|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 10 Input01  Welcome to the Input lesson series. In this series, you will learn how to receive inputs to the Raspberry Pi using the keyboard. We will start with just revealing the input, and then move to a more traditional text prompt.  This first input lesson teaches some theory about drivers and linking, as well as about keyboards and ends up displaying text on the screen.   |  | | --- | | **Contents**   * [1 Getting Started](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#gs) * [2 USB](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#usb) * [3 Linking](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#linking) * [4 Keyboards](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#keyboards) * [5 The Nut Behind the Wheel](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#driver) * [6 Updates Available](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#ua) * [7 Look Up Tables](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#lut) * [8 Notepad OS](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html#notepad) |   1 Getting Started  It is expected that you have completed the OK series, and it would be helpful to have completed the Screen series. Many of the files from that series will be called, without comment. If you do not have these files, or prefer to use a correct implementation, download the template for this lesson from the [Downloads](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html) page. If you're using your own implementation, please remove everything after your call to SetGraphicsAddress.  2 USB  The USB standard was designed to make simple hardware in exchange for complex software.  As you are no doubt aware, the Raspberry Pi model B has two USB ports, commonly used for connecting a mouse and keyboard. This was a very good design decision, USB is a very generic connector, and many different kinds of device use it. It's simple to build new devices for, simple to write device drivers for, and is highly extensible thanks to USB hubs. Could it get any better? Well, no, in fact for an Operating Systems developer this is our worst nightmare. The USB standard is huge. I really mean it this time, it is over 700 pages, before you've even thought about connecting a device.  I spoke to a number of other hobbyist Operating Systems developers about this and they all say one thing: don't bother. "It will take too long to implement", "You won't be able to write a tutorial on it" and "It will be of little benefit". In many ways they are right, I'm not able to write a tutorial on the USB standard, as it would take weeks. I also can't teach how to write device drivers for all the different devices, so it is useless on its own. However, I can do the next best thing: Get a working USB driver, get a keyboard driver, and then teach how to use these in an Operating System. I set out searching for a free driver that would run in an operating system that doesn't even know what a file is yet, but I couldn't find one. They were all too high level. So, I attempted to write one. Everybody was right, this took weeks to do. However, I'm pleased to say I did get one that works with no external help from the Operating System, and can talk to a mouse and keyboard. It is by no means complete, efficient, or correct, but it does work. It has been written in C and the full source code can be found on the downloads page for those interested.  So, this tutorial won't be a lesson on the USB standard (at all). Instead we'll look at how to work with other people's code.  3 Linking  Linking allows us to make reusable code 'libraries' that anyone can use in their program.  Since we're about to incorporate external code into the Operating System, we need to talk about linking. Linking is a process which is applied to programs or Operating System to link in functions. What this means is that when a program is made, we don't necessarily code every function (almost certainly not in fact). Linking is what we do to make our program link to functions in other people's code. This has actually been going on all along in our Operating Systems, as the linker links together all of the different files, each of which is compiled separately.  Programs often just call libraries, which call other libraries and so on until eventually they call an Operating System library which we would write.  There are two types of linking: static and dynamic. Static linking is like what goes on when we make our Operating Systems. The linker finds all the addresses of the functions, and writes them into the code, before the program is finished. Dynamic linking is linking that occurs after the program is 'complete'. When it is loaded, the dynamic linker goes through the program and links any functions which are not in the program to libraries in the Operating System. This is one of the jobs our Operating System should eventually be capable of, but for now everything will be statically linked.  The USB driver I have written is suitable for static linking. This means I give you the compiled code for each of my files, and then the linker finds functions in your code which are not defined in your code, and links them to functions in my code. On the [Downloads](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html) page for this lesson is a makefile and my USB driver, which you will need to continue. Download them and replace the makefile in your code with this one, and also put the driver in the same folder as that makefile.  4 Keyboards  In order to get input into our Operating System, we need to understand at some level how keyboards actually work. Keyboards have two types of keys: Normal and Modifier keys. The normal keys are the letters, numbers, function keys, etc. They constitute almost every key on the keyboard. The modifiers are up to 8 special keys. These are left shift, right shift, left control, right control, left alt, right alt, left GUI and right GUI. The keyboard can detect any combination of the modifier keys being held, as well as up to 6 normal keys. Every time a key changes (i.e. is pushed or released), it reports this to the computer. Typically, keyboards also have three LEDs for Caps Lock, Num Lock and Scroll Lock, which are controlled by the computer, not the keyboard itself. Keyboards may have many more lights such as power, mute, etc.  In order to help standardise USB keyboards, a table of values was produced, such that every keyboard key ever is given a unique number, as well as every conceivable LED. The table below lists the first 126 of values.   | Table 4.1 USB Keyboard Keys | | | | | | | | | --- | --- | --- | --- | --- | --- | --- | --- | | **Number** | **Description** | **Number** | **Description** | **Number** | **Description** | **Number** | **Description** | | 4 | a and A | 5 | b and B | 6 | c and C | 7 | d and D | | 8 | e and E | 9 | f and F | 10 | g and G | 11 | h and H | | 12 | i and I | 13 | j and J | 14 | k and K | 15 | l and L | | 16 | m and M | 17 | n and N | 18 | o and O | 19 | p and P | | 20 | q and Q | 21 | r and R | 22 | s and S | 23 | t and T | | 24 | u and U | 25 | v and V | 26 | w and W | 27 | x and X | | 28 | y and Y | 29 | z and Z | 30 | 1 and ! | 31 | 2 and @ | | 32 | 3 and # | 33 | 4 and $ | 34 | 5 and % | 35 | 6 and ^ | | 36 | 7 and & | 37 | 8 and \* | 38 | 9 and ( | 39 | 0 and ) | | 40 | Return (Enter) | 41 | Escape | 42 | Delete (Backspace) | 43 | Tab | | 44 | Spacebar | 45 | - and \_ | 46 | = and + | 47 | [ and { | | 48 | ] and } | 49 | \ and | | 50 | # and ~ | 51 | ; and : | | 52 | ' and " | 53 | ` and ~ | 54 | , and < | 55 | . and > | | 56 | / and ? | 57 | Caps Lock | 58 | F1 | 59 | F2 | | 60 | F3 | 61 | F4 | 62 | F5 | 63 | F6 | | 64 | F7 | 65 | F8 | 66 | F9 | 67 | F10 | | 68 | F11 | 69 | F12 | 70 | Print Screen | 71 | Scroll Lock | | 72 | Pause | 73 | Insert | 74 | Home | 75 | Page Up | | 76 | Delete forward | 77 | End | 78 | Page Down | 79 | Right Arrow | | 80 | Left Arrow | 81 | Down Arrow | 82 | Up Arrow | 83 | Num Lock | | 84 | Keypad / | 85 | Keypad \* | 86 | Keypad - | 87 | Keypad + | | 88 | Keypad Enter | 89 | Keypad 1 and End | 90 | Keypad 2 and Down Arrow | 91 | Keypad 3 and Page Down | | 92 | Keypad 4 and Left Arrow | 93 | Keypad 5 | 94 | Keypad 6 and Right Arrow | 95 | Keypad 7 and Home | | 96 | Keypad 8 and Up Arrow | 97 | Keypad 9 and Page Up | 98 | Keypad 0 and Insert | 99 | Keypad . and Delete | | 100 | \ and | | 101 | Application | 102 | Power | 103 | Keypad = | | 104 | F13 | 105 | F14 | 106 | F15 | 107 | F16 | | 108 | F17 | 109 | F18 | 110 | F19 | 111 | F20 | | 112 | F21 | 113 | F22 | 114 | F23 | 115 | F24 | | 116 | Execute | 117 | Help | 118 | Menu | 119 | Select | | 120 | Stop | 121 | Again | 122 | Undo | 123 | Cut | | 124 | Copy | 125 | Paste | 126 | Find | 127 | Mute | | 128 | Volume Up | 129 | Volume Down |  |  |  |  |   The full list can be found in section 10, page 53 of [HID Usage Tables 1.12](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads/hut1_12v2.pdf).  5 The Nut Behind the Wheel  These summaries and the code they describe form an API - Application Product Interface.  Normally, when you work with someone else's code, they provide a summary of their methods, what they do and roughly how they work, as well as how they can go wrong. Here is a table of the relevant instructions required to use my USB driver.   | Table 5.1 Keyboard related functions in CSUD | | | | | --- | --- | --- | --- | | **Function** | **Arguments** | **Returns** | **Description** | | UsbInitialise | None | r0 is result code | This method is the all-in-one method that loads the USB driver, enumerates all devices and attempts to communicate with them. This method generally takes about a second to execute, though with a few USB hubs plugged in this can be significantly longer. After this method is completed methods in the keyboard driver become available, regardless of whether or not a keyboard is indeed inserted. Result code explained below. | | UsbCheckForChange | None | None | Essentially provides the same effect as UsbInitialise, but does not provide the same one time initialisation. This method checks every port on every connected hub recursively, and adds new devices if they have been added. This should be very quick if there are no changes, but can take up to a few seconds if a hub with many devices is attached. | | KeyboardCount | None | r0 is count | Returns the number of keyboards currently connected and detected. UsbCheckForChange may update this. Up to 4 keyboards are supported by default. Up to this many keyboards may be accessed through this driver. | | KeyboardGetAddress | r0 is index | r0 is address | Retrieves the address of a given keyboard. All other functions take a keyboard address in order to know which keyboard to access. Thus, to communicate with a keyboard, first check the count, then retrieve the address, then use other methods. Note, the order of keyboards that this method returns may change after calls to UsbCheckForChange. | | KeyboardPoll | r0 is address | r0 is result code | Reads in the current key state from the keyboard. This operates via polling the device directly, contrary to the best practice. This means that if this method is not called frequently enough, a key press could be missed. All reading methods simply return the value as of the last poll. | | KeyboardGetModifiers | r0 is address | r0 is modifier state | Retrieves the status of the modifier keys as of the last poll. These are the shift, alt control and GUI keys on both sides. This is returned as a bit field, such that a 1 in the bit 0 means left control is held, bit 1 means left shift, bit 2 means left alt, bit 3 means left GUI and bits 4 to 7 mean the right versions of those previous. If there is a problem r0 contains 0. | | KeyboardGetKeyDownCount | r0 is address | r0 is count | Retrieves the number of keys currently held down on the keyboard. This excludes modifier keys. Normally, this cannot go above 6. If there is an error this method returns 0. | | KeyboardGetKeyDown | r0 is address, r1 is key number | r0 is scan code | Retrieves the scan code (see Table 4.1) of a particular held down key. Normally, to work out which keys are down, call KeyboardGetKeyDownCount and then call KeyboardGetKeyDown up to that many times with increasing values of r1 to determine which keys are down. Returns 0 if there is a problem. It is safe (but not recommended) to call this method without calling KeyboardGetKeyDownCount and interpret 0s as keys not held. Note, the order or scan codes can change randomly (some keyboards sort numerically, some sort temporally, no guarantees are made). | | KeyboardGetKeyIsDown | r0 is address, r1 is scan code | r0 is status | Alternative to KeyboardGetKeyDown, checks if a particular scan code is among the held down keys. Returns 0 if not, or a non-zero value if so. Faster when detecting particular scan codes (e.g. looking for ctrl+c). On error, returns 0. | | KeyboardGetLedSupport | r0 is address | r0 is LEDs | Checks which LEDs a particular keyboard supports. Bit 0 being 1 represents Number Lock, bit 1 represents Caps Lock, bit 2 represents Scroll Lock, bit 3 represents Compose, bit 4 represents Kana, bit 5 represents Power, bit 6 represents Mute and bit 7 represents Compose. As per the USB standard, none of these LEDs update automatically (e.g. Caps Lock must be set manually when the Caps Lock scan code is detected). | | KeyboardSetLeds | r0 is address, r1 is LEDs | r0 is result code | Attempts to turn on/off the specified LEDs on the keyboard. See below for result code values. See KeyboardGetLedSupport for LEDs' values. |   Result codes are an easy way to handle errors, but often more elegant solutions exist in higher level code.  Several methods return 'result codes'. These are commonplace in C code, and are just numbers which represent what happened in a method call. By convention, 0 always indicates success. The following result codes are used by this driver.   | Table 5.2 - CSUD Result Codes | | | --- | --- | | **Code** | **Description** | | 0 | Method completed successfully. | | -2 | Argument: A method was called with an invalid argument. | | -4 | Device: The device did not respond correctly to the request. | | -5 | Incompatible: The driver is not compatible with this request or device. | | -6 | Compiler: The driver was compiled incorrectly, and is broken. | | -7 | Memory: The driver ran out of memory. | | -8 | Timeout: The device did not respond in the expected time. | | -9 | Disconnect: The device requested has disconnected, and cannot be used. |   The general usage of the driver is as follows:   1. Call UsbInitialise 2. Call UsbCheckForChange 3. Call KeyboardCount 4. If this is 0, go to 2. 5. For each keyboard you support:    1. Call KeyboardGetAddress    2. Call KeybordGetKeyDownCount    3. For each key down:       1. Check whether or not it has just been pushed       2. Store that the key is down    4. For each key stored:       1. Check whether or not key is released       2. Remove key if released 6. Perform actions based on keys pushed/released 7. Go to 2.   Ultimately, you may do whatever you wish to with the keyboard, and these methods should allow you to access all of its functionality. Over the next 2 lessons, we shall look at completing the input side of a text terminal, similarly to most command line computers, and interpreting the commands. In order to do this, we're going to need to have keyboard inputs in a more useful form. You may notice that my driver is (deliberately) unhelpful, because it doesn't have methods to deduce whether or not a key has just been pushed down or released, it only has methods about what is currently held down. This means we'll need to write such methods ourselves.  6 Updates Available  Repeatedly checking for updates is called 'polling'. This is in contrast to interrupt driven IO, where the device sends a signal when data is ready.  First of all, let's implement a method KeyboardUpdate which detects the first keyboard and uses its poll method to get the current input, as well as saving the last inputs for comparison. We can then use this data with other methods to translate scan codes to keys. The method should do precisely the following:   1. Retrieve a stored keyboard address (initially 0). 2. If this is not 0, go to 9. 3. Call UsbCheckForChange to detect new keyboards. 4. Call KeyboardCount to detect how many keyboards are present. 5. If this is 0 store the address as 0 and return; we can't do anything with no keyboard. 6. Call KeyboardGetAddress with parameter 0 to get the first keyboard's address. 7. Store this address. 8. If this is 0, return; there is some problem. 9. Call KeyboardGetKeyDown 6 times to get each key currently down and store them 10. Call KeyboardPoll 11. If the result is non-zero go to 3. There is some problem (such as disconnected keyboard).   To store the values mentioned above, we will need the following values in the .data section.  .section .data .align 2 KeyboardAddress: .int 0 KeyboardOldDown: .rept 6 .hword 0 .endr  **.hword num** inserts the half word constant **num** into the file directly.  **.rept num [commands] .endr** copies the commands **commands** to the output **num** times.  Try to implement the method yourself. My implementation for this is as follows:   1. .section .text .globl KeyboardUpdate KeyboardUpdate: push {r4,r5,lr}  kbd .req r4 ldr r0,=KeyboardAddress ldr kbd,[r0]   We load in the keyboard address.   1. teq kbd,#0 bne haveKeyboard$   If the address is non-zero, we have a keyboard. Calling UsbCheckForChanges is slow, and so if everything works we avoid it.   1. getKeyboard$: bl UsbCheckForChange   If we don't have a keyboard, we have to check for new devices.   1. bl KeyboardCount   Now we see if a new keyboard has been added.   1. teq r0,#0 ldreq r1,=KeyboardAddress streq r0,[r1] beq return$   There are no keyboards, so we have no keyboard address.   1. mov r0,#0 bl KeyboardGetAddress   Let's just get the address of the first keyboard. You may want to allow more.   1. ldr r1,=KeyboardAddress str r0,[r1]   Store the keyboard's address.   1. teq r0,#0 beq return$ mov kbd,r0   If we have no address, there is nothing more to do.   1. saveKeys$:   mov r0,kbd mov r1,r5 bl KeyboardGetKeyDown  ldr r1,=KeyboardOldDown add r1,r5,lsl #1 strh r0,[r1] add r5,#1 cmp r5,#6 blt saveKeys$  Loop through all the keys, storing them in KeyboardOldDown. If we ask for too many, this returns 0 which is fine.   1. mov r0,kbd bl KeyboardPoll   Now we get the new keys.   1. teq r0,#0 bne getKeyboard$  return$: pop {r4,r5,pc} .unreq kbd   Finally we check if KeyboardPoll worked. If not, we probably disconnected.  With our new KeyboardUpdate method, checking for inputs becomes as simple as calling this method at regular intervals, and it will even check for disconnections etc. This is a useful method to have, as our actual key processing may differ based on the situation, and so being able to get the current input in its raw form with one method call is generally applicable. The next method we ideally want is KeyboardGetChar, a method that simply returns the next key pressed as an ASCII character, or returns 0 if no key has just been pressed. This could be extended to support typing a key multiple times if it is held for a certain duration, and to support the 'lock' keys as well as modifiers.  To make this method it is useful if we have a method KeyWasDown, which simply returns 0 if a given scan code is not in the KeyboardOldDown values, and returns a non-zero value otherwise. Have a go at implementing this yourself. As always, a solution can be found on the downloads page.  7 Look Up Tables  In many areas of programming, the larger the program, the faster it is. Look up tables are large, but are very fast. Some problems can be solved by a mixture of look up tables and normal functions.  The KeyboardGetChar method could be quite complex if we write it poorly. There are 100s of scan codes, each with different effects depending on the presence or absence of the shift key or other modifiers. Not all of the keys can be translated to a character. For some characters, multiple keys can produce the same character. A useful trick in situations with such vast arrays of possibilities is look up tables. A look up table, much like in the physical sense, is a table of values and their results. For some limited functions, the simplest way to deduce the answer is just to precompute every answer, and just return the correct one by retrieving it. In this case, we could build up a sequence of values in memory such that the nth value into the sequence is the ASCII character code for the scan code n. This means our method would simply have to detect if a key was pressed, and then retrieve its value from the table. Further, we could have a separate table for the values when shift is held, so that the shift key simply changes which table we're working with.  After the **.section .data** command, copy the following tables:  .align 3 KeysNormal:  .byte 0x0, 0x0, 0x0, 0x0, 'a', 'b', 'c', 'd' .byte 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l' .byte 'm', 'n', 'o', 'p', 'q', 'r', 's', 't' .byte 'u', 'v', 'w', 'x', 'y', 'z', '1', '2' .byte '3', '4', '5', '6', '7', '8', '9', '0' .byte '\n', 0x0, '\b', '\t', ' ', '-', '=', '[' .byte ']', '\\', '#', ';', '\'', '`', ',', '.' .byte '/', 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, '/', '\*', '-', '+' .byte '\n', '1', '2', '3', '4', '5', '6', '7' .byte '8', '9', '0', '.', '\\', 0x0, 0x0, '='  .align 3 KeysShift:  .byte 0x0, 0x0, 0x0, 0x0, 'A', 'B', 'C', 'D' .byte 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L' .byte 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T' .byte 'U', 'V', 'W', 'X', 'Y', 'Z', '!', '"' .byte '£', '$', '%', '^', '&', '\*', '(', ')' .byte '\n', 0x0, '\b', '\t', ' ', '\_', '+', '{' .byte '}', '|', '~', ':', '@', '¬', '<', '>' .byte '?', 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 .byte 0x0, 0x0, 0x0, 0x0, '/', '\*', '-', '+' .byte '\n', '1', '2', '3', '4', '5', '6', '7' .byte '8', '9', '0', '.', '|', 0x0, 0x0, '='  **.byte num** inserts the byte constant **num** into the file directly.  Most assemblers and compilers recognise escape sequences; character sequences such as \t which insert special characters instead.  These tables map directly the first 104 scan codes onto the ASCII characters as a table of bytes. We also have a separate table describing the effects of the shift key on those scan codes. I've used the ASCII null character (0) for all keys without direct mappings in ASCII (such as the function keys). Backspace is mapped to the ASCII backspace character (8 denoted \b), enter is mapped to the ASCII new line character (10 denoted \n) and tab is mapped to the ASCII horizontal tab character (9 denoted \t).  The KeyboardGetChar method will need to do the following:   1. Check if KeyboardAddress is 0. If so, return 0. 2. Call KeyboardGetKeyDown up to 6 times. Each time:    1. If key is 0, exit loop.    2. Call KeyWasDown. If it was, go to the next key.    3. If the scan code is more than 103, go to the next key.    4. Call KeyboardGetModifiers    5. If shift is held, load the address of KeysShift. Otherwise load KeysNormal.    6. Read the ASCII value from the table.    7. If it is 0, go to the next key otherwise return this ASCII code and exit. 3. Return 0.   Try to implement this yourself. My implementation is presented below:   1. .globl KeyboardGetChar KeyboardGetChar: ldr r0,=KeyboardAddress ldr r1,[r0] teq r1,#0 moveq r0,#0 moveq pc,lr   Simple check to see if we have a keyboard.   1. push {r4,r5,r6,lr} kbd .req r4 key .req r6 mov r4,r1 mov r5,#0 keyLoop$:   mov r0,kbd mov r1,r5 bl KeyboardGetKeyDown  r5 will hold the index of the key, r4 holds the keyboard address.   * 1. teq r0,#0 beq keyLoopBreak$   If a scan code is 0, it either means there is an error, or there are no more keys.   * 1. mov key,r0 bl KeyWasDown teq r0,#0 bne keyLoopContinue$   If a key was already down it is uninteresting, we only want ot know about key presses.   * 1. cmp key,#104 bge keyLoopContinue$   If a key has a scan code higher than 104, it will be outside our table, and so is not relevant.   * 1. mov r0,kbd bl KeyboardGetModifiers   We need to know about the modifier keys in order to deduce the character.   * 1. tst r0,#0b00100010 ldreq r0,=KeysNormal ldrne r0,=KeysShift   We detect both a left and right shift key as changing the characters to their shift variants. Remember, a **tst** instruction computes the logical AND and then compares it to zero, so it will be equal to 0 if and only if both of the shift bits are zero.   * 1. ldrb r0,[r0,key]   Now we can load in the key from the look up table.   * 1. teq r0,#0 bne keyboardGetCharReturn$ keyLoopContinue$: add r5,#1 cmp r5,#6 blt keyLoop$   If the look up code contains a zero, we must continue. To continue, we increment the index, and check if we've reached 6.   1. keyLoopBreak$: mov r0,#0 keyboardGetCharReturn$: pop {r4,r5,r6,pc} .unreq kbd .unreq key   We return our key here, if we reach keyLoopBreak$, then we know there is no key held, so return 0.  8 Notepad OS  Now we have our KeyboardGetChar method, we can make an operating system that just types what the user writes to the screen. For simplicity we'll ignore all the unusual keys. In 'main.s' delete all code after **bl SetGraphicsAddress**. Call UsbInitialise, set r4 and r5 to 0, then loop forever over the following commands:   1. Call KeyboardUpdate 2. Call KeyboardGetChar 3. If it is 0, got to 1 4. Copy r4 and r5 to r1 and r2 then call DrawCharacter 5. Add r0 to r4 6. If r4 is 1024, add r1 to r5 and set r4 to 0 7. If r5 is 768 set r5 to 0 8. Go to 1   Now compile this and test it on the Pi. You should almost immediately be able to start typing text to the screen when the Pi starts. If not, please see our troubleshooting page.  When it works, congratulations, you've achieved an interface with the computer. You should now begin to realise that you've almost got a primitive operating system together. You can now interface with the computer, issuing it commands, and receive feedback on screen. In the next tutorial, [Input02](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input02.html) we will look at producing a full text terminal, in which the user types commands, and the computer executes them. |  |