|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 8 Screen03  The Screen03 lesson builds on Screen02, by teaching how to draw text, and also a small feature on the command line arguments of the operating system. It is assumed you have the code for the [Lesson 7: Screen02](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen02.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 String Theory](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html#stringtheory) * [2 Characters](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html#characters) * [3 Strings](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html#strings) * [4 Your Wish is My Command Line](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html#ywimcl) * [5 Hello World](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html#helloworld) |   1 String Theory  So, our task for this operating system is to draw text. We have several problems to address, the most pressing of which is probably about storing text. Unbelievably text has been one of the biggest flaws on computers to date. What should have been a straightforward type of data has brought down operating systems, crippled otherwise wonderful encryption, and caused many problems for users of different alphabets. Nevertheless, it is an incredibly important type of data, as it is an excellent link between the computer and the user. Text can be sufficiently structured that the operating system understands it, as well as sufficiently readable that humans can use it.  Variable data types such as text require much more complex handling.  So how exactly is text stored? Simply enough, we have some system by which we give each letter a unique number, and then store a sequence of such numbers. Sounds easy. The problem is that the number of numbers is not fixed. Some pieces of text are longer than others. With storing ordinary numbers, we have some fixed limit, e.g. 32 bits, and then we can't go beyond that, we write methods that use numbers of that length, etc. In terms of text, or strings as we often call it, we want to write functions that work on variable length strings, otherwise we would need a lot of functions! This is not a problem for numbers normally, as there are only a few common number formats (byte, word, half, double).  Buffer overrun attacks have plagued computers for years. Recently, the Wii, Xbox and Playstation 2 all suffered buffer overrun attacks, as well as large systems like Microsoft's Web and Database servers.  So, how do we determine how long the string is? I think the obvious answer is just to store how long the string is, and then to store the characters that make it up. This is called length prefixing, as the length comes before the string. Unfortunately, the pioneers of computer science did not agree. They felt it made more sense to have a special character called the null terminator (denoted \0) which represents when a string ends. This does indeed simplify many string algorithms, as you just keep working until the null terminator. Unfortunately this is the source of many security issues. What if a malicious user gives you a very long string? What if you didn't have enough space to store it. You might run a string copying function that copies until the null terminator, but because the string is so long, it overwrites your program. This may sound far fetched, but nevertheless, such buffer overrun attacks are incredibly common. Length prefixing mitigates this problem as it is easy to deduce the size of the buffer required to store the string. As an operating system developer, I leave it to you to decide how best to store text.  The next thing we need to establish is how best to map characters to numbers. Fortunately, this is reasonably well standardised, so you have two major choices, Unicode and ASCII. Unicode maps almost every single useful symbol that can be written to a number, in exchange for having a lot more numbers, and a more complicated encoding system. ASCII uses one byte per character, and so only stores the Latin alphabet, numbers, a few symbols and a few special characters. Thus, ASCII is very easy to implement, compared to Unicode, in which not every character takes the same space, making string algorithms tricky. Normally operating systems use ASCII for strings which will not be displayed to end users (but perhaps to developers or experts), and Unicode for displaying messages to users, as Unicode can support things like Japanese characters, and so could be localised.  Fortunately for us, this decision is irrelevant at the moment, as the first 128 characters of both are exactly the same, and are encoded exactly the same.   | Table 1.1 ASCII/Unicode symbols 0-127 | | | | | | | | | | | | | | | | | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | | **00** | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI | | **10** | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US | | **20** |  | ! | " | # | $ | % | & | . | ( | ) | \* | + | , | - | . | / | | **30** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? | | **40** | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | | **50** | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | \_ | | **60** | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | | **70** | p | q | r | s | t | u | v | w | x | y | z | { | | | } | ~ | DEL |   The table shows the first 128 symbols. The hexadecimal representation of the number for a symbol is the row value added to the column value, for example A is 4116. What you may find surprising is the first two rows, and the very last value. These 33 special characters are not printed at all. In fact, these days, many are ignored. They exist because ASCII was originally intended as a system for transmitting data over computer networks, and so a lot more information than just the symbols had to be sent. The key special symbols that you should learn are NUL, the null terminator character I mentioned before, HT, horizontal tab is what we normally refer to as a tab and LF, the line feed character is used to make a new line. You may wish to research and use the other characters for special meanings in your operating system.  2 Characters  So, now that we know a bit about strings, we can start to think about how they're displayed. The fundamental thing we need to do in order to be able to display a string is to be able to display a character. Our first task will be making a DrawCharacter function which takes in a character to draw and a location, and then draws the character.  The true type font format used in many Operating Systems is so powerful, it has its own assembly language built in to ensure letters look correct at any resolution.  Naturally, this leads to a discussion about fonts. We already know there are many ways to display any given letter in accordance with font choice. So how does a font work? In the very early days of computer science, a font was just a series of little pictures of all the letters, called a bitmap font, and all the draw character method would do is copy one of the pictures to the screen. The trouble with this is when people want to resize the text. Sometimes we need big letters, and sometimes small. Although we could keep drawing new pictures for every font at every size with every character, this would get tedious. Thus, vector fonts were invented. in vector fonts, rather than containing an image of the font, the font file contains a description of how to draw it, e.g. an 'o' could be circle with radius half that of the maximum letter height. Modern operating systems use such fonts almost exclusively, as they are perfect at any resolution.  Unfortunately, much though I would love to include an implementation of one of the vector font formats, it would take up the remainder of this website. Thus, we will implement a bitmap font, though if you wish to make a decent graphical operating system, a vector font would be useful.   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |   On the downloads page, I have included several '.bin' files in the font section. These are just raw binary data files for a few fonts. For this tutorial, pick your favourite from the monospace, monochrome, 8x16 section. Download it and store it in the 'source' directory as 'font.bin'. These files are just monochrome images of each of the letters in turn, with each letter being exactly 8 by 16 pixels. Thus, each takes 16 bytes, the first byte being the top row, the second the next, etc.  The diagram shows the 'A' character in the monospace, monochrome, 8x16 font Bitstream Vera Sans Mono. In the file, we would find this starting at the 4116 × 1016 = 41016th byte as the following sequence in hexadecimal:  00, 00, 00, 10, 28, 28, 28, 44, 44, 7C, C6, 82, 00, 00, 00, 00  We're going to use a monospace font here, because in a monospace font every character is the same size. Unfortunately, yet another complication with most fonts is that the character's widths vary, leading to more complex display code. I've included a few other fonts on the downloads page, as well as an explanation of the format I've stored them all in.  So let's get down to business. Copy the following to 'drawing.s' after the **.int 0** of graphicsAddress.  .align 4 font: .incbin "font.bin"  **.incbin "file"** inserts the binary data from the file **file**.  This code copies the font data from the file to the address labelled font. We've used an **.align 4** here to ensure each character starts on a multiple of 16 bytes, which can be used for a speed trick later.  Now we want to write the draw character method. I'll give the pseudo code for this, so you can try to implement it yourself if you want to. Conventionally >> means logical shift right.  function drawCharacter(r0 is character, r1 is x, r2 is y)  if character > 127 then exit  set charAddress to font + character × 16 for row = 0 to 15  set bits to readByte(charAddress + row) for bit = 0 to 7  if test(bits >> bit, 0x1) then setPixel(x + bit, y + row)  next  next return r0 = 8, r1 = 16  end function  If implemented directly, this is deliberately not very efficient. With things like drawing characters, efficiency is a top priority, as we will do it a lot. Let's explore some improvements that bring this closer to optimal assembly code. Firstly, we have a × 16, which by now you should spot is the same as a logical shift left by 4 places. Next we have a variable row, which is only ever added to charAddress and to y. Thus, we can eliminate it by increasing these variables instead. The only issue now is how to tell when we've finished. This is where the **.align 4** comes in handy. We know that charAddress will start with the low nibble containing 0. This means we can see how far into the character data we are by checking that low nibble.  Though we can eliminate the need for bits, we must introduce a new variable to do so, so it is best left in. The only other improvement that can be made is to remove the nested bits >> bit.  function drawCharacter(r0 is character, r1 is x, r2 is y)  if character > 127 then exit  set charAddress to font + character << 4 loop  set bits to readByte(charAddress) set bit to 8 loop  set bits to bits << 1 set bit to bit - 1 if test(bits, 0x100) then setPixel(x + bit, y)  until bit = 0 set y to y + 1 set chadAddress to chadAddress + 1  until charAddress AND 0b1111 = 0 return r0 = 8, r1 = 16  end function  Now we've got code that is much closer to assembly code, and is near optimal. Below is the assembly code version of the above.  .globl DrawCharacter DrawCharacter: cmp r0,#127 movhi r0,#0 movhi r1,#0 movhi pc,lr  push {r4,r5,r6,r7,r8,lr} x .req r4 y .req r5 charAddr .req r6 mov x,r1 mov y,r2 ldr charAddr,=font add charAddr, r0,lsl #4  lineLoop$:  bits .req r7 bit .req r8 ldrb bits,[charAddr] mov bit,#8  charPixelLoop$:  subs bit,#1 blt charPixelLoopEnd$ lsl bits,#1 tst bits,#0x100 beq charPixelLoop$  add r0,x,bit mov r1,y bl DrawPixel  teq bit,#0 bne charPixelLoop$  charPixelLoopEnd$: .unreq bit .unreq bits add y,#1 add charAddr,#1 tst charAddr,#0b1111 bne lineLoop$  .unreq x .unreq y .unreq charAddr  width .req r0 height .req r1 mov width,#8 mov height,#16  pop {r4,r5,r6,r7,r8,pc} .unreq width .unreq height  3 Strings  Now that we can draw characters, we can draw text. We need to make a method that, for a given string, draws each character in turn, at incrementing positions. To be nice, we shall also implement new lines and tabs. It's decision time as far as null terminators are concerned, and if you want to make your operating system use them, feel free by changing the code below. To avoid the issue, I will have the length of the string passed as an argument to the DrawString function, along with the address of the string, and the x and y coordinates.  function drawString(r0 is string, r1 is length, r2 is x, r3 is y)  set x0 to x for pos = 0 to length - 1  set char to loadByte(string + pos) set (cwidth, cheight) to DrawCharacter(char, x, y) if char = '\n' then /\* \n is short hand for the character LF \*/  set x to x0 set y to y + cheight  otherwise if char = '\t' then /\* \t is short hand for the character HT \*/  set x1 to x until x1 > x0  set x1 to x1 + 5 × cwidth  loop set x to x1  otherwise  set x to x + cwidth  end if  next  end function  Once again, this function isn't that close to assembly code. Feel free to try to implement it either directly or by simplifying it. I will give the simplification and then the assembly code below.  Clearly the person who wrote this function wasn't being very efficient (me in case you were wondering). Once again we have a pos variable that just increments and is added to something else, which is completely unnecessary. We can remove it, and instead simultaneously decrement length until it is 0, saving the need for one register. The rest of the function is probably fine, except for that annoying multiplication by five. A key thing to do here would be to move the multiplication outside the loop; multiplication is slow even with bit shifts, and since we're always adding the same constant multiplied by 5, there is no need to recompute this. It can in fact be implemented in one operation using the argument shifting in assembly code, so I shall rephrase it like that.  function drawString(r0 is string, r1 is length, r2 is x, r3 is y)  set x0 to x until length = 0  set length to length - 1 set char to loadByte(string) set (cwidth, cheight) to DrawCharacter(char, x, y) if char = '\n' then  set x to x0 set y to y + cheight  otherwise if char = '\t' then  set x1 to x set cwidth to cwidth + cwidth << 2 until x1 > x0  set x1 to x1 + cwidth  loop set x to x1  otherwise  set x to x + cwidth  end if set string to string + 1  loop  end function  In assembly code:  .globl DrawString DrawString: x .req r4 y .req r5 x0 .req r6 string .req r7 length .req r8 char .req r9 push {r4,r5,r6,r7,r8,r9,lr}  mov string,r0 mov x,r2 mov x0,x mov y,r3 mov length,r1  stringLoop$:  subs length,#1 blt stringLoopEnd$  ldrb char,[string] add string,#1  mov r0,char mov r1,x mov r2,y bl DrawCharacter cwidth .req r0 cheight .req r1  teq char,#'\n' moveq x,x0 addeq y,cheight beq stringLoop$  teq char,#'\t' addne x,cwidth bne stringLoop$  add cwidth, cwidth,lsl #2 x1 .req r1 mov x1,x0  stringLoopTab$:  add x1,cwidth cmp x,x1 bge stringLoopTab$  mov x,x1 .unreq x1  b stringLoop$  stringLoopEnd$: .unreq cwidth .unreq cheight  pop {r4,r5,r6,r7,r8,r9,pc} .unreq x .unreq y .unreq x0 .unreq string .unreq length  **subs reg,#val** subtracts **val** from the register **reg** and compares the result with 0.  This code makes clever use of a new operation, **subs** which subtracts one number from another, stores the result and then compares it with 0. In truth, all comparisons are implemented as a subtraction and then comparison with 0, but the result is normally discarded. This means that this operation is as fast as **cmp**.  4 Your Wish is My Command Line  Now that we can print strings, the challenge is to find an interesting one to draw. Normally in tutorials such as this, people just draw "Hello World!", but after all we've done so far, I feel that is a little patronising (feel free to do so if it helps). Instead we're going to draw our command line.  A convention has been made for computers running ARM. When they boot, it is important they are given certain information about what they have available to them. Most all processors have some way of ascertaining this information, and on ARM this is by data left at the address 10016. The format of the data is as follows:   1. The data is broken down into a series of 'tags'. 2. There are nine types of tag: 'core', 'mem', 'videotext', 'ramdisk', 'initrd2', 'serial' 'revision', 'videolfb', 'cmdline'. 3. Each can only appear once, but all but the 'core' tag don't have to appear. 4. The tags are placed from 0x100 in order one after the other. 5. The end of the list of tags always contains 2 words which are 0. 6. Every tag's size in bytes is a multiple of 4. 7. Each tag starts with the size of the tag in words in the tag, including this number. 8. This is followed by a half word containing the tag's number. These are numbered from 1 in the order above ('core' is 1, 'cmdline' is 9). 9. This is followed by a half word containing 544116. 10. After this comes the data of the tag, which varies depending on the tag. The size of the data in words + 2 is always the same as the length mentioned above. 11. A 'core' tag is either 2 or 5 words in length. If it is 2, there is no data, if it is 5, it has 3 words. 12. A 'mem' tag is always 4 words in length. The data is the first address in a block of memory, and the length of that block. 13. A 'cmdline' tag contains a null terminated string which is the parameters of the kernel.   Almost all Operating Systems support the notion of programs having a 'command line'. The idea is to provide a common mechanism for choosing the desired behaviour of the program.  On the current version of the Raspberry Pi, only the 'core', 'mem' and 'cmdline' tags are present. You may find these useful later, and a more complete reference for these is on our Raspberry Pi reference page. The one we're interested in at the moment is the 'cmdline' tag, because it contains a string. We're going to write some code to search for the command line tag, and, if found, to print it out with each item on a new line. The command line is just a list of things that either the graphics processor or the user thought it might be nice for the Operating System to know. On the Raspberry Pi, this includes the MAC Address, serial number and screen resolution. The string itself is just a list of expressions such as 'key.subkey=value' separated by spaces.  Let's start by finding the 'cmdline' tag. In a new file called 'tags.s' copy the following code.  .section .data tag\_core: .int 0 tag\_mem: .int 0 tag\_videotext: .int 0 tag\_ramdisk: .int 0 tag\_initrd2: .int 0 tag\_serial: .int 0 tag\_revision: .int 0 tag\_videolfb: .int 0 tag\_cmdline: .int 0  Looking through the list of tags will be a slow operation, as it involves a lot of memory access. Therefore, we only want to have to do it once. This code creates some data which will store the memory address of the first tag of each of the types. Then, to find a tag the following pseudo code will suffice.  function FindTag(r0 is tag)  if tag > 9 or tag = 0 then return 0 set tagAddr to loadWord(tag\_core + (tag - 1) × 4) if not tagAddr = 0 then return tagAddr if readWord(tag\_core) = 0 then return 0 set tagAddr to 0x100 loop forever  set tagIndex to readHalfWord(tagAddr + 4) if tagIndex = 0 then return FindTag(tag) if readWord(tag\_core+(tagIndex-1)×4) = 0 then storeWord(tagAddr, tag\_core+(tagIndex-1)×4) set tagAddr to tagAddr + loadWord(tagAddr) × 4  end loop  end function  This code is already quite well optimised and close to assembly. It is optimistic in that the first thing it tries is loading the tag directly, as all but the first time this should be the case. If that fails, it checks if the core tag has an address. Since there must always be a core tag, the only reason that it would not have an address is if it doesn't exist. If it does have an address, the tag we were looking for didn't. If it doesn't we need to find the addresses of all the tags. It does this by reading the number of the tag. If it is zero, that must mean we are at the end of the list. This means we've now filled in all the tags in our directory. Therefore if we run our function again, it will now be able to produce an answer. If the tag number is not zero, we check to see if this tag type already has an address. If not, we store the address of this tag in our directory. We then add the length of this tag in bytes to the tag address to find the next tag.  Have a go at implementing this code in assembly. You will need to simplify it. If you get stuck, my answer is below. Don't forget the **.section .text**!  .section .text .globl FindTag FindTag: tag .req r0 tagList .req r1 tagAddr .req r2  sub tag,#1 cmp tag,#8 movhi tag,#0 movhi pc,lr  ldr tagList,=tag\_core tagReturn$: add tagAddr,tagList, tag,lsl #2 ldr tagAddr,[tagAddr]  teq tagAddr,#0 movne r0,tagAddr movne pc,lr  ldr tagAddr,[tagList] teq tagAddr,#0 movne r0,#0 movne pc,lr  mov tagAddr,#0x100 push {r4} tagIndex .req r3 oldAddr .req r4 tagLoop$: ldrh tagIndex,[tagAddr,#4] subs tagIndex,#1 poplt {r4} blt tagReturn$  add tagIndex,tagList, tagIndex,lsl #2 ldr oldAddr,[tagIndex] teq oldAddr,#0 .unreq oldAddr streq tagAddr,[tagIndex]  ldr tagIndex,[tagAddr] add tagAddr, tagIndex,lsl #2 b tagLoop$  .unreq tag .unreq tagList .unreq tagAddr .unreq tagIndex  5 Hello World  Now that we have everything we need, we can draw our first string. In 'main.s' delete everything after **bl SetGraphicsAddress**, and replace it with the following:  mov r0,#9 bl FindTag ldr r1,[r0] lsl r1,#2 sub r1,#8 add r0,#8 mov r2,#0 mov r3,#0 bl DrawString loop$: b loop$  This code simply uses our FindTag method to find the 9th tag (cmdline) and then calculates its length and passes the command and the length to the DrawString method, and tells it to draw the string at 0,0. Now test this on the Raspberry Pi. You should see a line of text on the screen. If not please see our troubleshooting page.  Once it works, congratulations you've now got the ability to draw text. But there is still room for improvement. What if we wanted to write out a number, or a section of the memory or manipulate our command line? In [Lesson 9: Screen04](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html), we will look at manipulating text and displaying useful numbers and information. | 第8课 屏幕03  这一节课是基于屏幕02的，教授的内容是如何绘制文本。同时介绍一些操作系统的命令行参数的一些小特征。这里假设你拥有第7课：屏幕02的代码基础。  目录   1. 串理论 2. 字符 3. 串 4. 你想要的是自己的命令行 5. 你好世界 6. 串理论   那么，为了实现这个操作系统，我们的任务是绘制文本。但是首先有几个问题要声明一下。这些问题中，首当其中的可能要数如何存储这些文本。令人男难以置信的是，文本曾经是计算机想要数据化的最大的一个缺陷，而且至今仍然还是。对于使用不同字母表的用户来说，文本应该编成什么类型的码，一度导致操作系统的性能下降，并严重破坏了其他的优秀的加密系统。然而，文本仍然是最重要的的数据类型，因为它是计算机和用户之间的一个极其重要的链接。文本可以被有效地组织起来，以便操作系统可以理解它们，而且人类也可以阅读并使用这些文本。  像文本这样的多变的数据类型需要更多的复杂处理。  那么，文本到底是如何存储的呢？作为一个足够简单的例子，我们可以使用这样的一个系统：把每个字母都赋予一个独一无二的数字，然后我们就存储这些数字即可。听起来很简单，但是问题是表示这些字符的数值并不固定。有的文本片段长，有的短些。如果存储普通的数值，那么我们就会有固定的限制，例32个位。并且当我们写个函数来使用文本片段长度信息时，我们不能超越这个32位的限制。从我们经常使用的文本或者串的方面来考虑的话，我们想要写个函数来工作在各种长度的串上，否则的话我们将不得不需要一大堆的函数！一般情况下，这还不是数值的问题，因为存在很少的普通数值格式（字节，字，半字，双字）。  缓存溢出攻击已经困扰了计算机很多年了。最近，Wii、Xbox和Playstation 2都遭受到了缓存溢出攻击的困扰，更不用说像微软Web和Database服务器之类的更加巨大的系统了。  那么，我们如何决定串的长度呢？我认为明显的答案就是把字符串的长度存储起来，然后再存储串中的字符。这个放在串的前面的信息被称为长度前缀。很不幸，计算机科学的前辈们并不同意这么做。他们感觉使用一个特殊的字符来标识一个串的结束更加有道理（表示为\0，被称为空终止符）。这么做确实可以简化很多和串有关的算法，因为我们只需要工作到空终止符为止。很不幸的是，这却恰恰是很多安全问题的根源。如果一个怀有恶意的用户给你发送一个非常长的字符串会怎样？你可能要运行一个字符串拷贝函数来，该函数的终止的条件是遇到该字符串的空终止符。但是因为这个字符串非常长，这将导致你的程序会被覆写。这听起来有些天方夜谭，但是就是这么个缓存溢出攻击却是极其的常见。长度前缀可以缓解这个问题，因为可以据此很容易地计算出要保存该字符串所需要的缓存大小。作为一名操作系统的开发者，我把这个问题留给你来决定。  下一个我们要确定的事情是：如何最优地把字符映射为数值。很幸运，这些都有大致的标准。因为你大致可以有两种主要的选择：Unicode和ASCII。Unicode编码方案把每个可以写出的单个有用的符号映射为一个数值，作为代价，这种方案生成的数值会有更多的数字，且编码系统更加复杂。ASCII使用一个字节来存储拉丁字母、数字和一些符号以及一些特殊符号。因此，相比于Unicode，ASCII非常容易实现。因为在Unicode种，并不是每个字符都占据相同的空间，这就会导致字符串的算法必须使用些小技巧来满足效率要求。一般情况下，操作系统内部使用的是ASCII来使用字符串，他们并不显示给终端用户（或许开发者或者专家可能会用到。），而Unicode则会用来给用户显示消息。这是因为Unicode可以支持像日语之类的字符，因此这就可以本地化了。  很幸运，此时此刻的这个决策无关紧要。两种编码系统的头128个字符都是一样的，而且也被编码成一模一样。   | 表1.1 ASCII/Unicode 符号0-127 | | | | | | | | | | | | | | | | | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | | **00** | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI | | **10** | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US | | **20** |  | ! | " | # | $ | % | & | . | ( | ) | \* | + | , | - | . | / | | **30** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? | | **40** | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | | **50** | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | \_ | | **60** | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | | **70** | p | q | r | s | t | u | v | w | x | y | z | { | | | } | ~ | DEL |   这张表格展示了头128个字符。一个字符的数值的十六进制表达式是其行值加上列值。例如A就是十六进制的41。你可能已经发现了，头两行以及最后一行的值，是不是很令人惊奇？这33个特殊字符是不能被打印出来的。事实上，当今的计算机会忽略很多这些字符。它们的存在的原因是因为起初ASCII的发明主要是为了在计算机网络上传送数据，许多大量的信息需要这些特殊的字符来协助传输过程。最关键的特殊字符是NUL，你必须好好了解它。它表示的是空终止字符。而HT表示水平指标符，LF表示行回退字符，通常用来建立一个新的行。或许你对搞明白其他特殊字符的涵义有浓厚的兴趣，那就尽管学习吧！把它们用到你的操作系统中吧。   1. 字符   既然我们已经略微了解了字符串的一些知识，那么我们就可以考虑如何显示它们了。要想显示一串字符，首先要做的是显示一个字符。我们的第一个任务就是编写一个DrawCharacter函数。该函数需要一个字符和该字符的位置的输入，然后把这个字符绘制出来。  在许多操作系统中使用的真类型字体格式非常强大。它内嵌自己的汇编语言，以确保在任何分辨率下文本都能够正确显示。  很自然地，我们这里将要讨论一下字体。根据字体的选择，我们有很多种方法来显示它们。那么，字体是如何工作的呢？在早期的计算机科学中，字体就是一系列字母的图片，这些被称为位图字体。所有和绘制字符有关的函数都是将这些图片复制到屏幕上去。但是困难发生在当人们想要把文本的大小改变时。有时我们需要一些大的字，但有时却需要小的。尽管我们可以为每个字符的每个尺寸定义新的图片，但是这会变得很冗长，令人感到厌烦。因而，矢量字体被发明了出来。相比于保存字体的图片，矢量字体的做法却是仅仅保存如何绘制字体的描述。例如，字符o就被描述为最高字母高度一半的圆。现代的操作系统专门使用这样的字体，因为它们可以在任何分辨率下完美显示。  很不幸的是，尽管我也很想去囊括一种矢量字体格式的实现，但是这恐怕将会占据这个教程的剩下的全部空间。因此，我将只实现一种位图字体。如果你想要做一个优雅的美观的图形操作系统，矢量字体将非常有用。   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |   在下载页，我已经收录了一些“.bin”文件，并把它们放在了字体一节中。这些文件仅仅是一些字体的原生二进制数据文件。你可以从monospace、monochrome以及8x16的小节中选择你最喜欢的，然后下载到你的电脑里，存储位置为“source”目录，文件名为“font.bin”。这些文件保存的是每个字母的monochrome图片，而且一个接着一个，每个字母精确的限定在8x16个像素内。因此，每个字母占据16个字节，这16个字节中，第一个字节位于顶行，第二个位于第二行，以此类推。  图片展示了字符A的monospace、monochrome以及8x16字体的位流信息。在文件中，我们会发现下面的16进制序列的起始位置为4116 × 1016 = 41016字节处：  00, 00, 00, 10, 28, 28, 28, 44, 44, 7C, C6, 82, 00, 00, 00, 00  我们这里将会用到monospace字体，因为monospace字体中每个字符的大小都是一样的。很不幸，一小部分字符的宽度是变化的，这就会导致显示代码的负责度增加。在下载页中，我还包含了其他一些字体，作为一种扩展，我把它们都保存了下来。  那么，让我们开始正题吧。把下面的代码复制到文件“”的函数graphicsAddress的.int 0的后面。  .align 4 font: .incbin "font.bin"  指令.incbin “file”的意思是从file文件中读取二进制数据。  这段代码从文件中拷贝字体数据，并把数据复制到标签我为font的地址处。这里我使用了.align 4，以便确保每个字符开始的地址是16字节的整数倍数，这样就可以加快字符数据的存取速度。  现在，我们要编写绘制字符的函数了。这里我给出伪代码。如果你愿意，你可以自己尝试着去实现它。一般说来，操作符>>的意思是右移。  function drawCharacter(r0 is character, r1 is x, r2 is y)  if character > 127 then exit  set charAddress to font + character × 16 for row = 0 to 15  set bits to readByte(charAddress + row) for bit = 0 to 7  if test(bits >> bit, 0x1) then setPixel(x + bit, y + row)  next  next return r0 = 8, r1 = 16  end function  如果直接这么实现，很明显，它并不十分有效。像绘制字符这样的事情，效率是头等大事，因为我们会大量地使用到这些函数。让我们尝试一些改进，以便把我们的汇编代码最优化。首先，我们发现一个x16的操作。你应该注意到它实际上和左移4位是等价的。紧接着是一个可变的行，这个只是被加到charAddress和y上。因此，我们可以通过增加这些变量的方式来取代它。最后就是完成时的如果告知的事了。这就是.align 4的用处了。我们知道charAddress函数会以包含0的低半个字节的位置处开始。这就意味着我们可以通过检测在字符数据中的低半个字节。  尽管我们消除了位操作的需要，但是我们必须引入一个新的变量来弥补。 |