嵌入式系统程序设计

大连理工大学软件学院
嵌入式教研室
赖晓晨

嵌入式C编程综述

- 软件架构
- 屏幕操作
- 键盘操作
- 内存操作
- 性能优化

一、软件架构

- 模块划分
- 任务模式
- 中断服务程序
- 硬件驱动
- C的面向对象化

1、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。

模块划分的方法

- 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；
- 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；
- 仅在模块内部使用的函数和全局变量需在.c文件开头冠以static关键字声明；
- 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。

一个不好的例子

```
/*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 */
```

结果。。。

一个好的例子

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

/*module1 .c*/
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 */
```

两种类型模块

- 一个嵌入式系统通常包括两类模块：
  - 硬件驱动模块，一种特定硬件对应一个模块；
  - 软件功能模块，其模块的划分应满足低耦合、高内聚的要求。

2、任务模式

- 单任务、多任务
  - 单任务：微观串行、宏观串行
  - 多任务：微观串行、宏观并行

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：
   - 初试化部分硬件设备；
   - 初始化各软件模块；
   - 进入死循环（无限循环），调用各模块的处理函数

无限循环的两种方案

1. while(1)
```
{
}
```
好：一目了然

2. for(;;)
```
{
}
```
不好：含义不明确

3、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识中断服务程序(ISR)，类似于__interrupt
- 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。

中断服务程序的特点

- 中断服务程序需要满足如下要求：
  - 不能返回值；
  - 不能向ISR传递参数；
  - ISR应该尽可能的短小精悍；
  - printf(char * lpFormatString,…)函数会带来性能问题，不能在ISR中采用。

中断服务程序模型

- 在项目的开发中，设计一个队列，在中断服务程序中，只是将中断类型添加入该队列中，在主程序的无限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。

中断服务程序模型

```
typedef struct tagIntQueue      /* 存放中断的队列 */
{
      int intType;              /* 中断类型 */
      struct tagIntQueue *next;
}IntQueue;
IntQueue* lpIntQueueHead;
__interrupt ISRexample ()
{
      int intType;
      intType = GetSystemType(); /* 得到中断类型 */
      QueueAddTail(lpIntQueueHead, intType);
                              /* 在队列尾加入新的中断 */
}
```

主程序模型

```
While(1)
{
    if( !IsIntQueueEmpty() )
    {
      intType = GetFirstInt();
      switch(intType)
      {       case xxx:     /*中断处理部分代码 */
                        …
                        break;
              case xxx:
                        …
                        break;
                   …
      }
    }
}
```

4、硬件驱动模块

- 一个硬件驱动模块通常应包括如下函数：
  - 中断服务程序ISR
  - 硬件初始化
    - 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）；
    - 将中断服务程序入口地址写入中断向量表；

硬件驱动模块（续）

```
/* 设置中断向量表 */
m_myPtr = make_far_pointer(0l);
/* 返回void far型指针void far * */
m_myPtr += ITYPE_UART;
/* ITYPE_UART： uart中断服务程序中断号，
    相对于中断向量表首地址的偏移 */
m_myPtr
/* UART _Isr：UART的中断服务程序首地址 */
```

硬件驱动模块（续）

- 一个硬件驱动模块通常应包括如下函数：
  - 设置CPU针对该硬件的控制线
    - 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；
    - 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）。

硬件驱动模块（续）

- 一个硬件驱动模块通常应包括如下函数：
  - 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；而对于时钟，其驱动模块则需提供获取时间、设置时间等函数。

5、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"。

用C程序模拟一个最简单的"类":

```
#define C_Class struct


C_Class A
{
     C_Class A *A_this;          /* this指针 */
     void (*foo)(C_Class A *A_this);
     int (*parea)(int length, int width);
                                 /* 行为：函数指针 */
     int a;                      /* 数据 */
     int b;
};
```

二、屏幕操作

- 汉字处理
- 系统时间显示
- 定时器
- 菜单操作
- 模拟MessageBox

1、汉字处理

- 嵌入式系统汉字库的特点

　　嵌入式系统中经常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。

　　但是一部手机则通常需要包括较完整的汉字库。

完整字库

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

然后乘上一个汉字字模占用的字节数即可，以16*16点阵字库为例，即：

(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。

仅使用少量汉字

```
# define EX_FONT_WORD(value) (value)
# define EX_FONT_UNICODE_VAL(value) (value)

typedef struct _wide_unicode_font16x16
{
    unsigned char word[3];
    int value; // 内码
    unsigned char data[32]; /* 字模点阵 */
}Unicode;

#define CHINESE_CHAR_NUM   4 /* 汉字数量 */
```

仅使用少量汉字（续）

```
Unicode chinese[CHINESE_CHAR_NUM] =
{
    {
        EX_FONT_WORD("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
        }
    },
    {   ...    }, {   ...    }, {   ...    }
}
```

仅使用少量汉字（续）

- 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。

2、系统时间显示

- 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。

系统时间显示（续）

```
#define BYTE unsigned char
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;
    }
}
```

3、定时器

- 希望显示会闪烁的“:”，用来分隔时间单位。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

思考：定时器的用途

- 请举例说明定时器都在哪些场合得到应用

思考：定时器的用途

- 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；
- 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；
- 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。

思考：定时器的用途

```
void ShowDot()
{
  static BOOL bShowDot = TRUE;
  if(bShowDot)
  {
    showChar(':',xPos,yPos);
  }
  else
  {
    showChar(' ',xPos,yPos);
  }
  bShowDot = ! bShowDot;
}
```

4、菜单操作

- 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。

菜单1　菜单2　菜单3　菜单4

菜单实现方法1

```
/* 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;
    ...
  }
}
```

菜单实现方法1（续）

```
/* 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;
    …
  }
}
```

菜单实现方法2

```
/* 将菜单的属性和操作“封装”在一起，声明菜单项的“类” */
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;
```

菜单实现方法2（续）

```
/* 用结构体数组来实现每一个菜单项 */
static SysMenu menu[MENU_NUM] =
{
  {     “menu1”, 0, 48, menu1OnOk, menu1OnCancel   }
  ,
  {     " menu2", 7, 48, menu2OnOk, menu2OnCancel   }
  ,
  {     " menu3", 7, 48, menu3OnOk, menu3OnCancel   }
  …
};
```

菜单实现方法2（续）

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

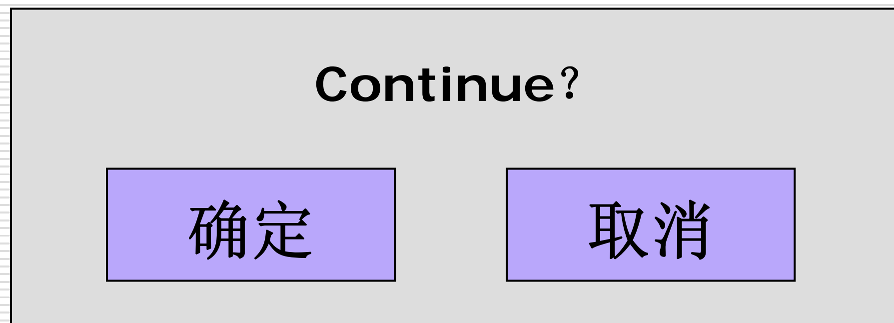
菜单实现方法2（续）

- 结论

　　程序被大大简化了，也开始具有很好的可扩展性！利用了面向对象中的封装思想，使程序结构清晰，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。

5、模拟MessageBox函数

- 在屏幕上绘制带有“确定”或者“确定”、“取消”的消息对话框。



```
******************************************
/* 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;
  }
```

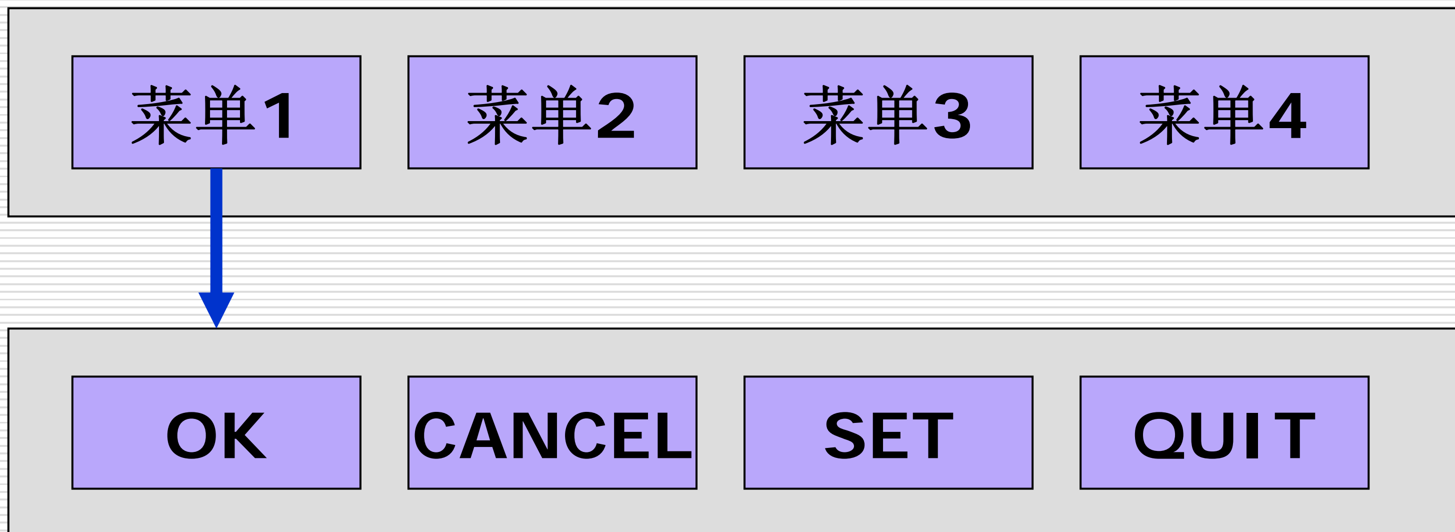
```
DrawRect(0, 0, 239, yPos+High+16+4); /* 绘制外框 */
  /* MessageBox是模式对话框，阻塞运行，等待按键 */
  while( (keyValue != KEY_OK) || (keyValue != KEY_CANCEL) )
  {
    keyValue = getSysKey();
  }
  /* 返回按键类型 */
  if(keyValue== KEY_OK)
  {
    return ID_OK;
  }
  else
  {
    return ID_CANCEL;
  }
}
```

三、键盘操作

- 处理功能键
- 处理数字键
- 整理用户输入

1、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。



win32机制

- **WIN32**编程中用到"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。

嵌入式系统采取的策略

- 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；
- 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；
- 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。

嵌入式系统采取的策略

```
/* 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 */
```

```
/* 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++;            }

    if(i < ELEMENT_NUM)         /* "消息映射" */
    {
        switch(keyValue)
        {
        case OK:
                element[i].OnOk();
                break;
                ...
        }
    }
}
```

2、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：

处理数字键（续）

```
/* 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;


/* 接收用户数字输入的数组 */
InputNum inputElement[NUM_LENGTH];
```

处理数字键（续）

```
extern void onNumKey(BYTE num) /* 数字按键处理函数 */
{
    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();
    }
}
```

3、整理用户输入

```
/* 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);
    }
}
```

整理用户输入（续）

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

思考?

- 如何使效率使更高?

思考?

- 如何使效率使更高?

　　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。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。

四、内存操作

- 数据指针
- 函数指针
- 数组和动态内存
- 关键字**volatile**

1、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。

利用指针读写内存

- 读写内存某地址处内容

```
unsigned char *p = (unsigned char *)0xF000;
*p=11;
```

- p++(或++p)的结果等同于：
  p = p+sizeof(指针p的关联类型)

2、函数指针

- 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；
- 调用函数实际上等同于“跳转指令＋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；
- 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

软启动

- 函数本质是指令集合；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！

```
/* 定义一个无参数、无返回类型的函数指针类型 */
typedef void (*lpFunction) ( );
/* 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)0xFFFF0;
lpReset(); /* 调用函数 */
```

思考：

- 函数本质是指令集合；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！

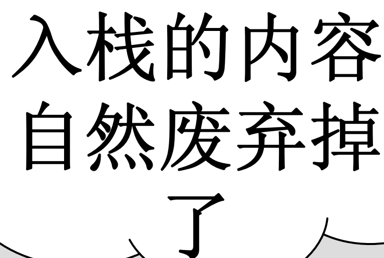
```
/* 定义一个无参数、无返回类型的函数指针
typedef void (*lpFunction) ( );
/* 定义一个函数指针，指向CPU启动后所
执行第一条指令的位置 */
lpFunction lpReset = (lpFunction)0xFFFF0;
lpReset(); /* 调用函数 */
```

此函数调用
如何返回？

回顾一下机器启动过程

- 函数本质是指令集合；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！

```
/* 定义一个无参数、无返回类型的函数指针
typedef void (*lpFunction) ( );
/* 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)0xFFFF0;
lpReset(); /* 调用函数 */
```



3、数组和动态内存

- 动态申请内存的原则是**malloc()**和**free()** 要成对出现

一个不好的例子

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

一个不好的例子

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

用户需要知道 function（）函数实现的细节

应该改为

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


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

原则

- 尽可能的选用数组，而不用动态申请的内存，而且数组不能越界访问
- 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！

4、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;
/*读取I/O空间0x100端口的内容存入a变量*/
a = inWord(0x100);
b = a;
/*再次读取I/O空间0x100端口的内容存入a变量*/
a = inWord (0x100);
c = a;
```

关键字volatile （续）

- 很可能被编译器优化为：

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

- 应改为：volatile int a;

volatile应用场合

- 并行设备的硬件寄存器（如：状态寄存器，上例中的代码属于此类）；
- 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）；
- 多线程应用中被几个任务共享的变量。

五、性能优化

- 用宏来代替函数
- 使用寄存器变量
- 内嵌汇编
- 利用硬件特性
- 活用位操作

1、使用宏来代替函数

- 宏定义“像”函数；
- 宏定义不是函数，因而需要括上所有“参数”；
- 宏定义可能产生副作用。

宏的副作用

- 如下程序段结果无法预料，但是使用函数可以避免这种情况

```
#define MIN(A,B) （（A）<= (B) ? (A) : (B) )
least = MIN(*p++, b);
//展开后为:
( (*p++) <= (b) ? (*p++) : (b) )
```

2、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。

寄存器变量说明

- 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；
- 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）。

3、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

4、利用硬件特性

- 把程序拷贝到ram中运行
- UART设备的buffer占满后再中断
- 尽量采用DMA方式存取数据

5、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。

位操作举例1

```
/* 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);
```

位操作举例2

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

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

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