The Keyboard

A keyboard is the most common way for a user to give a computer input, therefore it is vital that you create a driver of some sort for handling and managing the keyboard. When you get down to it, getting the basics of the keyboard isn't too bad. Here we will show the basics: how to get a key when it is pressed, and how to convert what's called a 'scancode' to standard ASCII characters that we can understand properly.

A scancode is simply a key number. The keyboard assigns a number to each key on the keyboard; this is your scancode. The scancodes are numbered generally from top to bottom and left to right, with some minor exceptions to keep layouts backwards compatible with older keyboards. You must use a lookup table (an array of values) and use the scancode as the index into this table. The lookup table is called a keymap, and will be used to translate scancodes into ASCII values rather quickly and painlessly. One last note about a scancode before we head into code is that if bit 7 is set (test with 'scancode & 0x80'), then this is the keyboard's way of telling us that a key was just released. Create yourself a 'kb.h' and do all your standard proceedures like adding a line for GCC and adding a file to LD's command line.

```
/* KBDUS means US Keyboard Layout. This is a scancode table
*  used to layout a standard US keyboard. I have left some
*  comments in to give you an idea of what key is what, even
*  though I set it's array index to 0. You can change that to
*  whatever you want using a macro, if you wish! */
unsigned char kbdus[128] =
{
    0,  27, '1', '2', '3', '4', '5', '6', '7', '8',         /* 9 */
  '9', '0', '-', '=', '\b',   /* Backspace */
  '\t',                       /* Tab */
  'q', 'w', 'e', 'r',         /* 19 */
  't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',   /* Enter key */
    0,                        /* 29   - Control */
  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',         /* 39 */
'\'', '`',   0,               /* Left shift */
'\\', 'z', 'x', 'c', 'v', 'b', 'n',                         /* 49 */
  'm', ',', '.', '/',   0,                                  /* Right shift */
  '*',
    0,    /* Alt */
  ' ',    /* Space bar */
    0,    /* Caps lock */
    0,    /* 59 - F1 key ... > */
    0,   0,   0,   0,   0,   0,   0,   0,
    0,    /* < ... F10 */
    0,    /* 69 - Num lock*/
    0,    /* Scroll Lock */
    0,    /* Home key */
    0,    /* Up Arrow */
    0,    /* Page Up */
  '-',
    0,    /* Left Arrow */
    0,
    0,    /* Right Arrow */
  '+',
    0,    /* 79 - End key*/
    0,    /* Down Arrow */
    0,    /* Page Down */
    0,    /* Insert Key */
    0,    /* Delete Key */
    0,   0,   0,
    0,    /* F11 Key */
    0,    /* F12 Key */
```

```
    0,    /* All other keys are undefined */
};
```

Sample keymap. Add this array to your 'kb.c'

Converting a scancode to an ASCII value is easy with this:

mychar = kbdus[scancode];

Note that although we leave comments for the function keys and shift/control/alt, we leave them as 0's in the array: You need to think up some random values such as ASCII values that you normally wouldn't use so that you can trap them. I'll leave this up to you, but you should keep a global variable to be used as a key status variable. This keystatus variable will have 1 bit set for ALT, one for CONTROL, and one for SHIFT. It's also a good idea to have one for CAPSLOCK, NUMLOCK, and SCROLLLOCK. This tutorial will explain how to set the keyboard lights, but we will leave it up to you to actually write the code for it.

The keyboard is attached to the computer through a special microcontroller chip on your mainboard. This keyboard controller chip has 2 channels: one for the keyboard, and one for the mouse. Also note that it is through this keyboard controller chip that you would enable the A20 address line on the processor to allow you to access memory past the 1MByte mark (GRUB enables this, you don't need to worry about it). The keyboard controller, being a device accessible by the system, has an address on the I/O bus that we can use for access and control. The keyboard controller has 2 main registers: a Data register at 0x60, and a Control register at 0x64. Anything that the keyboard wants to send the computer is stored into the Data register. The keyboard will raise IRQ1 whenever it has data for us to read. Observe:

```
/* Handles the keyboard interrupt */
void keyboard_handler(struct regs *r)
{
    unsigned char scancode;

    /* Read from the keyboard's data buffer */
    scancode = inportb(0x60);

    /* If the top bit of the byte we read from the keyboard is
    *  set, that means that a key has just been released */
    if (scancode & 0x80)
    {
        /* You can use this one to see if the user released the
        *  shift, alt, or control keys... */
    }
    else
    {
        /* Here, a key was just pressed. Please note that if you
        *  hold a key down, you will get repeated key press
        *  interrupts. */

        /* Just to show you how this works, we simply translate
        *  the keyboard scancode into an ASCII value, and then
        *  display it to the screen. You can get creative and
        *  use some flags to see if a shift is pressed and use a
        *  different layout, or you can add another 128 entries
        *  to the above layout to correspond to 'shift' being
        *  held. If shift is held using the larger lookup table,
        *  you would add 128 to the scancode when you look for it */
        putch(kbdus[scancode]);
    }
```

```
}
```

This might look intimidating, but it's 80% comments ;) Add to 'kb.c'

As you can see, the keyboard will generate an IRQ1 telling us that it has data ready for us to grab. The keyboard's data register exists at 0x60. When the IRQ happens, we call this handler which reads from port 0x60. This data that we read is the keyboard's scancode. For this example, we check if the key was pressed or released. If it was just pressed, we translate the scancode to ASCII, and print that character out with one line. Write a 'keyboard_install' function that calls 'irq_install_handler' to install the custom keyboard handler for 'keyboard_handler' to IRQ1. Be sure to make a call to 'keyboard_install' from inside 'main'.

In order to set the lights on your keyboard, you must send the keyboard controller a command. There is a specific proceedure for sending the keyboard a command. You must first wait for the keyboard controller to let you know when it's not busy. To do this, you read from the Control register (When you read from it, it's called a Status register) in a loop, breaking out when the keyboard isn't busy:

if ((inportb(0x64) & 2) == 0) break;

After that loop, you may write the command byte to the Data register. You don't write to the control register itself except for in special cases. To set the lights on the keyboard, you first send the command byte 0xED using the described method, then you send the byte that says which lights are to be on or off. This byte has the following format: Bit0 is Scroll lock, Bit1 is Num lock, and Bit2 is Caps lock.

Now that you have basic keyboard support, you may wish to expand upon the code. This section on the keyboard was more to show you how to do the basics rather than give an extremely detailed overview of all of the keyboard controller's functions. Note that you use the keyboard controller to enable and handle the PS/2 mouse port. The auxilliary channel on the keyboard controller manages the PS/2 mouse. Up to this point we have a kernel that can draw to the screen, handle exceptions, handle IRQs, handle the timer, and handle the keyboard. Click to find what's next in store for your kernel development.