第2天　汇编语言学习与Makefile入门

- 介绍文本编辑器
- 继续开发
- 先制作启动区
- Makefile入门

1　介绍文本编辑器

笔者要向大家推荐一个文本编辑器TeraPad，可以从下面这个网站下载，这是一款免费软件（在此感谢寺尾进先生的慷慨奉献！）。

http://www5f.biglobe.ne.jp/~t-susumu/library/tpad.html[1]

[1] 这个编辑器是日文版的，译者推荐一个可编辑中文的文本编辑器Notepad++，可以从这个网站下载：http://notepad-plus-plus.org/。

这也是个免费软件。下载以后解压缩，大家可以在解压后的文件夹里找到“Notepad++”，然后双击鼠标左键就可以安装软件了。

大家下载的时候，可能版本会升级，所以文件名也许会略有不同。它的使用方法与记事本（notepad）基本上是一样的。它有很多选项，大家可以根据自己的喜好进行相应的设置。这里介绍几个非常有用的设置。

设置中文模式方法：

从菜单选择“Encoding”→“Character set”→“Chinese”→“GB2312（Simplified）”

大家可以按照如下步骤设置Tab键所对应的字符数。从菜单选择“Settings”→“Preference”，会弹出一个对话框，选择“Language　　Menu/Tab Settings”，就会显示出语言和TAB键的设置窗口。在TAB键设置的下半部可以看到TAB键的宽度设置，默认值是4。如果要用空格代替TAB，则勾选“Replace by space”前面的选择框就可以了。其他还有显示文章行号，显示换行符、文件结束符等很多设置。笔者没有设置显示这些符号，因为这样画面看起来比较整洁。不过人各有所好，大家可以试一下各种设置，选择一组自己喜欢的。设置完成后，请按“OK”按钮关闭对话框。——译者注

笔者虽然从昨天开始介绍了很多免费软件，但并没有强制大家使用的意思，如果大家已经有了自己喜欢的二进制编辑器或者文本编辑器的话，那就还用它们吧。即便使用不同的软件，开发出来的程序也是一样的，所以笔者没有特意把这些免费软件放在光盘里。大家不用太在意笔者推荐的软件，尽管用自己喜欢的就是了。

2　继续开发

昨天我们还没有详细地讲解helloos.nas中的注释部分，其中要掌握程序核心之前的内容和启动区以外的内容，需要具备软盘方面的一些具体知识，而这在以后我们还会讲到，所以这两部分暂时先保留。

这样一来，尚未讲解清楚的就只有程序核心部分了，那么我们下面就把它改写成更简单易懂的形式吧。先把projects/02_day中的helloos3复制到tolset中，然后打开其中的helloos.nas文件。这个文件太长了，我们节选一部分来讲解。

helloos.nas节选

```
; hello-os
; TAB=4

        ORG     0x7c00          ; 指明程序的装载地址

; 以下的记述用于标准FAT12格式的软盘

        JMP     entry
        DB      0x90

---（中略）---

; 程序核心

entry:
        MOV     AX,0            ; 初始化寄存器
        MOV     SS,AX
        MOV     SP,0x7c00
        MOV     DS,AX
        MOV     ES,AX

        MOV     SI,msg
putloop:
        MOV     AL,[SI]
        ADD     SI,1            ; 给SI加1
        CMP     AL,0

        JE      fin
        MOV     AH,0x0e         ; 显示一个文字
        MOV     BX,15           ; 指定字符颜色
        INT     0x10            ; 调用显卡BIOS
        JMP     putloop
fin:
        HLT                     ; 让CPU停止，等待指令
        JMP     fin             ; 无限循环

msg:
        DB      0x0a, 0x0a      ; 换行2次
        DB      "hello, world"
        DB      0x0a            ; 换行
        DB      0
```

这段程序里有很多新指令，我们从上到下依次来看看。

■■■■■

首先是ORG指令。这个指令会告诉nask，在开始执行的时候，把这些机器语言指令装载到内存中的哪个地址。如果没有它，有几个指令就不能被正确地翻译和执行。另外，有了这条指令的话，美元符（$）的含义也随之变化，它不再是指输出文件

的第几个字节，而是代表将要读入的内存地址。

ORG指令来源于英文“origin”，意思是“源头、起点”。它会告诉nask，程序要从指定的这个地址开始，也就是要把程序装载到内存中的指定地址。这里指定的地址是0x7c00，至于指定它的原因我们会在后文（本节末尾）详述。

下一个是JMP指令，它相当于C语言的goto语句，来源于英文的jump，意思是“跳转”。简单吧！

再下面是“entry:”，这是标签的声明，用于指定JMP指令的跳转目的地等。这与C语言很像。entry这个词是“入口”的意思。

■■■■■

然后我们来看看MOV指令。MOV指令应该是最常用的指令了，即便在这段程序里，MOV指令的使用次数也仅次于DB指令。这个指令的功能非常简单，即赋值。虽然简单，但笔者认为，只要完全掌握了MOV指令，也就理解了汇编语言的一大半。所以，我们在这里详细地讲解一下这个指令。

“MOV　AX,0”，相当于“AX=0;”这样一个赋值语句。同样，“MOV　SS,AX”就相当于“SS=AX;”。或许有人会问：“这个AX和SS是什么东西？”这个问题我们待会儿再回答。

MOV命令源自英文“move”，意思是“移动”。“赋值”与“移动”虽然有些相似，但毕竟还是不同的。一般说来，如果我们把一个东西移走了，它原来所占用的位置就会空出来。但是，在执行了“MOV　SS,AX”语句之后，AX并没有变“空”，还保留着原来的值不变。所以这实际上是“赋值”，而不是“移动”。如果用“COPY”指令来打比方，理解起来就简单多了。至于为什么成了MOV指令，笔者也搞不明白。

■■■■■

现在来说说AX和SS。CPU里有一种名为寄存器的存储电路，在机器语言中就相当于变量的功能。具有代表性的寄存器有以下8个。各个寄存器本来都是有名字的，但现在知道这些名字的机会已经不多了，所以在这里顺便介绍一下。

AX——accumulator，累加寄存器

CX——counter，计数寄存器

DX——data，数据寄存器

BX——base，基址寄存器

SP——stack pointer，栈指针寄存器

BP——base pointer，基址指针寄存器

SI——source index，源变址寄存器

DI——destination index，目的变址寄存器

这些寄存器全都是16位寄存器，因此可以存储16位的二进制数。虽然它们都有上面这种正式名称，但在平常使用的时候，人们往往用简单的英文字母来代替，称它们为“AX寄存器”、“SI寄存器”等。

其实寄存器的全名还是很能说明它本来的意义的。比如在这8个寄存器中，不管使用哪一个，差不多都能进行同样的计算，但如果都用AX来进行各种运算的话，程序就可以写得很简洁。

“ADD CX,0x1234”编译成**81 C1 34 12**，是一个**4**字节的命令。

而 **“ADD AX,0x1234”**编译成**05 34 12**，是一个**3**字节的命令。

从上面例子可以看出，这里所说的“程序可以写得简洁”是指“用机器语言写程序”的情况，从汇编语言的源代码上是看不到这些区别的。如果我们不懂机器语言，就会有很多地方难以理解。

再说说别的寄存器，CX是为方便计数而设计的，BX则适合作为计算内存地址的基点。其他的寄存器也各有优点。

关于AX、CX、 DX、 BX 这几个寄存器名字的由来，虽然我们找不到缩写为X的单词，但这个X表示扩展（extend）的意思。之所以说扩展是因为在这之前CPU的寄存器都是8位的，而现在一下变成了16位，扩展了一倍，所以发明者在原来寄存器的名字后面加了个X，意思是说“扩张了一倍，了不起吧！”。大家可能注意到了这几个寄存器的排列顺序，它并不遵循名称的字母顺序。没错，其实这是按照机器语言中寄存器的编号顺序排列的，可不是笔者随手瞎写的哦。

这8个寄存器全部合起来也才只有16个字节。换句话说，就算我们把这8个寄存器都用上，CPU也只能存储区区16个字节。

另一方面，CPU中还有8个8位寄存器。

AL——累加寄存器低位（accumulator low）

CL——计数寄存器低位（counter low）

DL——数据寄存器低位（data low）

BL——基址寄存器低位（base low）

AH——累加寄存器高位（accumulator high）

CH——计数寄存器高位（counter high）

DH——数据寄存器高位（data high）

BH——基址寄存器高位（base high）

名字看起来有点像，其实这是有原因的：AX寄存器共有16位，其中0位到7位的低8位称为AL，而8位到15位的高8位称为AH。所以，如果以为“再加上这8个8位寄存器，CPU就又可以多保存8个字节了”就大错特错了，CPU还是那个CPU，依然只能存储区区16个字节。CPU的存储能力实在是太有限了。

那BP、SP、SI、DI怎么没分为“L”和“H”呢？能这么想，就说明大家已经做到举一反三了，但可惜的是这几个寄存器不能分为“L”和“H”。如果无论如何都要分别取高位或低位数据的话，就必须先用“MOV，AX，SI”将SI的值赋到AX中去，然后再用AL、AH来取值。这貌似是英特尔（Intel）的设计人员的思维模式。

“喂，我家的电脑是32位的，可不是16位。这样就能以32位为单位来处理数据了吧？那32位的寄存器在哪儿呀？”大家可能会有这样的疑问，下面笔者就来回答这个问题。

EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

这些就是32位寄存器。这次的程序虽然没有用到它们，但如果想用也是完全可以使用的。在16位寄存器的名字前面加上一个E就是32位寄存器的名字了。这个字母E其实还是来源于 “Extend”（扩展）这个词。在当时主流为16位的时代里，能扩展到32位算是个飞跃了。虽说EAX是个32位寄存器，但其实跟前面一样，它有一部分是与AX共用的，32位中的低16位就是AX，而高16位既没有名字，也没有寄存器编号。也就是说，虽然我们可以把EAX作为2个16位寄存器来用，但只有低16位用起来方便；如果我们要用高16位的话，就需要使用移位命令，把高16位移到低16位后才能用。

这么说来，就是32位的CPU也只能存储区区32字节，存储能力还真是小得可怜。

有的读者用的电脑可能是64位的，但我们这次不使用64位模式，所以这里也就不再赘述了。

关于寄存器本来笔者就想介绍到这儿，但是突然想起来，还有一个段寄存器（segment register），所以在这里一并给大家介绍一下吧。这些段寄存器都是16位寄存器。

ES——附加段寄存器（extra segment）

CS——代码段寄存器（code segment）

SS——栈段寄存器（stack segment）

DS——数据段寄存器（data segment）

FS——没有名称（segment part 2）

GS——没有名称（segment part 3）

关于段寄存器的具体内容，我们保留到明天再详细讲解。现在，我们暂时先在这些寄存器里放上0就可以了。

好，到这里寄存器已经讲得差不多了。

■■■■■

那么接下来我们继续看程序，下一个看不懂的语句应该是“MOV　SI,msg”吧。MOV是赋值，意思是SI=msg，而msg是下面将会出现的标号。“把标号赋值给寄存器？这到底是怎么回事？”为了理解这个谜团，我们先回到JMP指令。

前面我们已经看到了“JMP entry”这个指令，其实把它写成“JMP 0x7c50”也完全没有问题。本来JMP指令的基本形式就是跳转到指定的内存地址，因此这个指令就是让CPU去执行内存地址0x7c50的程序。

之所以可以用“JMP entry”来代替“JMP 0x7c50”，是因为entry就是0x7c50。在汇编语言中，所有标号都仅仅是单纯的数字。每个标号对应的数字，是由汇编语言编译器根据ORG指令计算出来的。编译器计算出的“标号的地方对应的内存地址”就是那个标号的值。

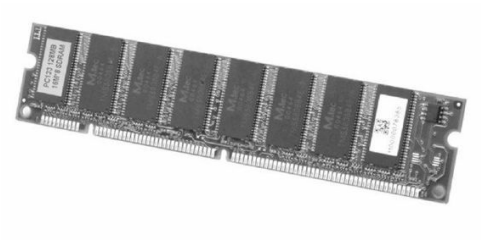
所以，如果我们在这个程序中写了“MOV　AX,entry”，那它就会把0x7c50代入到AX寄存器里，我们代入到AX寄存器中的就是这个简单的数字。大家可不要以为写在“entry”下面的程序也都被储存了，这是不可能的。

那么“MOV　SI,msg”会怎么样呢？由于在这里msg的地址是0x7c74，所以这个指令就是把0x7c74代入到SI寄存器中去。

■■■■■

下面我们来看“MOV　AL,[SI]”。如果这个命令是“MOV　AL,SI”的话，不用多说大家也都能明白它的意思，可这里用方括号把SI括了起来。如果在汇编语言中出现这个方括号，寄存器所代表的意思就完全不一样了。

这个记号代表“内存”。如果大家自己组装过电脑，就知道所谓“内存”，指的是256MB或512MB的那个零件。



内存

到现在为止，内存这个词我们已经使用了很多次了，可一直都还没有正式讲解过，那内存到底是什么呢？简单地用一句话来概括，它就是一个超大规模的存储单元“住宅区”。用“住宅区”来比喻内存再合适不过了，它能充分体现出存储单元紧密、整齐地排列在一起的样子。英语中memory是“记忆”的意思，这里我们把它译成“内存”。

通过对寄存器的讲解，现在大家都知道了CPU的存储能力很差，如果我们想让CPU处理大量信息，就必须给它另外准备一套用于存储的电路。因为即便是32位的CPU，把所有普通的寄存器都加在一起，最多也只能存储32个字节的数据。就算把段寄存器也全部用上，也才只有44字节。这么小的存储空间，就连启动电脑所必需的启动区数据都放不下。

现在大家已经知道了存储单元的必要性，那么我们下面就讲内存。内存并不在CPU的内部，而是在CPU的外面。所以对于CPU来说，内存实际上是外部存储器。这点很重要，就是说CPU要通过自己的一部分管脚（引线）向内存发送电信号，告诉内存说：“喂，把5678号地址的数据通过我的管脚传过来（严格说来，CPU和内存之间还有称为芯片（chipset）的控制单元）！”CPU向内存读写数据时，就是这样进行信息交换的。

CPU与内存之间的电信号交换，并不仅仅是为了存取数据。因为从根本上讲，程序本身也是保存在内存里的。程序一般都大于44字节，不可能保存在寄存器中，所以规定程序必须放在内存里。CPU在执行机器语言时，必须从内存一个命令一个命令地读取程序，顺序执行。

内存虽然如此重要，但它的位置却离CPU相当远。就算是只有10厘米左右的距离吧，可这与CPU中的半导体相比已经非常遥远了。所以，当CPU向内存请求数据或者输出数据的时候，内存需要花很长时间才能够完整无误地实现CPU的要求（CPU运行速度极快，所以即使在10厘米这么短的距离内传送电信号，所花的时间都不容忽视）。所以，虽然内存比寄存器的存储能力大很多个数量级，但使用内存时速度很慢。CPU访问内存的速度比访问寄存器慢很多倍，记住这一点，我们才能开发出执行速度快的程序来。

■■■■■

基础知识我们讲完了，下面再回到汇编语言。MOV指令的数据传送源和传送目的地不仅可以是寄存器或常数，也可以是内存地址。这个时候，我们就使用方括号（[　]）来表示内存地址。另外，BYTE、WORD、DWORD等英文词也都是汇编语言的保留字，下面举个例子吧。

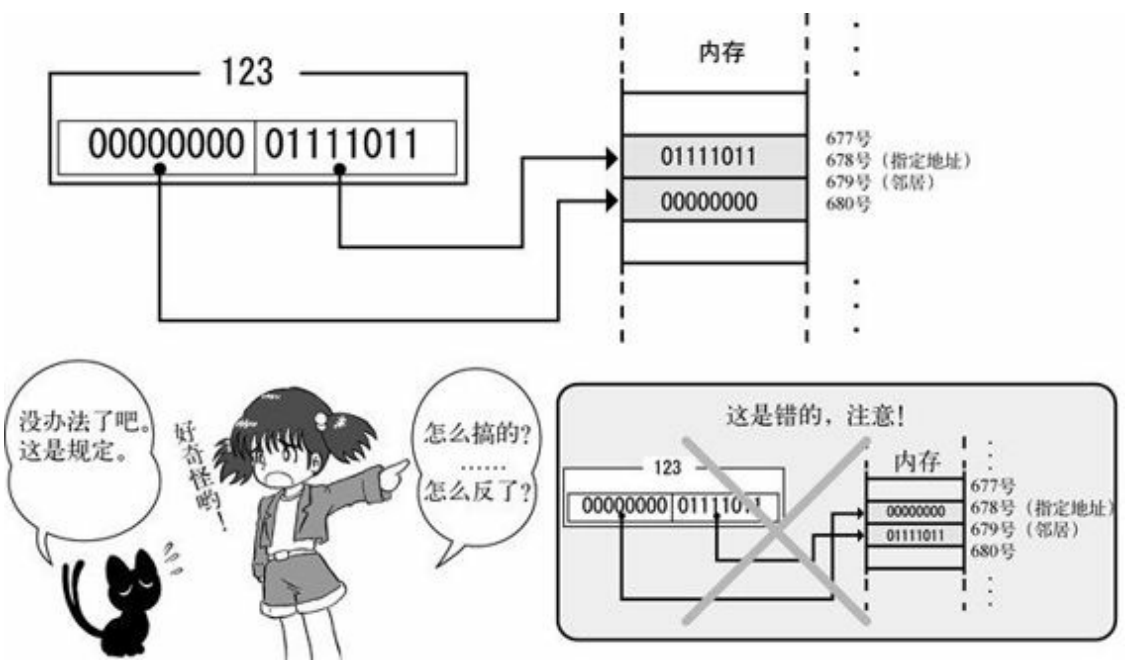
```
MOV BYTE [678],123
```

这个指令是要用内存的“678”号地址来保存“123”这个数值。虽然指令里有数字，看起来像那么回事，但实际上内存和CPU一样，根本就没有什么数值的概念。所谓的“678”，不过就是一大串开（ON）或者关（OFF）的电信号而已。当内存收到这一串信号时，电路中的某8个存储单元就会响应，这8个存储单元会记住代表“123”的开（ON）或关（OFF）的电信号。为什么是8位呢？这是因为指令里指定了“BYTE”。同样，我们还可以写成：

```
MOV WORD [678],123
```

在这种情况下，内存地址中的678号和旁边的679号都会做出反应，一共是16位。这时，123被解释成一个16位的数值，也就是0000000001111011，低位的01111011保存在678号，高位的00000000保存在旁边的679号。

像这样在汇编语言里指定内存地址时，要用下面这种方式来写：



数据大小 **[**地址**]**

这是一个固定的组合。如果我们指定“数据大小”为BYTE，那么使用的存储单元就只是地址所指定的字节。如果我们指定“数据大小”为WORD，则相邻的一个字节也会成为这个指令的操作对象。如果指定为DWORD，则与WORD相邻的两个字节，也都成为这个指令的操作对象（共4个字节）。这里所说的相邻，指的是地址增加方向的相邻。

至于内存地址的指定方法，我们不仅可以使用常数，还可以用寄存器。比如“BYTE [SI]”、“WORD　[BX]”等等。如果SI中保存的是987的话，“BYTE　[SI]”就会被解释成“BYTE [987]”，即指定地址为987的内存。

虽然我们可以用寄存器来指定内存地址，但可作此用途的寄存器非常有限，只有BX、BP、SI、DI这几个。剩下的AX、CX、DX、SP不能用来指定内存地址，这是因为CPU没有处理这种指令的电路，或者说没有表示这种处理的机器语言。没有对应的机器语言当然也就不能进行这样的处理了，如果有意见的话，就写邮件找英特尔的大叔们吧（笑）。笔者没有勇气找英特尔的大叔们抱怨，所以想把DX内存里的内容赋值给AL的时候，就会这样写：

```
MOV BX, DX
MOV AL, BYTE [BX]
```

■■■■■

根据以上说明我们知道可以用下面这个指令将SI地址的1字节内容读入到AL。

```
MOV AL, BYTE [SI]
```

可是MOV指令有一个规则[1]，那就是源数据和目的数据必须位数相同。也就是说，能向AL里代入的就只有BYTE，这样一来就可以省略BYTE，即可以写成：

```
MOV AL, [SI]
```

[1] 如果违反这一规则，比如写“MOV AX,CL”的话，汇编语言就找不到相对应的机器语言，编译时会出错。

哦，这样就与程序中的写法一样了。现在总算把这个指令解释清楚了，所以这个指令的意思就是“把SI地址的1个字节的内容读入AL中”。

■■■■■

ADD是加法指令。若以C语言的形式改写“ADD　SI,1”的话，就是SI=SI+1。“add”的英文原语意为“加”。

CMP是比较指令。或许有人想，比较指令是干什么的呢？简单说来，它是if语句的一部分。譬如 C 语言会有这种语句：

```
if(a==3){ 处理; }
```

即对a和3进行比较，将其翻译成机器语言时，必须先写“CMP　a,3”，告诉CPU比较的对象，然后下一步再写“如果二者相等，需要做什么”。

这里是“CMP　AL,0”，意思就是将AL中的值与0进行比较。这个指令源自英文中的compare，意为“比较”。

JE是条件跳转指令中之一。所谓条件跳转指令，就是根据比较的结果决定跳转或不跳转。就JE指令而言，如果比较结果相等，则跳转到指定的地址；而如果比较结果不等，则不跳转，继续执行下一条指令。因此，

```
CMP AL, 0
JE fin
```

这两条指令，就相当于：

```
if (AL == 0) { goto fin; }
```

这条指令源自于英文“jump　if　equal”，意思是如果相等就跳转。顺便说一句，fin是个标号，它表示“结束”（finish）的意思，笔者经常使用。

■■■■■

INT是软件中断指令。如果现在就讲中断机制的话，肯定会让人头昏脑胀的，所以我们暂时先把它看作一个函数调用吧。这个指令源自英文“interrupt”，是“中途打断”的意思。

电脑里有个名为BIOS的程序，出厂时就组装在电脑主板上的ROM[2]单元里。电脑厂家在BIOS中预先写入了操作系统开发人员经常会用到的一些程序，非常方便。BIOS是英文“basic　input　output　system”的缩写，直译过来就是“基本输入输出系统（程序）”。

[2]　只读存储器，不能写入，切断电源以后内容不会消失。ROM是“read　only memory”的缩写。

最近的BIOS功能非常多，甚至包括了电脑的设定画面，不过它的本质正如其名，就是为操作系统开发人员准备的各种函数的集合。而INT就是用来调用这些函数的指令。INT的后面是个数字，使用不同的数字可以调用不同的函数。这次我们调用的是0x10（即16）号函数，它的功能是控制显卡。

虽然制造厂家给我们准备好了BIOS，但其用法鲜为人知。不过这些很容易查到，笔者就做了一个关于BIOS的网页，下面给大家介绍一下。

http://community.osdev.info/?(AT)BIOS

比如我们现在想要显示文字，先假设一次只显示一个字，那么具体怎么做才能知道这个功能的使用方法呢？

首先，既然是要显示文字，就应该看跟显卡有关的函数。这么看来，INT　0x10好像

有点关系，于是在上面网页上搜索，然后就能找到以下内容（网页的原文为日语）。

显示一个字符

- AH=0x0e;
- AL=character code;
- BH=0;
- BL=color code;
- 返回值：无
- 注：beep、退格（back space）、CR、LF都会被当做控制字符处理

所以，如果大家按照这里所写的步骤，往寄存器里代入各种值，再调用INT 0x10，就能顺利地在屏幕上显示一个字符出来[3]。

[3] 因为这里的BL中放入了彩色字符码，所以一旦这里变更，显示的字符的颜色也应该变化。但笔者试了试，颜色并没有变。尚不清楚为什么只能显示白色，只能推测现在这个画面模式下，不能简单地指定字符颜色。

■■■■■

最后一个新出现的指令是HLT。这个指令很少用，会让它在第2天的内容里就登台亮相的，估计全世界就只有笔者了。不过由于笔者对它的偏好，就让笔者在这里多说两句吧（笑）。

HLT是让CPU停止动作的指令，不过并不是彻底地停止（如果要彻底停止CPU的动作，只能切断电源），而是让CPU进入待机状态。只要外部发生变化，比如按下键盘，或是移动鼠标，CPU就会醒过来，继续执行程序。说到这，请大家再仔细看看这个程序，我们会发现其实不管有没有HLT指令，JMP fin都是无限循环，不写HLT指令也可以。所以很少有人一开始就向初学者介绍HLT指令，因为这样只会让话变得很长。

然而笔者讨厌让CPU毫无意义地空转。如果没有HLT指令， CPU就会不停地全力去执行JMP指令，这会使CPU的负荷达到100%，非常费电。这多浪费呀。我们仅仅加上一个HLT指令，就能让CPU基本处于睡眠状态，可以省很多电。什么都不干，还要耗费那么多电，这就是浪费。即便是初学者，最好也要一开始就养成待机时使用HLT指令的习惯。或者说，恰恰应该在初学阶段，就养成这样的好习惯。这样既节能环保，又节约电费，或许还能延长电脑的使用寿命呢。

对了，HLT指令源自英文“halt”，意思是“停止”。

■■■■■

说了这么多，终于把这个程序从头到尾都讲完了。总结一下就是这样的：

用**C**语言改写后的**helloos.nas**程序节选

```
entry:
    AX = 0;
    SS = AX;
    SP = 0x7c00;
    DS = AX;
    ES = AX;
    SI = msg;
putloop:
    AL = BYTE [SI];
    SI = SI + 1;
    if (AL == 0) { goto fin; }
    AH = 0x0e;
    BX = 15;
    INT 0x10;
    goto putloop;
fin:
    HLT;
    goto fin;
```

就是有了这个程序，我们才能够把msg里写的数据，一个字符一个字符地显示出来，并且数据变成0以后，HLT指令就会让程序进入无限循环。“hello, world”就是这样显示出来的。

■■■■■

对了，我们还没有说ORG的0x7c00是怎么回事呢。ORG指令本身刚才已经讲过，就不再重复了，但这个0x7c00又是从哪儿冒出来的呢？换成1234是不是就不行啊？嗯，还真是不行，我们要是把它换成1234的话，程序马上就不动了。

大家所用的电脑里配置的，大概都是64MB，甚至512MB这样非常大的内存。那是不是这些内存我们想怎么用就能怎么用呢？也不是这样的。比如说，内存的0号地址，也就是最开始的部分，是BIOS程序用来实现各种不同功能的地方，如果我们随便使用的话，就会与BIOS发生冲突，结果不只是BIOS会出错，而且我们的程序也肯定会问题百出。另外，在内存的0xf0000号地址附近，还存放着BIOS程序本身，那里我们也不能使用。

内存里还有其他不少地方也是不能用的，所以我们作为操作系统开发者，不得不注意这一点。在我们作为一般用户使用Windows或Linux时，不用想这些麻烦事，因为操作系统已经都处理好了，而现在，我们成了操作系统开发者，就需要为用户来考虑这些问题了。只用语言文字来讲解内存哪个部分不能用的话，不够清楚直观，所以还是要画张地图。正好这里就有一张内存分布图，让我们一起来看看。

http://community.osdev.info/?(AT)memorymap

虽然称之为地图，可实际上根本就不像地图，网页的作者也太会偷工减料了吧。话说这个网页的作者，其实就是笔者本人，不好意思啦。大家要是仔细看的话，会发现其中很多东西都是不知所云（都是笔者不好，真是对不起），不过在“软件用途分类”这里，有一句话可是非常重要的，一定不能漏掉。

0x00007c00-0x00007dff ： 启动区内容的装载地址

程序中ORG指令的值就是这个数字。而且正是因为我们使用的是这个同样的数字，所以程序才能正常运行。

看到这，大家可能会问：“为什么是0x7c00呢？ 0x7000不是更简单、好记吗？”其实笔者也是这么想的，不过没办法，当初规定的就是0x7c00。做出这个规定的应该是IBM的大叔们，不过估计他们现在都成爷爷了。

一旦有了规定，人们就会以此为前提开发各种操作系统，因此以后就算有人说“现在地址变成0x7000-0x71ff了，请大家跟着改一下”，也只是空口号，不可能实现。因为硬要这么做的话，那现有的操作系统就必须全部加以改造才能在这台新电脑上运行，这样的电脑兼容性不好，根本就卖不出去。

今后也许大家还会提出很多疑问：“为什么是这样呢？”这些都是当年IBM和英特尔的大叔们规定的。如果非要深究的话，我们倒是也能找到一些当时时代背景下的原因，不过要把这些都说清楚的话，这本书恐怕还要再加厚一倍，所以关于这些问题我们就不过多解释了。

3　先制作启动区

考虑到以后的开发，我们不要一下子就用nask来做整个磁盘映像，而是先只用它来制作512字节的启动区，剩下的部分我们用磁盘映像管理工具来做，这样以后用起来就方便了。

如此一来，我们就有了projects/02_day的helloos4这个文件夹。

首先我们把heloos.nas的后半部分截掉了，这是因为启动区只需要最初的512字节。现在这个程序就仅仅是用来制作启动区的，所以我们把文件名也改为ipl.nas。

然后我们来改造asm.bat，将输出的文件名改成ipl.bin。另外，也顺便输出列表文件ipl.lst。这是一个文本文件，可以用来简单地确认每个指令是怎样翻译成机器语言的。到目前为止我们都没有输出过这个文件，那是因为1440KB的列表文件实在太大了，而这次只需要输出512字节，所以没什么问题。

另外我们还增加了一个makeimg.bat。它是以ipl.bin为基础，制作磁盘映像文件helloos.img的批处理文件。它利用笔者自己开发的磁盘映像管理工具edimg.exe，先读入一个空白的磁盘映像文件，然后在开头写入ipl.bin的内容，最后将结果输出为名为helloos.img的磁盘映像文件。详情请参考makeimg.bat的内容。

这样，从编译到测试的步骤就变得非常简单了，我们只要双击!cons，然后在命令行窗口中按顺序输入asm→makeimg→run这3个命令就完成了。

4　Makefile入门

到helloos4为止，做出来的程序与笔者最初开发时所写的源程序是完全一样的。在开发的过程中，笔者使用了一个名为Makefile的东西，在这里给大家介绍一下。

Makefile就像是一个非常聪明的批处理文件。

■■■■■

Makefile的写法相当简单。首先生成一个不带扩展名的文件Makefile，然后再用文本编辑器写入以下内容。

```
#文件生成规则

ipl.bin : ipl.nas Makefile
    ../z_tools/nask.exe ipl.nas ipl.bin ipl.lst

helloos.img : ipl.bin Makefile
    ../z_tools/edimg.exe   imgin:../z_tools/fdimg0at.tek \
        wbinimg src:ipl.bin len:512 from:0 to:0   imgout:helloos.img
```

#号表示注释。下一行“ipl.bin　:　ipl.nas　Makefile”的意思是，如果想要制作文件ipl.bin，就先检查一下ipl.nas和Makefile这两个文件是否都准备好了。如果这两个文件都有了，Make工具就会自动执行Makefile的下一行。

至于helloos.img，Makefile的写法也是完全一样的。其中的“\”是续行符号，表示这一行太长写不下，跳转到下一行继续写。

我们需要调用make.exe来让这个Makefile发挥作用。为了能更方便地从命令行窗口运行这个工具，我们来做个make.bat。make.bat就放在tolset的z_new_w文件夹中，可以直接把它复制过来用。

■■■■■

做好以上这些准备后，用!cons打开一个命令行窗口（console），然后输入“make　-r ipl.bin”。这样make.exe就会启动了，它首先读取Makefile文件，寻找制作ipl.bin的方法。因为ipl.bin的做法就写在Makefile里，make.exe找到了这一行就去执行其中的命令，顺利生成ipl.bin。然后我们再输入“make　-r　helloos.img”看看，果然它还是会启动make.exe，并按照Makefile指定的方法来执行。

到此为止好像也没什么特别的，我们再尝试一下把helloos.img和ipl.bin都删除后，再输入“make -r helloos.img”命令。make 首先很听话地试图生成helloos.img，但它会发现所需要的ipl.bin还不存在。于是就去Makefile里寻找ipl.bin的生成方法，找到后先生成ipl.bin，在确认ipl.bin顺利生成以后，就回来继续生成helloos.img。它很聪明吧。

下面，我们不删除文件，再输入命令“make　-r　helloos.img”执行一次的话，就会发现，仅仅输出一行“‘helloos.img’已是最新版本（‘helloos.img’is　up　to　date）”的信息，什么命令都不执行。也就是说，make知道helloos.img已经存在，没必要特意重

新再做一次了。它越来越聪明了吧。

让我们再考验考验make.exe。我们来编辑ipl.nas中的输出信息，把它改成“How are you?”并保存起来。而ipl.bin 和helloos.img保持刚才的样子不删除，在这种情况下我们再来执行一次“make- r helloos.img”。本以为这次它还会说没必要再生成一次呢，结果我们发现，make.exe又从ipl.bin开始重新生成输出文件。这也就是说，make.exe不仅仅判断输入文件是否存在，还会判断文件的更新日期，并据此来决定是否需要重新生成输出文件，真是太厉害了。

■■■■■

现在大家知道了Makefile比批处理文件高明，但每次都输入“make -r helloos.img”的话也很麻烦，其实有个可以省事的窍门。当然，可以将“make -r helloos.img”这个命令写成makeimg.bat，但这么做还是离不开批处理文件，所以我们换个别的方法，在Makefile里增加如下内容。

```
#命令
img :
    ../z_tools/make.exe -r helloos.img
```

修改之后，我们只要输入“make img”，就能达到与“make -r helloos.img”一样的效果。这样就省事多了。makeimg.bat已经没用了，把它删掉。另外顺便把下面内容也一并加进去吧。

```
asm :
    ../z_tools/make.exe -r ipl.bin

run :
    ../z_tools/make.exe img
    copy helloos.img ..\z_tools\qemu\fdimage0.bin
    ../z_tools/make.exe -C ../z_tools/qemu

install :
    ../z_tools/make.exe img
    ../z_tools/imgtol.com w a: helloos.img
```

这样一来，“run.bat”、“install.bat”也都用不着了。不但用不着，现在还更方便了呢。比如只要输入“make run”，它会首先执行“make img”，然后再启动模拟器。

到目前为止，我们为了节约时间，避免每次都从汇编语言的编译开始重新生成已有的输出文件，特意把批处理文件分成了几个小块。而现在有了Makefile，它会自动跳过没有必要的命令，这样不管任何时候，我们都可以放心地去执行“make img”了。而且就算直接“make run”也可以顺利运行。“make install”也是一样，只要把磁盘装到驱动器里，这个命令就会自动作出判断，如果已经有了最新的helloos.img就直接安装，没有的话就先自动生成新的helloos.img，然后安装。

■■■■■

笔者把以上这些都总结在projects/02_day下的helloos5文件夹里了，顺便又另外添加了几个命令。一个命令是“make clean”，它可以删除掉最终成果 （这里是helloos.img）以外的所有中间生成文件，把硬盘整理干净；还有一个命令是“make

src_only”，它可以把源程序以外的文件全都删除干净。另外，笔者还增加了make命令的默认动作，当执行不带参数的make时，就相当于执行“make img”命令（默认动作写在Makefile的最前头）。

功能增加了这么多，而文件数量却减少到5个，看上去清爽多了吧。像源文件这种真的必不可少的文件，多几个倒也没什么不好，但像批处理文件这种可有可无的东西太多，堆在那里乱糟糟的就会让人很不舒服。

这样整理一下，我们以后的开发工作就会更加轻松愉快了。

■■■■■

啊，有一点忘了告诉大家，这个make.exe是GNU项目组的人开发的，公开供大家免费使用的一款软件。gcc的作者也是这个GNU项目组。真是太感谢了！

按照现在的速度真的能在一个月后开发出一个操作系统吗？笔者也有点担心。不过应该没问题，虽说现在的进展比当初的计划稍慢一些，不过刚开始的时候说明肯定会多一些，等到后面用C语言来开发的时候，速度就能上来了。嗯，就是这样……笔者满怀希望地自言自语中（苦笑）。那么我们明天见！

COLUMN-1　数据也能“执行”吗？机器语言也能“显示”吗？

在helloos5中，如果我们把最开始的JMP entry 写成JMP msg，到底会怎样呢？……

首先，可不可以这么写呢？完全可以！nask不会报错，别的汇编语言也不会报错。在汇编语言里，标号归根到底不过就是一个表示内存地址的数字而已，至于JMP跳转的地方是机器语言还是字符编码，汇编语言中不考虑这些问题。

那么如果执行这个程序，CPU会怎么样呢？首先最初的命令是0A　0A，意思是“OR　CL,[BP+SI]”，也就是把CL寄存器的内容和BP+SI内存地址的内容做逻辑或（OR）运算（过几天会出现这个命令），结果放入CL寄存器。接着的命令是68 65 6C，也就是PUSH 0x6c65的意思（过几天这个命令也会出现），它将0x6c65储存进栈。……就这样， CPU执行的命令很混乱，但CPU只能按照电信号的指令来进行处理，所以即使不明其意，也会一板一眼地照单执行。

结果，要么画面上出现怪异的字符，要么软盘或硬盘上的数据突然被覆盖。虽然电脑并没有坏掉（因为CPU还在全速执行指令），但看上去却像坏了一样。所以大家一定不要尝试这样做。

不过人无完人，搞不好通宵写了一夜程序，稀里糊涂之下就将本应写entry的地方，错写成了msg，这也不是不可能发生。要是因为这而丢失了重要文件，可就损失惨重了，所以CPU具有预防这种事故的功能。但是这种功能只有在操作系统做了各种相应设置后才会起作用（几天后也会讲到）。所以，在开发操作系统的阶段，我们还不能指望这种保护功能。从某种程度上来说，我们操作系统开发者一直都是在提心吊胆地做开发。

那么反过来会怎样呢？也就是假设要把机器语言当作文字来显示，会出现什么结果呢？程序里有一句是“MOV SI,msg”，我们把它写成“MOV SI,entry”看看。首先画面上会显示一个编码是B8的字符（估计是个表情符号或者别的什么符号），下一个字符碰巧是00，所以显示就到此结束了。这种情况不会出现恶劣的后果，大家试一试也无妨。

通过以上的尝试，最终证明，不管是CPU还是内存，它们根本就不关心所处理的电信号到底代表什么意思。这么一来，说不定我们拿数码相机拍一幅风景照，把它作为磁盘映像文件保存到磁盘里，就能成为世界上最优秀的操作系统！这看似荒谬的情况也是有可能发生的。但从常识来看，这样做成的东西肯定会故障百出。反之，我们把做出的可执行文件作为一幅画来看，也没准能成为世界上最高水准的艺术品。不过可以想象的是，要么文件格式有错，要么显示出来的图是乱七八糟的。