STM32-C 语言知识点

__I、__O 、__IO 是什么意思？

http://www.chuxue123.com/forum.php?mod=viewthread&tid=122&extra=page%3D1

这是 ST 库里面的宏定义，定义如下：

```
#define    __I     volatile const       /*!< defines 'read only' permissions      */
#define    __O     volatile             /*!< defines 'write only' permissions     */
#define    __IO    volatile             /*!< defines 'read / write' permissions   */
```

显然，这三个宏定义都是用来替换成 volatile 和 const 的，所以我们先要了解 这两个关键字的作用：

volatile

简单的说，就是不让编译器进行优化，即每次读取或者修改值的时候，都必须重新从内存或者寄存器中读取或者修改。

一般说来，volatile 用在如下的几个地方：

1、中断服务程序中修改的供其它程序检测的变量需要加 volatile；

2、多任务环境下各任务间共享的标志应该加 volatile；

3、存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义；

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。

1)一个参数既可以是 const 还可以是 volatile 吗？解释为什么。

2); 一个指针可以是 volatile 吗？解释为什么。

3); 下面的函数有什么错误：

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

1)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。

2); 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。

3) 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;
}
```

const

只读变量，即变量保存在**只读静态存储区**。编译时，如何尝试修改只读变量，则编译器提示出错，就能防止误修改。

const 与 define

两者都可以用来定义常量，但是 const 定义时，定义了常量的类型，所以更精确一些（其实 const 定义的是只读变量，而不是常量）。#define 只是简单的文本替换，除了可以定义常量外，还可以用来定义一些简单的函数，有点类似内置函数。const 和 define 定义的常量可以放在头文件里面。（小注：可以多次声明，但只能定义一次）

const 与指针

```
int me;
const int * p1=&me;          //p1 可变，*p1 不可变          const 修饰的是 *p1，即 *p1 不可变
int * const p2=&me;          //p2 不可变，*p2 可变          const 修饰的是 p2，即 p2 不可变
const int *const p3=&me;   //p3 不可变，*p3 也不可变      前一个 const 修饰的是 *p3，后一个 const 修饰的是 p3，两者都不可变
```

前面介绍了 volatile 和 const 的用法，不知道大家了解了没？了解了后，下面的讲解就更加容易了：

__I：输入口。既然是输入，那么寄存器的值就随时会外部修改，那就不能进行优化，每次

都要重新从寄存器中读取。也不能写，即只读，不然就不是输入而是输出了。
__O：输出口，也不能进行优化，不然你连续两次输出相同值，编译器认为没改变，就忽略了后面那一次输出，假如外部在两次输出中间修改了值，那就影响输出
__IO：输入输出口，同上

为什么加下划线？

原因是：避免命名冲突
一般宏定义都是大写，但因为这里的字母比较少，所以再添加下划线来区分。这样一般都可以避免命名冲突问题，因为很少人这样命名，这样命名的人肯定知道这些是有什么用的。
经常写大工程时，都会发现老是命名冲突，要不是全局变量冲突，要不就是宏定义冲突，所以我们要尽量避免这些问题，不然出问题了都不知道问题在哪里。

static 和 extern 的区别

http://www.chuxue123.com/forum.php?mod=viewthread&tid=548&extra=page%3D1

一些基本概念：

1. 编译单元(模块)：

在 IDE 开发工具大行其道的今天，对于编译的一些概念很多人已经不再清楚了，很多程序员最怕的就是处理连接错误(LINK ERROR), 因为它不像编译错误那样可以给出你程序错误的具体位置，你常常对这种错误感到懊恼，但是如果你经常使用 gcc，makefile 等工具在 linux 或者嵌 入式下做开发工作的话，那么你可能非常的理解编译与连接的区别！当在 VC 这样的开发工具上编写完代码，点击编译按钮准备生成 exe 文件时，VC 其实做了两 步工作，第一步，将每个.cpp(.c)和相应.h 文件编译成 obj 文件；第二步，将工程中所有的 obj 文件进行 LINK 生成最终的.exe 文件，那么错 误就有可能在两个地方产生，一个是编译时的错误，这个主要是语法错误，另一个是连接错误，主要是重复定义变量等。我们所说的编译单元就是指在编译阶段生成 的每个 obj 文件，一个 obj 文件就是一个编译单元，也就是说一个 cpp(.c)和它相应的.h 文件共同组成了一个编译单元，一个工程由很多个编译单元组 成，每个 obj 文件里包含了变量存储的相对地址等 。

2. 声明与定义的区别

函数或变量在声明时，并没有给它实际的物理内存空间，它有时候可以保证你的程序编译通过， 但是当函数或变量定义的时候，它就在内存中有了实际的物理空间，如果你在编译模块中引用的外部变量没有在整个工程中任何一个地方定义的话， 那么即使它在编译时可以通过，在连接时也会报错，因为程序在内存中找不到这个变量！你也可以这样理解， 对同一个变量或函数的声明可以有多次，而定义只能有一次!

3. extern 的作用

extern 有两个作用，第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b); 则告诉编译器在编译 fun 这个函数名时按着 C 的规则去翻译相应的函数名而不是 C++的，C++ 的 规 则 在 翻 译 这 个 函 数 名 时 会 把 fun 这 个 名 字 变 得 面 目 全 非 ， 可 能 是 fun@aBc_int_int#%$也可能是别的，这要看编译器的"脾气"了 (不同的编译器采用的方法不一样)，为什么这么做呢，因为 C++支持函数的重载啊，在这里不去过多的论述这个问题，如果你有兴趣可以去网上搜索，相信你可 以得到满意的解释!

当 extern 不与"C"在一起修饰变量或函数时，如在头文件中: extern int g_Int; 它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块或者其他模块中使用，记住它是一个声明不是定义!也就是说 B 模块(编译 单元)要是引用模块(编译单元)A 中定义的全局变量或函数时，它只要包含 A 模块的头文件即可，在编译阶段，模块 B 虽然找不到该函数或变量，但它不会报错，它会在连接时从模块 A 生成的目标代码中找到此函数。

如果你对以上几个概念已经非常明白的话，那么让我们一起来看以下几种全局变量/常量的使用区别:

1. 用 extern 修饰的全局变量

以上已经说了 extern 的作用，下面我们来举个例子,如:

在 test1.h 中有下列声明:

[code=cpp] #ifndef TEST1H

#define TEST1H

```
extern char g_str[]; // 声明全局变量 g_str
    void fun1();
    #endif[/code]    在 test1.cpp 中
[code=cpp]    #include "test1.h"

    char g_str[] = "123456"; // 定义全局变量 g_str

    void fun1()
    {
       cout << g_str << endl;
    }[/code]
```

以上是 test1 模块，它的编译和连接都可以通过,如果我们还有 test2 模块也想使用 g_str, 只需要在原文件中引用就可以了

```
[code=cpp]    #include "test1.h"
    void fun2()
    {
       cout << g_str << endl;
    }[/code]
```

以上 test1 和 test2 可以同时编译连接通过，如果你感兴趣的话可以用 ultraEdit 打开 test1.obj,你可以在里面着"123456"这 个字符串,但是你却不能在 test2.obj 里面找到，这是因为 g_str 是整个工程的全局变量，在内存中只存在一份, test2.obj 这个编译单元不需要再有一份了，不然会在连接时报告重复定义这个错误!

有些人喜欢把全局变量的声明和定义放在一起，这样可以防止忘记了定义，如把上面 test1.h 改为

```
[code=cpp]    extern char g_str[] = "123456"; // 这个时候相当于没有 extern[/code]
```

然后把 test1.cpp 中的 g_str 的定义去掉,这个时候再编译连接 test1 和 test2 两个模块时，会报连接错误，这是因为你把全局变量 g_str 的定义放在了头文件之后，test1.cpp 这个模块包含了 test1.h 所以定义了一次 g_str,而 test2.cpp 也包含了 test1.h 所以再一次定义了 g_str, 这个时候连接器在连接 test1 和 test2 时发现两个 g_str。如果你非要把 g_str 的定义放在 test1.h 中的话，那么就把 test2 的代码 中#include "test1.h"去掉 换成:

```
[code=cpp]    extern char g_str[];
    void fun2()
    {
       cout << g_str << endl;
    }[/code]
```

这个时候编译器就知道 g_str 是引自于外部的一个编译模块了，不会在本模块中再重复定义一个出来，但是我想说这样做非常糟糕，因为你由于无法在 test2.cpp 中使用#include "test1.h", 那么 test1.h 中声明的其他函数你也无法使用了，除非也用都用 extern 修饰，这样的话你光声明的函数就要一大串，而且头文件的作用就是要给外部提 供接口使用的，所以 请记住， 只在头文件中做声明，真理总是这么简单。

2. 用 static 修饰的全局变量

首先，我要告诉你 static 与 extern 是一对"水火不容"的家伙，也就是说 extern 和 static 不能同时修饰一个变量；其次，static 修 饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用 static 声明了全局变量后，它也同时被定义了；最后，static 修饰全局变量的作用域 只能是本身的编译单元，也就是说它的"全局"只对本编译单元有效，其他编译单

元则看不到它,如:
test1.h:
[code=cpp] #ifndef TEST1H
#define TEST1H
static char g_str[] = "123456";
void fun1();
#endif
test1.cpp:
#include "test1.h"

void fun1()
{
cout << g_str << endl;
}[/code]
test2.cpp
[code=cpp] #include "test1.h"

void fun2()
{
cout << g_str << endl;
}[/code]
以上两个编译单元可以连接成功, 当你打开 test1.obj 时，你可以在它里面找到字符串"123456", 同时你也可以在 test2.obj 中找到它们，它们之所以可以连接成功而没有报重复定义的错误是因为虽然它们有相同的内容，但是存储的物理地址并不一样， 就像是两个不同变量赋了相同的值一样，而这两个变量分别作用于它们各自的编译单元。
也许你比较较真，自己偷偷的跟踪调试上面的代码,结果你发现两个编译单元（test1, test2）的 g_str 的内存地址相同，于是你下结论 static 修饰的变量也可以作用于其他模块，但是我要告诉你，那是你的编译器在欺骗你，大多数编 译器都对代码都有优化功能，以达到生成的目标程序更节省内存，执行效率更高，当编译器在连接各个编译单元的时候，它会把相同内容的内存只拷贝一份，比如上 面的"123456", 位于两个编译单元中的变量都是同样的内容，那么在连接的时候它在内存中就只会存在一份了， 如果你把上面的代码改成下面的样子，你马上就可以拆穿编译器的谎言:
test1.cpp:
[code=cpp] #include "test1.h"

void fun1()
{
g_str[0] = 'a';
cout << g_str << endl;
}[/code] test2.cpp
[code=cpp] #include "test1.h"

void fun2()
{

```
    cout << g_str << endl;
}

void main()
{
    fun1(); // a23456
    fun2(); // 123456
}[/code]
```

这个时候你在跟踪代码时，就会发现两个编译单元中的 g_str 地址并不相同，因为你在一处修改了它，所以编译器被强行的恢复内存的原貌，在内存中存在了两份拷贝给两个模块中的变量使用。

正是因为 static 有以上的特性，所以一般定义 static 全局变量时，都把它放在原文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！

3 const 修饰的全局常量（const 总结的 ms 还不是很好，下次要是有好文章再转载！）

const 修饰的全局常量用途很广，比如软件中的错误信息字符串都是用全局常量来定义的。const 修饰的全局常量据有跟 static 相同的特性(有条件的，感谢 sswv 的提醒，const 放在只读静态存储区)，即它们只能作用于本编译模块中，但是 const 可以与 extern 连用来声明该常量可以作用于其他编译模块中, 如

[code=cpp] extern const char g_str[];[/code] 然后在原文件中别忘了定义:

[code=cpp] const char g_str[] = "123456";[/code] 所以当 const 单独使用时它就与 static 相同，（前提是都在描述全局变量，如果在函数内部就不一样）而当与 extern 一起合作的时候，它的特性就跟 extern 的一样了！所以对 const 我没有什么 可以过多的描述，我只是想提醒你，const char* g_str = "123456" 与 const char g_str[] = "123465"是不同的，前面那个 const 修饰的是 char * 而不是 g_str,它的 g_str 并不是常量，它被看做是一个定义了的全局变量（可以被其他编译单元使用）， 所以如果你像让 char *g_str 遵守 const 的全局常量的规则，最好这么定义 const char* const g_str="123456".

一、c 程序存储空间布局

C 程序一直由下列部分组成：

1）正文段——CPU 执行的机器指令部分；一个程序只有一个副本；只读，防止程序由于意外事故而修改自身指令；

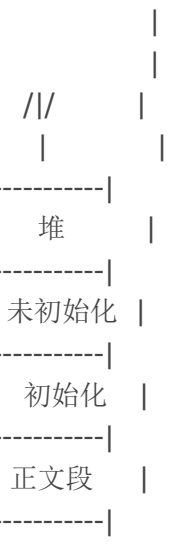
2）初始化数据段（数据段）——在程序中所有赋了初值的全局变量，存放在这里。

3）非初始化数据段（bss 段）——在程序中没有初始化的全局变量；内核将此段初始化为 0。

4）栈——增长方向：自顶向下增长；自动变量以及每次函数调用时所需要保存的信息（返回地址；环境信息）。

5）堆——动态存储分。

```
|-----------|
|           |
|-----------|
|    栈      |
|-----------|
|    |       |
|   /|/      |
```



二、 面向过程程序设计中的 static

1. 全局静态变量

在全局变量之前加上关键字 static，全局变量就被定义成为一个全局静态变量。

1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2）初始化：未经初始化的全局静态变量会被程序自动初始化为 0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

看下面关于作用域的程序：

```
//teststatic1.c
void display();
extern int n;
int main()
{
  n = 20;
  printf("%d/n",n);
  display();
  return 0;
}
//teststatic2.c
static int n;   //定义全局静态变量，自动初始化为 0，仅在本文件中可见
void display()
{
  n++;
  printf("%d/n",n);
}
```

文件分别编译通过，但 link 的时候 teststatic1.c 中的变量 n 找不到定义，产生错误。

定义全局静态变量的好处：

<1>不会被其他文件所访问，修改

<2>其他文件中可以使用相同名字的变量，不会发生冲突。

2. 局部静态变量

在局部变量之前加上关键字 static，局部变量就被定义成为一个局部静态变量。

1）内存中的位置：静态存储区

2）初始化：未经初始化的全局静态变量会被程序自动初始化为 0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

注：当 static 用来修饰局部变量的时候，它就改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问。

当 static 用来修饰全局变量的时候，它就改变了全局变量的作用域（在声明他的文件之外是不可见的），但是没有改变它的存放位置，还是在静态存储区中。

3. 静态函数

在函数的返回类型前加上关键字 static，函数就被定义成为静态函数。

函数的定义和声明默认情况下是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

例如：

[code=cpp]//teststatic1.c
void display();
static void staticdis();
int main()
{
  display();
  staticdis();
  renturn 0;
}
//teststatic2.c
void display()
{
  staticdis();
  printf("display() has been called /n");
}
static void staticdis()
{
  printf("staticDis() has been called/n");
}[/code]文件分别编译通过，但是连接的时候找不到函数 staticdis（）的定义，产生错误。

定义静态函数的好处：

<1> 其他文件中可以定义相同名字的函数，不会发生冲突

<2> 静态函数不能被其他文件所用。

存储说明符 auto，register，extern，static，对应两种存储期：自动存储期和静态存储期。auto 和 register 对应自动存储期。具有自动存储期的变量在进入声明该变量的程序块时被建立，它在该程序块活动时存在，退出该程序块时撤销。

关键字 extern 和 static 用来说明具有静态存储期的变量和函数。用 static 声明的局部变量具有静态存储持续期（static storage duration），或静态范围（static extent）。虽然他的值在函数调用之间保持有效，但是其名字的可视性仍限制在其局部域内。静态局部对象在程序执行到该对象的声明处时被首次初始化。

由于 static 变量的以上特性，可实现一些特定功能。

1． 统计次数功能

声明函数的一个局部变量，并设为 static 类型，作为一个计数器，这样函数每次被调用的时候就可以进行计数。这是统计函数被调用次数的最好的办法，因为这个变量是和函数息息相关的，而函数可能在多个不同的地方被调用，所以从调用者的角度来统计比较困难。代码如下：

[code=cpp]void count();
int main()
{
int i;
for (i = 1; i <= 3; i++)
  count();
  return 0;
}
void count()
{
static num = 0;
num++;
printf(" I have been called %d",num,"times/n");
}[/code]输出结果为：

I have been called 1 times.
I have been called 2 times.
I have been called 3 times.

C语言 # 与 ## 的妙用

http://www.chuxue123.com/forum.php?mod=viewthread&tid=899&extra=page%3D1

一、一般用法

我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.

用法:

```
[code=cpp]#include<cstdio>
#include<climits>

using namespace std;

#define STR(s)     #s

#define CONS(a,b)  int(a##e##b)

int main()
{
    printf(STR(vck));           // 输出字符串"vck"

    printf("%d", CONS(2,3));  // 2e3  输出:2000

    return 0;
}[/code]
```

二、当宏参数是另一个宏的时候

需要注意的是凡宏定义里有用'#'或'##'的地方宏参数是不会再展开.

1, 非'#'和'##'的情况

```
[code=cpp]#define TOW        (2)

#define MUL(a,b) (a*b)

printf("%d*%d=%d", TOW, TOW, MUL(TOW,TOW));[/code]
```

这行的宏会被展开为：

[code=cpp]printf("%d*%d=%d", (2), (2), ((2)*(2)));[/code]

MUL 里的参数 TOW 会被展开为(2).

2, 当有'#'或'##'的时候

[code=cpp]#define A (2)
#define STR(s) #s
#define CONS(a,b) int(a##e##b)

printf("int max: %s", STR(INT_MAX)); // INT_MAX #include<climits>[/code]

这行会被展开为：

[code=cpp]printf("int max: %s", "INT_MAX");[/code]

[code=cpp]printf("%s", CONS(A, A)); // compile error [/code]

这一行则是：

[code=cpp]printf("%s", int(AeA));[/code]

INT_MAX 和 A 都不会再被展开, 然而解决这个问题的方法很简单. 加多一层中间转换宏.

加这层宏的用意是把所有宏的参数在这层里全部展开, 那么在转换宏里的那一个宏(_STR)就能得到正确的宏参数.

[code=cpp]#define A (2)

#define _STR(s) #s
#define STR(s) _STR(s) // 转换宏

#define _CONS(a,b) int(a##e##b)
#define CONS(a,b) _CONS(a,b) // 转换宏

printf("int max: %s", STR(INT_MAX)); // INT_MAX,int 型的最大值，为一个变量

#include<climits>[/code]

输出为: int max: 0x7fffffff

STR(INT_MAX) --> _STR(0x7fffffff) 然后再转换成字符串；

[code=cpp]printf("%d", CONS(A, A));[/code]

输出为：200

CONS(A, A) --> _CONS((2), (2)) --> int((2)e(2))

三、'#'和'##'的一些应用特例

1、合并匿名变量名

[code=cpp]#define ___ANONYMOUS1(type, var, line) type var##line

#define __ANONYMOUS0(type, line) ___ANONYMOUS1(type, _anonymous, line)

#define ANONYMOUS(type) __ANONYMOUS0(type, __LINE__)
[/code]
例：ANONYMOUS(static int); 即: static int _anonymous70; 70 表示该行行号；

第一层：ANONYMOUS(static int); --> __ANONYMOUS0(static int, __LINE__);

第二层： --> ___ANONYMOUS1(static int, _anonymous, 70);

第三层： --> static int _anonymous70;

即每次只能解开当前层的宏，所以__LINE__在第二层才能被解开；

2、填充结构

[code=cpp]#define FILL(a) {a, #a}

enum IDD{OPEN, CLOSE};

typedef struct MSG{

IDD id;

const char * msg;

}MSG;

MSG _msg[] = {FILL(OPEN), FILL(CLOSE)};[/code]

相当于：

[code=cpp]MSG _msg[] = {{OPEN, "OPEN"},

{CLOSE, "CLOSE"}};[/code]

3、记录文件名

[code=cpp]#define _GET_FILE_NAME(f) #f

#define GET_FILE_NAME(f) _GET_FILE_NAME(f)

static char FILE_NAME[] = GET_FILE_NAME(__FILE__);
[/code]

4、得到一个数值类型所对应的字符串缓冲大小

[code=cpp]#define _TYPE_BUF_SIZE(type) sizeof #type

#define TYPE_BUF_SIZE(type) _TYPE_BUF_SIZE(type)

char buf[TYPE_BUF_SIZE(INT_MAX)];[/code]

--> char buf[_TYPE_BUF_SIZE(0x7fffffff)];

--> char buf[sizeof "0x7fffffff"];

这里相当于：

[code=cpp]char buf[11];[/code]

关于宏定义的用法

http://www.chuxue123.com/forum.php?mod=viewthread&tid=123&extra=page%3D1

Q 群中总会有人会问各种不同的问题，而很多问题很适合其他人参考了，故把它粘贴上来，以供参考：

如何解释下面这段代码：

```
#define LED1(a)   if (a)\
                      GPIO_SetBits(GPIOC,GPIO_Pin_3);\
                  else \
                      GPIO_ResetBits(GPIOC,GPIO_Pin_3)
```

首先，这个是用宏定义的方式包装成类似函数那样，但不是函数调用
你在代码中调用：

```
LED1(1);
```

实际上通过宏定义替换，代码会替换成：

```
if (1) GPIO_SetBits(GPIOC,GPIO_Pin_3); elseGPIO_ResetBits(GPIOC,GPIO_Pin_3)
```

宏定义中的 a 就被 调用时 的‘1’所替换掉，就类似函数那样，但不是函数。

另外，有没有发现替换后的代码只有一行，而不是宏定义中的多行呢？

回答这个问题，你就必须先了解 c 语言 中 反斜杠（ \）的作用：语义上表示，下一行是上一行的延续。也就是同一行。
当你的代码一行写的时候会太长，需要分行方便显示时，但代码又不能分行时，例如这里的宏定义，只能在一行定义好，那样就可以用过在结尾添加 反斜杠（ \） 来换行。表示 接着下一行，就是例子中的整个 if-else 语句都被 反斜杠（ \） 连接在同一行，所以替换后就仅仅一行而已。

注意，反斜杠（ \） 后面不能有任何字符，包括空格。

宏定义的用法

注意：宏定义不是函数！！

一般用来简化操作的，但又能避免函数调用那样需要进行切换环境，花费时间。例如：

```
#define max (a,b) (a>b?a:b)
#define MALLOC(n, type)   ((type *)  malloc( (n) * sizeof (type) ))
```

使用时，我只需：

```
a=max (a,b);                    //而不是 a=(a>b?a:b);
int *p=MALLOC(10,int);   //而不是 int *p= ((int *)  malloc( (10) * sizeof (int) ))
```

网上 copy 一篇不知出自哪里的文章：

1、防止一个头文件被重复包含

```
#ifndef COMDEF_H
#define COMDEF_H  //头文件内容
#endif
```

2、重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。

```
typedef  unsigned char       boolean;      /* Boolean value type. */

typedef  unsigned long int  uint32;       /* Unsigned 32 bit value */
typedef  unsigned short      uint16;       /* Unsigned 16 bit value */
typedef  unsigned char       uint8;        /* Unsigned 8  bit value */

typedef  signed long int     int32;        /* Signed 32 bit value */
typedef  signed short        int16;        /* Signed 16 bit value */
typedef  signed char         int8;        /* Signed 8  bit value */
```

3、得到指定地址上的一个字节或字

```
#define  MEM_B( x )  ( *( (byte *) (x) ) )
#define  MEM_W( x )  ( *( (word *) (x) ) )
```

4、求最大值和最小值

```
   #define  MAX( x, y ) ( ((x) > (y)) ? (x) : (y) )
   #define  MIN( x, y ) ( ((x) < (y)) ? (x) : (y) )
```

5、得到一个 field 在结构体(struct)中的偏移量

```
#define FPOS( type, field )   ( (dword) &(( type *) 0)-> field )
```

6、得到一个结构体中 field 所占用的字节数

```
#define FSIZ( type, field ) sizeof( ((type *) 0)->field )
```

7、按照 LSB 格式把两个字节转化为一个 Word

```
#define  FLIPW( ray ) ( (((word) (ray)[0]) * 256) + (ray)[1] )
```

8、按照 LSB 格式把一个 Word 转化为两个字节

```
#define  FLOPW( ray, val ) \
  (ray)[0] = ((val) / 256); \
  (ray)[1] = ((val) & 0xFF)
```

9、得到一个变量的地址（word 宽度）

```
#define  B_PTR( var )  ( (byte *) (void *) &(var) )
#define  W_PTR( var )  ( (word *) (void *) &(var) )
```

10、得到一个字的高位和低位字节

```
#define  WORD_LO(xxx)  ((byte) ((word)(xxx) & 255))
#define  WORD_HI(xxx)  ((byte) ((word)(xxx) >> 8))
```

11、返回一个比 X 大的最接近的 8 的倍数

```
#define RND8( x )         ((((x) + 7) / 8 ) * 8 )
```

12、将一个字母转换为大写

```
#define  UPCASE( c ) ( ((c) >= 'a' && (c) <= 'z') ? ((c) - 0x20) : (c) )
```

13、判断字符是不是 10 进值的数字

```
#define  DECCHK( c ) ((c) >= '0' && (c) <= '9')
```

14、判断字符是不是 16 进值的数字

```
#define  HEXCHK( c ) ( ((c) >= '0' && (c) <= '9') ||\
                       ((c) >= 'A' && (c) <= 'F') ||\
((c) >= 'a' && (c) <= 'f') )
```

15、防止溢出的一个方法

```
#define  INC_SAT( val )  (val = ((val)+1 > (val)) ? (val)+1 : (val))
```

16、返回数组元素的个数

```
#define  ARR_SIZE( a )  ( sizeof( (a) ) / sizeof( (a[0]) ) )
```

17、返回一个无符号数 n 尾的值 MOD_BY_POWER_OF_TWO(X,n)=X%(2^n)

```
#define MOD_BY_POWER_OF_TWO( val, mod_by ) \
           ( (dword)(val) & (dword)((mod_by)-1) )
```

18、对于 IO 空间映射在存储空间的结构，输入输出处理

```
  #define inp(port)          (*((volatile byte *) (port)))
  #define inpw(port)         (*((volatile word *) (port)))
  #define inpdw(port)        (*((volatile dword *)(port)))

  #define outp(port, val)    (*((volatile byte *) (port)) = ((byte) (val)))
  #define outpw(port, val)  (*((volatile word *) (port)) = ((word) (val)))
  #define outpdw(port, val) (*((volatile dword *) (port)) = ((dword) (val)))
```

19、使用一些宏跟踪调试

ＡＮＳＩ标准说明了五个预定义的宏名。它们是：

ＬＩＮＥ

ＦＩＬＥ

ＤＡＴＥ

ＴＩＭＥ

ＳＴＤＣ

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序也许还提供其它预定义的宏名。

_ＬＩＮＥ_及_ＦＩＬＥ_宏指令在有关#ｌｉｎｅ的部分中已讨论，这里讨论其余的宏名。

_ＤＡＴＥ_宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。

源代码翻译到目标代码的时间作为串包含在_ＴＩＭＥ_中。串形式为时：分：秒。

如果实现是标准的，则宏_ＳＴＤＣ_含有十进制常量 1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如:

当定义了_DEBUG，输出数据信息和所在文件所在行

```
#ifdef _DEBUG
#define DEBUGMSG(msg,date) printf(msg);printf("%d%d%d",date,_LINE_,_FILE_)
#else
      #define DEBUGMSG(msg,date)
#endif
```

20、宏定义防止 使用是错误

用小括号包含。

例如：#define ADD(a,b) （a+b）

用 do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;\

```
                a++;
```

应用时：if(....)

```
                DO(a,b); //产生错误
        else
                ......
```

解决方法: #difne DO(a,b) do{a+b;\

```
                a++;}while(0)
```

sizeof 作用

http://zhidao.baidu.com/link?url=9Sa2qEgZMR-xXtUb922pTtkyVCbXZag-4Q2--q9Xwqpv3jw-_ImfnRon-MX0upatQ3PtELgGprOpIwzFo0_xGq

比如不知道一个数据长度是多少不知道怎么给他分配内存的时候就要用到。如清空变量 A 内存：memset(变量 A,0x00,sizeof(变量 A));这个时候如果是字符串还可以说用数字来代替 sizeof，如果是一个大的结构体，那么清空的内存有多少多大就不太好算了所以还是 sizeof 方便而且不会出错。

又如 memcpy(变量 A，变量 B，sizeof(变量 A)); 这样就方便的以变量 A 的长度来复制内存。

sizeof 主要就是用来确保求数据类型长度的时候不出错，一个上百几千行的程序不可能写到后面还要去回到定义的位置再去看变量的数据长度一个 sizeof 就能确保无误。有的时候就正好当做整型数来用像：strncpy(str1,str2,sizeof(str2)-1);直接进行整型运算就行啦。

UNIX 时间戳查询

http://tool.chinaz.com/Tools/unixtime.aspx