系在一起。在 Linux 中，用 request_irq()函数来实现请求：

```
int request_irq (unsigned int irq ,void( * handler) int ,unsigned long type ,char * name);
```
参数 irq 为要中断请求号，参数 handler 为指向中断服务程序的指针，参数 type 用来确定是正常中断还是快速中断（正常中断指中断服务子程序返回后，内核可以执行调度程序来确定将运行哪一个进程；而快速中断是指中断服务子程序返回后，立即执行被中断程序，正常中断 type 取值为 0 ，快速中断 type 取值为 SA_INTERRUPT），参数 name 是设备驱动程序的名称。

3.字符设备驱动

我们必须为字符设备提供一个初始化函数，该函数用来完成对所控设备的初始化工作，并调用 register_chrdev() 函数注册字符设备。假设有一字符设备"exampledev"，则其 init 函数为：

```
void exampledev_init(void)
{
 if (register_chrdev(MAJOR_NUM, " exampledev ", &exampledev_fops))
    TRACE_TXT("Device exampledev driver registered error");
 else
    TRACE_TXT("Device exampledev driver registered successfully");
 …//设备初始化
}
```

其中，register_chrdev 函数中的参数 MAJOR_NUM 为主设备号，"exampledev"为设备名，exampledev_fops 为包含基本函数入口点的结构体，类型为 file_operations。当执行 exampledev_init 时，它将调用内核函数 register_chrdev，把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。

随着内核功能的加强，file_operations 结构体也变得更加庞大。但是大多数的驱动程序只是利用了其中的一部分，对于驱动程序中无需提供的功能，只需要把相应位置的值设为 NULL。对于字符设备来说，要提供的主要入口有：open ()、release ()、read ()、write ()、ioctl ()等。

open()函数 对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open () 函数：

```
int (*open)(struct inode * inode,struct file *filp);
```

其中参数 inode 为设备特殊文件的 inode (索引结点) 结构的指针，参数 filp 是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode->i_rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0 表示成功，负数表示存在错误) 等；

release()函数 当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数：

```
void (*release) (struct inode * inode,struct file *filp) ;
```

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

read()函数 当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read() 函数：

```
ssize_t (*read) (struct file * filp, char * buf, size_t count, loff_t * offp);
```

参数 buf 是指向用户空间缓冲区的指针，由用户进程给出，count 为用户进程要求读取的字节数，也由用户给出。

read() 函数的功能就是从硬设备或内核内存中读取或复制 count 个字节到 buf 指定的缓冲区中。在复制数据时要注意，驱动程序运行在内核中，而 buf 指定的缓冲区在用户内存区中，是不能直接在内核中访问使用的，因此，必须使用特殊的复制函数来完成复制工作，这些函数在 include/asm/uaccess.h 中被声明：

```
unsigned long copy_to_user (void * to, void * from, unsigned long len);
```

此外，put_user()函数用于内核空间和用户空间的单值交互（如 char、int、long）。

write( ) 函数 当设备特殊文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

write ()的功能是将参数 buf 指定的缓冲区中的 count 个字节内容复制到硬件或内核内存中，和 read() 一样，复制工作也需要由特殊函数来完成：

```
unsigned long copy_from_user(void *to, const void *from, unsigned long n);
```

此外，get_user()函数用于内核空间和用户空间的单值交互（如 char、int、long）。

ioctl() 函数 该函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息，函数原型为：

```
int (*ioctl) (struct inode * inode,struct file * filp,unsigned int cmd,unsigned long arg);
```

参数 cmd 为设备驱动程序要执行的命令的代码，由用户自定义，参数 arg 为相应的命令提供参数，类型可以是整型、指针等。

同样，在驱动程序中，这些函数的定义也必须符合命名规则，按照本文约定，设备“exampledev”的驱动程序的这些函数应分别命名为 exampledev_open 、 exampledev_ release 、 exampledev_read 、 exampledev_write 、 exampledev_ioctl ， 因此设备“exampledev”的基本入口点结构变量 exampledev_fops 赋值如下（对较早版本的内核）：

```
struct file_operations exampledev_fops {
  NULL ,
  exampledev_read ,
  exampledev_write ,
  NULL ,
  NULL ,
  exampledev_ioctl ,
  NULL ,
  exampledev_open ,
  exampledev_release ,
  NULL ,
  NULL ,
  NULL ,
  NULL
} ;
```

就目前而言，由于 file_operations 结构体已经很庞大，我们更适合用 GNU 扩展的 C 语法来初始化 exampledev_fops：

```
struct file_operations exampledev_fops = {
read: exampledev _read,
write: exampledev _write,
ioctl: exampledev_ioctl ,
open: exampledev_open ,
release : exampledev_release ,
};
```

看看第一章电路板硬件原理图，板上包含四个用户可编程的发光二极管（LED），这些 LED 连接在 ARM 处理器的可编程 I/O 口（GPIO）上，现在来编写这些 LED 的驱动：

```
#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
```

```
#include <linux/delay.h>
#include <asm/hardware.h>
#define DEVICE_NAME "leds" /*定义 led 设备的名字*/
#define LED_MAJOR 231 /*定义 led 设备的主设备号*/
static unsigned long led_table[] =
{
 /*I/O 方式 led 设备对应的硬件资源*/
 GPIO_B10, GPIO_B8, GPIO_B5, GPIO_B6,
};
/*使用 ioctl 控制 led*/
static int leds_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
 unsigned long arg)
{
 switch (cmd)
 {
    case 0:
    case 1:
      if (arg > 4)
      {
        return -EINVAL;
      }
      write_gpio_bit(led_table[arg], !cmd);
    default:
      return -EINVAL;
 }
}

static struct file_operations leds_fops =
{
 owner: THIS_MODULE, ioctl: leds_ioctl,
};
static devfs_handle_t devfs_handle;
static int __init leds_init(void)
{
 int ret;
 int i;
 /*在内核中注册设备*/
 ret = register_chrdev(LED_MAJOR, DEVICE_NAME, &leds_fops);
 if (ret < 0)
 {
    printk(DEVICE_NAME " can't register major number\n");
    return ret;
 }
 devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT, LED_MAJOR,
```

```
    0, S_IFCHR | S_IRUSR | S_IWUSR, &leds_fops, NULL);
 /*使用宏进行端口初始化，set_gpio_ctrl 和 write_gpio_bit 均为宏定义*/
 for (i = 0; i < 8; i++)
 {
    set_gpio_ctrl(led_table[i] | GPIO_PULLUP_EN | GPIO_MODE_OUT);
    write_gpio_bit(led_table[i], 1);
 }
 printk(DEVICE_NAME " initialized\n");
 return 0;
}

static void __exit leds_exit(void)
{
 devfs_unregister(devfs_handle);
 unregister_chrdev(LED_MAJOR, DEVICE_NAME);
}

module_init(leds_init);
module_exit(leds_exit);
```

使用命令方式编译 led 驱动模块：

```
#arm-linux-gcc -D__KERNEL__ -I/arm/kernel/include
-DKBUILD_BASENAME=leds -DMODULE -c -o leds.o leds.c
```

以上命令将生成 leds.o 文件，把该文件复制到板子的/lib 目录下，使用以下命令就可以安装 leds 驱动模块：

```
#insmod /lib/ leds.o
```

删除该模块的命令是：

```
#rmmod leds
```

4.块设备驱动

块设备驱动程序的编写是一个浩繁的工程，其难度远超过字符设备，上千行的代码往往只能搞定一个简单的块设备，而数十行代码就可能搞定一个字符设备。因此，非得有相当的基本功才能完成此项工作。下面先给出一个实例，即 mtdblock 块设备的驱动。我们通过分析此实例中的代码来说明块设备驱动程序的写法（由于篇幅的关系，大量的代码被省略，只保留了必要的主干）：

```
#include <linux/config.h>
#include <linux/devfs_fs_kernel.h>
static void mtd_notify_add(struct mtd_info* mtd);
static void mtd_notify_remove(struct mtd_info* mtd);
static struct mtd_notifier notifier = {
        mtd_notify_add,
        mtd_notify_remove,
        NULL
};
static devfs_handle_t devfs_dir_handle = NULL;
static devfs_handle_t devfs_rw_handle[MAX_MTD_DEVICES];
```

```
static struct mtdblk_dev {
	struct mtd_info *mtd; /* Locked */
	int count;
	struct semaphore cache_sem;
	unsigned char *cache_data;
	unsigned long cache_offset;
	unsigned int cache_size;
	enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;
} *mtdblks[MAX_MTD_DEVICES];

static spinlock_t mtdblks_lock;
/* this lock is used just in kernels >= 2.5.x */
static spinlock_t mtdblock_lock;

static int mtd_sizes[MAX_MTD_DEVICES];
static int mtd_blksizes[MAX_MTD_DEVICES];

static void erase_callback(struct erase_info *done)
{
	wait_queue_head_t *wait_q = (wait_queue_head_t *)done->priv;
	wake_up(wait_q);
}

static int erase_write (struct mtd_info *mtd, unsigned long pos,
                        int len, const char *buf)
{
	struct erase_info erase;
	DECLARE_WAITQUEUE(wait, current);
	wait_queue_head_t wait_q;
	size_t retlen;
	int ret;

	/*
	 * First, let's erase the flash block.
	 */

	init_waitqueue_head(&wait_q);
	erase.mtd = mtd;
	erase.callback = erase_callback;
	erase.addr = pos;
	erase.len = len;
	erase.priv = (u_long)&wait_q;

	set_current_state(TASK_INTERRUPTIBLE);
```

```
        add_wait_queue(&wait_q, &wait);

        ret = MTD_ERASE(mtd, &erase);
        if (ret) {
                set_current_state(TASK_RUNNING);
                remove_wait_queue(&wait_q, &wait);
                printk (KERN_WARNING "mtdblock: erase of region [0x%lx, 0x%x] "
                                     "on \"%s\" failed\n",
                        pos, len, mtd->name);
                return ret;
        }

        schedule(); /* Wait for erase to finish. */
        remove_wait_queue(&wait_q, &wait);

        /*
         * Next, writhe data to flash.
         */

        ret = MTD_WRITE (mtd, pos, len, &retlen, buf);
        if (ret)
                return ret;
        if (retlen != len)
                return -EIO;
        return 0;
}


static int write_cached_data (struct mtdblk_dev *mtdblk)
{
        struct mtd_info *mtd = mtdblk->mtd;
        int ret;

        if (mtdblk->cache_state != STATE_DIRTY)
                return 0;

        DEBUG(MTD_DEBUG_LEVEL2, "mtdblock: writing cached data for \"%s\" "
                        "at 0x%lx, size 0x%x\n", mtd->name,
                        mtdblk->cache_offset, mtdblk->cache_size);

        ret = erase_write (mtd, mtdblk->cache_offset,
                           mtdblk->cache_size, mtdblk->cache_data);
        if (ret)
                return ret;
```

```
        mtdblk->cache_state = STATE_EMPTY;
        return 0;
}

static int do_cached_write (struct mtdblk_dev *mtdblk, unsigned long pos,
                            int len, const char *buf)
{
...
}

static int do_cached_read (struct mtdblk_dev *mtdblk, unsigned long pos,
                           int len, char *buf)
{
...
}

static int mtdblock_open(struct inode *inode, struct file *file)
{
...
}

static release_t mtdblock_release(struct inode *inode, struct file *file)
{
        int dev;
        struct mtdblk_dev *mtdblk;
    DEBUG(MTD_DEBUG_LEVEL1, "mtdblock_release\n");

        if (inode == NULL)
                release_return(-ENODEV);

        dev = minor(inode->i_rdev);
        mtdblk = mtdblks[dev];

        down(&mtdblk->cache_sem);
        write_cached_data(mtdblk);
        up(&mtdblk->cache_sem);

        spin_lock(&mtdblks_lock);
        if (!--mtdblk->count) {
                /* It was the last usage. Free the device */
                mtdblks[dev] = NULL;
                spin_unlock(&mtdblks_lock);
                if (mtdblk->mtd->sync)
```

```
                mtdblk->mtd->sync(mtdblk->mtd);
        put_mtd_device(mtdblk->mtd);
        vfree(mtdblk->cache_data);
        kfree(mtdblk);
    } else {
        spin_unlock(&mtdblks_lock);
    }

    DEBUG(MTD_DEBUG_LEVEL1, "ok\n");

    BLK_DEC_USE_COUNT;
    release_return(0);
}


/*
 * This is a special request_fn because it is executed in a process context
 * to be able to sleep independently of the caller. The
 * io_request_lock (for <2.5) or queue_lock (for >=2.5) is held upon entry
 * and exit. The head of our request queue is considered active so there is
 * no need to dequeue requests before we are done.
 */
static void handle_mtdblock_request(void)
{
    struct request *req;
    struct mtdblk_dev *mtdblk;
    unsigned int res;

    for (;;) {
        INIT_REQUEST;
        req = CURRENT;
        spin_unlock_irq(QUEUE_LOCK(QUEUE));
        mtdblk = mtdblks[minor(req->rq_dev)];
        res = 0;

        if (minor(req->rq_dev) >= MAX_MTD_DEVICES)
            panic("%s : minor out of bound", __FUNCTION__);

        if (!IS_REQ_CMD(req))
            goto end_req;

        if ((req->sector + req->current_nr_sectors) > (mtdblk->mtd->size >> 9))
            goto end_req;
```

```
            // Handle the request
            switch (rq_data_dir(req))
            {
                  int err;

                  case READ:
                  down(&mtdblk->cache_sem);
                  err = do_cached_read (mtdblk, req->sector << 9,
                              req->current_nr_sectors << 9,
                              req->buffer);
                  up(&mtdblk->cache_sem);
                  if (!err)
                        res = 1;
                  break;

                  case WRITE:
                  // Read only device
                  if ( !(mtdblk->mtd->flags & MTD_WRITEABLE) )
                        break;

                  // Do the write
                  down(&mtdblk->cache_sem);
                  err = do_cached_write (mtdblk, req->sector << 9,
                              req->current_nr_sectors << 9,
                              req->buffer);
                  up(&mtdblk->cache_sem);
                  if (!err)
                        res = 1;
                  break;
            }

end_req:
            spin_lock_irq(QUEUE_LOCK(QUEUE));
            end_request(res);
      }
}

static volatile int leaving = 0;
static DECLARE_MUTEX_LOCKED(thread_sem);
static DECLARE_WAIT_QUEUE_HEAD(thr_wq);

int mtdblock_thread(void *dummy)
{
…
```

```
}

#define RQFUNC_ARG request_queue_t *q

static void mtdblock_request(RQFUNC_ARG)
{
	/* Don't do anything, except wake the thread if necessary */
	wake_up(&thr_wq);
}

static int mtdblock_ioctl(struct inode * inode, struct file * file,
		      unsigned int cmd, unsigned long arg)
{
	struct mtdblk_dev *mtdblk;

	mtdblk = mtdblks[minor(inode->i_rdev)];

	switch (cmd) {
	case BLKGETSIZE:   /* Return device size */
		return put_user((mtdblk->mtd->size >> 9), (unsigned long *) arg);


	case BLKFLSBUF:
		if(!capable(CAP_SYS_ADMIN))
			return -EACCES;

		fsync_dev(inode->i_rdev);
		invalidate_buffers(inode->i_rdev);
		down(&mtdblk->cache_sem);
		write_cached_data(mtdblk);
		up(&mtdblk->cache_sem);
		if (mtdblk->mtd->sync)
			mtdblk->mtd->sync(mtdblk->mtd);
		return 0;

	default:
		return -EINVAL;
	}
}

static struct block_device_operations mtd_fops =
{
	owner: THIS_MODULE,
	open: mtdblock_open,
```

```
        release: mtdblock_release,
        ioctl: mtdblock_ioctl
};

static void mtd_notify_add(struct mtd_info* mtd)
{
…
}

static void mtd_notify_remove(struct mtd_info* mtd)
{
        if (!mtd || mtd->type == MTD_ABSENT)
                return;

        devfs_unregister(devfs_rw_handle[mtd->index]);
}

int __init init_mtdblock(void)
{
        int i;

        spin_lock_init(&mtdblks_lock);
        /* this lock is used just in kernels >= 2.5.x */
        spin_lock_init(&mtdblock_lock);

#ifdef CONFIG_DEVFS_FS
        if (devfs_register_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME, &mtd_fops))
        {
                printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology
Devices.\n",
                        MTD_BLOCK_MAJOR);
                return -EAGAIN;
        }

        devfs_dir_handle = devfs_mk_dir(NULL, DEVICE_NAME, NULL);
        register_mtd_user(&notifier);
#else
        if (register_blkdev(MAJOR_NR,DEVICE_NAME,&mtd_fops)) {
                printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology
Devices.\n",
                        MTD_BLOCK_MAJOR);
                return -EAGAIN;
        }
#endif
```

```
        /* We fill it in at open() time. */
        for (i=0; i< MAX_MTD_DEVICES; i++) {
                mtd_sizes[i] = 0;
                mtd_blksizes[i] = BLOCK_SIZE;
        }
        init_waitqueue_head(&thr_wq);
        /* Allow the block size to default to BLOCK_SIZE. */
        blksize_size[MAJOR_NR] = mtd_blksizes;
        blk_size[MAJOR_NR] = mtd_sizes;

                BLK_INIT_QUEUE(BLK_DEFAULT_QUEUE(MAJOR_NR), &mtdblock_request,
&mtdblock_lock);

        kernel_thread (mtdblock_thread, NULL, CLONE_FS|CLONE_FILES|CLONE_SIGHAND);
        return 0;
}

static void __exit cleanup_mtdblock(void)
{
        leaving = 1;
        wake_up(&thr_wq);
        down(&thread_sem);
#ifdef CONFIG_DEVFS_FS
        unregister_mtd_user(&notifier);
        devfs_unregister(devfs_dir_handle);
        devfs_unregister_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME);
#else
        unregister_blkdev(MAJOR_NR,DEVICE_NAME);
#endif
        blk_cleanup_queue(BLK_DEFAULT_QUEUE(MAJOR_NR));
        blksize_size[MAJOR_NR] = NULL;
        blk_size[MAJOR_NR] = NULL;
}

module_init(init_mtdblock);
module_exit(cleanup_mtdblock);
```

从上述源代码中我们发现，块设备也以与字符设备 register_chrdev、unregister_ chrdev 函数类似的方法进行设备的注册与释放：

```
int register_blkdev(unsigned int major, const char *name, struct block_device_operations
*bdops);
int unregister_blkdev(unsigned int major, const char *name);
```

但是，register_chrdev 使用一个向 file_operations 结构的指针，而 register_blkdev 则使用 block_device_operations 结构的指针，其中定义的 open、release 和 ioctl 方法和字符设备的对应方法相

同，但未定义 read 或者 write 操作。这是因为，所有涉及到块设备的 I/O 通常由系统进行缓冲处理。
块驱动程序最终必须提供完成实际块 I/O 操作的机制，在 Linux 当中，用于这些 I/O 操作的方法称为“request（请求）”。在块设备的注册过程中，需要初始化 request 队列，这一动作通过 blk_init_queue 来完成，blk_init_queue 函数建立队列，并将该驱动程序的 request 函数关联到队列。在模块的清除阶段，应调用 blk_cleanup_queue 函数。
本例中相关的代码为：

```
BLK_INIT_QUEUE(BLK_DEFAULT_QUEUE(MAJOR_NR), &mtdblock_request, &mtdblock_lock);
blk_cleanup_queue(BLK_DEFAULT_QUEUE(MAJOR_NR));
```

每个设备有一个默认使用的请求队列，必要时，可使用 BLK_DEFAULT_QUEUE(major) 宏得到该默认队列。这个宏在 blk_dev_struct 结构形成的全局数组（该数组名为 blk_dev）中搜索得到对应的默认队列。blk_dev 数组由内核维护，并可通过主设备号索引。blk_dev_struct 接口定义如下：

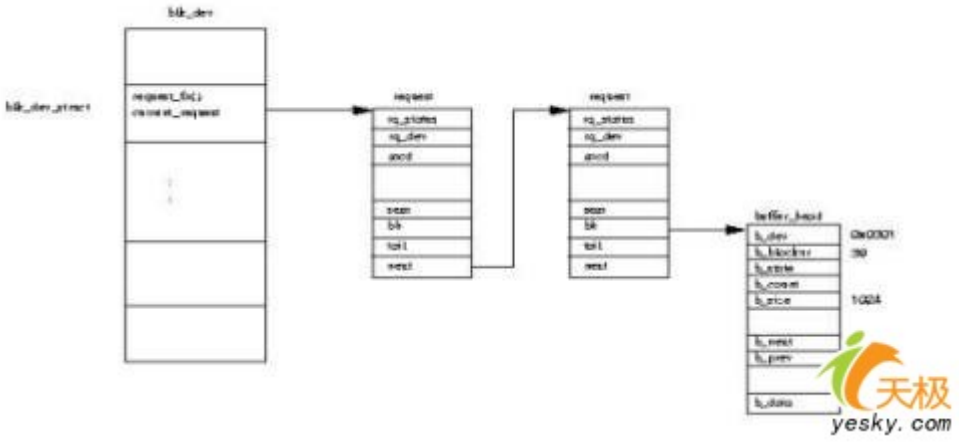
```
struct blk_dev_struct {
	/*
	 * queue_proc has to be atomic
	 */
	request_queue_t		request_queue;
	queue_proc		*queue;
	void			*data;
};
```

request_queue 成员包含了初始化之后的 I/O 请求队列，data 成员可由驱动程序使用，以便保存一些私有数据。
request_queue 定义为：

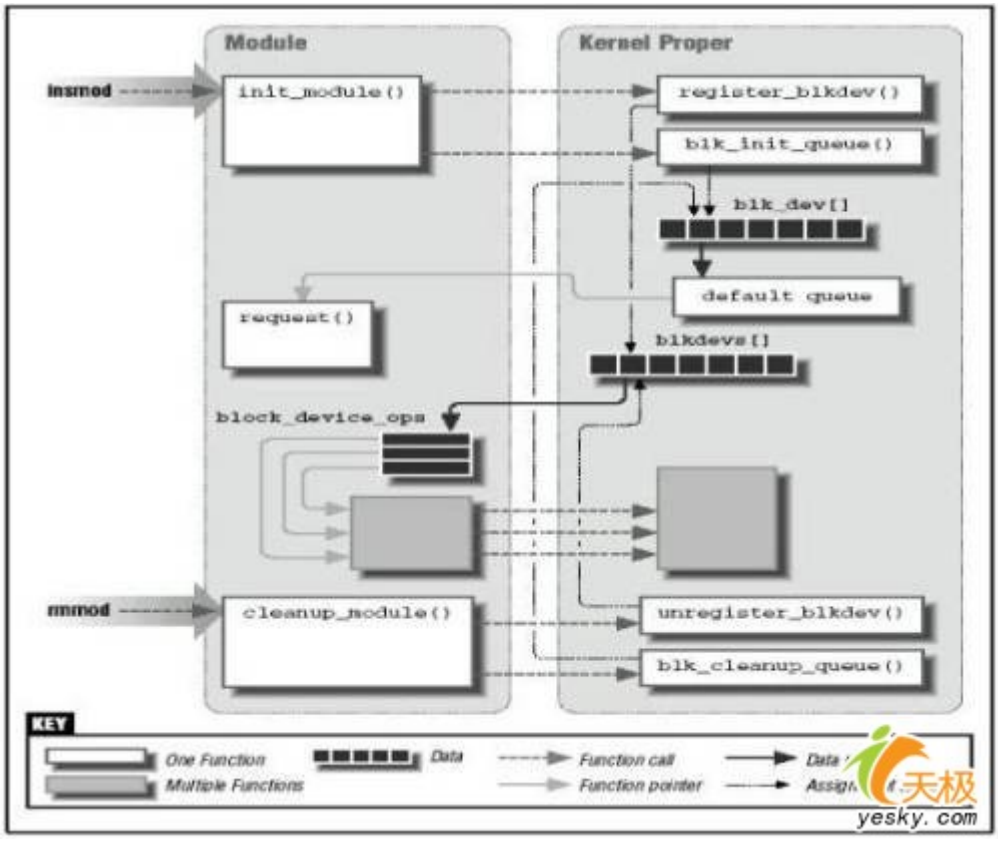
```
struct request_queue
{
	/*
	 * the queue request freelist, one for reads and one for writes
	 */
	struct request_list	rq[2];

	/*
	 * Together with queue_head for cacheline sharing
	 */
	struct list_head	queue_head;
	elevator_t		elevator;

	request_fn_proc		* request_fn;
	merge_request_fn	* back_merge_fn;
	merge_request_fn	* front_merge_fn;
	merge_requests_fn	* merge_requests_fn;
	make_request_fn		* make_request_fn;
	plug_device_fn		* plug_device_fn;
	/*
	 * The queue owner gets to use this for whatever they like.
	 * ll_rw_blk doesn't touch it.
```

```
     */
    void                * queuedata;

    /*
     * This is used to remove the plug when tq_disk runs.
     */
    struct tq_struct      plug_tq;

    /*
     * Boolean that indicates whether this queue is plugged or not.
     */
    char                plugged;

    /*
     * Boolean that indicates whether current_request is active or
     * not.
     */
    char                head_active;

    /*
     * Is meant to protect the queue in the future instead of
     * io_request_lock
     */
    spinlock_t              queue_lock;

    /*
     * Tasks wait here for free request
     */
    wait_queue_head_t wait_for_request;
};
```

下图表征了 blk_dev、blk_dev_struct 和 request_queue 的关系：



下图则表征了块设备的注册和释放过程：



5.小结

本章讲述了 Linux 设备驱动程序的入口函数及驱动程序中的内存申请、中断等，并分别以实例讲述了字符设备及块设备的驱动开发方法。

Trackback: http://tb.donews.net/TrackBack.aspx?PostId=1000099

[点击此处收藏本文] 发表于 2006 年 08 月 14 日 8:07 PM

基于 ARM 的嵌入式 Linux 移植真实体验（5）——应用实例

宋宝华 21cnbao@21cn.com 出处:dev.yesky.com

应用实例的编写实际上已经不属于 Linux 操作系统移植的范畴，但是为了保证本系列文章的完整性，这里提供一系列针对嵌入式 Linux 开发应用程序的实例。

编写 Linux 应用程序要用到如下工具：

（1）编译器：GCC

GCC 是 Linux 平台下最重要的开发工具，它是 GNU 的 C 和 C++编译器，其基本用法为：gcc [options]

[filenames]。
我们应该使用 arm-linux-gcc。
（2）调试器：GDB
gdb 是一个用来调试 C 和 C++程序的强力调试器，我们能通过它进行一系列调试工作，包括设置断点、观查变量、单步等。
我们应该使用 arm-linux-gdb。
（3）Make
GNU Make 的主要工作是读进一个文本文件，称为 makefile。这个文件记录了哪些文件由哪些文件产生，用什么命令来产生。Make 依靠此 makefile 中的信息检查磁盘上的文件，如果目的文件的创建或修改时间比它的一个依靠文件旧的话，make 就执行相应的命令，以便更新目的文件。
Makefile 中的编译规则要相应地使用 arm-linux-版本。
（4）代码编辑
可以使用传统的 vi 编辑器，但最好采用 emacs 软件，它具备语法高亮、版本控制等附带功能。
在宿主机上用上述工具完成应用程序的开发后，可以通过如下途径将程序下载到目标板上运行：
（1）通过串口通信协议 rz 将程序下载到目标板的文件系统中（感谢 Linux 提供了 rz 这样的一个命令）；
（2）通过 ftp 通信协议从宿主机上的 ftp 目录里将程序下载到目标板的文件系统中；
（3）将程序拷入 U 盘，在目标机上 mount　U 盘，运行 U 盘中的程序；
（4）如果目标机 Linux 使用 NFS 文件系统，则可以直接将程序拷入到宿主机相应的目录内，在目标机 Linux 中可以直接使用。

1. 文件编程

Linux 的文件操作 API 涉及到创建、打开、读写和关闭文件。
创建
```
int creat(const char *filename, mode_t mode);
```
参数 mode 指定新建文件的存取权限，它同 umask 一起决定文件的最终权限（mode&umask），其中 umask 代表了文件在创建时需要去掉的一些存取权限。umask 可通过系统调用 umask()来改变：
```
int umask(int newmask);
```
该调用将 umask 设置为 newmask，然后返回旧的 umask，它只影响读、写和执行权限。
打开
```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```
读写
在文件打开以后，我们才可对文件进行读写了，Linux 中提供文件读写的系统调用是 read、write 函数：
```
int read(int fd, const void *buf, size_t length);
int write(int fd, const void *buf, size_t length);
```
其中参数 buf 为指向缓冲区的指针，length 为缓冲区的大小（以字节为单位）。函数 read()实现从文件描述符 fd 所指定的文件中读取 length 个字节到 buf 所指向的缓冲区中，返回值为实际读取的字节数。函数 write 实现将把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中，返回值为实际写入的字节数。
以 O_CREAT 为标志的 open 实际上实现了文件创建的功能，因此，下面的函数等同 creat()函数：
```
int open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```
定位
对于随机文件，我们可以随机的指定位置读写，使用如下函数进行定位：
```
int lseek(int fd, offset_t offset, int whence);
```

lseek()将文件读写指针相对 whence 移动 offset 个字节。操作成功时，返回文件指针相对于文件头的位置。参数 whence 可使用下述值：
SEEK_SET：相对文件开头
SEEK_CUR：相对文件读写指针的当前位置
SEEK_END：相对文件末尾
offset 可取负值，例如下述调用可将文件指针相对当前位置向前移动 5 个字节：

```
lseek(fd, -5, SEEK_CUR);
```

由于 lseek 函数的返回值为文件指针相对于文件头的位置，因此下列调用的返回值就是文件的长度：

```
lseek(fd, 0, SEEK_END);
```

关闭
只要调用 close 就可以了，其中 fd 是我们要关闭的文件描述符：

```
int close(int fd);
```

下面我们来编写一个应用程序，在当前目录下创建用户可读写文件“example.txt”，在其中写入“Hello World”，关闭文件，再次打开它，读取其中的内容并输出在屏幕上：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define LENGTH 100
main()
{
 int fd, len;
 char str[LENGTH];

 fd = open("hello.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); /* 创建并打开文件 */
 if (fd)
 {
 write(fd, "Hello, Software Weekly", strlen("Hello, software weekly")); /* 写入 Hello, software weekly 字符串 */
 close(fd);
 }

 fd = open("hello.txt", O_RDWR);
 len = read(fd, str, LENGTH); /* 读取文件内容 */
 str[len] = '\0';
 printf("%s\n", str);
 close(fd);
}
```

2. 进程控制 / 通信编程

进程控制中主要涉及到进程的创建、睡眠和退出等，在 Linux 中主要提供了 fork、exec、clone 的进程创建方法，sleep 的进程睡眠和 exit 的进程退出调用，另外 Linux 还提供了父进程等待子进程结束的系统调用 wait。

fork

对于没有接触过 Unix/Linux 操作系统的人来说，fork 是最难理解的概念之一，因为它执行一次却返回两个值，

以前“闻所未闻”。先看下面的程序：

```
int main()
{
 int i;
 if (fork() == 0)
 {
    for (i = 1; i < 3; i++)
      printf("This is child process\n");
 }
 else
 {
    for (i = 1; i < 3; i++)
      printf("This is parent process\n");
 }
}
```

执行结果为：

```
This is child process
This is child process
This is parent process
This is parent process
```

fork 在英文中是“分叉”的意思，一个进程在运行中，如果使用了 fork，就产生了另一个进程，于是进程就“分叉”了。当前进程为父进程，通过 fork()会产生一个子进程。对于父进程，fork 函数返回子程序的进程号而对于子程序，fork 函数则返回零，这就是一个函数返回两次的本质。

exec

在 Linux 中可使用 exec 函数族，包含多个函数（execl、execlp、execle、execv、execve 和 execvp），被用于启动一个指定路径和文件名的进程。exec 函数族的特点体现在：某进程一旦调用了 exec 类函数，正在执行的程序就被干掉了，系统把代码段替换成新的程序（由 exec 类函数执行）的代码，并且原有的数据段和堆栈段也被废弃，新的数据段与堆栈段被分配，但是进程号却被保留。也就是说，exec 执行的结果为：系统认为正在执行的还是原先的进程，但是进程对应的程序被替换了。

fork 函数可以创建一个子进程而当前进程不死，如果我们在 fork 的子进程中调用 exec 函数族就可以实现既让父进程的代码执行又启动一个新的指定进程，这很好。fork 和 exec 的搭配巧妙地解决了程序启动另一程序的执行但自己仍继续运行的问题，请看下面的例子：

```
char command[MAX_CMD_LEN];
void main()
{
 int rtn; /* 子进程的返回数值 */
 while (1)
 {
    /* 从终端读取要执行的命令 */
    printf(">");
    fgets(command, MAX_CMD_LEN, stdin);
    command[strlen(command) - 1] = 0;
    if (fork() == 0)
    {
```

```
    /* 子进程执行此命令 */
    execlp(command, command);
    /* 如果 exec 函数返回，表明没有正常执行命令，打印错误信息*/
    perror(command);
    exit(errorno);
  }
  else
  {
    /* 父进程，等待子进程结束，并打印子进程的返回值 */
    wait(&rtn);
    printf(" child process return %d\n", rtn);
  }
 }
}
```

这个函数实现了一个 shell 的功能，它读取用户输入的进程名和参数，并启动对应的进程。

clone

clone 是 Linux2.0 以后才具备的新功能，它较 fork 更强（可认为 fork 是 clone 要实现的一部分），可以使得创建的子进程共享父进程的资源，并且要使用此函数必须在编译内核时设置 clone_actually_works_ok 选项。

clone 函数的原型为：

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

此函数返回创建进程的 PID，函数中的 flags 标志用于设置创建子进程时的相关选项。

来看下面的例子：

```
int variable, fd;

int do_something() {
  variable = 42;
  close(fd);
  _exit(0);
}

int main(int argc, char *argv[]) {
  void **child_stack;
 char tempch;

  variable = 9;
  fd = open("test.file", O_RDONLY);
  child_stack = (void **) malloc(16384);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack, CLONE_VM|CLONE_FILES, NULL);
  sleep(1);   /* 延时以便子进程完成关闭文件操作、修改变量   */

  printf("The variable is now %d\n", variable);
```

```
    if (read(fd, &tempch, 1) < 1) {
        perror("File Read Error");
        exit(1);
    }
    printf("We could read from the file\n");
    return 0;
}
```

运行输出：

```
The variable is now 42
File Read Error
```

程序的输出结果告诉我们，子进程将文件关闭并将变量修改（调用 clone 时用到的 CLONE_VM、CLONE_FILES 标志将使得变量和文件描述符表被共享），父进程随即就感觉到了，这就是 clone 的特点。

sleep

函数调用 sleep 可以用来使进程挂起指定的秒数，该函数的原型为：

```
unsigned int sleep(unsigned int seconds);
```

该函数调用使得进程挂起一个指定的时间，如果指定挂起的时间到了，该调用返回 0；如果该函数调用被信号所打断，则返回剩余挂起的时间数（指定的时间减去已经挂起的时间）。

exit

系统调用 exit 的功能是终止本进程，其函数原型为：

```
void _exit(int status);
```

_exit 会立即终止发出调用的进程，所有属于该进程的文件描述符都关闭。参数 status 作为退出的状态值返回父进程，在父进程中通过系统调用 wait 可获得此值。

wait

wait 系统调用包括：

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait 的作用为发出调用的进程只要有子进程，就睡眠到它们中的一个终止为止； waitpid 等待由参数 pid 指定的子进程退出。

Linux 的进程间通信（IPC，InterProcess Communication）通信方法有管道、消息队列、共享内存、信号量、套接口等。套接字通信并不为 Linux 所专有，在所有提供了 TCP/IP 协议栈的操作系统中几乎都提供了 socket，而所有这样操作系统，对套接字的编程方法几乎是完全一样的。管道分为有名管道和无名管道，无名管道只能用于亲属进程之间的通信，而有名管道则可用于无亲属关系的进程之间；消息队列用于运行于同一台机器上的进程间通信，与管道相似；共享内存通常由一个进程创建，其余进程对这块内存区进行读写；信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。

下面是一个使用信号量的例子，该程序创建一个特定的 IPC 结构的关键字和一个信号量，建立此信号量的索引，修改索引指向的信号量的值，最后清除信号量：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>
void main()
{
 key_t unique_key; /*  定义一个 IPC 关键字*/
 int id;
```

```
struct sembuf lock_it;
union semun options;
int i;

unique_key = ftok(".", 'a'); /* 生成关键字，字符'a'是一个随机种子*/
/* 创建一个新的信号量集合*/
id = semget(unique_key, 1, IPC_CREAT | IPC_EXCL | 0666);
printf("semaphore id=%d\n", id);
options.val = 1; /*设置变量值*/
semctl(id, 0, SETVAL, options); /*设置索引 0 的信号量*/

/*打印出信号量的值*/
i = semctl(id, 0, GETVAL, 0);
printf("value of semaphore at index 0 is %d\n", i);

/*下面重新设置信号量*/
lock_it.sem_num = 0; /*设置哪个信号量*/
lock_it.sem_op = - 1; /*定义操作*/
lock_it.sem_flg = IPC_NOWAIT; /*操作方式*/
if (semop(id, &lock_it, 1) == - 1)
{
   printf("can not lock semaphore.\n");
   exit(1);
}

i = semctl(id, 0, GETVAL, 0);
printf("value of semaphore at index 0 is %d\n", i);

/*清除信号量*/
semctl(id, 0, IPC_RMID, 0);
}
```

3. 线程控制 / 通信编程

Linux 本身只有进程的概念，而其所谓的“线程”本质上在内核里仍然是进程。大家知道，进程是资源分配的单位，同一进程中的多个线程共享该进程的资源（如作为共享内存的全局变量）。Linux 中所谓的“线程”只是在被创建的时候“克隆”(clone)了父进程的资源，因此，clone 出来的进程表现为“线程”。Linux 中最流行的线程机制为 LinuxThreads，它实现了一种 Posix 1003.1c “pthread”标准接口。
线程之间的通信涉及同步和互斥，互斥体的用法为：

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); //按缺省的属性初始化互斥体变量 mutex
pthread_mutex_lock(&mutex); // 给互斥体变量加锁
... //临界资源
phtread_mutex_unlock(&mutex); // 给互斥体变量解锁
```

同步就是线程等待某个事件的发生。只有当等待的事件发生线程才继续执行，否则线程挂起并放弃处理器。当

多个线程协作时，相互作用的任务必须在一定的条件下同步。Linux 下的 C 语言编程有多种线程同步机制，最典型的是条件变量(condition variable)。而在头文件 semaphore.h 中定义的信号量则完成了互斥体和条件变量的封装，按照多线程程序设计中访问控制机制，控制对资源的同步访问，提供程序设计人员更方便的调用接口。下面的生产者/消费者问题说明了 Linux 线程的控制和通信：

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16
struct prodcons
{
 int buffer[BUFFER_SIZE];
 pthread_mutex_t lock;
 int readpos, writepos;
 pthread_cond_t notempty;
 pthread_cond_t notfull;
};
/* 初始化缓冲区结构 */
void init(struct prodcons *b)
{
 pthread_mutex_init(&b->lock, NULL);
 pthread_cond_init(&b->notempty, NULL);
 pthread_cond_init(&b->notfull, NULL);
 b->readpos = 0;
 b->writepos = 0;
}
/* 将产品放入缓冲区,这里是存入一个整数*/
void put(struct prodcons *b, int data)
{
 pthread_mutex_lock(&b->lock);
 /* 等待缓冲区未满*/
 if ((b->writepos + 1) % BUFFER_SIZE == b->readpos)
 {
    pthread_cond_wait(&b->notfull, &b->lock);
 }
 /* 写数据,并移动指针 */
 b->buffer[b->writepos] = data;
 b->writepos++;
 if (b->writepos > = BUFFER_SIZE)
    b->writepos = 0;
 /* 设置缓冲区非空的条件变量*/
 pthread_cond_signal(&b->notempty);
 pthread_mutex_unlock(&b->lock);
}

/* 从缓冲区中取出整数*/
```

```
int get(struct prodcons *b)
{
 int data;
 pthread_mutex_lock(&b->lock);
 /* 等待缓冲区非空*/
 if (b->writepos == b->readpos)
 {
    pthread_cond_wait(&b->notempty, &b->lock);
 }
 /* 读数据,移动读指针*/
 data = b->buffer[b->readpos];
 b->readpos++;
 if (b->readpos > = BUFFER_SIZE)
    b->readpos = 0;
 /* 设置缓冲区未满的条件变量*/
 pthread_cond_signal(&b->notfull);
 pthread_mutex_unlock(&b->lock);
 return data;
}

/* 测试:生产者线程将 1 到 10000 的整数送入缓冲区,消费者线
程从缓冲区中获取整数,两者都打印信息*/
#define OVER ( - 1)
struct prodcons buffer;
void *producer(void *data)
{
 int n;
 for (n = 0; n < 10000; n++)
 {
    printf("%d --->\n", n);
    put(&buffer, n);
 } put(&buffer, OVER);
 return NULL;
}

void *consumer(void *data)
{
 int d;
 while (1)
 {
    d = get(&buffer);
    if (d == OVER)
      break;
    printf("--->%d \n", d);
```

```
 }
 return NULL;
}

int main(void)
{
 pthread_t th_a, th_b;
 void *retval;
 init(&buffer);
 /* 创建生产者和消费者线程*/
 pthread_create(&th_a, NULL, producer, 0);
 pthread_create(&th_b, NULL, consumer, 0);
 /* 等待两个线程结束*/
 pthread_join(th_a, &retval);
 pthread_join(th_b, &retval);
 return 0;
}
```

4.小结

本章主要给出了 Linux 平台下文件、进程控制与通信、线程控制与通信的编程实例。至此，一个完整的，涉及硬件原理、Bootloader、操作系统及文件系统移植、驱动程序开发及应用程序编写的嵌入式 Linux 系列讲解就全部结束了。

深入浅出 Linux 设备驱动编程

宋宝华 21cnbao@21cn.com yesky

1. Linux 内核模块

Linux 设备驱动属于内核的一部分，Linux 内核的一个模块可以以两种方式被编译和加载：
（1）直接编译进 Linux 内核，随同 Linux 启动时加载；
（2）编译成一个可加载和删除的模块，使用 insmod 加载（modprobe 和 insmod 命令类似，但依赖于相关的配置文件），rmmod 删除。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。
下面我们给出一个内核模块的例子：

```
#include <linux/module.h> //所有模块都需要的头文件
#include <linux/init.h>     // init&exit 相关宏
MODULE_LICENSE("GPL");

static int __init hello_init (void)
```

```
{
 printk("Hello module init\n");
 return 0;
}

static void __exit hello_exit (void)
{
 printk("Hello module exit\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

分析上述程序，发现一个 Linux 内核模块需包含模块初始化和模块卸载函数，前者在 insmod 的时候运行，后者在 rmmod 的时候运行。初始化与卸载函数必须在宏 module_init 和 module_exit 使用前定义，否则会出现编译错误。

程序中的 MODULE_LICENSE("GPL")用于声明模块的许可证。

如果要把上述程序编译为一个运行时加载和删除的模块，则编译命令为：

```
gcc –D__KERNEL__ -DMODULE –DLINUX –I /usr/local/src/linux2.4/include -c –o hello.o hello.c
```

由此可见，Linux 内核模块的编译需要给 gcc 指示–D__KERNEL__ -DMODULE –DLINUX 参数。-I 选项跟着 Linux 内核源代码中 Include 目录的路径。

下列命令将可加载 hello 模块：

```
insmod ./hello.o
```

下列命令完成相反过程：

```
rmmod hello
```

如果要将其直接编译入 Linux 内核，则需要将源代码文件拷贝入 Linux 内核源代码的相应路径里，并修改 Makefile。

我们有必要补充一下 Linux 内核编程的一些基本知识：

内存

在 Linux 内核模式下，我们不能使用用户态的 malloc()和 free()函数申请和释放内存。进行内核编程时，最常用的内存申请和释放函数为在 include/linux/kernel.h 文件中声明的 kmalloc()和 kfree()，其原型为：

```
void *kmalloc(unsigned int len, int priority);
void kfree(void *__ptr);
```

kmalloc 的 priority 参数通常设置为 GFP_KERNEL，如果在中断服务程序里申请内存则要用 GFP_ATOMIC 参数，因为使用 GFP_KERNEL 参数可能会引起睡眠，不能用于非进程上下文中（在中断中是不允许睡眠的）。由于内核态和用户态使用不同的内存定义，所以二者之间不能直接访问对方的内存。而应该使用 Linux 中的用户和内核态内存交互函数（这些函数在 include/asm/uaccess.h 中被声明）：

```
unsigned long copy_from_user(void *to, const void *from, unsigned long n);
unsigned long copy_to_user (void * to, void * from, unsigned long len);
```

copy_from_user、copy_to_user 函数返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。include/asm/uaccess.h 中定义的 put_user 和 get_user 用于内核空间和用户空间的单值交互（如 char、int、long）。

这里给出的仅仅是关于内核中内存管理的皮毛，关于 Linux 内存管理的更多细节知识，我们会在本文第 9 节《内存与 I/O 操作》进行更加深入地介绍。

输出

在内核编程中，我们不能使用用户态 C 库函数中的 printf()函数输出信息，而只能使用 printk()。但是，内核中 printk()函数的设计目的并不是为了和用户交流，它实际上是内核的一种日志机制，用来记录下日志信息或者给出警告提示。
每个 printk 都会有个优先级，内核一共有 8 个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。如果优先级数字比 int console_loglevel 变量小的话，消息就会打印到控制台上。如果 syslogd 和 klogd 守护进程在运行的话，则不管是否向控制台输出，消息都会被追加进/var/log/messages 文件。klogd 只处理内核消息，syslogd 处理其他系统消息，比如应用程序。
模块参数
2.4 内核下，include/linux/module.h 中定义的宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。
在装载内核模块时，用户可以向模块传递一些参数：
```
insmod modname var=value
```
如果用户未指定参数，var 将使用模块内定义的缺省值。

2.字符设备驱动程序

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得 Windows 的设备操作犹如文件一般。在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作，如 open ()、close ()、read ()、write () 等。
Linux 主要将设备分为二类：字符设备和块设备。字符设备是指设备发送和接收数据以字符的形式进行；而块设备则以整个数据缓冲区的形式进行。字符设备的驱动相对比较简单。
下面我们来假设一个非常简单的虚拟字符设备：这个设备中只有一个 4 个字节的全局变量 int global_var，而这个设备的名字叫做“gobalvar”。对“gobalvar”设备的读写等操作即是对其中全局变量 global_var 的操作。
驱动程序是内核的一部分，因此我们需要给其添加模块初始化函数，该函数用来完成对所控设备的初始化工作，并调用 register_chrdev() 函数注册字符设备：
```
static int __init gobalvar_init(void)
{
 if (register_chrdev(MAJOR_NUM, " gobalvar ", &gobalvar_fops))
    {
    //…注册失败
    }
 else
    {
    //…注册成功
    }
}
```
其中，register_chrdev 函数中的参数 MAJOR_NUM 为主设备号,“gobalvar”为设备名，gobalvar_fops 为包含基本函数入口点的结构体，类型为 file_operations。当 gobalvar 模块被加载时，gobalvar_init 被执行，它将调用内核函数 register_chrdev，把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。
与模块初始化函数对应的就是模块卸载函数，需要调用 register_chrdev()的“反函数” unregister_chrdev()：
```
static void __exit gobalvar_exit(void)
```

```
{
 if (unregister_chrdev(MAJOR_NUM, " gobalvar "))
    {
    //…卸载失败
    }
 else
    {
    //…卸载成功
    }
}
```

随着内核不断增加新的功能，file_operations 结构体已逐渐变得越来越大，但是大多数的驱动程序只是利用了其中的一部分。对于字符设备来说，要提供的主要入口有：open ()、release ()、read ()、write ()、ioctl ()、llseek()、poll()等。

open()函数 对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open () 函数：

```
int (*open)(struct inode * ,struct file *);
```

其中参数 inode 为设备特殊文件的 inode（索引结点）结构的指针，参数 file 是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0 表示成功，负数表示存在错误) 等；

release()函数 当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数：

```
void (*release) (struct inode * ,struct file *) ;
```

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

read()函数 当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read() 函数：

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
```

用来从设备中读取数据。当该函数指针被赋为 NULL 值时，将导致 read 系统调用出错并返回-EINVAL（“Invalid argument，非法参数”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

globalvar_read 函数中内核空间与用户空间的内存交互需要借助第 2 节所介绍的函数：

```
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
 …
 copy_to_user(buf, &global_var, sizeof(int));
 …
}
```

write( ) 函数 当设备特殊文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

向设备发送数据。如果没有这个函数，write 系统调用会向调用程序返回一个-EINVAL。如果返回值非负，则表示成功写入的字节数。

globalvar_write 函数中内核空间与用户空间的内存交互需要借助第 2 节所介绍的函数：

```
static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
 *off)
{
 …
 copy_from_user(&global_var, buf, sizeof(int));
```

```
...
}
```
ioctl() 函数 该函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息，函数原型为：
```
int (*ioctl) (struct inode * ,struct file * ,unsigned int ,unsigned long);
```
unsigned int 参数为设备驱动程序要执行的命令的代码，由用户自定义，unsigned long 参数为相应的命令提供参数，类型可以是整型、指针等。如果设备不提供 ioctl 入口点，则对于任何内核未预先定义的请求，ioctl 系统调用将返回错误（-ENOTTY，“No such ioctl fordevice，该设备无此 ioctl 命令”）。如果该设备方法返回一个非负值，那么该值会被返回给调用程序以表示调用成功。
llseek()函数该函数用来修改文件的当前读写位置，并将新位置作为（正的）返回值返回，原型为：
```
loff_t (*llseek) (struct file *, loff_t, int);
```
poll()函数 poll 方法是 poll 和 select 这两个系统调用的后端实现，用来查询设备是否可读或可写，或是否处于某种特殊状态，原型为：
```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```
我们将在“设备的阻塞与非阻塞操作”一节对该函数进行更深入的介绍。
设备“gobalvar”的驱动程序的这些函数应分别命名为 gobalvar_open、gobalvar_ release、gobalvar_read、gobalvar_write、gobalvar_ioctl，因此设备“gobalvar”的基本入口点结构变量 gobalvar_fops 赋值如下：
```
struct file_operations gobalvar_fops = {
read: gobalvar_read,
write: gobalvar_write,
};
```
上述代码中对 gobalvar_fops 的初始化方法并不是标准 C 所支持的，属于 GNU 扩展语法。
完整的 globalvar.c 文件源代码如下：
```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM 254    //主设备号

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);

//初始化字符设备驱动的 file_operations 结构体
struct file_operations globalvar_fops =
{
 read: globalvar_read, write: globalvar_write,
};
static int global_var = 0;    //“globalvar”设备的全局变量

static int __init globalvar_init(void)
```

```
{
 int ret;

 //注册设备驱动
 ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
 if (ret)
 {
    printk("globalvar register failure");
 }
 else
 {
    printk("globalvar register success");
 }
 return ret;
}

static void __exit globalvar_exit(void)
{
 int ret;

 //注销设备驱动
 ret = unregister_chrdev(MAJOR_NUM, "globalvar");
 if (ret)
 {
    printk("globalvar unregister failure");
 }
 else
 {
    printk("globalvar unregister success");
 }
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
 //将 global_var 从内核空间复制到用户空间
 if (copy_to_user(buf, &global_var, sizeof(int)))
 {
    return - EFAULT;
 }
 return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
 *off)
```

```
{
 //将用户空间的数据复制到内核空间的 global_var
 if (copy_from_user(&global_var, buf, sizeof(int)))
 {
    return - EFAULT;
 }
 return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

运行

```
gcc –D__KERNEL__ -DMODULE –DLINUX –I /usr/local/src/linux2.4/include -c –o globalvar.o globalvar.c
```

编译代码，运行

```
inmod globalvar.o
```

加载 globalvar 模块，再运行

```
cat /proc/devices
```

发现其中多出了“254 globalvar”一行，如下图：

接着我们可以运行：

```
mknod /dev/globalvar c 254 0
```

创建设备节点，用户进程通过/dev/globalvar 这个路径就可以访问到这个全局变量虚拟设备了。我们写一个用户态的程序 globalvartest.c 来验证上述设备：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd, num;
 //打开“/dev/globalvar”
 fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
 if (fd != -1 )
 {
  //初次读 globalvar
    read(fd, &num, sizeof(int));
    printf("The globalvar is %d\n", num);

  //写 globalvar
    printf("Please input the num written to globalvar\n");
    scanf("%d", &num);
    write(fd, &num, sizeof(int));

  //再次读 globalvar
```

```
    read(fd, &num, sizeof(int));
    printf("The globalvar is %d\n", num);

    //关闭"/dev/globalvar"
    close(fd);
 }
 else
 {
    printf("Device open failure\n");
 }
}
```

编译上述文件：

```
gcc –o globalvartest.o globalvartest.c
```

运行

```
./globalvartest.o
```

可以发现"globalvar"设备可以正确的读写。

3. 设备驱动中的并发控制

在驱动程序中，当多个线程同时访问相同的资源时（驱动程序中的全局变量是一种典型的共享资源），可能会引发"竞态"，因此我们必须对共享资源进行并发控制。Linux 内核中解决并发控制的最常用方法是自旋锁与信号量（绝大多数时候作为互斥锁使用）。

自旋锁与信号量"类似而不类"，类似说的是它们功能上的相似性，"不类"指代它们在本质和实现机理上完全不一样，不属于一类。

自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环查看是否该自旋锁的保持者已经释放了锁，"自旋"就是"在原地打转"。而信号量则引起调用者睡眠，它把进程从运行队列上拖出去，除非获得锁。这就是它们的"不类"。

但是，无论是信号量，还是自旋锁，在任何时刻，最多只能有一个保持者，即在任何时刻最多只能有一个执行单元获得锁。这就是它们的"类似"。

鉴于自旋锁与信号量的上述特点，一般而言，自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用；信号量适合于保持时间较长的情况，会只能在进程上下文使用。如果被保护的共享资源只在进程上下文访问，则可以以信号量来保护该共享资源，如果对共享资源的访问时间非常短，自旋锁也是好的选择。但是，如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

与信号量相关的 API 主要有：

定义信号量

```
struct semaphore sem;
```

初始化信号量

```
void sema_init (struct semaphore *sem, int val);
```

该函数初始化信号量，并设置信号量 sem 的值为 val

```
void init_MUTEX (struct semaphore *sem);
```

该函数用于初始化一个互斥锁，即它把信号量 sem 的值设置为 1，等同于 sema_init (struct semaphore *sem, 1)；

```
void init_MUTEX_LOCKED (struct semaphore *sem);
```

该函数也用于初始化一个互斥锁，但它把信号量 sem 的值设置为 0，等同于 sema_init (struct semaphore *sem, 0)；

获得信号量

```
void down(struct semaphore * sem);
```

该函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文使用；

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 down 类似，不同之处为，down 不能被信号打断，但 down_interruptible 能被信号打断；

```
int down_trylock(struct semaphore * sem);
```

该函数尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文使用。

释放信号量

```
void up(struct semaphore * sem);
```

该函数释放信号量 sem，唤醒等待者。

与自旋锁相关的 API 主要有：

定义自旋锁

```
spinlock_t spin;
```

初始化自旋锁

```
spin_lock_init(lock)
```

该宏用于动态初始化自旋锁 lock

获得自旋锁

```
spin_lock(lock)
```

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放；

```
spin_trylock(lock)
```

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，实际上不再“在原地打转”；

释放自旋锁

```
spin_unlock(lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用；

除此之外，还有一组自旋锁使用于中断情况下的 API。

下面进入对并发控制的实战。首先，在 globalvar 的驱动程序中，我们可以通过信号量来控制对 int global_var 的并发访问，下面给出源代码：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM 254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);
```

```
struct file_operations globalvar_fops =
{
 read: globalvar_read, write: globalvar_write,
};
static int global_var = 0;
static struct semaphore sem;

static int __init globalvar_init(void)
{
 int ret;
 ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
 if (ret)
 {
    printk("globalvar register failure");
 }
 else
 {
    printk("globalvar register success");
    init_MUTEX(&sem);
 }
 return ret;
}

static void __exit globalvar_exit(void)
{
 int ret;
 ret = unregister_chrdev(MAJOR_NUM, "globalvar");
 if (ret)
 {
    printk("globalvar unregister failure");
 }
 else
 {
    printk("globalvar unregister success");
 }
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
 //获得信号量
 if (down_interruptible(&sem))
 {
    return - ERESTARTSYS;
 }
```

```
 //将 global_var 从内核空间复制到用户空间
 if (copy_to_user(buf, &global_var, sizeof(int)))
 {
     up(&sem);
     return - EFAULT;
 }

 //释放信号量
 up(&sem);

 return sizeof(int);
}

ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
 *off)
{
 //获得信号量
 if (down_interruptible(&sem))
 {
     return - ERESTARTSYS;

 }

 //将用户空间的数据复制到内核空间的 global_var
 if (copy_from_user(&global_var, buf, sizeof(int)))
 {
     up(&sem);
     return - EFAULT;
 }

 //释放信号量
 up(&sem);

 return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

接下来，我们给 globalvar 的驱动程序增加 open()和 release()函数，并在其中借助自旋锁来保护对全局变量 int globalvar_count（记录打开设备的进程数）的访问来实现设备只能被一个进程打开（必须确保 globalvar_count 最多只能为 1）：

```
#include <linux/module.h>
#include <linux/init.h>
```

```
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM 254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);
static int globalvar_open(struct inode *inode, struct file *filp);
static int globalvar_release(struct inode *inode, struct file *filp);

struct file_operations globalvar_fops =
{
 read: globalvar_read, write: globalvar_write, open: globalvar_open, release:
    globalvar_release,
};

static int global_var = 0;
static int globalvar_count = 0;
static struct semaphore sem;
static spinlock_t spin = SPIN_LOCK_UNLOCKED;


static int __init globalvar_init(void)
{
 int ret;
 ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
 if (ret)
 {
    printk("globalvar register failure");
 }
 else
 {
    printk("globalvar register success");
    init_MUTEX(&sem);
 }
 return ret;
}

static void __exit globalvar_exit(void)
{
 int ret;
```

```
 ret = unregister_chrdev(MAJOR_NUM, "globalvar");
 if (ret)
 {
    printk("globalvar unregister failure");
 }
 else
 {
    printk("globalvar unregister success");
 }
}

static int globalvar_open(struct inode *inode, struct file *filp)
{

 //获得自选锁
 spin_lock(&spin);

 //临界资源访问
 if (globalvar_count)
 {
    spin_unlock(&spin);
    return - EBUSY;
 }
 globalvar_count++;

 //释放自选锁
 spin_unlock(&spin);

 return 0;
}

static int globalvar_release(struct inode *inode, struct file *filp)
{
 globalvar_count--;
 return 0;
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
 *off)
{
 if (down_interruptible(&sem))
 {
    return - ERESTARTSYS;
 }
```

```
 if (copy_to_user(buf, &global_var, sizeof(int)))
 {
    up(&sem);
    return - EFAULT;
 }
 up(&sem);
 return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len,
 loff_t *off)
{
 if (down_interruptible(&sem))
 {
    return - ERESTARTSYS;

 }
 if (copy_from_user(&global_var, buf, sizeof(int)))
 {
    up(&sem);
    return - EFAULT;
 }
 up(&sem);
 return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

为了上述驱动程序的效果，我们启动两个进程分别打开/dev/globalvar。在两个终端中调用./globalvartest.o 测试程序，当一个进程打开/dev/globalvar 后，另外一个进程将打开失败，输出“device open failure”，如下图：

4. 设备的阻塞与非阻塞操作

阻塞操作是指，在执行设备操作时，若不能获得资源，则进程挂起直到满足可操作的条件再进行操作。非阻塞操作的进程在不能进行设备操作时，并不挂起。被挂起的进程进入 sleep 状态，被从调度器的运行队列移走，直到等待的条件被满足。

在 Linux 驱动程序中，我们可以使用等待队列（wait queue）来实现阻塞操作。wait queue 很早就作为一个基本的功能单位出现在 Linux 内核里了，它以队列为基础数据结构，与进程调度机制紧密结合，能够用于实现核心的异步事件通知机制。等待队列可以用来同步对系统资源的访问，上节中所讲述 Linux 信号量在内核中也是由等待队列来实现的。

下面我们重新定义设备“globalvar”，它可以被多个进程打开，但是每次只有当一个进程写入了一个数据之后本进程或其它进程才可以读取该数据，否则一直阻塞。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/wait.h>
#include <asm/semaphore.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM 254

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t*);

struct file_operations globalvar_fops =
{
 read: globalvar_read, write: globalvar_write,
};

static int global_var = 0;
static struct semaphore sem;
static wait_queue_head_t outq;
static int flag = 0;

static int __init globalvar_init(void)
{
 int ret;
 ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);
 if (ret)
 {
    printk("globalvar register failure");
 }
 else
 {
    printk("globalvar register success");
    init_MUTEX(&sem);
    init_waitqueue_head(&outq);
 }
 return ret;
}

static void __exit globalvar_exit(void)
```

```
{
 int ret;
 ret = unregister_chrdev(MAJOR_NUM, "globalvar");
 if (ret)
 {
    printk("globalvar unregister failure");
 }
 else
 {
    printk("globalvar unregister success");
 }
}


static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
 *off)
{
 //等待数据可获得
 if (wait_event_interruptible(outq, flag != 0))
 {
    return - ERESTARTSYS;
 }

 if (down_interruptible(&sem))
 {
    return - ERESTARTSYS;
 }

 flag = 0;
 if (copy_to_user(buf, &global_var, sizeof(int)))
 {
    up(&sem);
    return - EFAULT;
 }

 up(&sem);

 return sizeof(int);
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len,
 loff_t *off)
{
 if (down_interruptible(&sem))
```

```
{
    return - ERESTARTSYS;
}
if (copy_from_user(&global_var, buf, sizeof(int)))
{
    up(&sem);
    return - EFAULT;
}
up(&sem);
flag = 1;
//通知数据可获得
wake_up_interruptible(&outq);

return sizeof(int);
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

编写两个用户态的程序来测试，第一个用于阻塞地读/dev/globalvar，另一个用于写/dev/globalvar。只有当后一个对/dev/globalvar 进行了输入之后，前者的 read 才能返回。

读的程序为：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd, num;

 fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
 if (fd != - 1)
 {
    while (1)
    {
      read(fd, &num, sizeof(int)); //程序将阻塞在此语句，除非有针对 globalvar 的输入
      printf("The globalvar is %d\n", num);

      //如果输入是 0，则退出
      if (num == 0)
      {
        close(fd);
        break;
      }
    }
```

```
 }
 else
 {
    printf("device open failure\n");
 }
}
```
写的程序为：
```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd, num;

 fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
 if (fd != - 1)
 {
    while (1)
    {
      printf("Please input the globalvar:\n");
      scanf("%d", &num);
      write(fd, &num, sizeof(int));

      //如果输入 0，退出
      if (num == 0)
      {
        close(fd);
        break;
      }
    }
 }
 else
 {
    printf("device open failure\n");
 }
}
```
打开两个终端，分别运行上述两个应用程序，发现当在第二个终端中没有输入数据时，第一个终端没有输出（阻塞），每当我们在第二个终端中给 globalvar 输入一个值，第一个终端就会输出这个值，如下图：
关于上述例程，我们补充说一点，如果将驱动程序中的 read 函数改为：
```
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t
 *off)
{
 //获取信号量：可能阻塞
```

```
 if (down_interruptible(&sem))
 {
    return - ERESTARTSYS;
 }

 //等待数据可获得：可能阻塞
 if (wait_event_interruptible(outq, flag != 0))
 {
    return - ERESTARTSYS;
 }
 flag = 0;

 //临界资源访问
 if (copy_to_user(buf, &global_var, sizeof(int)))
 {
    up(&sem);
    return - EFAULT;
 }

 //释放信号量
 up(&sem);

 return sizeof(int);
}
```

即交换 wait_event_interruptible(outq, flag != 0)和 down_interruptible(&sem)的顺序，这个驱动程序将变得不可运行。实际上，当两个可能要阻塞的事件同时出现时，即两个 wait_event 或 down 摆在一起的时候，将变得非常危险，死锁的可能性很大，这个时候我们要特别留意它们的出现顺序。当然，我们应该尽可能地避免这种情况的发生！

+还有一个与设备阻塞与非阻塞访问息息相关的论题，即 select 和 poll，select 和 poll 的本质一样，前者在 BSD Unix 中引入，后者在 System V 中引入。poll 和 select 用于查询设备的状态，以便用户程序获知是否能对设备进行非阻塞的访问，它们都需要设备驱动程序中的 poll 函数支持。

驱动程序中 poll 函数中最主要用到的一个 API 是 poll_wait，其原型如下：

```
void poll_wait(struct file *filp, wait_queue_heat_t *queue, poll_table * wait);
```

poll_wait 函数所做的工作是把当前进程添加到 wait 参数指定的等待列表（poll_table）中。下面我们给 globalvar 的驱动添加一个 poll 函数：

```
static unsigned int globalvar_poll(struct file *filp, poll_table *wait)
{
 unsigned int mask = 0;

 poll_wait(filp, &outq, wait);

 //数据是否可获得？
 if (flag != 0)
 {
```

```
    mask |= POLLIN | POLLRDNORM; //标示数据可获得
 }

 return mask;
}
```

需要说明的是，poll_wait 函数并不阻塞，程序中 poll_wait(filp, &outq, wait)这句话的意思并不是说一直等待 outq 信号量可获得，真正的阻塞动作是上层的 select/poll 函数中完成的。select/poll 会在一个循环中对每个需要监听的设备调用它们自己的 poll 支持函数以使得当前进程被加入各个设备的等待列表。若当前没有任何被监听的设备就绪，则内核进行调度（调用 schedule）让出 cpu 进入阻塞状态，schedule 返回时将再次循环检测是否有操作可以进行，如此反复；否则，若有任意一个设备就绪，select/poll 都立即返回。

我们编写一个用户态应用程序来测试改写后的驱动。程序中要用到 BSD Unix 中引入的 select 函数，其原型为：

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select()监视的读、写和异常处理的文件描述符集合，numfds 的值是需要检查的号码最高的文件描述符加 1。timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 时间后若没有文件描述符准备好则返回。struct timeval 数据结构为：

```
struct timeval
{
    int tv_sec; /* seconds */
    int tv_usec; /* microseconds */
};
```

除此之外，我们还将使用下列 API：

FD_ZERO(fd_set *set)——清除一个文件描述符集；

FD_SET(int fd,fd_set *set)——将一个文件描述符加入文件描述符集中；

FD_CLR(int fd,fd_set *set)——将一个文件描述符从文件描述符集中清除；

FD_ISSET(int fd,fd_set *set)——判断文件描述符是否被置位。

下面的用户态测试程序等待/dev/globalvar 可读，但是设置了 5 秒的等待超时，若超过 5 秒仍然没有数据可读，则输出“No data within 5 seconds”：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
 int fd, num;
 fd_set rfds;
 struct timeval tv;

 fd = open("/dev/globalvar", O_RDWR, S_IRUSR | S_IWUSR);
 if (fd != - 1)
```

```
{
   while (1)
   {
     //查看 globalvar 是否有输入
     FD_ZERO(&rfds);
     FD_SET(fd, &rfds);
     //设置超时时间为 5s
     tv.tv_sec = 5;
     tv.tv_usec = 0;
     select(fd + 1, &rfds, NULL, NULL, &tv);

     //数据是否可获得？
     if (FD_ISSET(fd, &rfds))
     {
       read(fd, &num, sizeof(int));
       printf("The globalvar is %d\n", num);

       //输入为 0，退出
       if (num == 0)
       {
         close(fd);
         break;
       }
     }
     else
       printf("No data within 5 seconds.\n");
   }
}
else
{
   printf("device open failure\n");
}
}
```

开两个终端，分别运行程序：一个对 globalvar 进行写，一个用上述程序对 globalvar 进行读。当我们在写终端给 globalvar 输入一个值后，读终端立即就能输出该值，当我们连续 5 秒没有输入时，“No data within 5 seconds”在读终端被输出.