C语言语法高级应用第2版

**作者: 向仔州**

目录

[结构体字节对齐问题 2](#_Toc159859643)

[用结构体方式接受串口的数据包，结构体对数据内容进行了取名，方便使用 5](#_Toc159859644)

[\_\_attribute\_\_ 用法 10](#_Toc159859645)

[C语言中\_\_attribute\_\_ ((at())绝对定位的应用 10](#_Toc159859646)

[C语言 ，变量位域定义，如 unsigned char ch: 6; 这种用法 11](#_Toc159859647)

[#和##连接符使用 16](#_Toc159859648)

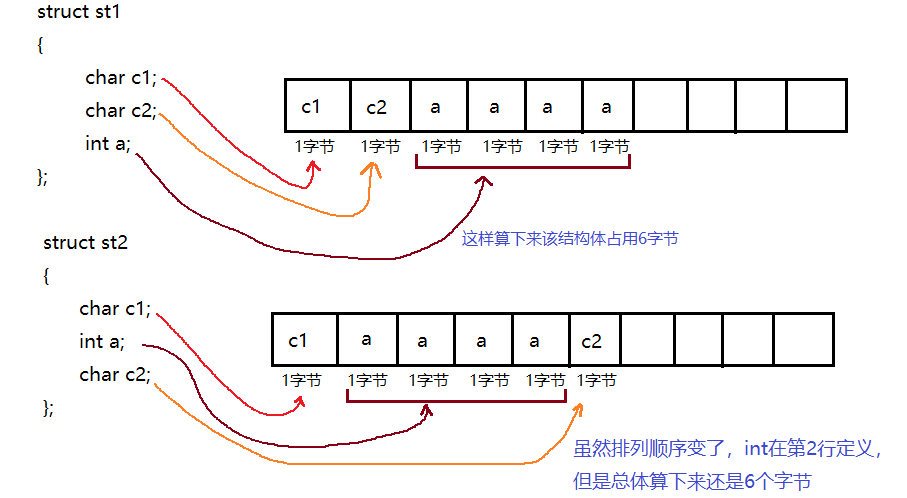
[联合体配合结构体和一维数组的高级应用 18](#_Toc159859649)

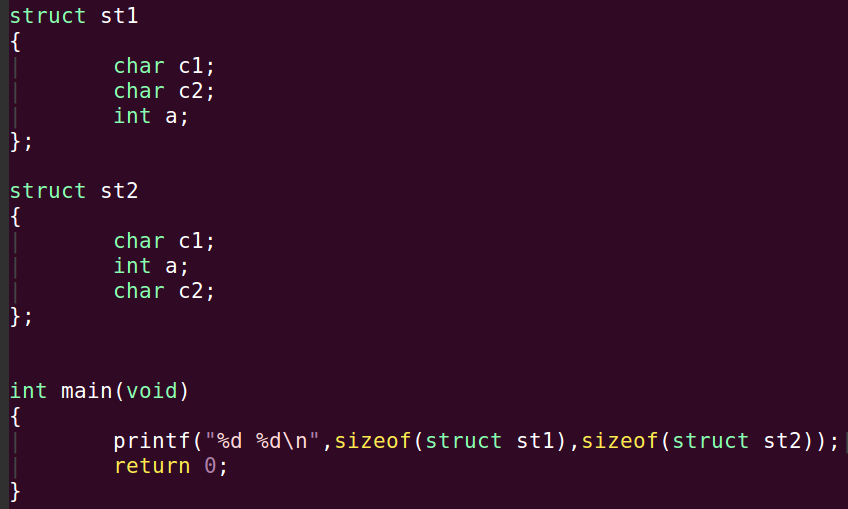
[函数宏(主要是为了降低压栈和弹栈的开销，方便移植。不然直接建立函数来实现更好) 22](#_Toc159859650)

[钩子函数使用方式 25](#_Toc159859651)

[C语言实现MVC模式实现多人开发 26](#_Toc159859652)

## 结构体字节对齐问题



 两个结构体大小不一致

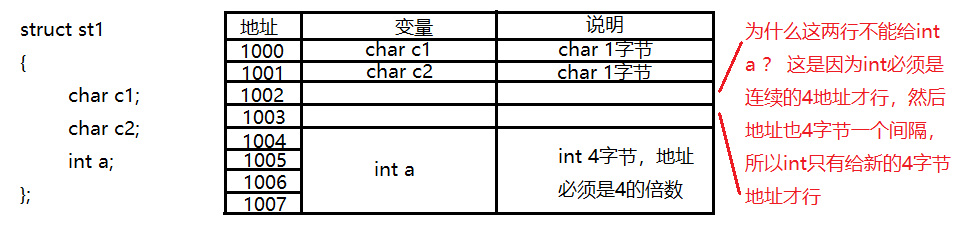
不对啊！！都是6个字节的变量，怎么一个结构体是8字节，一个结构体12字节呢？

难道是结构体内部变量没有按照顺序对齐的原因？

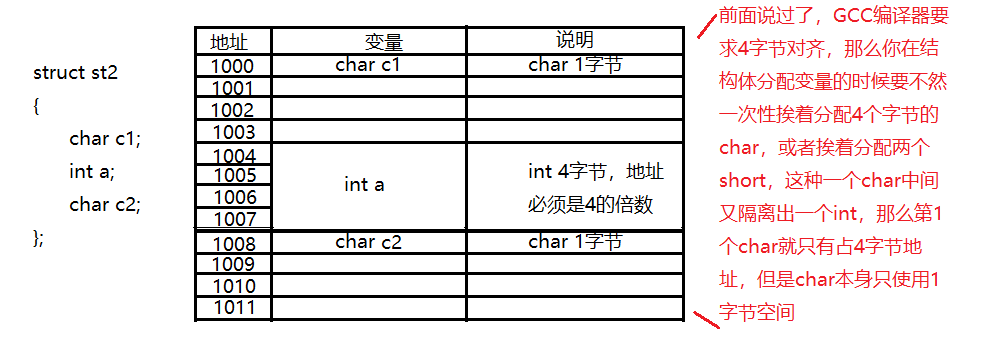
其实影响结构体大小的因素，1.字节对齐(是按照1字节，4字节，8字节还是16字节对齐)?

2.结构体成员变量大小也有关系

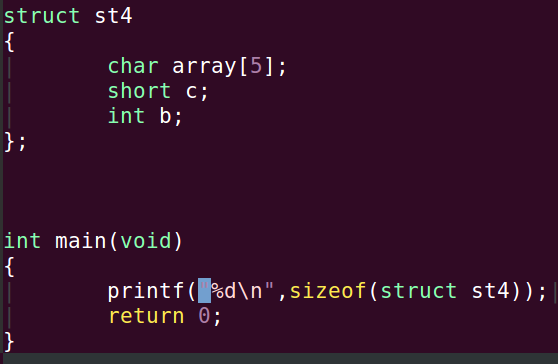
3.不同类型的变量对齐

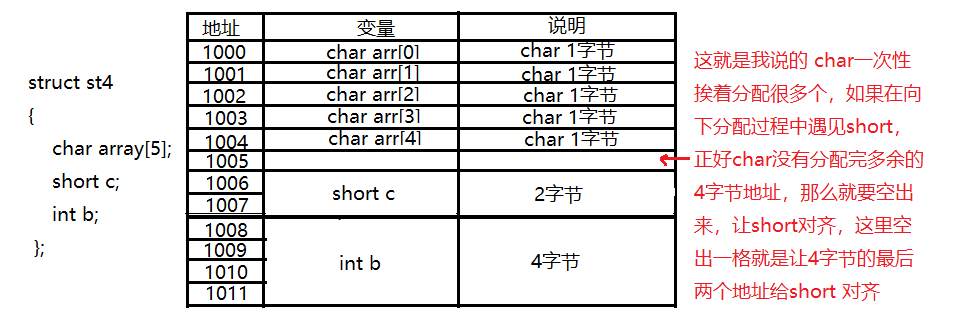
因为GCC编译器默认是4字节对齐，那么int满足4字节，但是int前面的char是1字节，而且地址是4字节一排，4字节一排，第1排被两个char用了，那么就只有让第2排承载int，所以第一排浪费了两个字节空出来。结果就是8字节。 

注意，结构体里面变量定义的先后顺序也是有关系的，请看下面



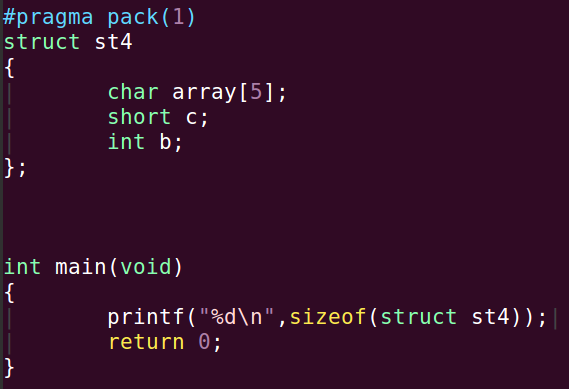
所以这个结构体大小占用12字节。

  这个应该是11字节，怎么算出12字节的？

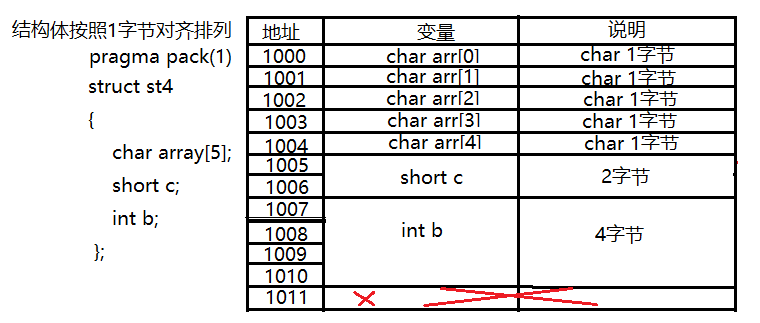


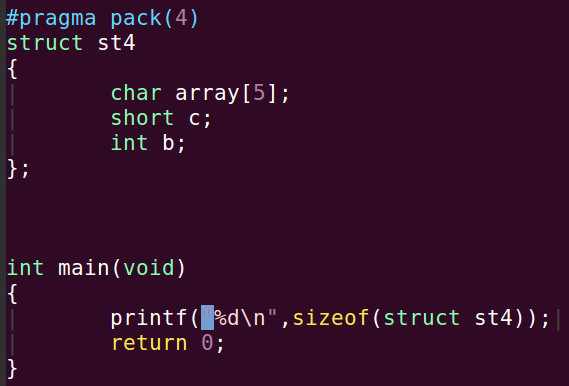
#pragma pack( n ) 内存对齐函数

n = 1表示按照1字节对齐，n=2表示按照2字节对齐，这样对齐方式是紧密排列了

  你看计算出来只有11字节

Pragma必须放在结构体前面，才能让下面结构体按照设置的对齐方式生效



4字节对齐，又回到GCC编译器默认的情况了



一般不建议去修改pragma对齐方式，因为如果你的程序要直接用在其它系统或者电脑平台上运行，可能会出现一些问题，建议就用GCC默认对齐方式。

#pragma pack(pop) //pack(pop) 就是让以下结构体恢复到系统默认对应方式

#pragma pack(4) // CC 结构体4字节对齐

struct CC {

double d;

char b;

int a;

short c;

};

#pragma pack() //BB结构体又恢复到系统默认对齐方式

struct BB{

double d;

char b;

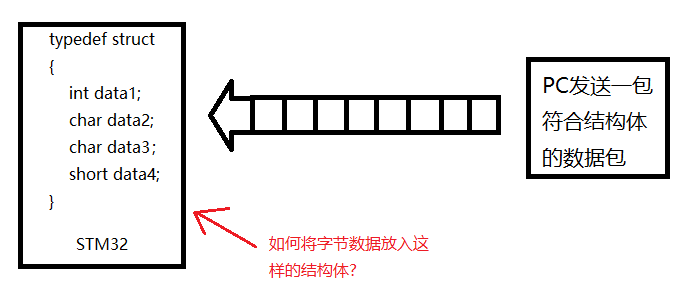
int a;

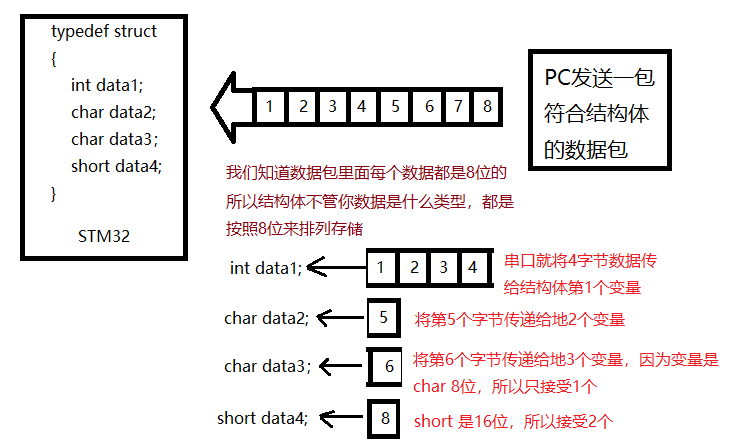
short c;

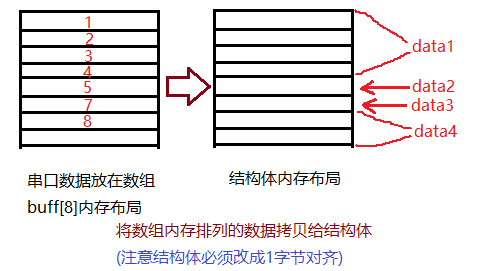
};

## 用结构体方式接受串口的数据包，结构体对数据内容进行了取名，方便使用

需要上一节#pragma pack 字节对齐的知识

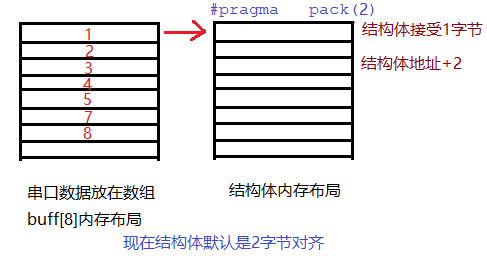




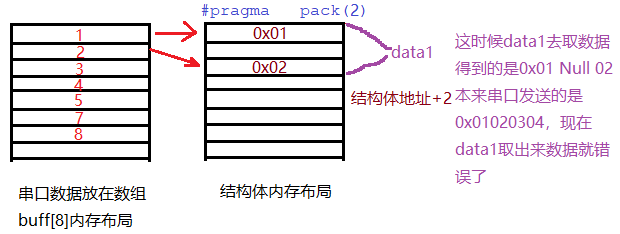


为什么结构体要改成1字节对齐？

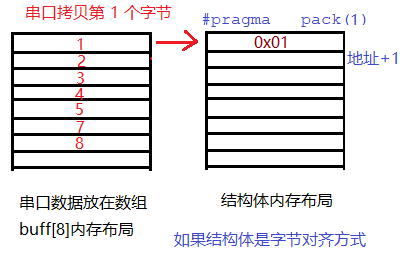
因为串口是单字节数组，所以串口数据拷贝给结构体只能1字节1字节拷贝，如果结构体是2字节对齐，那么结构体收到串口拷贝的1个字节之后，结构体地址+2

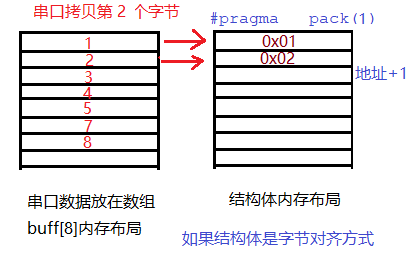


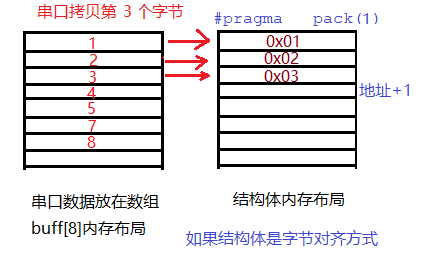


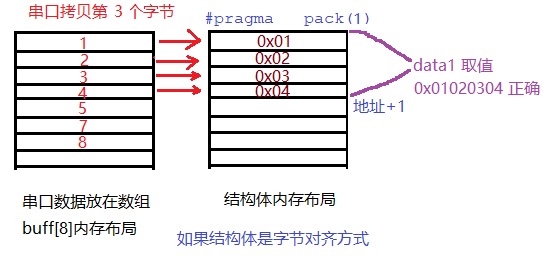


这就是结构体地址2字节对齐的问题









这就是为什么要单字节对齐的原因，因为串口拷贝给结构体的数据是按照单字节拷贝的。

#pragma pack(1) //以下结构体按照1字节对齐

typedef struct

{

uint16\_t Preamble; //前导码

uint32\_t DeviceAddr; //得到本设备地址

uint8\_t FunCode; //功能码

uint16\_t StartAddr; //起始地址

uint16\_t DataNum; //数据个数

uint16\_t CRC16; //CRC校验

}ModbusMaster\_Rcv\_Data; //接受串口的数据打印出来

ModbusMaster\_Rcv\_Data MBrcvdata; 创建一个结构体存放串口拷贝的数据

uint8\_t data[20] = {0xCE,0xA0,0x1f,0x2f,0x3f,0x4f,0x55,0x11,0x12,0x30,0x30,0x22,0xff};

如果这一帧data是串口发过来的数据

memcpy(&MBrcvdata,data,20 ); //将串口缓存的数据拷贝给结构体

直接打印结构体里面的数据，里面的数据就是串口每一段数据的标识

printf("Preamble1 = %x\r\n", MBrcvdata.Preamble);

printf("Preamble1 = %x\r\n", MBrcvdata. DeviceAddr);

printf("Preamble1 = %x\r\n", MBrcvdata. StartAddr);

打印结果

C:\Users\xiang\AppData\Local\Temp\1606370384(1).png

C:\Users\xiang\AppData\Local\Temp\1606370430(1).png

C:\Users\xiang\AppData\Local\Temp\1606370456(1).png

我们发现数据大小端是反的0xce是低位，0xa0是高位，貌似根据数组排列这样是正确的。

但是结构体输出也必须和数组排列一致才对，比如Preamble1 输出应该是0xcea0，因为数组是data[0]=0xce，data[1] = 0xa0

所以我们需要做大小端反转

/\*uint16\_t数据大端转小端\*/

void BigLittle\_endian16(uint16\_t \*StuData)

{

uint16\_t ret = 0;

uint16\_t tmpL;

uint16\_t tmpH;

ret = \*StuData;

tmpL = (ret >>8 ) & 0x00ff;

tmpH = (ret << 8 ) & 0xFF00;

\*StuData = tmpH | tmpL;

}

/\*uint32\_t数据大端转小端\*/

void BigLittle\_endian32(uint32\_t \*StuData)

{

uint32\_t ret = 0;

uint32\_t tmpL1;

uint32\_t tmpL2;

uint32\_t tmpH1;

uint32\_t tmpH2;

/\*4f 3f 2f 1f\*/

/\*1f 2f 3f 4f\*/

ret = \*StuData;

tmpL1 = ( ret >> 24 ) & 0x000000ff;

tmpL2 = ( ret >> 8 ) & 0x0000ff00;

tmpH1 = ( ret << 8 ) & 0x00ff0000;

tmpH2 = ( ret << 24) & 0xff000000;

\*StuData = tmpH2 | tmpH1 | tmpL2 | tmpL1 ;

}

使用以上函数进行大小端转换，转换16位，和32位两种，8位不需要转。

直接传入1字节对齐的结构体可以转换吗？BigLittle\_endian16(&MBrcvdata.Preamble);

记住这样绝对不行，会出现程序宕机

因为BigLittle\_endian16都是进行16位(2字节)变量运算的

所以只有把MBrcvdata结构体的数据赋值给编译器默认的字节对齐结构体变量。

所以我要创建一个默认的字节对齐，相同的结构体变量。

#pragma pack() //使用pack() 将以下结构体变成编译器默认的结构体对齐方式

typedef struct

{

#pragma pack(1) //以下结构体按照1字节对齐

typedef struct

{

uint16\_t Preamble; //前导码

uint32\_t DeviceAddr; //得到本设备地址

uint8\_t FunCode; //功能码

uint16\_t StartAddr; //起始地址

uint16\_t DataNum; //数据个数

uint16\_t CRC16; //CRC校验

}ModbusMaster\_Rcv\_Data;

uint16\_t Preamble; //前导码

uint32\_t DeviceAddr; //得到本设备地址

uint8\_t FunCode; //功能码

uint16\_t StartAddr; //起始地址

uint16\_t DataNum; //数据个数

uint16\_t CRC16; //CRC校验

}ModbusMaster\_Rcv\_cache;

因为Rcv\_cache是系统默认对齐方式，所以将1字节对齐的结构拷贝过来

ModbusMaster\_Rcv\_Data MBrcvdata; //1字节对齐

ModbusMaster\_Rcv\_cache cache; //系统默认对齐方式

/\*将1字节对齐的数据拷贝给默认对齐的结构体\*/

cache.Preamble = MBrcvdata.Preamble;

cache.DeviceAddr = MBrcvdata.DeviceAddr;

cache.FunCode = MBrcvdata.FunCode;

cache.StartAddr = MBrcvdata.StartAddr;

/\*把默认对齐的结构体放入大小端转换程序就可以了\*/

printf("Preamble1 = %x\r\n",cache.Preamble);

BigLittle\_endian16(&cache.Preamble);//16位大小端转换

printf("Preamble2 = %x\r\n",cache.Preamble);

printf("DeviceAddr1 = %x\r\n",cache.DeviceAddr);

BigLittle\_endian32(&cache.DeviceAddr);

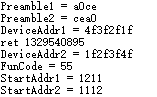
printf("DeviceAddr2 = %x\r\n",cache.DeviceAddr);

printf("FunCode = %x\r\n",cache.FunCode);

printf("StartAddr1 = %x\r\n",cache.StartAddr);

BigLittle\_endian16(&cache.StartAddr);//16位大小端转换

printf("StartAddr2 = %x\r\n",cache.StartAddr);

 大小端就转换过来了，a0ce转换成cea0 和数组对上了。其它也一样

## \_\_attribute\_\_ 用法

[**C语言中\_\_attribute\_\_ ((at())绝对定位的应用**](https://www.cnblogs.com/xyelectric/p/9046048.html)

**\_\_attribute\_\_( at(绝对地址) )**的作用分两个，一个是绝对定位到Flash，另个一是绝对定位到RAM。

**定位到flash中**，一般用于固化的信息，如出厂设置的参数，上位机配置的参数，ID卡的ID号，flash标记等等

const u16 gFlashDefValue[512] \_\_attribute\_\_((at(0x0800F000))) = {0x1111,0x1111,0x1111,0x0111,0x0111,0x0111};//定位在flash中,其他flash补充为00

const u16 gflashdata\_\_attribute\_\_((at(0x0800F000))) = 0xFFFF;//地址为0x0800F000

**定位到RAM中**，一般用于数据量比较大的缓存，如串口的接收缓存，再就是某个位置的特定变量

u8 USART2\_RX\_BUF[USART2\_REC\_LEN] \_\_attribute\_\_ ((at(0X20001000)));//接收缓冲,最大USART\_REC\_LEN个字节,起始地址为0X20001000.

注意:

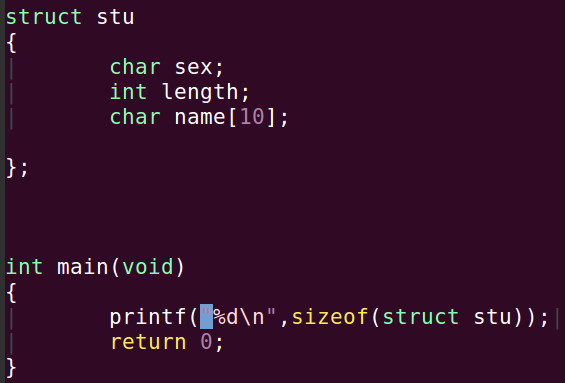
绝对定位不能在函数中定义，局部变量是定义在栈区的，栈区由MDK自动分配、释放，不能定义为绝对地址，只能放在函数外定义

定义的长度不能超过栈或Flash的大小，否则，造成栈、Flash溢出

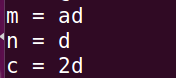
\_\_attribute\_\_((\_\_aligned\_\_(n)))对变量的影响

\_\_attribute\_\_（（aligned(n)））和pragma效果一样用于字节对齐，

\_\_attribute\_\_（（aligned(n)））和pragma有什么区别呢？ n=1 ，1字节对齐，n=4, 4字节对齐 …..

  20个字节，理论上只有15个

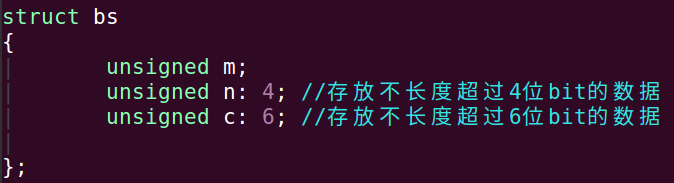
## C语言 ，变量位域定义，如 unsigned char ch: 6; 这种用法

你看，输出结果，m是完整正确的，n和c变量都被截取了丢掉了部分。

这是为什么呢？

这就是位域运算，给某个变量限制存放数据的范围。感觉是不是有点多此一举了？



在这个结构体中：

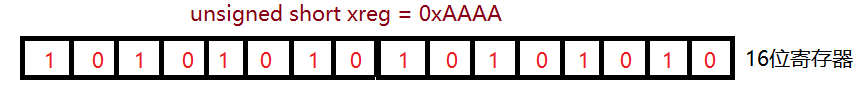
m可以存放很大的数据

但是n只能存放不超过4位的数据，也就是最大二进制为1111(0xf)，所以n变量的a值被截取掉了，很正常。

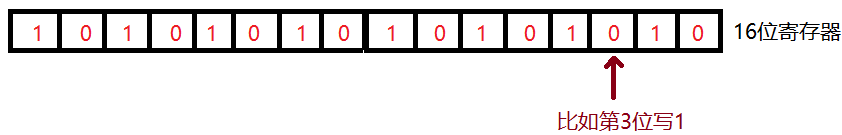
同样c只能存放最大不超过6位的数据，也就是最大二进制位111111(0x3f)，所以ad (1010 1101) 截取高两位就是(10 1101) 2d，。

这种位域运算这么折腾到底有什么用？

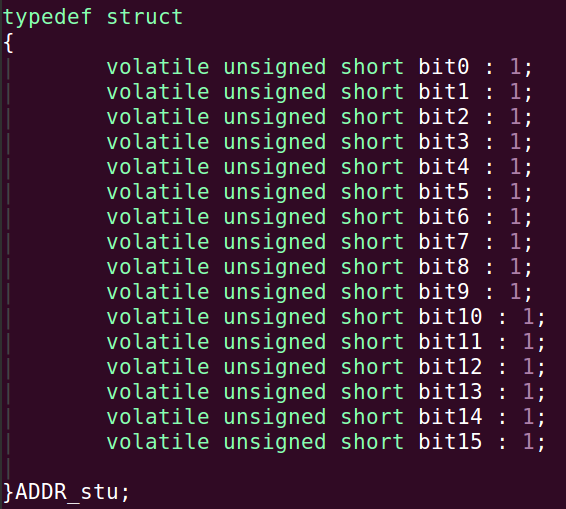
我用一个16位的unsigned short变量来模拟16位寄存器



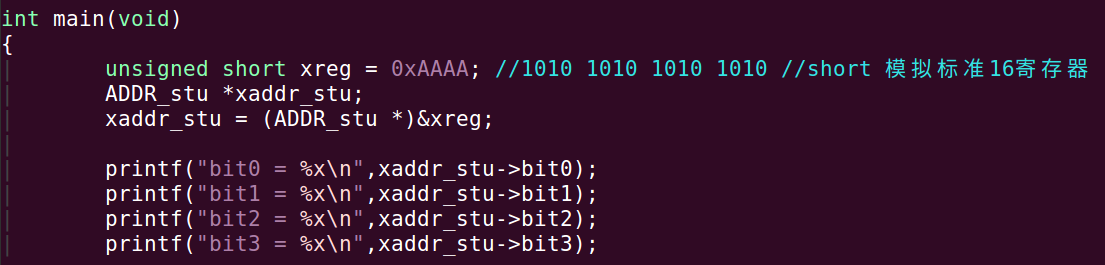
如果我想修改其中某一位，不使用移位运算，有更好的修改方式吗？



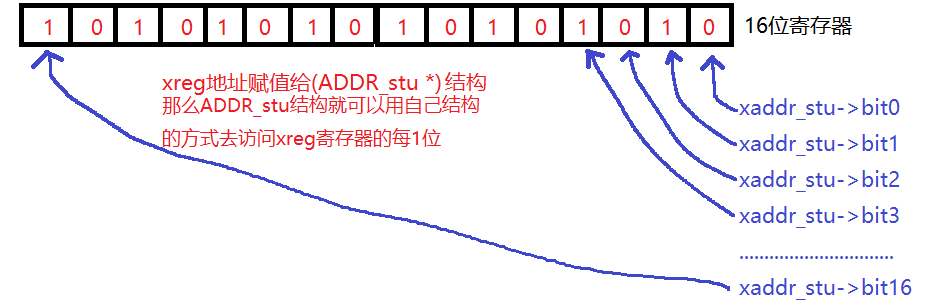
我用这种位域方式就很方便



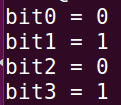
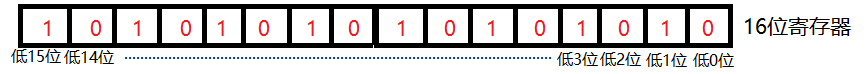
一定要用volatile 来修饰，不然编译器在编译的时候很可能把unsigned short xx : 1位运算，优化成常规的unsigned short 16位变量



将我们模拟的寄存器xreg地址强制转换成位域赋值给位域变量



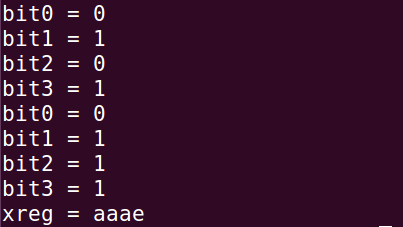
我们打印看看是否将寄存器转换到结构体位域成功

 对比查看打印正确

我们写1位进去看看是不是对的呢？



修改变量里面的某1位



未修改

xreg本身第2位确实被修改了

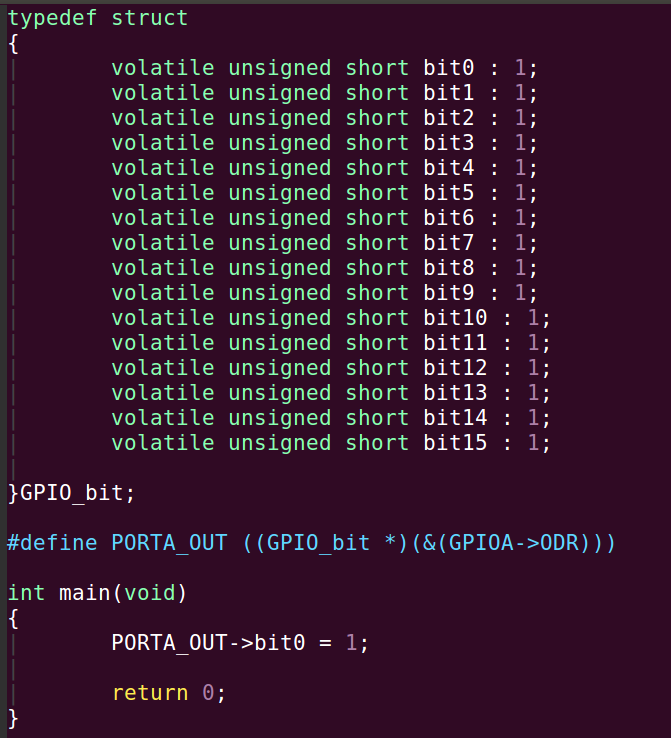
修改后，第2位修改成功

0xaaae = 1010 1010 1010 1110 你看xreg变量第2位被修改

这种方式用在单片机寄存器操作就很方便了，我可以直接放弃移位运算(1<<2)这种麻烦的操作。

下面操作STM32单片机GPIO管脚的某一位就可以用这种位域方法

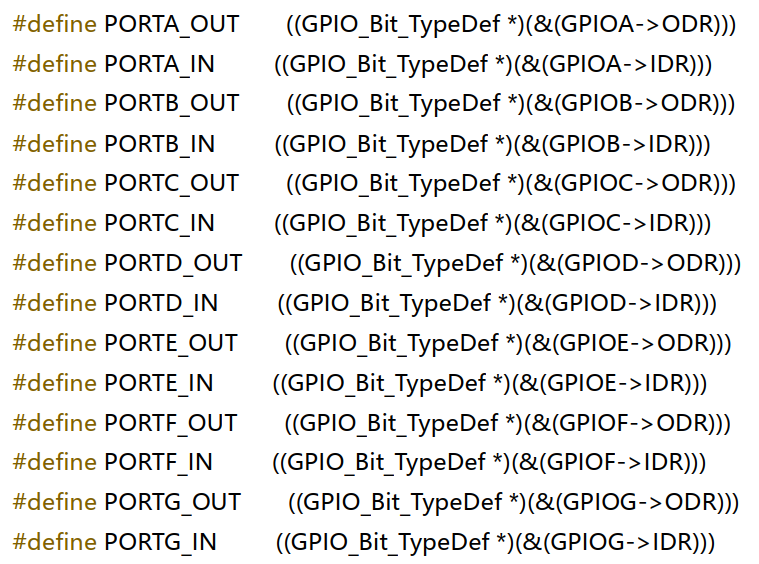




因为GPIOA 的ODR是16位的寄存器，所以我们可以用unsigned short来转换

这就是操作GPIOA bit0位的方法，因为地址要在STM32上才能访问， 所以这里只能浏览代码结构。

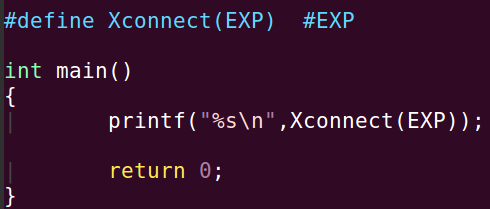




可以将所有的GPIO都定义成这样

## #和##连接符使用

#号使用，主要是将宏里面的变量名变成字符显示

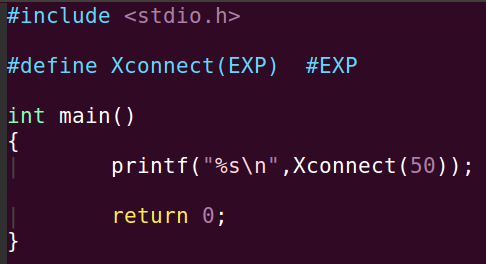


向宏定义传入EXP

单#号就是将#后面的英文或者变量值变成字符串形式，这里就是将EXP变成字符串



EXP变成字符被打印出来

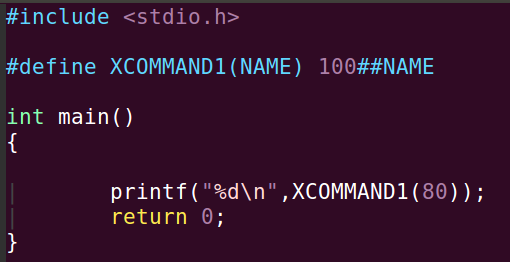
 数字变成字符

传入数字，也会变成字符被打印出来

##连接符

( ## 是变量连接符，将两个字符连接成一个变量,<记住是将两个字符连接成一个变量，或者是拼接成值>，而不是连接两个字符串)

##将前后两个参数进行拼接 (比如100#80 = 10080)



100##80拼接后

10080

将100和NAME变量传入的值拼接起来



在单片机访问寄存器中配合上一节位域的概念，就可以操作寄存器中某一位。

#define PORTA\_OUT ((GPIO\_Bit\_TypeDef \*)(&(GPIOA->ODR)))  
#define PORTA\_IN ((GPIO\_Bit\_TypeDef \*)(&(GPIOA->IDR)))

#define PAout(n) (PORTA\_OUT->bit##n) //这样用连接符就可以操作某位的高低电平  
#define PAin(n) (PORTA\_IN->bit##n)

如: PAout(1) = 1；就相当于PORTA\_OUT - >bit1 ，表示GPIOA 1 输出高电平

#define GPIO\_SET(X) (X##\_PORT)->BSRR = ((uint32\_t)1 << X##\_PIN) //比如这种？怎么理解？

GPIO\_SET(LED\_PDM); //这是GPIO\_SET宏真正的操作方式

这种操作方式需要满足以下几个条件

#define LED\_PDM\_PORT GPIOB //GPIOB地址

#define LED\_PDM\_PIN 1

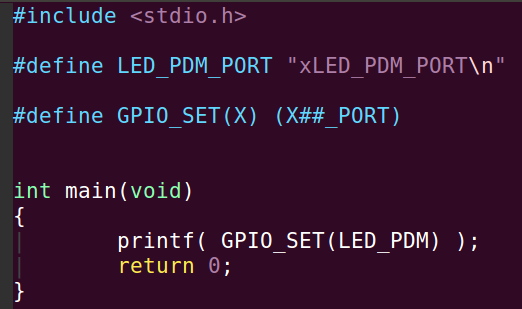
宏定义了GPIO地址和引脚

GPIO\_SET(LED\_PDM); =

#define GPIO\_SET(X) (X##\_PORT)->BSRR = ((uint32\_t)1 << X##\_PIN) 替换如下

代入 LED\_PDM\_PORT->BSRR = ((uint32\_t)1 << LED\_PDM \_PIN)

就是将LED\_PDM拼接进去了，不用先定义LED\_PDM变量，因为是连接符，所以就是传入的字符

 运行成功

这种LED\_PDM\_PORT事先就定义好了的，只是为了方便用连接符来调用。

#define PWM\_START(X) \

X##\_TM->BDTR |= TIM\_BDTR\_MOE; \

X##\_TM->CR1 |= TIM\_CR1\_CEN

PWM\_START(CWD\_NEN); //就是这种使用方法

我事先在宏里面定义了CWD\_NEN连接的几种可能性

#define CWD\_NEN\_PORT GPIOA

#define CWD\_NEN\_PIN 0

#define CWD\_NEN\_CC 3

#define CWD\_NEN\_CH 3

#define CWD\_NEN\_AF 4

#define CWD\_NEN\_TM TIM2

如果PWM\_START(CWD\_NEN); 就相当于如下：

#define PWM\_START(CWD\_NEN) //启动TIM2

CWD\_NEN \_TM->BDTR |= TIM\_BDTR\_MOE;

CWD\_NEN \_TM->CR1 |= TIM\_CR1\_CEN

就是替换成TIM2

#define PWM\_START(CWD\_NEN)

TIM2->BDTR |= TIM\_BDTR\_MOE;

TIM2->CR1 |= TIM\_CR1\_CEN

如果我用TIM14呢？

#define WWD\_NEN\_TM TIM14

那就是： PWM\_START(WWD\_NEN);

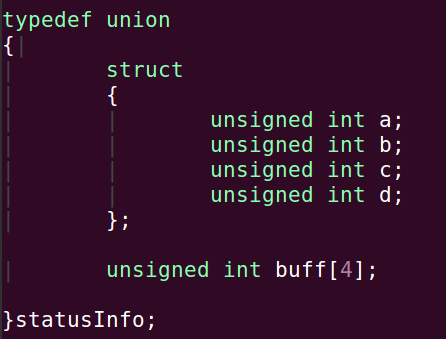
#define PWM\_START(WWD\_NEN) //启动TIM14

TIM14->BDTR |= TIM\_BDTR\_MOE;

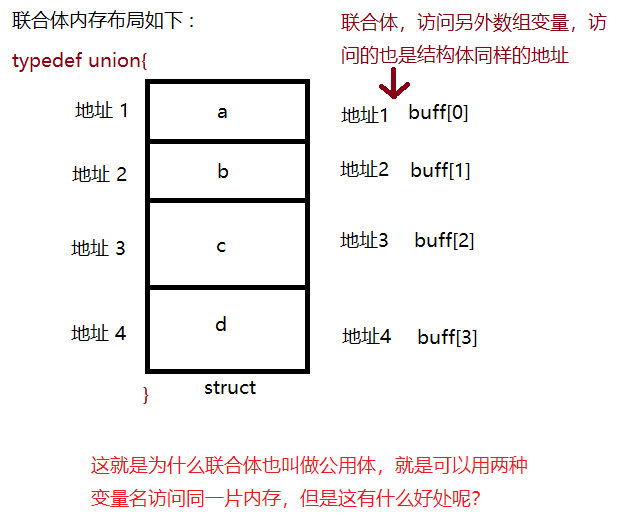
TIM14->CR1 |= TIM\_CR1\_CEN

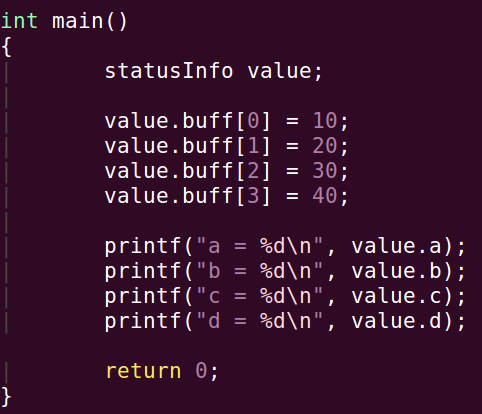
这就是连接符方便调用寄存器的方法，用人容易看懂的方式去使用寄存器，只不过代价就是要预先写很多这种宏。

## 联合体配合结构体和一维数组的高级应用

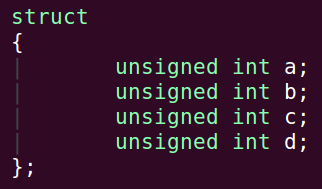


如果我要读写结构体里面某个变量，可以用数组的方式去访问



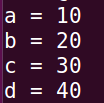


其实也就是向同一片内存结构体写数据



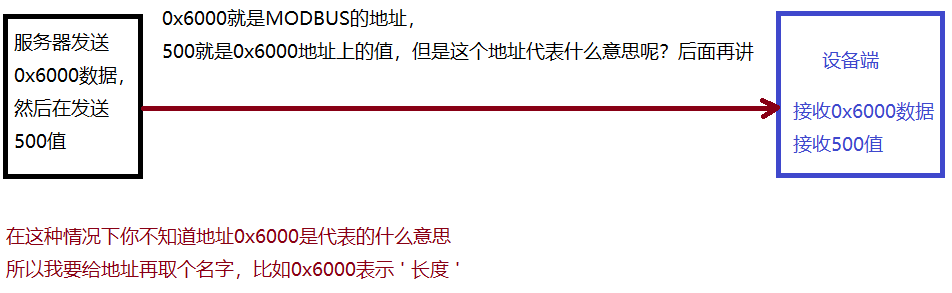
比如我向数组里面写数据

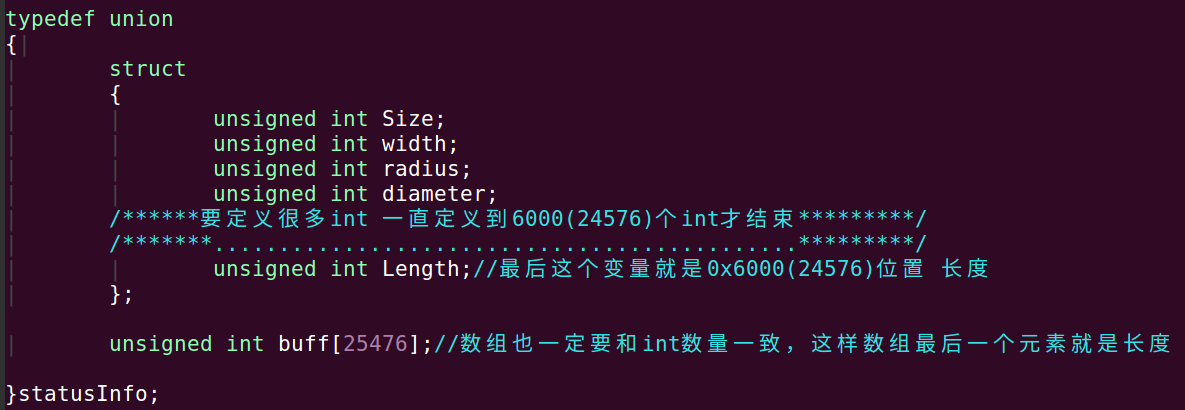


 读出来看看，其实就是我value.buff写入的数据

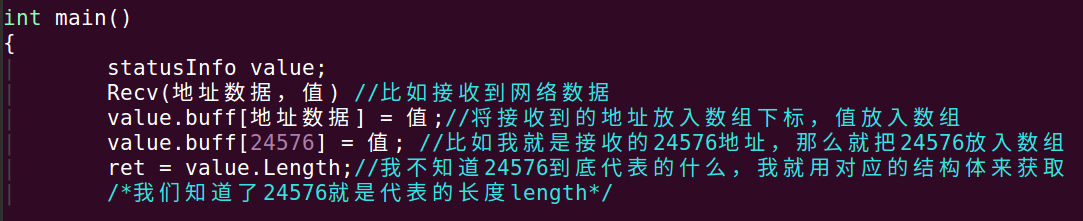
这样做有什么好处？用两个不同的变量来访问同一个内存空间？

比如在单片机MODBUS TCP协议的应用中





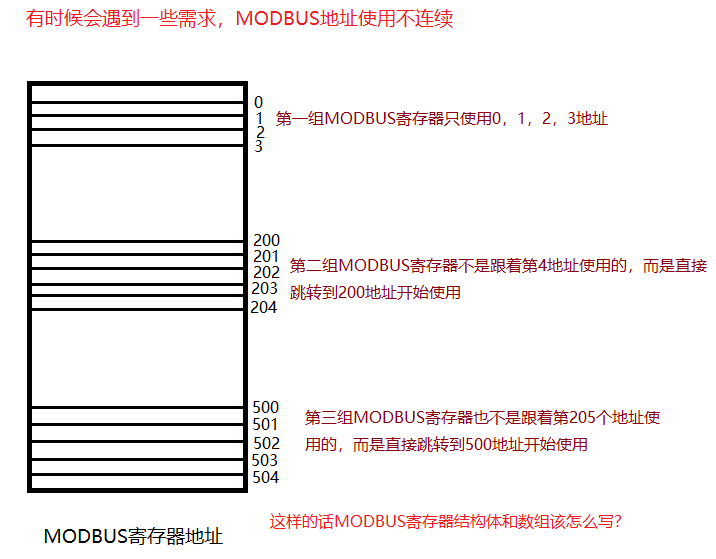
下面来讲解如何使用这个数据结构

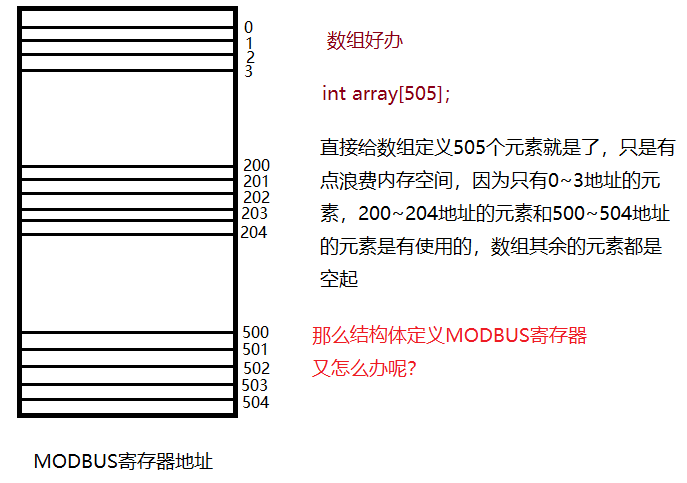


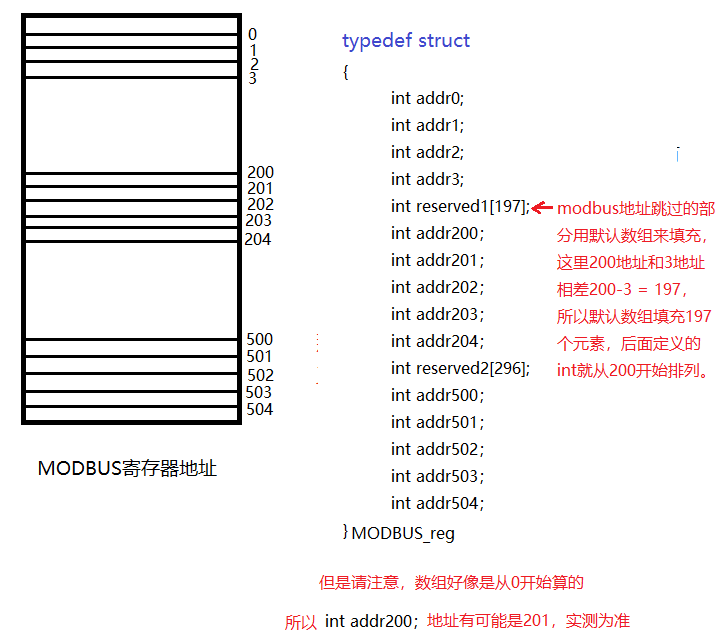
这就是设备端为了人类看得懂，所以用结构体变量来获取MODBUS 发过来的值。

如果服务器一次性发送1000个MODBUS的数据，你不能定义1000个变量去接收，你最好用数组去接收，然后用结构体来获取。

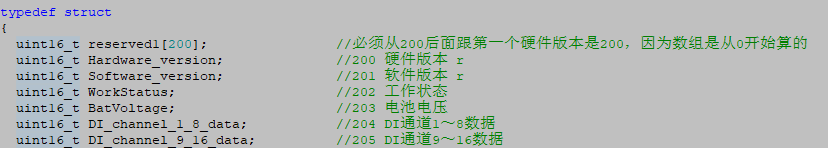
联合体在MODBUS协议应用的时候，寄存器排列方式注意事项

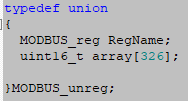






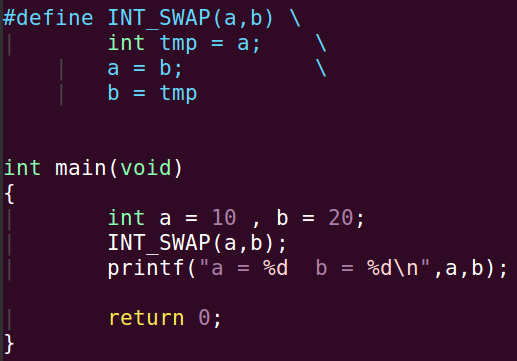
你看，在项目中我要访问200的地址，数组最后一个下标200地址还不对，因为数组是从0开始计算的，所以数组最后200下标的地址可能是199。所以数组后面定义的变量才是200地址



 最后使用枚举的方式操作MODBUS寄存器

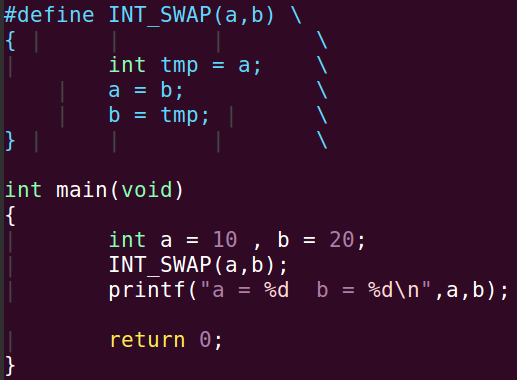
## 函数宏(主要是为了降低压栈和弹栈的开销，方便移植。不然直接建立函数来实现更好)

函数宏，即包含多条语句的宏定义，其通常为某一被频繁调用的功能的语句封装，且不想通过函数方式封装来降低额外的弹栈压栈开销。

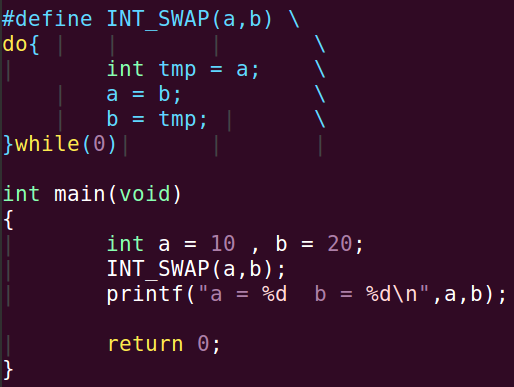
 

两个变量交互值。这种宏方式当使用 if、while 等语句且不使用花括号，直接调用宏INT\_SWAP时

程序运行到第1个 分号 int tmp = a; 就结束了。不会运行a=b; 和b = tmp

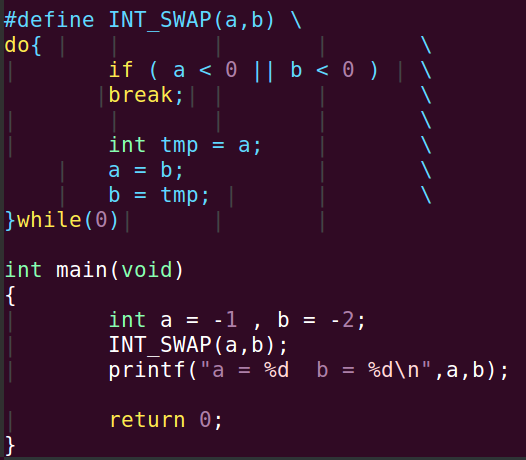
加入花括号之后也能正常运行

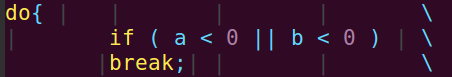
do{...}while(0) 表示只执行一遍 {} 内的语句

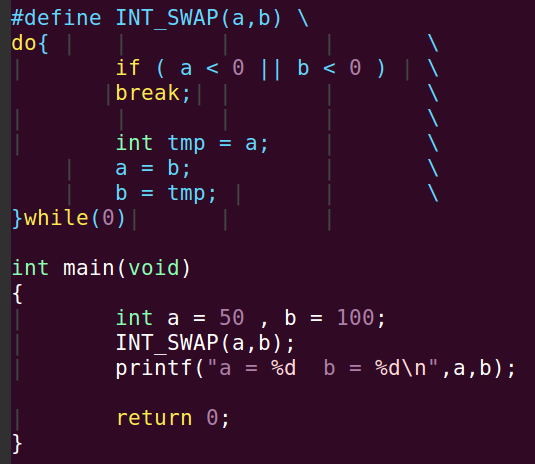
用do…. while有什么好处呢？

由于 do{...}while(0) 实际为 while 循环，因此可以使用关键字 break 提前结束循环。利用该特性，可以为函数宏添加参数检测。例如：

数据没有发生交换，因为发现a和b都 < 0 所以 在break提前退出了

提前退出

a和b不小于0，数据交换

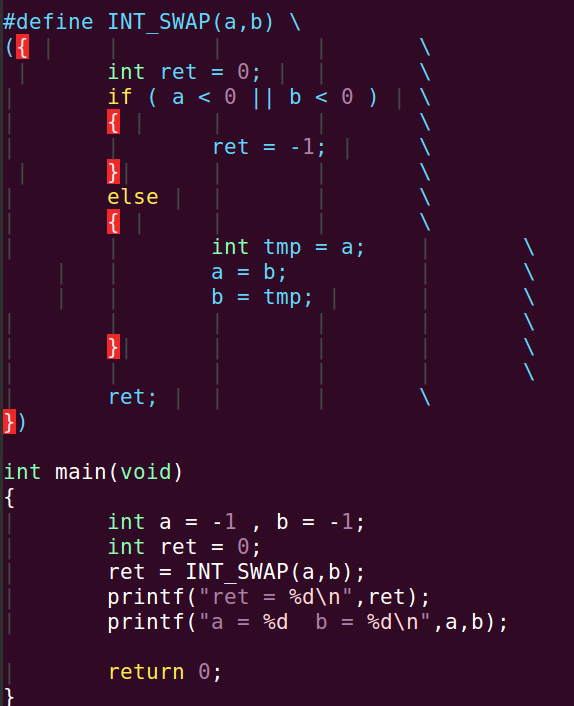
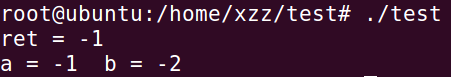
如果我想让函数宏运行完之后有返回值怎么做呢？

有些项目中用函数宏做判断，然后返回判断结果，以供其它程序识别

({}) 大括号外加小括号 ， 为 GNU C 扩展的语法，非 C 语言的原生语法。

与 do{...}while(0) 不同的是，({}) 不能提前退出函数宏

所以({})可以当做函数来用，还可以返回值

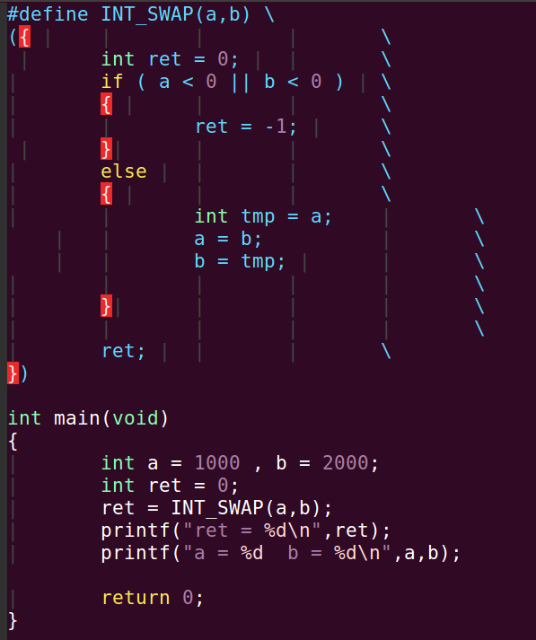
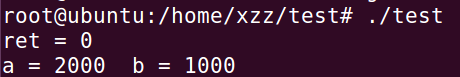
函数结尾执行的变量就是返回值

如果传入的值大于0，ret为默认值，就是int ret = 0这句

如果传入的值小于0，ret为-1

这种有({})括号的宏，里面的代码可以当做函数来写，但是记住不能用return退出哦

我将b改成了-2，函数宏返回-1，证明a和b < 0 没有交换

a和b 大于0，返回0，变量进行交换

## 钩子函数使用方式

钩子函数就是函数指针的一种，也是为了完成简单的分层，让一个功能可以方便多个人开发的一种设计模式。

#include <stdio.h>

int max(int a, int b){

    if(a > b)

        return a;

    return b;

}

int min(int a, int b){

    if(a > b)

        return b;

    return a;

}

int main( int argc, char \*argv[] )

{

    int (\*pFun)(int x, int y); //函数指针

    int a = 1;

    int b = 2;

    int ret;//定义一个变量来接收调用函数后的返回值

一般这样使用函数指针没有什么意义

    pFun = max;

    ret = pFun(a, b);//相当于直接调用max(a,b);

    printf("%d\n", ret);

    return 0;

}

一般需要两个C文件，比如A.c和B.c，一个工程师维护A.c文件，一个工程师维护B.c文件。

比如A.c实现

#include <stdio.h>

int (\*global\_pFun)(int a, int b);//由于后面要在注册函数中进行挂钩操作，所以把这个函数指针定义成全局变量

int RegFun(int (\*pFun)(int a, int b)){ //一定是先注册，这个函数交个头文件

    global\_pFun = pFun;

    return 0;

}

int main( int argc, char \*argv[] )

{

一定要等B工程师注册之后，再运行钩子函数，不然会出现空函数BUG

    int a = 1;

    int b = 2;

    int ret;

    ret = global\_pFun(a, b);//直接使用这个钩子函数，函数内部怎么实现的，交由B工程师

    printf("%d\n", ret);

    return 0;

}

现在B工程师实现

#include "A.h" //包含A工程师提供的h文件接口

//B工程师自己实现的子函数

int max(int a, int b){

    if(a > b)

        return a;

    return b;

}

int main( int argc, char \*argv[] )

{

    int a = 1;

    int b = 2;

    int ret;

    RegFun(max); //注册B工程师实现的函数

    //调用A工程师实现的main函数.......

    return 0;

}

这就是钩子函数的意义，一定是定义全局函数指针，然后让B工程师实现具体内容，A工程师再去调用B工程师实现的功能。这种方式的好处就是A工程师可以先把钩子函数实现起走，然后A工程师可以先调用钩子函数的空函数，等B工程师有时间了，把B工程师实现的功能注册进来。

## C语言实现MVC模式实现多人开发

首先，我们需要定义三个主要组件：模型(model)、视图(view)和控制器(controller)。每个组件都有自己的职责和相应的操作。

1. 模型(model): 模型包含了程序所处理的数据结构和算法。它负责存储和管理数据，并提供对这些数据进行修改或查询的接口。下面是一个示例的模型部分的代码：

typedef struct {

int data; // 数据成员

} Model;

void model\_setData(Model\* m, int value) {

m->data = value;

}

int model\_getData(const Model\* m) {

return m->data;

}

1. 视图(view): 视图负责将模型的数据展示给用户。它会从模型获取数据，然后根据需求显示到界面上。下面是一个示例的视图部分的代码

#include <stdio.h>

void view\_displayData(const Model\* m) {

printf("The current data is %d\n", model\_getData(m));

}

1. 控制器(controller): 控制器负责处理用户输入和更新模型。当用户与界面交互时，控制器会调用合适的模型函数来更新数据。下面是一个示例的控制器部分的代码

void controller\_updateData() {

Model m;

// 初始化模型数据为0

model\_setData(&m, 0);

// 模拟用户输入

char input[5];

printf("Enter a new value for the data (max 4 digits): ");

scanf("%s", input);

// 转换字符串为整数并设置到模型中

if (sscanf(input, "%d", &m.data) == 1 && m.data >= 0 && m.data <= 9999) {

printf("New data set to %d\n", m.data);

// 更新完数据后，调用视图函数显示最新的数据

view\_displayData(&m);

} else {

printf("Invalid input!\n");

}

}

以上就是C语言实现的最基本MVC模式方法。

能多人开发的MVC代码编写方式

