|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 11 Input02  The Input02 lesson builds on Input01, by building a simple command line interface where the user can type commands and the computer interprets and displays them. It is assumed you have the code for the [Lesson 11: Input01](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 Terminal 1](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html#textbuffer) * [2 Showing the Text](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html#display) * [3 Printing Lines](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html#print) * [4 Standard Input](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html#userinput) * [5 The Terminal: Rise of the Machine](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html#terminator) |   1 Terminal 1  In the early days of computing, there would usually be one large computer in a building, and many 'terminals' which sent commands to it. The computer would take it in turns to execute different incoming commands.  Almost every operating system starts life out as a text terminal. This is typically a black screen with white writing, where you type commands for the computer to execute on the keyboard, and it explains how you've mistyped them, or very occasionally, does what you want. This approach has two main advantages: it provides a simple, robust control mechanism for the computer using only a keyboard and monitor, and it is done by almost every operating system, so is widely understood by system administrators.  Let's analyse what we want to do precisely:   1. Computer turns on, displays some sort of welcome message 2. Computer indicates its ready for input 3. User types a command, with parameters, on the keyboard 4. User presses return or enter to commit the command 5. Computer interprets command and performs actions if command is acceptable 6. Computer displays messages to indicate if command was successful, and also what happened 7. Loop back to 2   One defining feature of such terminals is that they are unified for both input and output. The same screen is used to enter inputs as is used to print outputs. This means it is useful to build an abstraction of a character based display. In a character based display, the smallest unit is a character, not a pixel. The screen is divided into a fixed number of characters which have varying colours. We can build this on top of our existing screen code, by storing the characters and their colours, and then using the DrawCharacter method to push them to the screen. Once we have a character based display, drawing text becomes a matter of drawing a line of characters.  In a new file called terminal.s copy the following code:  .section .data .align 4 terminalStart: .int terminalBuffer terminalStop: .int terminalBuffer terminalView: .int terminalBuffer terminalColour: .byte 0xf .align 8 terminalBuffer: .rept 128\*128 .byte 0x7f .byte 0x0 .endr terminalScreen: .rept 1024/8 \* 768/16 .byte 0x7f .byte 0x0 .endr  This sets up the data we need for the text terminal. We have two main storages: terminalBuffer and terminalScreen. terminalBuffer is storage for all of the text we have displayed. It stores up to 128 lines of text (each containing 128 characters). Each character consists of an ASCII character code and a colour, all of which are initially set to 0x7f (ASCII delete) and 0 (black on a black background). terminalScreen stores the characters that are currently displayed on the screen. It is 128 by 48 characters, similarly initialised. You may think that we only need this terminalScreen, not the terminalBuffer, but storing the buffer has 2 main advantages:   1. We can easily see which characters are different, so we only have to draw those. 2. We can 'scroll' back through the terminal's history because it is stored (to a limit).   You should always try to design systems that do the minimum amount of work, as they run much faster for things which don't often change.  The differing trick is really common on low power Operating Systems. Drawing the screen is a slow operation, and so we only want to draw thing that we absolutely have to. In this system, we can freely alter the terminalBuffer, and then call a method which copies the bits that change to the screen. This means we don't have to draw each character as we go along, which may save time in the long run on very long sections of text that span many lines.  The other values in the .data section are as follows:  **terminalStart**  The first character which has been written in terminalBuffer.  **terminalStop**  The last character which has been written in terminalBuffer.  **terminalView**  The first character on the screen at present. We can use this to scroll the screen.  **temrinalColour**  The colour to draw new characters with.  Circular buffers are an example of an **data structure**. These are just ideas we have for organising data, that we sometimes implement in software.  Diagram showing hellow world being inserted into a circular buffer of size 5.  The reason why terminalStart needs to be stored is because termainlBuffer should be a circular buffer. This means that when the buffer is completely full, the end 'wraps' round to the start, and so the character after the very last one is the first one. Thus, we need to advance terminalStart so we know that we've done this. When wokring with the buffer this can easily be implemented by checking if the index goes beyond the end of the buffer, and setting it back to the beginning if it does. Circular buffers are a common and clever way of storing a lot of data, where only the most recent data is important. It allows us to keep writing indefinitely, while always being sure there is a certain amount of recent data available. They're often used in signal processing or compression algorithms. In this case, it allows us to store a 128 line history of the terminal, without any penalties for writing over 128 lines. If we didn't have this, we would have to copy 127 lines back a line very time we went beyond the 128th line, wasting valuable time.  I've mentioned the terminalColour here a few times. You can implement this however you, wish, however there is something of a standard on text terminals to have only 16 colours for foreground, and 16 colours for background (meaning there are 162 = 256 combinations). The colours on a CGA terminal are defined as follows:   | Table 1.1 - CGA Colour Codes | | | --- | --- | | **Number** | **Colour (R, G, B)** | | 0 | Black (0, 0, 0) | | 1 | Blue (0, 0, ⅔) | | 2 | Green (0, ⅔, 0) | | 3 | Cyan (0, ⅔, ⅔) | | 4 | Red (⅔, 0, 0) | | 5 | Magenta (⅔, 0, ⅔) | | 6 | Brown (⅔, ⅓, 0) | | 7 | Light Grey (⅔, ⅔, ⅔) | | 8 | Grey (⅓, ⅓, ⅓) | | 9 | Light Blue (⅓, ⅓, 1) | | 10 | Light Green (⅓, 1, ⅓) | | 11 | Light Cyan (⅓, 1, 1) | | 12 | Light Red (1, ⅓, ⅓) | | 13 | Light Magenta (1, ⅓, 1) | | 14 | Yellow (1, 1, ⅓) | | 15 | White (1, 1, 1) |   Brown was used as the alternative (dark yellow) was unappealing and not useful.  We store the colour of each character by storing the fore colour in the low nibble of the colour byte, and the background colour in the high nibble. Apart from brown, all of these colours follow a pattern such that in binary, the top bit represents adding ⅓ to each component, and the other bits represent adding ⅔ to individual components. This makes it easy to convert to RGB colour values.  We need a method, TerminalColour, to read these 4 bit colour codes, and then call SetForeColour with the 16 bit equivalent. Try to implement this on your own. If you get stuck, or have not completed the Screen series, my implementation is given below:  .section .text TerminalColour: teq r0,#6 ldreq r0,=0x02B5 beq SetForeColour  tst r0,#0b1000 ldrne r1,=0x52AA moveq r1,#0 tst r0,#0b0100 addne r1,#0x15 tst r0,#0b0010 addne r1,#0x540 tst r0,#0b0001 addne r1,#0xA800 mov r0,r1 b SetForeColour  2 Showing the Text  The first method we really need for our terminal is TerminalDisplay, one that copies the current data from terminalBuffer to terminalScreen and the actual screen. As mentioned, this method should do a minimal amount of work, because we need to be able to call it often. It should compare the text in terminalBuffer with that in terminalDisplay, and copy it across if they're different. Remember, terminalBuffer is a circular buffer running, in this case, from terminalView to terminalStop or 128\*48 characters, whichever comes sooner. If we hit terminalStop, we'll assume all characters after that point are 7f16 (ASCII delete), and have colour 0 (black on a black background).  Let's look at what we have to do:   1. Load in terminalView, terminalStop and the address of terminalDisplay. 2. For each row:    1. For each column:       1. If view is not equal to stop, load the current character and colour from view       2. Otherwise load the character as 0x7f and the colour as 0       3. Load the current character from terminalDisplay       4. If the character and colour are equal, go to 10       5. Store the character and colour to terminalDisplay       6. Call TerminalColour with the background colour in r0       7. Call DrawCharacter with r0 = 0x7f (ASCII delete, a block), r1 = x, r2 = y       8. Call TerminalColour with the foreground colour in r0       9. Call DrawCharacter with r0 = character, r1 = x, r2 = y       10. Increment the position in terminalDisplay by 2       11. If view and stop are not equal, increment the view position by 2       12. If the view position is at the end of textBuffer, set it to the start       13. Increment the x co-ordinate by 8    2. Increment the y co-ordinate by 16   Try to implement this yourself. If you get stuck, my solution is given below:   1. .globl TerminalDisplay TerminalDisplay: push {r4,r5,r6,r7,r8,r9,r10,r11,lr} x .req r4 y .req r5 char .req r6 col .req r7 screen .req r8 taddr .req r9 view .req r10 stop .req r11  ldr taddr,=terminalStart ldr view,[taddr,#terminalView - terminalStart] ldr stop,[taddr,#terminalStop - terminalStart] add taddr,#terminalBuffer - terminalStart add taddr,#128\*128\*2  mov screen,taddr   I go a little wild with variables here. I'm using taddr to store the location of the end of the textBuffer for ease.   1. mov y,#0 yLoop$:   Start off the y loop.   * 1. mov x,#0 xLoop$:   Start off the x loop.   * + 1. teq view,stop ldrneh char,[view]   I load both the character and the colour into char simultaneously for ease.   * + 1. moveq char,#0x7f   This line complements the one above by acting as though a black delete character was read.   * + 1. ldrh col,[screen]   For simplicity I load both the character and colour into col simultaneously.   * + 1. teq col,char beq xLoopContinue$   Now we can check if anything has changed with a teq.   * + 1. strh char,[screen]   We can also easily save the current value.   * + 1. lsr col,char,#8 and char,#0x7f lsr r0,col,#4 bl TerminalColour   I split up char into the colour in col and the character in char with a bitshift and an and, then use a bitshift to get the background colour to call TerminalColour.   * + 1. mov r0,#0x7f mov r1,x mov r2,y bl DrawCharacter   Write out a delete character which is a coloured block.   * + 1. and r0,col,#0xf bl TerminalColour   Use an and to get the low nibble of col then call TerminalColour.   * + 1. mov r0,char mov r1,x mov r2,y bl DrawCharacter   Write out the character we're supposed to write.   * + 1. xLoopContinue$: add screen,#2   Increment the screen pointer.   * + 1. teq view,stop addne view,#2   Increment the view pointer if necessary.   * + 1. teq view,taddr subeq view,#128\*128\*2   It's easy to check for view going past the end of the buffer because the end of the buffer's address is stored in taddr.   * + 1. add x,#8 teq x,#1024 bne xLoop$   We increment x and then loop back if there are more characters to go.   * 1. add y,#16 teq y,#768 bne yLoop$   We increment y and then loop back if there are more characters to go.  pop {r4,r5,r6,r7,r8,r9,r10,r11,pc} .unreq x .unreq y .unreq char .unreq col .unreq screen .unreq taddr .unreq view .unreq stop  Don't forget to clean up at the end!  3 Printing Lines  Now we have our TerminalDisplay method, which will automatically display the contents of terminalBuffer to terminalScreen, so theoretically we can draw text. However, we don't actually have any drawing routines that work on a character based display. A quick method that will come in handy first of all is TerminalClear, which completely clears the terminal. This can actually very easily be achieved with no loops. Try to deduce why the following method suffices:  .globl TerminalClear TerminalClear: ldr r0,=terminalStart add r1,r0,#terminalBuffer-terminalStart str r1,[r0] str r1,[r0,#terminalStop-terminalStart] str r1,[r0,#terminalView-terminalStart] mov pc,lr  Now we need to make a basic method for character based displays; the Print function. This takes in a string address in r0, and a length in r1, and simply writes it to the current location at the screen. There are a few special characters to be wary of, as well as special behaviour to ensure that terminalView is kept up to date. Let's analyse what it has to do:   1. Check if string length is 0, if so return 2. Load in terminalStop and terminalView 3. Deduce the x-coordinate of terminalStop 4. For each character:    1. Check if the character is a new line    2. If so, increment bufferStop to the end of the line storing a black on black delete character.    3. Otherwise, copy the character in the current terminalColour    4. Check if we're at the end of a line    5. If so, check if the number of characters between terminalView and terminalStop is more than one screen    6. If so, increment terminalView by one line    7. Check if terminalView is at the end of the buffer, replace it with the start if so    8. Check if terminalStop is at the end of the buffer, replace it with the start if so    9. Check if terminalStop equals terminalStart, increment terminalStart by one line if so    10. Check if terminalStart is at the end of the buffer, replace it with the start if so 5. Store back terminalStop and terminalView.   See if you can implement this yourself. My solution is provided below:   1. .globl Print Print: teq r1,#0 moveq pc,lr   This quick check at the beginning makes a call to Print with a string of length 0 almost instant.   1. push {r4,r5,r6,r7,r8,r9,r10,r11,lr} bufferStart .req r4 taddr .req r5 x .req r6 string .req r7 length .req r8 char .req r9 bufferStop .req r10 view .req r11  mov string,r0 mov length,r1  ldr taddr,=terminalStart ldr bufferStop,[taddr,#terminalStop-terminalStart] ldr view,[taddr,#terminalView-terminalStart] ldr bufferStart,[taddr] add taddr,#terminalBuffer-terminalStart add taddr,#128\*128\*2   I do a lot of setup here. bufferStart contains terminalStart, bufferStop contains terminalStop, view contains terminalView, taddr is the address of the end of terminalBuffer.   1. and x,bufferStop,#0xfe lsr x,#1   As per usual, a sneaky alignment trick makes everything easier. Because of the aligment of terminalBuffer, the x-coordinate of any character address is simply the last 8 bits divided by 2.   * 1. charLoop$: ldrb char,[string] and char,#0x7f teq char,#'\n' bne charNormal$   We need to check for new lines.   * 1. mov r0,#0x7f clearLine$: strh r0,[bufferStop] add bufferStop,#2 add x,#1 teq x,#128 blt clearLine$  b charLoopContinue$   Loop until the end of the line, writing out 0x7f; a delete character in black on a black background.   * 1. charNormal$: strb char,[bufferStop] ldr r0,=terminalColour ldrb r0,[r0] strb r0,[bufferStop,#1] add bufferStop,#2 add x,#1   Store the current character in the string and the terminalColour to the end of the terminalBuffer and then increment it and x.   * 1. charLoopContinue$: cmp x,#128 blt noScroll$   Check if x is at the end of a line; 128.   * 1. mov x,#0 subs r0,bufferStop,view addlt r0,#128\*128\*2 cmp r0,#128\*(768/16)\*2   Set x back to 0 and check if we're currently showing more than one screen. Remember, we're using a circular buffer, so if the difference between bufferStop and view is negative, we're actually wrapping around the buffer.   * 1. addge view,#128\*2   Add one lines worth of bytes to the view address.   * 1. teq view,taddr subeq view,taddr,#128\*128\*2   If the view address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.   * 1. noScroll$: teq bufferStop,taddr subeq bufferStop,taddr,#128\*128\*2   If the stop address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.   * 1. teq bufferStop,bufferStart addeq bufferStart,#128\*2   Check if bufferStop equals bufferStart. If so, add one line to bufferStart.   * 1. teq bufferStart,taddr subeq bufferStart,taddr,#128\*128\*2   If the start address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.   1. subs length,#1 add string,#1 bgt charLoop$ 2. Loop until the string is done. 3. charLoopBreak$: sub taddr,#128\*128\*2 sub taddr,#terminalBuffer-terminalStart str bufferStop,[taddr,#terminalStop-terminalStart] str view,[taddr,#terminalView-terminalStart] str bufferStart,[taddr]  pop {r4,r5,r6,r7,r8,r9,r10,r11,pc} .unreq bufferStart .unreq taddr .unreq x .unreq string .unreq length .unreq char .unreq bufferStop .unreq view   Store back the variables and return.  This method allows us to print arbitrary text to the screen. Throughout, I've been using the colour variable, but no where have we actually set it. Normally, terminals use special combinations of characters to change the colour. For example ASCII Escape (1b16) followed by a number 0 to f in hexadecimal could set the foreground colour to that CGA colour number. You can try implementing this yourself; my version is in the further examples section on the download page.  4 Standard Input  By convention, in many programming languages, every program has access to stdin and stdout, which are an input and and output stream linked to the terminal. This is still true on graphical programs, though many don't use it.  Now we have an output terminal that in theory can print out text and display it. That is only half the story however, we want input. We want to implement a method, ReadLine, which stores the next line of text a user types to a location given in r0, up to a maximum length given in r1, and returns the length of the string read in r0. The tricky thing is, the user annoyingly wants to see what they're typing as they type it, they want to use backspace to delete mistakes and they want to use return to submit commands. They probably even want a flashing underscore character to indicate the computer would like input! These perfectly reasonable requests make this method a real challenge. One way to achieve all of this is to store the text they type in memory somewhere along with its length, and then after every character, move the terminalStop address back to where it started when ReadLine was called and calling Print. This means we only have to be able to manipulate a string in memory, and then make use of our Print function.  Lets have a look at what ReadLine will do:   1. If the maximum length is 0, return 0 2. Retrieve the current values of terminalStop and terminalView 3. If the maximum length is bigger than half the buffer size, set it to half the buffer size 4. Subtract one from maximum length to ensure it can store our flashing underscore or a null terminator 5. Write an underscore to the string 6. Write the stored terminalView and terminalStop addresses back to the memory 7. Call Print on the current string 8. Call TerminalDisplay 9. Call KeyboardUpdate 10. Call KeyboardGetChar 11. If it is a new line character go to 16 12. If it is a backspace character, subtract 1 from the length of the string (if it is > 0) 13. If it is an ordinary character, write it to the string (if the length < maximum length) 14. If the string ends in an underscore, write a space, otherwise write an underscore 15. Go to 6 16. Write a new line character to the end of the string 17. Call Print and TerminalDisplay 18. Replace the new line with a null terminator 19. Return the length of the string   Convince yourself that this will work, and then try to implement it yourself. My implementation is given below:   1. .globl ReadLine ReadLine: teq r1,#0 moveq r0,#0 moveq pc,lr   Quick special handling for the zero case, which is otherwise difficult.   1. string .req r4 maxLength .req r5 input .req r6 taddr .req r7 length .req r8 view .req r9  push {r4,r5,r6,r7,r8,r9,lr}  mov string,r0 mov maxLength,r1 ldr taddr,=terminalStart ldr input,[taddr,#terminalStop-terminalStart] ldr view,[taddr,#terminalView-terminalStart] mov length,#0   As per the general theme, I do a lot of initialisations early. input contains the value of terminalStop and view contains terminalView. Length starts at 0.   1. cmp maxLength,#128\*64 movhi maxLength,#128\*64   We have to check for unusually large reads, as we can't process them beyond the size of the terminalBuffer (I suppose we CAN, but it would be very buggy, as terminalStart could move past the stored terminalStop).   1. sub maxLength,#1   Since the user wants a flashing cursor, and we ideally want to put a null terminator on this string, we need 1 spare character.   1. mov r0,#'\_' strb r0,[string,length]   Write out the underscore to let the user know they can input.   1. readLoop$: str input,[taddr,#terminalStop-terminalStart] str view,[taddr,#terminalView-terminalStart]   Save the stored terminalStop and terminalView. This is important to reset the terminal after each call to Print, which changes these variables. Strictly speaking it can change terminalStart too, but this is irreversible.   1. mov r0,string mov r1,length add r1,#1 bl Print   Write the current input. We add 1 to the length for the underscore.   1. bl TerminalDisplay   Copy the new text to the screen.   1. bl KeyboardUpdate   Fetch the latest keyboard input.   1. bl KeyboardGetChar   Retrieve the key pressed.   1. teq r0,#'\n' beq readLoopBreak$ teq r0,#0 beq cursor$ teq r0,#'\b' bne standard$   Break out of the loop if we have an enter key. Also skip these conditions if we have a null terminator and process a backspace if we have one.   1. delete$: cmp length,#0 subgt length,#1 b cursor$   Remove one from the length to delete a character.   1. standard$: cmp length,maxLength bge cursor$ strb r0,[string,length] add length,#1   Write out an ordinary character where possible.   1. cursor$: ldrb r0,[string,length] teq r0,#'\_' moveq r0,#' ' movne r0,#'\_' strb r0,[string,length]   Load in the last character, and change it to an underscore if it isn't one, and a space if it is.   1. b readLoop$ readLoopBreak$:   Loop until the user presses enter.   1. mov r0,#'\n' strb r0,[string,length]   Store a new line at the end of the string.   1. str input,[taddr,#terminalStop-terminalStart] str view,[taddr,#terminalView-terminalStart] mov r0,string mov r1,length add r1,#1 bl Print bl TerminalDisplay   Reset the terminalView and terminalStop and then Print and TerminalDisplay the final input.   1. mov r0,#0 strb r0,[string,length]   Write out the null terminator.   1. mov r0,length pop {r4,r5,r6,r7,r8,r9,pc} .unreq string .unreq maxLength .unreq input .unreq taddr .unreq length .unreq view   Return the length.  5 The Terminal: Rise of the Machine  So, now we can theoretically interact with the user on the terminal. The most obvious thing to do is to put this to the test! In 'main.s' delete everything after **bl UsbInitialise** and copy in the following code:  reset$:  mov sp,#0x8000 bl TerminalClear  ldr r0,=welcome mov r1,#welcomeEnd-welcome bl Print  loop$:  ldr r0,=prompt mov r1,#promptEnd-prompt bl Print  ldr r0,=command mov r1,#commandEnd-command bl ReadLine  teq r0,#0 beq loopContinue$  mov r4,r0  ldr r5,=command ldr r6,=commandTable  ldr r7,[r6,#0] ldr r9,[r6,#4] commandLoop$:  ldr r8,[r6,#8] sub r1,r8,r7  cmp r1,r4 bgt commandLoopContinue$  mov r0,#0 commandName$:  ldrb r2,[r5,r0] ldrb r3,[r7,r0] teq r2,r3 bne commandLoopContinue$ add r0,#1 teq r0,r1 bne commandName$  ldrb r2,[r5,r0] teq r2,#0 teqne r2,#' ' bne commandLoopContinue$  mov r0,r5 mov r1,r4 mov lr,pc mov pc,r9 b loopContinue$  commandLoopContinue$:  add r6,#8 mov r7,r8 ldr r9,[r6,#4] teq r9,#0 bne commandLoop$  ldr r0,=commandUnknown mov r1,#commandUnknownEnd-commandUnknown ldr r2,=formatBuffer ldr r3,=command bl FormatString  mov r1,r0 ldr r0,=formatBuffer bl Print  loopContinue$:  bl TerminalDisplay b loop$  echo:  cmp r1,#5 movle pc,lr  add r0,#5 sub r1,#5  b Print  ok:  teq r1,#5 beq okOn$ teq r1,#6 beq okOff$ mov pc,lr  okOn$:  ldrb r2,[r0,#3] teq r2,#'o' ldreqb r2,[r0,#4] teqeq r2,#'n' movne pc,lr mov r1,#0 b okAct$  okOff$:  ldrb r2,[r0,#3] teq r2,#'o' ldreqb r2,[r0,#4] teqeq r2,#'f' ldreqb r2,[r0,#5] teqeq r2,#'f' movne pc,lr mov r1,#1  okAct$:  mov r0,#16 b SetGpio  .section .data .align 2 welcome: .ascii "Welcome to Alex's OS - Everyone's favourite OS" welcomeEnd: .align 2 prompt: .ascii "\n> " promptEnd: .align 2 command:  .rept 128  .byte 0  .endr  commandEnd: .byte 0 .align 2 commandUnknown: .ascii "Command `%s' was not recognised.\n" commandUnknownEnd: .align 2 formatBuffer:  .rept 256  .byte 0  .endr  formatEnd:  .align 2 commandStringEcho: .ascii "echo" commandStringReset: .ascii "reset" commandStringOk: .ascii "ok" commandStringCls: .ascii "cls" commandStringEnd:  .align 2 commandTable: .int commandStringEcho, echo .int commandStringReset, reset$ .int commandStringOk, ok .int commandStringCls, TerminalClear .int commandStringEnd, 0  This code brings everything together into a simple command line operating system. The commands available are echo, reset, ok and cls. echo copies any text after it back to the terminal, reset resets the operating system if things go wrong, ok has two functions: ok on turns the OK LED on, and ok off turns the OK LED off, and cls clears the terminal using TerminalClear.  Have a go with this code on the Raspberry Pi. If it doesn't work, please see our troubleshooting page.  When it works, congratulations you've completed a basic terminal Operating System, and have completed the input series. Unfortunately, this is as far as these tutorials go at the moment, but I hope to make more in the future. Please send feedback to [awc32@cam.ac.uk](mailto:awc32@cam.ac.uk).  You're now in position to start building some simple terminal Operating Systems. My code above builds up a table of available commands in commandTable. Each entry in the table is an int for the address of the string, and an int for the address of the code to run. The last entry has to be commandStringEnd, 0. Try implementing some of your own commands, using our existing functions, or making new ones. The parameters for the functions to run are r0 is the address of the command the user typed, and r1 is the length. You can use this to pass inputs to your commands. Maybe you could make a calculator program, perhaps a drawing program or a chess program. Whatever ideas you've got, give them a go! | 第十一课 输入02  Input02的课程是建立在Input01的基础上的。这个课程将建立一个简单的命令行界面，在这个界面上用户可以输入命令，然后计算机进行解释然后显示执行的结果。这里有个假设，你已经拥有了第十课的操作系统代码。  目录  1. 终端1  2. 显示文本  3. 打印行文本  4. 标准输入  5. 终端：机器的出现  1. 终端1  在计算机的早期年代里，通常有个巨大的建筑物来容纳一台体型巨大的计算机，而且通常有很多的终端来把命令输入到计算机中。计算机会挨个儿执行这些输入的命令。  几乎所有的操作系统都是以一个文本终端来作为生命的起始。典型的情况是黑色的屏幕上显示着白的文字。你可以用键盘来输入命令以便计算机执行。如果你输入错误了，它会告诉你为什么错了。几乎很少的情况下，它可以执行你输入的命令。这种方式有两个优点：它提供了一个很简单的健壮的计算机控制机制，它只需要一个键盘和一台监视器。几乎每个操作系统都是这么做的，系统管理人员也是基本都理解它们。  让我们精确地分析一下我们想要做什么：  1. 计算机开机，显示一些欢迎信息  2. 计算机显示它已经准备好了接收输入  3. 用户利用键盘输入一个命令和参数  4. 用户敲击回车按键以便提交命令  5. 如果命令可以接受，计算机将解释命令并执行它们  6. 如果命令成功地执行了，计算机将会显示信息以及发生了什么的信息  7. 返回到第二步  那样一个终端得其中一个定义是它们是输入和输出的统一。同样一块屏幕，既可以用作输入也可以用作输出。这意味着：建立一个基于字符的显示的一种抽象将是很有用的。在一个基于字符的显示中，最小的单元是字符，而不是一个像素。屏幕具有固定数量的字符，这些字符可以有一些颜色。我们可以存储字符和它们的颜色数值，以便在我们已经拥有的屏幕代码的基础上建立这么一个系统。然后使用函数DrawCharacter来把字符和它们的颜色属性推到屏幕上。一旦我们拥有了一个基于字符的显示系统，那么绘制文本就变成了一件绘制一行字符的事情。  在一个叫做“terminal.s”的新文件里，把下面的代码拷贝进去：  .section .data .align 4 terminalStart: .int terminalBuffer terminalStop: .int terminalBuffer terminalView: .int terminalBuffer terminalColour: .byte 0xf .align 8 terminalBuffer: .rept 128\*128 .byte 0x7f .byte 0x0 .endr terminalScreen: .rept 1024/8 \* 768/16 .byte 0x7f .byte 0x0 .endr  这段代码为我们的文本终端准备了需要的数据。我们有两个主要的存储仓库：terminalBuffer和terminalScreen。库terminalBuffer用来存储我们要显示的所有的文本。它可以存储128行文本（每行文本包含128个字符）。每个字符由一个ASCII码和一个颜色数值组成，所有的这些数值的默认值是0x7f（ASCII码的删除）和0（黑色背景上的黑色）。库terminalScreen用来存储当前显示在屏幕上的字符。这个库中可以存储128乘48个字符，和上面提到的具有相同的初始化值。你可以决定我们只需要terminalScreen库，而并不需要terminalBuffer库。但，存储缓存有两个主要的优点：  1. 我们可以很容易地那些字符是不同的，因而我们只需要绘制这些不同的。  2. 我们依据终端历史纪录来“回滚”，就是因为它们被存储了起来（当然它们是由限制的）。  你应该总是试着这么做：以最小的代价来完成工作。其中原因是：工作内容分为不经常改变的和经常改变的两个部分，对于前者，我们做最小的功，对于后者我们静观其变，尽可能地不做功。  在低功耗操作系统上，这些不同的技巧的确很常见。绘制屏幕是一个很慢的操作，所以，我们只绘制那些我们必须要绘制的东西。在这样的系统中，我们可以很随意地改变terminalBuffer库的内容，然后调用一个函数，该函数仅仅复制那些改变了的内容的位信息到屏幕上。这意味着我们不必绘制所有的字符。这么做的目的是，当我们要绘制包括很多行的文本时节约时间。  在.data段中的其他的数值列在这里：  terminalStart  写入库terminalBuffer中的第一个字符。  terminalStop  写入库terminalBuffer中的最后一个字符。  terminalView  当前在屏幕上的第一个字符。我们可以利用这个信息来滚动屏幕。  terminalColour  绘制新字符时用到的颜色数值。  回环缓冲器是数据结构的一种。这里我们仅仅是组织数据的一个想法，它并不是一件实实在在的硬件。它需要我们在软件中实现。  Diagram showing hellow world being inserted into a circular buffer of size 5. |