第三章

先说说系统是怎么使用硬件的，有这四种方式：

(1)程序I/O方式（Programmed I/O方式）

CPU需要时刻对外设设备状态进行循环检查，直到确定该字已经在I/O控制器的数据寄存器中。CPU和设备只能串行工作，CPU效率相当低。

(2)中断驱动方式

允许I/O设备主动打断CPU的运行并且请求服务，使得其向I/O控制器发送读命令。具体实现是这样的：

对于I/O控制器来说，从CPU接收一个读命令，然后从外围设备读取数据，一旦数据读入到I/O控制器的数据寄存器，便通过控制线给CPU一个中断信号，表示数据准备好了，等待CPU请求该数据，I/O控制器收到CPU的取数据请求后，将数据放到数据总线上，传到CPU的寄存器中。

CPU：CPU发出读命令，然后保存当前运行程序的上下文，去执行其他程序，在每个指令周期的末尾，CPU检查中断，有来自I/O控制器的中断的时候，CPU保存当前正在运行程序的上下文，去运行中断处理程序处理该中断。这时，CPU从I/O控制器读一个字的数据传到寄存器，并且存入内存，CPU恢复发出I/O命令的程序的上下文，继续运行。  
但是数据的每个字在存储器和I/O控制器间的传输需要经过CPU，导致中断驱动方式还是会消耗较多的CPU时间。

(3)DMA方式

在I/O设备和内存之间加上DMA控制器使得数据进行直接传输而不经过CPU。

DMA方式特点：基本单位是数据块，传送的数据是从设备直接送入内存或者相反，只有在一个或者多个数据块的开始结束时，CPU干预。整块数据的传送是在DMA控制器控制下完成。

工作方式：CPU接收到I/O设备的DMA请求时候，给I/O控制器发送一条命令，启动DMA控制器，然后DMA控制器和存储器交互，传送整个数据块而不需要CPU的参与。传送完成后，DMA控制器发送中断信号给处理器。只有传送开始和结束的时候才需要CPU的参与。

(4)通道控制方式

I/O通道是指专门负责输入/输出的处理机，因此属于硬件技术。  
CPU要完成一组相关的读写操作以及有关控制时候，向I/O通道发送一条I/O指令，给出所要执行的通道程序的首地址和要访问的I/O设备，通道接受指令后，执行通道程序完成CPU指定的I/O任务，数据传送结束时向CPU发送中断请求。

通道方式由通道控制传输的数据块大小以及传输的内存位置，一个通道可以控制多台设备与内存的数据交换。

这些都是硬件的一些东西，那么我们的操作系统到底是如何做操作的呢？这里就需要明白一些I/O的内容了。这里请参考<X86架构的部分内容>

此时，我们希望能做做一些输出了。这里就从I/O系统转移到了具体的显卡部分了。做图形或者文字的输出，需要设置我们的视频内存(video memory)，有如下几个模式：

|  |  |
| --- | --- |
| 00 | text 40\*25 16 color (mono) |
| 01 | text 40\*25 16 color |
| 02 | text 80\*25 16 color (mono) |
| 03 | text 80\*25 16 color |
| 04 | CGA 320\*200 4 color |
| 05 | CGA 320\*200 4 color (m) |
| 06 | CGA 640\*200 2 color |
| 07 | MDA monochrome text 80\*25 |
| 08 | PCjr |
| 09 | PCjr |
| 0A | PCjr |
| 0B | reserved |
| 0C | reserved |
| 0D | EGA 320\*200 16 color |
| 0E | EGA 640\*200 16 color |
| 0F | EGA 640\*350 mono |
| 10 | EGA 640\*350 16 color |
| 11 | VGA 640\*480 mono |
| 12 | VGA 640\*480 16 color |
| 13 | VGA 320\*200 256 color |

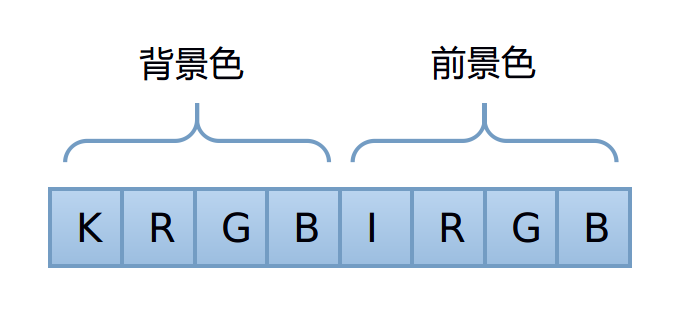
这一内容的实现，可以通过手动设置VGA的寄存器，或者在实模式中通过BIOS的 int 10h调用set video Mode服务实现，但所幸，GRUB已经为我们做好了这项设置了，设置为了80x25的视频模式， 也就是mode 03，所以我们就可以坐享其成啦。

实际上，所有在PC上工作的显卡，在加电初始化之后都会自动初始化到80\*25的文本模式。在这个模式下，屏幕被划分为25行，每行可以显示80个字符，所以一屏可以显示2000个字符。

先说说字符是怎么显示的：

这里的显示，用到的是内码，内码定义了字符在内存中存储的形式，而英文编码就是大家所熟知的ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）码了。对应的关系很简单，从0xB8000这个地址开始，每2个字节表示屏幕上显示的一个字符。一个字节是ASCII码，一个字节是附加码，表示颜色等信息。

颜色含义如下图：



对应颜色表：



所以，例如来讲使用0x00作为属性字节就是黑色黑色（你将看不到任何东西）。 0x07是黑底浅灰DOS默认），0x1F是蓝白色（Win9x的蓝屏死机），0x2a是绿色 - 单色怀旧风格。

具体实现是这样的：我们把我们的要显示的内容，根据设定好的模式，设置好我们文字的格式，并把它们放到内存中合适的位置，显卡会周期性的读取这里的数据，并且把它们按顺序显示在屏幕上。

那么具体来讲，CPU是如何控制我们的显卡呢，或者说，我们把我们要显示的内容放到内存中后，它们是如何显示在屏幕上呢？而我们要显示的内容，又应该放在哪里呢？

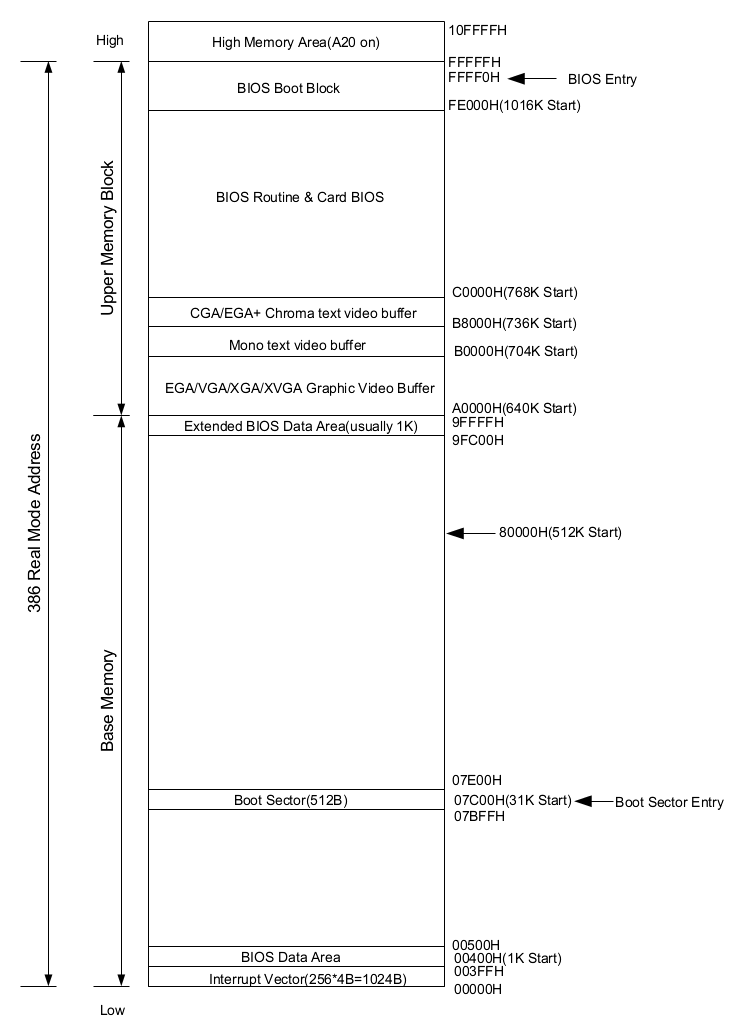
先来说一个概念，MMIO，指的是memory mapped I/O，此外还有PMIO(端口映射I/O)，这两种方法是在CPU和外围设备之间执行输入/输出（I/O）的基本方法。

MMIO，就是说，把I/O设备的内存和寄存器映射到我们的主存中，这样的话，访问对于的地址，就相当于访问对于的设备了。I/O设备会监视CPU的地址总线，然后响应CPU对于相关的设备的读取，把数据总线连到设备的寄存器上。这一过程的往往是通过特定的CPU的指令来实现的，比如in和out这种基于x86和x86-64架构在微处理芯片上设置的指令，它可以在EAX(或者AX，AL)和I/O设备特定的端口通信。I/O设备具有与通用存储器分离的地址空间，通过CPU物理接口上的额外“I/O”引脚或专用于I/O的整条总线来实现。

系统的硬件通过特定的设计，可以使得连接在地址总线上的设备只响应特定的地址访问， 而其它地址访问不会触发它们。这一部分由地址解码电路实现，它可以建立系统的地址映射。比如一种常见的映射方式：

|  |  |  |
| --- | --- | --- |
| Address range ([hexadecimal](https://en.wikipedia.org/wiki/Hexadecimal)) | Size | Device |
| 0000–7FFF | 32 KiB | RAM |
| 8000–80FF | 256 bytes | General-purpose I/O |
| 9000–90FF | 256 bytes | Sound controller |
| A000–A7FF | 2 KiB | Video controller/text-mapped display RAM |
| C000–FFFF | 16 KiB | ROM |

还记得在我们的linker.ld里面，我们开始的地址吗？没错，就是从1MB开始的，那么在1MB之下，是什么呢？在这之下，是一些属于外部设备的内存，这就是上面说的MMIO了，如图所示：



可以看到， 如果是VGA模式，我们就从0xA0000开始，是我们显示的内容，而0xB0000开始，则是我们的黑白显示。在0xB8000开始，是彩色文字显示。

这里我们不做复杂的显示，仅使用显卡的文本模式。上图中的0xB8000～0xBFFFF这个地址段便是映射到文本模式的显存的。当访问这些地址的时候，实际上读写的就是显存区域了。

再进一步的，我们的显卡，到底是如何访问内存的呢？

有三种方式。第一种，就是framebuffer。CPU搞一块内存名叫Framebuffer，里面放上要显示的东西，显卡有个部件叫DIsplay Controller会扫描那块内存，然后把内容显示到屏幕上。第二种，DMA。就是硬件设备直接从内存取数据，当然需要软件先配置，这就是graphics driver的活儿。在显卡驱动里，DMA还有个专用的名字叫Blit。第三种，内存共享。Intel的平台，显存和内存本质都是主存。区别是CPU用的需要MMU映射，GPU用的需要GPU的MMU叫做GTT映射。所以共享内存的方法很简单，把同一个物理页既填到MMU页表里，也填到GTT页表里。

啰嗦几句，说说驱动，驱动是我们听到很多的东西了，那么驱动到底干了什么，又是怎么干的呢？以显卡驱动为例，目前的显卡驱动，不是单纯的一个独立的驱动模块，而是几个驱动模块的集合。用户态和内核态驱动都有。以Linux桌面系统为例，按照模块划分，内核驱动有drm/i915模块， 用户驱动包括libdrm, Xorg的DDX和DIX，3D的LibGL, Video的Libva等等，各个用户态驱动可能相互依赖，相互协作，如果按照功能划分的话，大概分成5大类，display, 2D, 3D, video, 以及General Purpose Computing 通用计算。Display是关于如何显示内容，比如分辨率啊，刷新率啊，多屏显示啊。2D现在用的很少了，基本就是画点画线加速，快速内存拷贝（也就是一种DMA）。3D就复杂了，基本现在2D的事儿也用3D干。3D涉及很多计算机图形学的知识，我的短板，我就不多说了。Video是指硬件加速的视频编解码。通用计算就是对于OpenCL，OpenCV，CUDA这些框架的支持。

理解驱动程序最重要的一句话是，寄存器是软件控制硬件的唯一途径。所以如何控制显卡，答案就是靠读写显卡提供的寄存器。

所以总结下来，我们**要编写一个简单的驱动**，来管理我们的硬件。这个驱动仅仅设计端口的读写操作，而设计的核心在于有了MMIO，我们的硬件已经在内存的底层映射了可以操作的端口，显示的话，我们需要在0xB8000～0xBFFFF输出显示内容。操作端口，实现对硬件的控制。于是，实现屏幕的输出。

对于显示控制单元来说，它们被编制在独立的I/O空间里，需要用特殊的in/out指令去读写。这里相关的控制寄存器多达300多个，显然无法一一映射到I/O端口的地址空间。对此工程师们解决方案是，将一个端口作为内部寄存器的索引：0x3D4，再通过0x3D5端口来设置相应寄存器的值。

在具体的设置之前，我们首先需要几个端口读写函数的实现。因为C语言并没有直接操作端口的方法，而且频繁的内联汇编麻烦又容易出错。所以好的做法就是定义几个端口读写函数。代码如下：

/libs/common.c

1. #include "common.h"
2. // 端口写一个字节
3. inline void outb(uint16\_t port, uint8\_t value)
4. {
5. asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
6. }
7. // 端口读一个字节
8. inline uint8\_t inb(uint16\_t port)
9. {
10. uint8\_t ret;
11. asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));
12. return ret;
13. }
14. // 端口读一个字
15. inline uint16\_t inw(uint16\_t port)
16. {
17. uint16\_t ret;
18. asm volatile ("inw %1, %0" : "=a" (ret) : "dN" (port));
19. return ret;
20. }

对应的头文件如下：

/include/common.h

1. #ifndef INCLUDE\_COMMON\_H\_
2. #define INCLUDE\_COMMON\_H\_
3. #include "types.h"
4. // 端口写一个字节
5. void outb(uint16\_t port, uint8\_t value);
6. // 端口读一个字节
7. uint8\_t inb(uint16\_t port);
8. // 端口读一个字
9. uint16\_t inw(uint16\_t port);
10. #endif

这一部分用了一些不常见的内容，介绍下：

1. 内联函数，用inline关键字来修饰，这样的声明建议编译器在编译的时候，直接把函数代码插入调用的地方，但这仅仅是建议，一来因为大量的内联会大量占用空间，再者编译器实际执行中也会有自己的优化。这样的好处就是在实际使用执行中不需要函数调用了，节省了额外的时间开支。
2. volatile 关键字，它是一个变数声明限定词。这个限定会告诉编译器，它所修饰的变数的值，可能会在任何时候被意外地更新，即便在上下午中并没有更新它的语句。也就是说，为了保证我们的程序的正确性，我们不能允许编译器为了提高效率，直接用寄存器中存储过的变量的值，我们需要寄存器严格从内存中读取我们的变量，以保证结果的正确性。换言之，使用volatile对变量做声明，就阻止了编译器对这部分代码/变量进行优化。
3. 内联汇编，即使用ASM，这样可以在高级语言中直接嵌入低级的汇编语言，可以提高我们代码的效率，同时又不需要过多复杂的底层代码的书写。

其格式如下：

1. asm (volatile) ( 汇编语句
2. : 输出操作数 // 非必需
3. : 输入操作数 // 非必需
4. : 其他被污染的寄存器 // 非必需
5. );

(1) 第1行是汇编语句，用双引号引起来， 多条语句用 ; 或者 \n\t 来分隔。

(2) 第2行是输出操作数，都是 "=?"(var) 的形式， var 可以是任意内存变量（输出结果会存到这个变量中） ? 一般是下面这些标识符 （表示内联汇编中用什么来代理这个操作数）：

a,b,c,d,S,D 分别代表 eax,ebx,ecx,edx,esi,edi 寄存器

r 上面的寄存器的任意一个（谁闲着就用谁）

m 内存

i 立即数（常量，只用于输入操作数）

g 寄存器、内存、立即数 都行（gcc你看着办）

在汇编中用 %序号 来代表这些输入/输出操作数， 序号从 0 开始。为了与操作数区分开来， 寄存器用两个%引用，如：%%eax

1. 第3行是输入操作数，都是 "?"(var) 的形式  ? 除了可以是上面的那些标识符，还可以是输出操作数的序号， 表示用 var 来初始化该输出操作数， 上面的程序中 %0 和 %1 就是一个东西，初始化为 1（a的值）。
2. 第4行标出那些在汇编代码中修改了的、 又没有在输入/输出列表中列出的寄存器， 这样 gcc 就不会擅自使用这些"危险的"寄存器。 还可以用 "memory" 表示在内联汇编中修改了内存， 之前缓存在寄存器中的内存变量需要重新读取。

讲完后，用asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));作为例子说明一下，b是byte的意思，outb %1,%0, 是说把%0的值输出到%1端口中。后面我们没有输出操作数，所以是两个:作为分隔，后面的输入操作数，输入了我们的端口值和要写入端口的值。端口用dN修饰，d表示是edx，而N是用在out指令中的，表示[0,255]的常数。再后面a表示eax寄存器。怎么样，这下就清楚了吧。

多说一点，如果你对C语言很熟悉的话，这些功能的实现你可能会首先想到宏和宏函数，是的，宏确实功能强大，而且使用方便，但宏有一个问题就是它对二进制代码不可见，它仅仅是C语言预处理阶段的一种文本替换工具。这就导致一旦我们的函数出了问题，我们是无法通过编译来查找的。同时，虽然我们的操作系统比较简单，但全部做完也有几千行代码，这时候很容易发生没有预料到的文本替换，问题就比较大了，而且排查很困难。

这一部分写完之后，来说说我们对颜色的定义以及屏幕操作函数的实现。

/include/console.h

1. #ifndef INCLUDE\_CONSOLE\_H\_
2. #define INCLUDE\_CONSOLE\_H\_
3. #include "types.h"
4. typedef
5. enum real\_color {
6. rc\_black = 0,
7. rc\_blue = 1,
8. rc\_green = 2,
9. rc\_cyan = 3,
10. rc\_red = 4,
11. rc\_magenta = 5,
12. rc\_brown = 6,
13. rc\_light\_grey = 7,
14. rc\_dark\_grey = 8,
15. rc\_light\_blue = 9,
16. rc\_light\_green = 10,
17. rc\_light\_cyan = 11,
18. rc\_light\_red = 12,
19. rc\_light\_magenta = 13,
20. rc\_light\_brown = 14,
21. rc\_white = 15
22. } real\_color\_t;
23. // 清屏操作
24. void console\_clear();
25. // 屏幕输出一个字符(带颜色)
26. void console\_putc\_color(char c, real\_color\_t back, real\_color\_t fore);
27. // 屏幕打印一个以 \0 结尾的字符串(默认黑底白字)
28. void console\_write(char \*cstr);
29. // 以黑底白字做一个字符的打印输出，模仿putchar()
30. void putchar(char c);
31. // 屏幕打印一个以 \0 结尾的字符串(带颜色)
32. void console\_write\_color(char \*cstr, real\_color\_t back, real\_color\_t fore);
33. // 屏幕输出一个十六进制的整型数
34. void console\_write\_hex(uint32\_t n, real\_color\_t back, real\_color\_t fore);
35. // 屏幕输出一个十进制的整型数
36. void console\_write\_dec(uint32\_t n, real\_color\_t back, real\_color\_t fore);
37. #endif // INCLUDE\_CONSOLE\_H\_

/drivers/console.c

1. #include "console.h"
2. #include "common.h"
3. // VGA 的显示缓冲的起点是 0xB8000
4. static uint16\_t \*video\_memory = (uint16\_t \*)0xB8000;
5. // 屏幕"光标"的坐标
6. static uint8\_t cursor\_x = 0;
7. static uint8\_t cursor\_y = 0;
8. // 移动光标
9. static void move\_cursor()
10. {
11. // 屏幕是 80 字节宽
12. uint16\_t cursorLocation = cursor\_y \* 80 + cursor\_x;
13. outb(0x3D4, 14); // 告诉 VGA 我们要设置光标的高字节
14. outb(0x3D5, cursorLocation >> 8); // 发送高 8 位
15. outb(0x3D4, 15); // 告诉 VGA 我们要设置光标的低字节
16. outb(0x3D5, cursorLocation); // 发送低 8 位
17. }
18. // 屏幕滚动操作
19. static void scroll()
20. {
21. // attribute\_byte 被构造出一个黑底白字的描述格式
22. uint8\_t attribute\_byte = (0 << 4) | (15 & 0x0F);
23. uint16\_t blank = 0x20 | (attribute\_byte << 8); // space 是 0x20
24. // cursor\_y 到 25 的时候，就该换行了
25. if (cursor\_y >= 25) {
26. // 将所有行的显示数据复制到上一行，第一行永远消失了...
27. int i;
28. for (i = 0 \* 80; i < 24 \* 80; i++) {
29. video\_memory[i] = video\_memory[i+80];
30. }
31. // 最后的一行数据现在填充空格，不显示任何字符
32. for (i = 24 \* 80; i < 25 \* 80; i++) {
33. video\_memory[i] = blank;
34. }
35. // 向上移动了一行，所以 cursor\_y 现在是 24
36. cursor\_y = 24;
37. }
38. }
39. // 清屏操作
40. void console\_clear()
41. {
42. uint8\_t attribute\_byte = (0 << 4) | (15 & 0x0F);
43. uint16\_t blank = 0x20 | (attribute\_byte << 8);
44. int i;
45. for (i = 0; i < 80 \* 25; i++) {
46. video\_memory[i] = blank;
47. }
48. cursor\_x = 0;
49. cursor\_y = 0;
50. move\_cursor();
51. }
52. // 屏幕输出一个字符(带颜色)
53. void console\_putc\_color(char c, real\_color\_t back, real\_color\_t fore)
54. {
55. uint8\_t back\_color = (uint8\_t)back;
56. uint8\_t fore\_color = (uint8\_t)fore;
57. uint8\_t attribute\_byte = (back\_color << 4) | (fore\_color & 0x0F);
58. uint16\_t attribute = attribute\_byte << 8;
59. // 0x08 是 退格键 的 ASCII 码
60. // 0x09 是 tab 键 的 ASCII 码
61. if (c == 0x08 && cursor\_x) {
62. cursor\_x--;
63. } else if (c == 0x09) {
64. cursor\_x = (cursor\_x+8) & ~(8-1);
65. } else if (c == '\r') {
66. cursor\_x = 0;
67. } else if (c == '\n') {
68. cursor\_x = 0;
69. cursor\_y++;
70. } else if (c >= ' ') {
71. video\_memory[cursor\_y\*80 + cursor\_x] = c | attribute;
72. cursor\_x++;
73. }
74. // 每 80 个字符一行，满80就必须换行了
75. if (cursor\_x >= 80) {
76. cursor\_x = 0;
77. cursor\_y ++;
78. }
79. // 如果需要的话滚动屏幕显示
80. scroll();
81. // 移动硬件的输入光标
82. move\_cursor();
83. }
84. // 屏幕打印一个以 \0 结尾的字符串(默认黑底白字)
85. void console\_write(char \*cstr)
86. {
87. while (\*cstr) {
88. console\_putc\_color(\*cstr++, rc\_black, rc\_white);
89. }
90. }
91. // 以黑底白字打印输出一个字符，模仿了putchar()的实现
92. void putchar(char c)
93. {
94. console\_putc\_color(c, rc\_black, rc\_white);
95. }
96. // 屏幕打印一个以 \0 结尾的字符串(带颜色)
97. void console\_write\_color(char \*cstr, real\_color\_t back, real\_color\_t fore)
98. {
99. while (\*cstr) {
100. console\_putc\_color(\*cstr++, back, fore);
101. }
102. }
103. // 屏幕输出一个十六进制的整型数
104. void console\_write\_hex(uint32\_t n, real\_color\_t back, real\_color\_t fore)
105. {
106. int tmp;
107. char noZeroes = 1;
108. console\_write\_color("0x", back, fore);
109. int i;
110. for (i = 28; i >= 0; i -= 4) {
111. tmp = (n >> i) & 0xF;
112. if (tmp == 0 && noZeroes != 0) {
113. continue;
114. }
115. noZeroes = 0;
116. if (tmp >= 0xA) {
117. console\_putc\_color(tmp-0xA+'a', back, fore);
118. } else {
119. console\_putc\_color(tmp+'0', back, fore);
120. }
121. }
122. }
123. // 屏幕输出一个十进制的整型数
124. void console\_write\_dec(uint32\_t n, real\_color\_t back, real\_color\_t fore)
125. {
126. if (n == 0) {
127. console\_putc\_color('0', back, fore);
128. return;
129. }
130. uint32\_t acc = n;
131. char c[32];
132. int i = 0;
133. while (acc > 0) {
134. c[i] = '0' + acc % 10;
135. acc /= 10;
136. i++;
137. }
138. c[i] = 0;
139. char c2[32];
140. c2[i--] = 0;
141. int j = 0;
142. while(i >= 0) {
143. c2[i--] = c[j++];
144. }
145. console\_write\_color(c2, back, fore);
146. }

参考注释应该能有一个比较清晰的认识了，此外我对几个重点的地方做个说明。

首先在这个文件里大量使用了static 修饰符。对于其修饰的全局变量来说，比如\*video\_memory，cursor\_x，cursor\_y，这几个变量作用域都只限于文件内（只在文件内可见）即“内部连接”，因而限定了变量的作用域。对于其修饰的函数也是一样的情况。原因在于我们这里的全局变量，函数只在本模块文件内被使用，限定其作用域。可以尽可能的向外部暴露较少的接口。

第5行的static uint16\_t \*video\_memory = (uint16\_t \*)0xB8000 正是我们之前提到的显存的位置，是我们开始放置字符的地方。

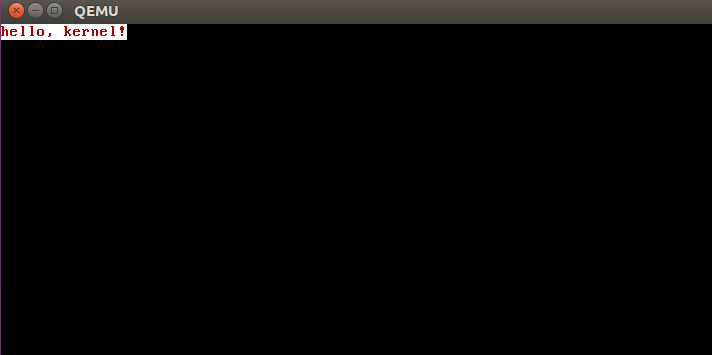
第8,9行的cursor\_x 和 cursor\_y 这两个变量指明了逻辑上的当前输出位置，但  
是并没有实际上移动硬件的显示"光标"，根据move\_cursor()函数实现了根据这两个变量的值移动光标的功能。里面的outb中的端口都是规定值，不多解释了。有兴趣可以自行查询。

有了这些内容，我们就有了最基本的输出函数了。修改一下我们的entry.c，来实验一下：

/entry.c

1. #include"console.h"
2. int kern\_entry()
3. {
4. console\_clear();
5. console\_write\_color("hello, kernel!\n",rc\_white,rc\_red);
6. return 0;
7. }

按照我们设置的颜色，就输出了需要的内容了：



最后列出我们此时的目录：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── drivers

│   └── console.c

├── entry.c

├── floppy.img

├── include

│   ├── common.h

│   ├── console.h

│   └── types.h

├── libs

│   └── common.c

├── Makefile

└── scripts

└── kernel.ld

5 directories, 10 files

<http://www.jianshu.com/p/d66aa643bf40>

<https://en.wikipedia.org/wiki/Memory-mapped_I/O>

<https://www.zhihu.com/question/20722310/answer/50344183>

<https://en.wikipedia.org/wiki/Inline_function>

https://github.com/1184893257/simplelinux/blob/master/inlineasm.md

<https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Extended-Asm.html>