问题：

① 编译和连接哪个环节会出问题？
② 显示出的错误信息是什么？
③ 这个错误信息可能与哪个文件相关？

(2) 用学习汇编语言时使用的 link.exe 对 tc.exe 生成的 f.obj 文件进行连接，生成 f.exe。用 Debug 加载 f.exe，察看整个程序的汇编代码。思考相关的问题。

问题：

① f.exe 的程序代码总共有多少字节？
② f.exe 的程序能正确返回吗？
③ f 函数的偏移地址是多少？

(3) 写一个程序 m.c。

```
main()
{
   *(char far *)(0xb8000000+160*10+80)='a';

   *(char far *)(0xb8000000+160*10+81)=2;
}
```

用 tc.exe 对 m.c 进行编译，连接，生成 m.exe，用 Debug 察看 m.exe 整个程序的汇编代码。思考相关的问题。

问题：

① m.exe 的程序代码总共有多少字节？
② m.exe 能正确返回吗？
③ m.exe 程序中的 main 函数和 f.exe 中的 f 函数的汇编代码有何不同？

(4) 用 Debug 对 m.exe 进行跟踪：①找到对 main 函数进行调用的指令的地址；②找到整个程序返回的指令。注意：使用 g 命令和 p 命令。

(5) 思考如下几个问题：

① 对 main 函数调用的指令和程序返回的指令是哪里来的？
② 没有 main 函数时，出现的错误信息里有和“c0s”相关的信息；而前面在搭建开发环境时，没有 c0s.obj 文件 tc.exe 就无法对程序进行连接。是不是 tc.exe 把 c0s.obj 和用户程序的.obj 文件一起进行连接生成.exe 文件？
③ 对用户程序的 main 函数进行调用的指令和程序返回的指令是否就来自 c0s.obj 文件？
④ 我们如何看到 c0s.obj 文件中的程序代码呢？
⑤ c0s.obj 文件里有我们设想的代码吗？

(6) 用 link.exe 对 c:\minic 目录下的 c0s.obj 进行连接，生成 c0s.exe。

用 Debug 分别察看 c0s.exe 和 m.exe 的汇编代码。注意：从头开始察看，两个文件中的程序代码有何相同之处？

(7) 用 Debug 找到 m.exe 中调用 main 函数的 call 指令的偏移地址，从这个偏移地址开始向后察看 10 条指令；然后用 Debug 加载 c0s.exe，从相同的偏移地址开始向后察看 10 条指令。对两处的指令进行对比。

(8) 从上我们可以看出，tc.exe 把 c0s.obj 和用户.obj 文件一同进行连接，生成.exe 文件。按照这个方法生成的.exe 文件中的程序的运行过程如下。

① c0s.obj 里的程序先运行，进行相关的初始化，比如，申请资源、设置 DS、SS 等寄存器；

② c0s.obj 里的程序调用 main 函数，从此用户程序开始运行；

③ 用户程序从 main 函数返回到 c0s.obj 的程序中；

④ c0s.obj 的程序接着运行，进行相关的资源释放，环境恢复等工作；

⑤ c0s.obj 的程序调用 DOS 的 int 21h 例程的 4ch 号功能，程序返回。

看来，C 程序必须从 main 函数开始，是 C 语言的规定，这个规定不是在编译时保证的(tc.exe 对 f.c 的编译是可以通过的)，也不是连接的时候保证的(虽然，tc.exe 文件对 f.obj 文件不能连接成 f.exe，但 link.exe 却可以)，而是用如下的机制保证的。

首先，C 开发系统提供了用户写的应用程序正确运行所必须的初始化和程序返回等相关程序，这些程序存放在相关的.obj 文件(比如，c0s.obj)中。

其次，需要将这些文件和用户.obj 文件一起进行连接，才能生成可正确运行的.exe 文件。

第三，连接在用户.obj 文件前面的由 C 语言开发系统提供的.obj 文件里的程序要对 main 函数进行调用。

基于这种机制，我们只要改写 c0s.obj，让它调用其他函数，编程时就可以不写 main 函数了。

下面，我们用汇编语言编一个程序 c0s.asm，然后把它编译为 c0s.obj，替代 c:\minic 目录下的 c0s.obj。

程序 c0s.asm：

```
assume cs:code

data segment

  db 128 dup (0)
```

```
data ends

code segment

  start: mov ax,data
         mov ds,ax
         mov ss,ax
         mov sp,128

         call s

         mov ax,4c00h
         int 21h

s:

code ends

end start
```

用 masm.exe 对 c0s.asm 进行编译，生成 c0s.obj，把这个 c0s.obj 复制到 c:\minic 目录下覆盖由 tc2.0 提供的 c0s.obj。

(9) 在 c:\minic 目录下，用 tc.exe 将 f.c 重新进行编译，连接，生成 f.exe。这次能通过连接吗？f.exe 可以正确运行吗？用 Debug 察看 f.exe 的汇编代码。

(10)在新的 c0s.obj 的基础上，写一个新的 f.c，向安全的内存空间写入从“a”到“h”的 8 个字符。分析、理解 f.c。

程序 f.c：

```
#define Buffer ((char *)*(int far *)0x200)

f()
{
   Buffer=0;

   Buffer[10]=0;

   while(Buffer[10]!=8)
   {
      Buffer[Buffer[10]]='a'+Buffer[10];

      Buffer[10]++;
   }
}
```

注意，完成上面的相关试验后，把 c:\minic 目录下的 c0s.obj 文件恢复为 tc2.0 提供的 c0s.obj 文件。

研究试验 5　函数如何接收不定数量的参数

用 c:\minic 下的 tc.exe 完成下面的试验。

(1) 写一个程序 a.c：

```
void showchar(char a,int b);

main()
{
  showchar('a',2);
}

void showchar(char a,int b)
{
   *(char far *)(0xb8000000+160*10+80)=a;

   *(char far *)(0xb8000000+160*10+81)=b;
}
```

用 tc.exe 对 a.c 进行编译，连接，生成 a.exe。用 Debug 加载 a.exe，对函数的汇编代码进行分析。解答这两个问题：main 函数是如何给 showchar 传递参数的？showchar 是如何接收参数的？

(2) 写一个程序 b.c：

```
void showchar(int,int,...);

main()
{
   showchar(8,2,'a','b','c','d','e','f','g','h');
}

void showchar(int n,int color,...)
{
   int a;

   for(a=0;a!=n;a++)
   {
      *(char far *)(0xb8000000+160*10+80+a+a)=*(int *)(_BP+8+a+a);

      *(char far *)(0xb8000000+160*10+81+a+a)=color;
   }
}
```

分析程序 b.c，深入理解相关的知识。

思考：showchar 函数是如何知道要显示多少个字符的？printf 函数是如何知道有多少个参数的？

(3) 实现一个简单的 printf 函数，只需要支持“%c、%d”即可。

附　　注

附注 1　Intel 系列微处理器的 3 种工作模式

微机中常用的 Intel 系列微处理器的主要发展过程是：8080，8086/8088，80186，80286，80386，80486，Pentium，PentiumⅡ，PentiumⅢ，Pentium4。

8086/8088 是一个重要的阶段，8086 和 8088 是略有区别的两个功能相同的 CPU。8088 被 IBM 用在了它所生产的第一台微机上，该微机的结构事实上成为以后微机的基本结构。

80386 是第二个重要的型号，随着微机应用及性能的发展，在微机上构造可靠的多任务操作系统的问题日益突出。人们希望(或许是一种潜在的希望，一旦被挖掘出来，便形成了一个最基本的需求)自己的 PC 机能够稳定地同时运行多个程序，同时处理多项工作；或将 PC 机用作主机服务器，运行 UNIX 那样的多用户系统。

8086/8088 不具备实现一个完善的多任务操作系统的功能。为此 Intel 开发了 80286，80286 具备了对多任务系统的支持。但对 8086/8088 的兼容却做得不好。这妨碍了用户对原 8086 机上的程序的使用。IBM 最早基于 80286 开发了多任务系统 OS/2，结果犯了一个战略错误。

随后 Intel 又开发了 80386 微处理器，这是一个划时代的产品。它可以在以下 3 个模式下工作。

(1)　实模式：工作方式相当于一个 8086。

(2)　保护模式：提供支持多任务环境的工作方式，建立保护机制(这与 VAX 等小型机类似)。

(3)　虚拟 8086 模式：可从保护模式切换至其中的一种 8086 工作方式。这种方式的提供使用户可以方便地在保护模式下运行一个或多个原 8086 程序。

以后的各代微处理器都提供了上述 3 种工作模式。

你也许会说：“喂，先生，你说的太抽象了，这 3 种模式我如何感知？”

其实 CPU 的这 3 种模式只要用过 PC 机的人都经历过。任何一台使用 Intel 系列 CPU 的 PC 机只要一开机，CPU 就工作在实模式下。如果你的机器装的是 DOS，那么在 DOS 加载后 CPU 仍以实模式工作。如果你的机器装的是 Windows，那么 Windows 加载后，将由 Windows 将 CPU 切换到保护模式下工作，因为 Windows 是多任务系统，它必须在保护模式下运行。如果你在 Windows 中运行一个 DOS 下的程序，那么 Windows 将 CPU 切换

到虚拟 8086 模式下运行该程序。或者是这样，你点击开始菜单在程序项中进入 MS-DOS 方式，这时，Windows 也将 CPU 切换到虚拟 8086 模式下运行。

可以从保护模式直接进入能运行原 8086 程序的虚拟 8086 模式是很有意义的，这为用户提供了一种机制，可以在现有的多任务系统中方便地运行原 8086 系统中的程序。这一点，在 Windows 中我们都可以体会到，你在 Windows 中想运行一个原 DOS 中的程序，只用鼠标点击一下它的图标即可。80286CPU 的缺陷在于，它只提供了实模式和保护模式，但没有提供虚拟 8086 模式。这使基于 80286 构造的多任务系统，不能方便地运行原 8086 系统中的程序。如果运行原 8086 系统中的程序，需要重新启动计算机，使 CPU 工作在实模式下才行。这意味着什么？意味着将给用户造成很大的不方便。假设你使用的是基于 80286 构造的 Windows 系统，就会发生这样的情况：你正在用 Word 写一篇论文，其中用到了一些从前的数据，你必须运行原 DOS 下的 DBASE 系统来看一下这些数据。这时你只能停下现有的工作，重新启动计算机，进入实模式工作。你看完了数据，继续写论文，可过了一会儿，你发现又有些数据需要参考，于是你又得停下现有的工作，重新启动计算机……

幸运的是，我们用的 Windows 是基于 80386 的，我们可以以这样轻松的方式工作，开两个窗口，一个是工作于保护模式的 Word，一个是工作于虚拟 8086 模式的 DBASE，我们可以方便地在两个窗口中切换，只要用鼠标点一下就行。

前面讲过，我们在 8086PC 机的基础上学习汇编语言。但现在知道，我们实际的编程环境是当前 CPU 的实模式。当然，有些程序也可以在虚拟 8086 模式下运行。

如果你仔细阅读了上面的内容，或已具备相关的知识，你会发现，从 80386 到当前的 CPU，提供 8086 实模式的目的是为了兼容。现今 CPU 的真正有效力的工作模式是支持多任务操作系统的保护模式。这也许会引发你的一个疑问：“为什么我们不在保护模式下学习汇编语言?”

类似的问题很多，我们都希望学习更新的东西，但学习的过程是客观的。任何合理的学习过程(尽可能排除走弯路、盲目探索、不成系统)都是一个循序渐进的过程。我们必须先通过一个易于全面把握的事物，来学习和探索一般的规律和方法。信息技术是一个发展非常快、日新月异的技术，新的东西不断出现，使人在学习的时候往往无所适从。在你的身边不断有这样的故事出现：COOL 先生用了 3 天(或更短)的时间就学会了某某语言，并开始用它编写软件。在这个故事的感召下，一个初学者也去尝试，但完全是另外一种结果。COOL 先生的快速学习只是露出水面的冰山一角，深藏水下的是他的较为系统的相关基础知识和相关的技术。在开始的时候学习保护模式下的编程，是不现实的，保护模式下所涉及的东西对初学者来说太复杂。你必须知道很多知识后，才能开始编写第一个小程序。相比之下 8086 就合适得多。

附注 2　补　　码

以 8 位的数据为例，对于无符号数来说是从 00000000b~11111111b 到 0~255 一一对应的。那么我们如何对有符号数进行编码呢？即我们如何用 8 位数据表示有符号数呢？

既然表示的数有符号，则必须要能够区分正、负。

首先，我们可以考虑用 8 位数据的最高位来表示符号，1 表示负，0 表示正，而用其他位表示数值。如下：

```
00000000b：0
00000001b：1
00000010b：2
01111111b：127
10000000b：？
10000001b：-1
10000010b：-2
11111111b：-127
```

可见，用上面的表示方法，8 位数据可以表示-127~127 的 254 个有符号数。从这里我们看出一些问题，8 位数据可以表示 255 种不同的信息，也就是说应该可以表示 255 个有符号数，可用上面的方法，只能表示 254 个有符号数。注意，用上面的方法，00000000b 和 10000000b 都表示 0，一个是 0，一个是-0，当然不可能有-0。可以看出，这种表示有符号数的方法是有问题的，它并不能正确地表示有符号数。

我们再考虑用反码来表示，这种思想是，我们先确定用 00000000b~01111111b 表示 0~127，然后再用它们按位取反后的数据表示负数。如下：

```
00000000b：0                11111111b：？
00000001b：1                11111110b：-1
00000010b：2                11111101b：-2
01111111b：127              10000000b：-127
```

可以看出，用反码表示有符号数存在同样的问题，0 出现重码。

为了解决这种问题，采用一种称为补码的编码方法。这种思想是：先确定用 00000000b~01111111b 表示 0~127，然后再用它们按位取反加 1 后的数据表示负数。如下：

```
00000000b：0                11111111b+1=00000000b：0
00000001b：1                11111110b+1=11111111b：-1
00000010b：2                11111101b+1=11111110b：-2
01111111b：127              10000000b+1=10000001b：-127
```

观察上面的数据，我们可以发现，在补码方案中：

(1) 最高位为 1，表示负数；

(2) 正数的补码取反加 1 后，为其对应的负数的补码；负数的补码取反加 1 后，为其绝对值。比如：

1 的补码为：00000001b，取反加 1 后为：11111111b，表示-1；

-1 的补码为:11111111b，取反加 1 后为：00000001b，其绝对值为 1。

我们从一个负数的补码不太容易看出它所表示的数据，比如：11010101b 表示的数据是多少？

但是我们利用补码的特性，将 11010101b 取反加 1 后为：00101011b。可知 11010101b 表示的负数的绝对值为：2BH，则 11010101b 表示的负数为-2BH。

那么-20 的补码是多少呢？

用补码的特性，-20 的绝对值是 20，00010100b，将其取反加 1 后为：11101100b。可知-20H 的补码为：11101100b。

那么 10000000b 表示多少呢？

10000000b 取反加 1 后为：10000000b，其大小为 128，所以 10000000b 表示-128。

8 位补码所表示的数的范围：-128~127。

补码为有符号数的运算提供了方便，运算后的结果依旧满足补码规则。

比如：

```
  计算          补码表示
   10          00001010b
+(-20)         11101100b
  -10          11110110b
```

附注 3 汇编编译器(masm.exe)对 jmp 的相关处理

1. 向前转移

```
s:  :
    :
    :
    jmp s (jmp short s、jmp near ptr s、jmp far ptr s)
```

编译器中有一个地址计数器(AC)，编译器在编译程序过程中，每读到一个字节 AC 就加 1。当编译器遇到一些伪操作的时候，也会根据具体情况使 AC 增加，如 db、dw 等。

在向前转移时，编译器可以在读到标号 s 后记下 AC 的值 as，在读到 jmp ... s 后记下

AC 的值 aj。编译器可以用 as-aj 算出位移量 disp。

此时，编译器作如下处理。

(1) 如果 disp∈[-128,127]，则不管汇编指令格式是：

```
jmp s
jmp short s
jmp near ptr s
jmp far ptr s
```

中的哪一种，都将它转变为 jmp short s 所对应的机器码。

jmp short s 所对应的机器码格式为：EB disp(占两个字节)

编译，连接以下程序，用 Debug 进行反汇编查看。

```
assume cs:code
code segment
s:    jmp s
      jmp short s
      jmp near ptr s
      jmp far ptr s
code ends
end s
```

(2) 如果 disp∈[-32768,32767]，则：

对于 jmp short s 将产生编译错误；

对于 jmp s、jmp near ptr s 将产生 jmp near ptr s 所对应的机器码，jmp near ptr s 所对应的机器码格式为：E9 disp(占 3 个字节)；

对于 jmp far ptr s 将产生相应的编码，jmp far ptr s 所对应的机器码格式为：EA 偏移地址 段地址(占 5 个字节)。

编译，连接以下程序。

```
assume cs:code
code segment
s:    db 100 dup (0b8h,0,0)
      jmp short s
      jmp s
      jmp near ptr s
      jmp far ptr s
code ends
end s
```

编译中将产生错误，错误是由 jmp short s 引起的，去掉 jmp short s 后再编译就可以通过。用 Debug 进行反汇编查看。

2. 向后转移

```
    jmp s (jmp short s、jmp near ptr s、jmp far ptr s)
     :
     :
s:   :
```

在这种情况下，编译器先读到 jmp ... s 指令。由于它还没有读到标号 s ，所以编译器此时还不能确定标号 s 处的 AC 值。也就是说，编译器不能确定位移量 disp 的大小。

此时，编译器将 jmp ... s 指令都当作 jmp short s 来读取，记下 jmp ... s 指令的位置和 AC 的值 aj ，并作如下处理。

对于 jmp short s ，编译器生成 EB 和 1 个 nop 指令(相当于预留 1 个字节的空间，存放 8 位 disp)；

对于 jmp s 和 jmp near ptr s，编译器生成 EB 和两个 nop 指令(相当于预留两个字节的空间，存放 16 位 disp)；

对于 jmp far ptr s，编译器生成 EB 和 4 个 nop 指令(相当于预留 4 个字节的空间，存放段地址和偏移地址)。

作完以上处理后，编译器继续工作，当向后读到标号 s 时，记下 AC 的值 as，并计算出转移的位移量：disp=as−aj。

此时，编译器作如下处理。

(1) 当 disp∈[−128,127]时，不管指令格式是：

```
jmp short s
jmp s
jmp near ptr s
jmp far ptr s
```

中的哪一种，都在前面记下的 jmp ... s 指令位置处添上 jmp short s 对应的机器码(格式为：EB disp)。

注意，此时，对于 jmp s 和 jmp near ptr s 格式，在机器码 EB disp 后还有 1 条 nop 指令；对于 jmp far ptr s 格式，在机器码 EB disp 后还有 3 条 nop 指令。

编译，连接以下程序，用 Debug 进行反汇编查看。

```
assume cs:code
code segment
begin:jmp short s
      jmp s
      jmp near ptr s
      jmp far ptr s
s:    mov ax,0
code ends
end begin
```

(2) 当 disp∈ [−32768,32767]时，则：

对于 jmp short s，将产生编译错误；

对于 jmp s、jmp near ptr s，在前面记下的 jmp ... s 指令位置处添上 jmp near ptr s 所对应的机器码(格式为：E9 disp)；

对于 jmp far ptr s，在前面记下的 jmp ... s 指令位置处添上相应的代码。

编译，连接以下程序。

```
assume cs:code
code segment
begin:jmp short s
      jmp s
      jmp near ptr s
      jmp far ptr s
      db 100 dup (0b8h,0,0)
s:    mov ax,2
code ends
end begin
```

在编译中将产生错误，错误是由 jmp short s 引起的，去掉 jmp short s 后再编译就可通过。用 Debug 进行反汇编查看。

附注 4　用栈传递参数

这种技术和高级语言编译器的工作原理密切相关。我们下面结合 C 语言的函数调用，看一下用栈传递参数的思想。

用栈传递参数的原理十分简单，就是由调用者将需要传递给子程序的参数压入栈中，子程序从栈中取得参数。我们看下面的例子。

```
;说明：计算(a-b)^3，a、b 为字型数据
;参数：进入子程序时，栈顶存放 IP，后面依次存放 a、b
;结果：(dx:ax)=(a-b)^3

difcube:push bp
        mov bp,sp
        mov ax,[bp+4]     ;将栈中 a 的值送入 ax 中
        sub ax,[bp+6]     ;减栈中 b 的值
        mov bp,ax
        mul bp
        mul bp
        pop bp
        ret 4
```

指令 ret n 的含义用汇编语法描述为：

```
pop ip
add sp,n
```

因为用栈传递参数，所以调用者在调用程序的时候要向栈中压入参数，子程序在返回的时候可以用 ret n 指令将栈顶指针修改为调用前的值。调用上面的子程序之前，需要压入两个参数，所以用 ret 4 返回。

我们看一下如何调用上面的程序，设 a=3、b=1，下面的程序段计算 (a-b)^3：

```
mov ax,1
push ax
mov ax,3
push ax                ;注意参数压栈的顺序
call difcube
```

程序的执行过程中栈的变化如下。

(1) 假设栈的初始情况如下：

```
1000:0000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                                    ↑
                                                  ss:sp
```

(2) 执行以下指令：

```
mov ax,1
push ax
mov ax,3
push ax
```

栈的情况变为：

```
                                                  a     b
1000:0000   00 00 00 00 00 00 00 00 00 00 00 00 03 00 01 00
                                                  ↑
                                                ss:sp
```

(3) 执行指令 call difcube，栈的情况变为：

```
                                              IP    a     b
1000:0000   00 00 00 00 00 00 00 00 00 00 XX XX 03 00 01 00
                                              ↑
                                            ss:sp
```

(4) 执行指令 push bp ，栈的情况变为：

```
                                          bp   IP    a     b
1000:0000   00 00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                          ↑
                                        ss:sp
```

(5) 执行指令 mov bp,sp ，ss:bp 指向 1000:8

(6) 执行以下指令：

```
mov ax,[bp+4]            ;将栈中 a 的值送入 ax 中
```

```
sub ax,[bp+6]            ;减栈中 b 的值
mov bp,ax
mul bp
mul bp
```

(7) 执行指令 pop bp，栈的情况变为：

```
                                          IP     a      b
1000:0000   00 00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                            ↑
                                         ss:sp
```

(8) 执行指令 ret 4，栈的情况变为：

```
1000:0000   00 00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                                        ↑
                                                     ss:sp
```

下面，我们通过一个 C 语言程序编译后的汇编语言程序，看一下栈在参数传递中的应用。要注意的是，在 C 语言中，局部变量也在栈中存储。

C 程序

```
void add(int,int,int);

main()
{
 int a=1;
 int b=2;
 int c=0;
 add(a,b,c);
 c++;
}

void add(int a,int b,int c)
{
 c=a+b;
}
```

编译后的汇编程序

```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
```

```
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

附注 5　公式证明

问题：计算 X/n (X<65536*65536，n≠0)

在计算过程中要保证不会出现除法溢出。

分析：

(1) 在计算过程中不会出现除法溢出，也就是说，在计算过程中除法运算的商要小于 65536。

设：X/n=(H*65536+L)/n= (H/n)*65536+(L/n)
H=int(X/65536)
L=rem(X/65536)
因为 H<65536 ，L<65536　所以
将计算 X/n 转化为计算：(H/n)*65536+(L/n) 可以消除溢出的可能性。

(2) 将计算 X/n 分解为计算：

(H/n)*65536+(L/n)；H=int(X/65536)；L=rem(X/65536)

DIV 指令只能得出余数和商，而我们只保留商。余数必然小于除数，一次正确的除法运算只能丢掉一个余数。

我们虽然在具体处理时进行了两次除法运算 H/n 和 L/n；但这实质上是一次除法运算 X/n 问题的分解。也就是说，为保证最终结果的正确，两次除法运算只能丢掉一个余数。

在这个问题中，H/n 产生的余数是绝对不能丢的，因为丢掉了它(设为 r)就相当于丢掉了 r*65536(这是一个相当大的误差)。

那么如何处理 H/n 产生的余数呢？

我们知道：H=int(H/n)*n+rem(H/n)

所以有：

(H/n)*65536+(L/n)
=[int(H/n)*n+rem(H/n)]/n*65536+(L/n)
=int(H/n)*65536+rem(H/n)*65536/n+L/n
=int(H/n)*65536+[rem(H/n)*65536+L]/n

现在将计算 X/n 转化为计算：

int(H/n)*65536+[rem(H/n)*65536+L]/n
H=int(X/65536)；L=rem(X/65536)

在这里要进行两次除法运算：

第一次：H/n
第二次：[rem(H/n)*65536+L]/n

我们知道第一次不会产生除法溢出。

现证明第二次：

① L≤65535
② rem(H/n)≤n−1
由②有：
③ rem(H/n)*65536≤(n−1)*65536
由①，③有：
④ rem(H/n)*65536+L≤(n−1)*65536+65535
由④有：
⑤ [rem(H/n)*65536+L]/n≤[(n−1)*65536+65535]/n
由⑤有：
⑥ [rem(H/n)*65536+L]/n≤65536−(1/n)

所以 [rem(H/n)*65536+L]/n 不会产生除法溢出。

则：X/n=int(H/n)*65536+[rem(H/n)*65536+L]/n
H=int(X/65536)；L=rem(X/65536)