# 目 录

[目 录 I](#_Toc507401378)

[1 C语言基础 1](#_Toc507401379)

[1.1 结构体成员乱序初始化 1](#_Toc507401380)

[1.2 const、static变量存放位置 1](#_Toc507401381)

[1.3 #if 0的作用 3](#_Toc507401382)

[1.4 可变参数宏定义 3](#_Toc507401383)

[1.5 Bit位操作 4](#_Toc507401384)

[1.6 函数声明和定义不一致问题 4](#_Toc507401385)

[1.7 do{…}while(0)的妙用 5](#_Toc507401386)

[1.8 宏定义中#和##符号的使用及宏定义展开问题 5](#_Toc507401387)

[1.9 二进制文件和文本文件的区别 6](#_Toc507401388)

[1.10 数组和结构体最后一个逗号 6](#_Toc507401389)

[1.11 宏定义中的空格 6](#_Toc507401390)

[1.12 memset\_u8设置陷阱 7](#_Toc507401391)

[1.13 结构体的定义是否占用内存空间 7](#_Toc507401392)

[1.14 ->的历史原因 7](#_Toc507401393)

[1.15 不同结构体成员的数据访问一致性问题 8](#_Toc507401394)

[1.16 C程序空间分布 8](#_Toc507401395)

[2 相关格式 11](#_Toc507401396)

[2.1 编号 11](#_Toc507401397)

[2.1.1 1-9）版 11](#_Toc507401398)

[2.1.2 1-10）版 11](#_Toc507401399)

[2.1.3 （1-10）版 12](#_Toc507401400)

[2.1.4 1. 代码版 12](#_Toc507401401)

[2.2 图表 13](#_Toc507401402)

[2.2.1 技术 13](#_Toc507401403)

[参考文献 15](#_Toc507401404)

# C语言基础

* 1. 结构体成员乱序初始化

乱序初始化是C99标准新加的，比较直观的一种初始化方式。相比顺序初始化而言，乱序初始化就如其名，成员可以不按照顺序初始化，而且可以只初始化部分成员，扩展性较好。Linux内核中采用这种方式初始化struct。

乱序初始化有两种方式，一种是用点（.）符号，一种是用冒号（：）。方式1是C99标准，方式2是GCC的扩展，强烈建议使用第一种方式。乱序初始化使用方式如下：

1. //结构体定义
2. **typedef** struct \_data\_t **{**
3. int a**;**
4. int b**;**
5. int c**;**
6. **}**data\_t**;**
7. //乱序初始化结构体1
8. data\_t data\_one **=** **{**
9. **.**c **=** 20**,**
10. **.**b **=** 30**,**
11. **.**a **=** 10**,**
12. **};**
13. //乱序初始化结构体2
14. data\_t data\_two **=** **{**
15. b**:**40**,**
16. c**:**20**,**
17. **}**;
    1. const、static变量存放位置

以下述代码为例，其对应const变量和static变量的存放位置如图 1‑1所示。

1. static int val\_a **=** 1 **;** // 初始化的静态变量
2. int val\_b **=** 2 **;** // 初始化的全局变量
3. const int val\_c **=** 3 **;** // const全局变量
4. static int val\_d **;** // 未初始化的静态变量
5. int val\_e **;** // 未初始化的全局变量
6. int main**()**
7. **{**
8. static int val\_f **=** 5**;** // 初始化的局部静态变量
9. static int val\_g**;** // 未初始化局部静态变量
10. int val\_h **=** 6**;** // 初始化局部变量
11. int val\_i**;** // 未初始化局部变量
12. const int val\_j **=** 7**;** // const局部变量
13. **return** 0**;**
14. **}**

|  |  |  |
| --- | --- | --- |
|  | 0xFFFFFF |  |
| val\_e | 0x405058 | 未初始化的全局变量 |
|  |  |  |
| val\_g | 0x40500C | 未初始化局部静态变量 |
| val\_d | 0x405008 | 未初始化的静态变量 |
|  |  |  |
| val\_c | 0x403024 | const 全局变量 |
|  |  |  |
| val\_f | 0x402008 | 初始化的局部静态变量 |
| val\_b | 0x402004 | 初始化的全局变量 |
| val\_a | 0x402000 | 初始化的静态变量 |
|  |  |  |
| val\_h | 0x28FF1C | 初始化局部变量 |
| val\_i | 0x28FF18 | 未初始化局部变量 |
| val\_j | 0x28FF14 | const局部变量 |
|  |  |  |
| 0x000000 |

图 1‑1 变量存放位置

1. static无论是全局变量还是局部变量都存储在全局/静态区域，在编译期就为其分配内存，在程序结束时释放，例如：val\_a、val\_d、val\_f、val\_g；
2. const全局变量存储在只读数据段，编译期最初将其保存在符号表中，第一次使用时为其分配内存，在程序结束时释放，例如：val\_c；const局部变量存储在栈中，代码块结束时释放，例如：val\_j；
3. 全局变量存储在全局/静态区域，在编译期为其分配内存，在程序结束时释放，例如：val\_b、val\_e；
4. 局部变量存储在栈中，代码块结束时释放，例如：val\_h、val\_i。

注：当全局变量和静态局部变量未赋初值时，系统自动置为0。

* 1. #if 0的作用

#if 0

code

#endif

作用和/\* \*/一样，从而解决嵌套注释/\* /\* \*/ \*/失败的问题。如果想让code生效，只需把#if 0改成#if 1。

* 1. 可变参数宏定义

在C99中规定宏也可以像函数一样带可变的参数，如：

#define Log(fmt, ...) printf(fmt, ##\_\_VA\_ARGS\_\_)

其中，**...**表示可变参数列表，\_\_VA\_ARGS\_\_在预处理中，会被实际的参数集（实参列表）所替换。宏定义只是在预处理阶段简单地替换，如果可变参数列表为空，宏定义将会多出一个"**,**"符号，即Log**(**"abc"**)**替换后为printf**(**"abc"**,** **)**的形式，而使用“##”则不进行连接，所以允许省略可变参数。

1. #define MGS\_SYS\_ERR "-E-:"
2. #define MSG(...) debugmsg\_printf\_func(\_\_VA\_ARGS\_\_)
3. #define MSG\_ERR(...) (CONFIG\_MSG\_LVL < MSG\_LVL\_ERR) ? (void)0 : \ debugmsg\_printf\_func(MGS\_SYS\_ERR \_\_VA\_ARGS\_\_)
4. void dualcore\_mutex\_lock**(**HANDLE mutex\_id**)**
5. **{**
6. ASSERT**(NULL** **!=** mutex\_id**);**
7. u32 ret **=** WaitForSingleObject**(**mutex\_id**,** INFINITE**);**
8. ASSERT**(**WAIT\_OBJECT\_0 **==** ret**);**
9. **}**
10. void dualcore\_mutex\_unlock**(**HANDLE mutex\_id**)**
11. **{**
12. ASSERT**(NULL** **!=** mutex\_id**);**
13. ReleaseMutex**(**mutex\_id**);**
14. **}**
15. void debugmsg\_printf\_func**(**const char **\***fmt**,** **...)**
16. **{**
17. va\_list args**;**//定义一个va\_list型的变量,这个变量是指向参数的指针.
18. char msg\_buf**[**MSG\_STR\_BUF\_LEN\_MAX**];**//MSG\_STR\_BUF\_LEN\_MAX is 256
19. u32 msg\_len**;**
20. #if(DUAL\_CORE == mENABLE)
21. dualcore\_mutex\_lock**(**MUTEX\_MSGOUT**);**
22. #endif
23. va\_start**(**args**,** fmt**);**//用va\_start宏初始化变量,该宏参数固定
24. vsnprintf**(**msg\_buf**,sizeof(**msg\_buf**),** fmt**,** args**);**//若生成的字符串长度大于sizeof，则复制前sizeof个字符到msg\_buf，同时返回原串的长度（不包含终止符）
25. va\_end**(**args**);**//用va\_end宏结束可变参数的获取
26. msg\_len **=** strlen**(**msg\_buf**);**
27. **if(sizeof(**msg\_buf**)** **-** 1 **==** msg\_len**)**
28. **{**
29. printf**(**"%s\r\n"**,**MSG\_BUF\_OVF\_STR**);**//"-E-:Msg Overflow!\r\n"
30. **}**
31. **else**
32. **{**
33. printf**(**"%s\r\n"**,** msg\_buf**);**
34. **}**
35. #if(DUAL\_CORE == mENABLE)
36. dualcore\_mutex\_unlock**(**MUTEX\_MSGOUT**);**
37. #endif
38. **}**
    1. Bit位操作

/\* define bits and bit ops \*/

#define BIT(x) ((u32)1 << (x))

#define SETBIT(val,bit) ((val)|=(bit))

#define CLRBIT(val,bit) ((val)&=(~(bit)))

#define WRBIT(val, bit) ((val) = (bit))

#define CHKBIT(val, bit) ((val)&(bit))

* 1. 函数声明和定义不一致问题

以如下代码为例，由于头文件里函数的声明与C文件里的实现返回值类型不一致，会导致比较隐蔽的C语言问题。

1. //crc.c
2. //此处没有包含crc.h头文件
3. unsigned int get\_crc**(**void**)**
4. **{**
5. **return** 0x12345678**;**
6. **}**
7. //crc.h
8. //声明时不加void等同于get\_crc(...)，调用者可以传递多个参数
9. unsigned short get\_crc**(**void**);**
10. //main.c
11. #include <stdio.h>
12. //该include相当于extern unsigned short get\_crc(void);
13. #include "crc.h"
14. int main **(**int argc**,** char **\***argv**[])**
15. **{**
16. unsigned int crc **=** get\_crc**();**
17. printf**(**"crc:%x\n"**,**crc**);**
18. **return** 0**;**
19. **}**

编译执行: gcc -Wall -o test main.c crc.c，没有警告或者报错，在x86平台下输出:5678，而在powerpc平台下输出：12345678。

由于crc.c文件中没有包含crc.h头文件，所以main.c中函数的声明中默认带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有明显的区别：extern int f();和int f();

该问题解决办法很简单，把函数声明（原型）放在头文件中，而定义则放在另一个包含了该头文件的源文件中，这样编译器就能发现不一致的情况从而报错提醒我们。

类似问题还有函数指针定义时类型不匹配问题，不过此时编译器会有警告，从而避免错误的发生，代码示例如下：

1. #include <stdio.h>
2. //typedef void (\*print\_fn)()相当于(...)，输出为258
3. **typedef** void **(\***print\_fn**)(**unsigned char val**);**
4. //函数定义与函数指针的参数类型不一致
5. void print\_val**(**unsigned int val**)**
6. **{**
7. printf**(**"val is %d\n"**,** val**);**
8. **}**
9. int main **(**int argc**,** char **\***argv**[])**
10. **{**
11. print\_fn fn\_ptr **=** print\_val**;**
12. fn\_ptr**(**2 **+** 256**);**//输出为2
13. **return** 0**;**
14. **}**
    1. do{…}while(0)的妙用

do{...}while(0) 并非循环，但可以用来提高代码的健壮性，属于C语言特性。

1. 辅助定义复杂的宏，避免引用的时候出错；

#define AB1 a; b; //下面语句中b分支不受cond条件的约束if(cond) AB1;

#define AB2 {a; b;}//下面语句编译出错：if(cond) AB2; else …;

#define AB3 a, b //有运算符优先级问题

#define AB4 do{a; b;} while(0)

1. 避免使用goto语句，将主体使用do()while(0)包含起来，用break语句做跳出；
2. 避免空宏引起的warning；

#define EMPTYMICRO do{}while(0)

1. 替代{}实现局部作用域，使得代码分块更直观，同时可以缩小作用域。
   1. 宏定义中#和##符号的使用及宏定义展开问题

#是把宏参数变为一个字符串，##是把两个宏参数连接在一起。有一道经典的C语言问题，程序如下：

1. #include <stdio.h>
2. #define f(a,b) a##b
3. #define g(a) #a
4. #define h(a) g(a)
6. int main**(**int argc**,** char **\***argv**[])**
7. **{**
8. //#define h(a) g(a)参数前无#及##操作,则先展开宏参数
9. printf**(**"%s\n"**,** h**(**f**(**1**,**2**)));**
10. //#define g(a) #a宏参数前有#,不继续展开宏参数
11. printf**(**"%s\n"**,** g**(**f**(**1**,**2**)));**
12. **return** 0**;**
13. **}**

答案：第一行：12 第二行：f(1**,** 2)

说明：在展开当前宏函数时，如果形参有#或##则不进行宏参数的展开，否则先展开宏参数，再展开当前宏。

* 1. 二进制文件和文本文件的区别

1. 文本文件操作函数

文本文件操作函数读写的内容当字符看待，读时把\r\n替换成\n，写时把\n替换成\r\n，且读到\0x1A就设置EOF，其主要函数为：fputc、fgetc、fputs、fgets、fprintf、fscanf。

1. 二进制文件读写函数

二进制文件操作函数读写的内容当字节看待，因此没有数据转换，避免了数据精度的丢失，同时也避免转换成文本文件格式，使得二进制文件的读写速度相比较于文本文件要快一点，且读操作遇到\0x1A仍继续，需要使用feof函数来判断文件是否结束，其主要函数为：fread、fwrite。

* 1. 数组和结构体最后一个逗号

数组和结构体初始化列表中最后一个元素后面多加一个逗号结果与不加逗号一致，即没有什么意义，不会对数组和结构体的大小造成影响，但能使以后代码维护时少输入一个逗号，便于自动化脚本交互。

* 1. 宏定义中的空格

1. 对于有参数的宏定义，宏定义时在宏名与带参数的括号之间不应加空格，否则将空格以后的字符都作为替代字符串的一部分；
2. 带参数的宏定义只是进行简单的字符替换，而宏展开则是在编译前进行的，在展开时并不分配内存单元，不进行值的传递处理，因此宏定义替换不会占用运行时间，只占用编译时间。
3. 宏不存在类型问题，宏名无类型，宏的参数也无类型，只是一个符号代表，展开时代入指定的字符串即可，宏定义时字符串可以是任何类型的数据。
4. 转义字符是在反斜杠"\"后面加一个字符，但如果反斜杠用于一行的结尾，表示本行与下一行连接起来，此时反斜杠后不可有空格，否则"\"就成了"\ "。
5. 字符串" "中永远不包含宏
   1. memset\_u8设置陷阱

长度为u16或者u32类型的寄存器，不能使用memset\_u8来设置，因为寄存器的硬件设计可能约束了该寄存器只接受长度为u16或者u32的数据；

此外，如果Dram的总线位宽是16位，那么memset\_u8的设置过程可能是RMW操作，即硬件控制器先读出原来的U16的数据，然后修改，最后再写入数据。

* 1. 结构体的定义是否占用内存空间

变量与内存地址在编译的时候就一一对应了，而结构体类型主要封装一些属性来组成新的数据类型，因此结构体的定义只是在编译时用于确定结构体变量占用内存的大小，结构体的定义并不占用内存空间，只有结构体的变量才会分配内存，且结构体变量的内存地址与结构体中第一个成员在内存中的地址相同。

C中union的使用和struct一样，只有先定义了共用体变量后才能进行引用，而且不能直接引用共用体变量，只能引用共用体变量中的某个成员。例如，以下代码直接引用union的使用方式是错误的，因为val的存储区中存在好几种类型，分别占用不同长度的存储区，仅写共用体变量名val会使编译器无法确定究竟输出的哪一个成员的值。

1. union test val**;**
2. printf**(**"%d\n"**,** val**);**

由于union中的成员共享内存，所以不能在union中定义静态或引用类型的变量，同时union中也不允许存放带有构造函数、析构函数和复制构造函数等类的对象，但是可以存放对应类对象的指针，因为编译器无法保证类的构造函数和析构函数得到正确的调用，由此可能发生内存泄漏。所以，我们在C++中使用union时，尽量保持C语言中的union使用风格。

* 1. ->的历史原因

从机器码和汇编的角度来看，不存在变量，也不存在结构体，只有寄存器和一个叫做内存的大数组，所以变量是C语言对内存地址的一种抽象，变量代表的是一个位置。例如，C中的a = b，从汇编的角度来看更像是\*A = \*B，其中A和B各是一个内存地址，也就是指针。以此为背景，再来看结构体是什么，以及结构体的成员是什么，假设我们有如下代码：

1. struct Point **{**
2. int x**;**
3. int y**;**
4. **};**
5. struct Point p**;**
6. struct Point **\***pp **=** **&**p**;**

从现代语义上讲，*p*就是一个结构体对象，*x*和*y*各是其成员。而从汇编的语义上讲，*p*是一个不完整的地址，或者说，半个地址，再或者说，一个指向的东西是虚构出来的地址，而*x*和*y*各是在Point结构中的地址偏移量。也就是说，必须有*p*和*x*或者*p*和*y*同时出现，才能形成一个完整的地址，而单独的一个*p*没有意义。

早期的C就是在这样的模型上建立的，所以对于早期的C而言，\**pp*没有意义。\**pp*表示你取得了一个struct，而这个struct不能塞在任何一个寄存器里，编译器和CPU都无法表达这个东西，这时候只有*p.x*和*p.y*有意义，因为它们具有真实的地址。所以对于早期的C而言，(\**pp*).x = 1是错的，因为作为这个赋值的目标地址表达式的一部分，\**pp*这个中间结果没法直译到机器码。早期的C就是这样一个看起来怪异的语义，但它更贴近于机器的表达。所以早期的C就发明了->，表示解引用和取成员的偏移这两个操作紧密结合起来变成一个单独的操作，于是才能写*pp*->*x* = 1，这就是它存在的历史原因。现在这个历史原因现在已经不重要了，现代的符合标准的C编译器都知道(\**pp*).x和*pp* ->x是等价的。

* 1. 不同结构体成员的数据访问一致性问题

结构体对象*a*中有两个成员变量*b*和*c*，且系统中有两个线程，某一时刻，线程1需要操作*a.b*，线程2需要操作*a.c*，例如*a.b* = 5和*a.c* = 10，这两条语句实际上是对两个不同的内存地址进行写操作，也就相当于对两个不同的变量进行操作，所以这两个线程可同时运行而无需考虑对象*a*的数据一致性问题。不过还是要具体问题具体分析，如果结构体中的两个变量并非简单的int类型，而是结构体或C++对象，这些结构体或对象中又有指针，这些指针又指向了同一个东西，就容易出问题。

* 1. C的代码内存布局

C语言程序分为映像和运行时两种状态。在编译、连接后形成的映像中，即存放在存储盘上可执行文件，其中只包含代码段、只读数据段、读写数据段和其他段（调试的段、动态库共享库链接表的段）。在程序运行之前，将动态生成未初始化数据段（BSS）。在程序的运行时还将另外形成堆（Heap）区域和栈（Stack）区域。可执行文件的内存分布如图 1‑2所示。

BSS段：存放程序中未初始化的全局变量和未初始化的static变量。

数据段：存放程序中已初始化的全局（静态）变量，分为读写数据段和只读数据段。

代码段：属于只读的内存区域，存放程序执行代码和只读的常数变量，例如字符串常量等。

栈：存放函数内部的变量、参数和返回地址，其在函数被调用时自动分配。

堆：存放进程运行中被动态分配的内存段。



图 1‑2 可执行文件中的段在内存中如何布局

一般来说，直接定义的全局变量在未初始化数据段，如果该变量有初始化则在已初始化数据段（RW Data），加上const修饰符后具有只读属性，其实不管全局变量还是局部变量，都只代表一个地址，所以在编译之后，变量的定义语句都不会占用目标文件中代码段的空间，经过初始化的全局变量占用数据段的空间，而经过初始化的局部变量在运行时占用栈空间，同时会在目标文件的代码段中保留赋值语句。代码示例如下：

1. int a **=** 10**;**//全局(静态)初始化数据段
2. char **\***p1**;**//全局未初始化数据段
3. const char ro**[]** **=** "const data"**;**//ro具有全局(静态)初始化数据段中的只读属性
4. int main**(**void**)**
5. **{**
6. static int b **=** 0**;**//全局(静态)初始化数据段
7. short c**;**//栈中占2字节
8. char s**[]** **=** "abcde"**;**//s在栈上占4字节；"abcde"在只读数据段占6字节
9. char **\***p2 **=** "123456"**;**//p2在栈上占4字节；"123456\0"在常量区占7字节
10. p1 **=** **(**char **\*)**malloc**(**a**\*sizeof(**char**));**//在堆区分配10字节的区域
11. strcpy**(**p1**,** "XXXX"**);**//"XXXX"在只读数据段占5字节
12. free**(**p1**);**
13. **return** 0**;**
14. **}**

可执行文件的内存分布有如下三点注意事项：

1. BSS段只保存没有值的变量，而目标文件不需要保存这些变量的映像。运行时所需要的BSS段大小记录在目标文件中，但BSS段并不占据目标文件的任何空间；
2. 局部变量在编译之后并不进入目标文件中，它们在运行时创建，但是经过初始化的局部变量相当于定义和赋值两部分，所以代码中仅会保留赋值部分；
3. const类型的存储跟一般的变量没有区别，在外部定义的存储在全局数据区，static的存储在静态数据区，在函数内部定义的存储在栈，const跟非const在存储上没区别，只不过是read only的。
   1. ARM程序的组成

ARM程序是指在ARM系统中正在执行的程序，而非保存在ROM中的bin映像（Image）文件。一个ARM程序包含3部分：RO，RW和ZI，各部分含义如下：

1. RO是程序中的指令和常量；
2. RW是程序中的已初始化变量；
3. ZI是程序中的未初始化的变量。

ARM映像文件是指烧录到ROM中的bin文件，也称为Image文件。Image文件包含RO和RW数据，而不包含ZI数据，因为ZI数据都是0，没必要包含，只要程序运行之前将ZI数据所在的区域一律清零即可，包含进去反而浪费存储空间。

* 1. 指针的加法

一般地，T\* p = ...; p += i; //p是增加i\*sizeof(T)。

数组名的类型为指向数组第一个元素的常量指针，数组名的使用如下述代码所示，但有两种例外：

1. 数组名作为sizeof操作符的操作数时，sizeof返回整个数组的长度，而不是指向数组的指针的长度。
2. 数组名作为单目运算符&的操作数时，也就是说取该数组名的地址产生的是一个指向数组的指针，而不是指向一个指向某个指针常量值的指针。
3. int a**[**5**];**
4. a //&a[0]
5. a**+**2 //&a[2]
6. **&**a //int (\*)[5]
7. **&**a**+**1//下一个a[5]数组

有阴影的寄存器，表示在物理上这个寄存器对应2个寄存器，一个是程序员可以写入或读出的寄存器，称为preload register(预装载寄存器)，另一个是程序员看不见的、但在操作中真正起作用的寄存器，称为shadow register(影子寄存器)；preload register的内容可以随时传送到shadow register，即两者是连通的(permanently)，或者在每一次更新事件(UEV)时才把preload register的内容传送到shadow register。

设计preload register和shadow register的好处是，所有真正需要起作用的寄存器(shadow register)可以在同一个时间(发生更新事件时)被更新为所对应的preload register的内容，这样可以保证多个通道的操作能够准确地同步。如果没有shadow register，或者preload register和shadow register是直通的，即软件更新preload register时，同时更新了shadow register，因为软件不可能在一个相同的时刻同时更新多个寄存器，结果造成多个通道的时序不能同步，如果再加上其它因素(例如中断)，多个通道的时序关系有可能是不可预知的。

指某些寄存器的镜像寄存器，用于与原始寄存器同步，但当进入某些

Exception Handler处理时，Raw寄存器值发生变化，而Shadow寄存器值不变，

这样出Handler时，Raw寄存器值可从Shadow寄存器恢复。

（如果我和你谈的是同一个概念的话）

# 相关格式

* 1. 编号
     1. 1-9）版

1. static无论是全局变量还是局部变量都存储在全局/静态区域，在编译期就为其分配内存，在程序结束时释放，例如：val\_a、val\_d、val\_f、val\_g；
2. const全局变量存储在只读数据段，编译期最初将其保存在符号表中，第一次使用时为其分配内存，在程序结束时释放，例如：val\_c；const局部变量存储在栈中，代码块结束时释放，例如：val\_j；
3. 全局变量存储在全局/静态区域，在编译期为其分配内存，在程序结束时释放，例如：val\_b、val\_e；
   * 1. 1-10）版
4. D能量消耗，降低系统运行成本，并可以延长系统工作时间。近些年许多学者基于操
5. 作
6. 系
7. 统
8. 层
9. 的
10. D
11. P
12. M
13. 技
14. 术对这个问题进行了深入的研究。在本节中，我们
    * 1. （1-10）版
15. 任务之间支持抢占；
16. 备能量消耗，降低系统运行成本，并可以延长系统工作时间。近些年许多学者基于操作系统层的DPM技术对这个问题进行了深入的研究。在本节中，我们首先介
17. 绍
18. 基
19. 于
20. 动
21. 态
22. 调频
23. 备能量
24. 消
25. 系
    * 1. 1. 代码版
26. 低系统运行成本，并可以延长系统工作时间。近些年许多学者基于操作系的的的的统层的DPM技
27. 术
28. 对
29. 这
30. 个
31. 问
32. 题
33. 进
34. 行
35. **typedef** struct \_data\_t **{**
36. int a**;**
37. int b**;**
38. int c**;**
39. **}**data\_t**;**
40. //乱序初始化结构体1
41. data\_t data\_one **=** **{**
42. **.**c **=** 20**,**
43. **.**b **=** 30**,**
44. **.**a **=** 10**,**
45. **};**
46. //乱序初始化结构体2

能量消耗，降低系统运行成本，并

化来降低能耗，利用底层硬件提供的特性，在系统运行时降低系统总体功耗，从而节省大量的能量。

* 1. 图表
     1. 技术

**错误!未找到引用源。**中所示的设备，都是实际中使用的满足本文DPM模型的设备，这些设备的参数来自于文献**错误!未找到引用源。**。

表 2‑1 具有DPM功能的设备的功耗以及*T*be时间

|  |  |  |  |
| --- | --- | --- | --- |
| 设备 | *P*m,i(mW) | *T*m,i(ms) | *T*be(ms) |
| 传感器节点 | 1040/400/270/200/10 | 5/15/20/50 | 8/20/25/50 |
| 网卡（Linksys） | 0.76/0.0 | 2.75 | 3.61 |
| 硬盘（IBM） | 10.2/2.5 | 12.4 | 15.2 |
| Beowolf集群节点 | 1/0.766/0.1/0.01 | 3/7/70 | 6/10/100 |
| 笔记本LCD | 21.1/17.1 | 7.6 | 15.6 |
| 无线网卡 | 0.9/0 | 0.3 | 0.7 |

**错误!未找到引用源。**中的传感器节点具有四种低功耗模式。我们使用该传感器节点的设备参数以

及一些具体任务来描述最理想的DPM策略所需要考虑的综合标准，此外，设备处于睡眠状态所消耗的能量依赖于空闲时间的长度，如图 2‑1所示。



图 2‑1 空闲时间与能量之间凹函数曲线

考虑四个周期性任务组成的任务集*τ* = {*τ*1*, τ*2*, τ*3*, τ*4}，这些任务的释放时间以及执行时间分别为*φ*1=0，*φ*2=100，*φ*3=200，*φ*4=300，*C*1=15，*C*2=78，*C*3=90和*C*4=100，所有任务的周期*T*=400。

本文的章节安排如下：

1. 绪论 介绍本文的研究背景、研究现状以及本文的主要研究内容。

# 参考文献

1. ACPI. 2011. Advanced configuration and power interface standard. http://www.acpi.info