CMSC 412

Project 1: Keyboard and Screen Drivers

Due: February 11, 1998 (in recitation)

Introduction

Almost all computers need to operate with external devices. At the very least, you need to use the keyboard, the screen, and the disk drive. Device drivers are code that operate these devices and provide a layer of abstraction between the operating system and the device. That way, as long as the device driver provides certain functions that are well-defined, the operating system does not have to know how the device driver is implement.

In this project, you will be asked to write a screen and keyboard driver. For each of the drivers, you will be asked to write functions that provide the interface for their use. We will discuss the details of the project momentarily.

Cheating

This project is to be done individually. You are not to work in teams. You are not allowed to copy code nor to give code to other students in the class. This includes handing hard copies, or transmitting the code in any manner (e-mail, FTP, copying disks, etc.). You are not allowed to make modifications to code that you have received from anyone to disguise the code. All work should be your own. The current instructor and current TA's will provide some code to you which you may be permitted to use.

You are permitted to discuss general strategies as well as certain technical information (e.g., what the interrupts do, and so on). However, the coding should be done by yourself. You may be asked to explain the code you have written at any time. Those who cheat may be brought to the honor council.

Please refer to the Web page concerning issues of Academic Dishonesty that applies to all materials (projects, tests, etc.) covered in CMSC 412.

Preliminaries

Some Conventions

All functions that you write and is provided to will have its first letter capitalized. This will make it easier to distinguish the functions you write (or have been written for you), and the built-in functions (i.e., those that are in standard C libraries) which start with a lowercase letter.

Please make sure that the code provided to you conforms to this convention. We will do our best to make sure sample code adheres this convention, but their may be some instances where an oversight has occurred. Our apologies in advance.

Code Provided To You

You will be given the following files. You should write your code in C, not in C++.

scancode

This is a list of the scancodes. This should be placed in keyboard.c.

main.c This is the main function which is described in more detail below.

cmsc412.c We will also give you `cmsc412.h`. These two files will give you our versions of `malloc()`. This is needed because the normal `malloc()` is implemented using interrupts, and you will need a version which works with interrupts disabled. For this reason, you should use C and not C++ (since `new` is a form of `malloc`).You will not need this file if you are going to use arrays to implement your queues.

codes.h This defines a few useful constants including some scan codes.

makefile. If you create the files as suggested below, the makefile should work. You MUST submit a makefile with your code on this project and all future projects.

Overview

The goal of this project is to record the characters typed in at the keyboard and print them to a screen. The characters will be stored in a queue. In main(), you will write a loop that will keep checking if there are characters in the queue, and, if so, you will print them to the screen. If the Escape character is the next character to be printed, you will exit the loop and quit. This will involve restoring DOS back to its normal state (i.e., restoring interrupts).

The Main Program

The goal of the main program is to read the characters from the keyboard buffer and place it on the screen. The code can be found at the class Web site.

We explain some of the functions that will be useful to you. Some of these were found in project 0. `getvect()` is a built-in function which will return the address (which is a pointer) to an interrupt vector (the segment and offset). For example, in the code provided, the interrupt vector for interrupt 9h is saved in the variable, `old9h`. The function, `setvect`(), takes two arguments. The first argument is an integer which is an interrupt number. The second argument is the name of a function which will serve as the interrupt handler.

`setvect` is used to *hook* the interrupt handler, i.e., it places the address of the interrupt handler in the interrupt vector table entry for interrupt 9h. In C, the name of the function is equivalent to the address of the first instruction of that function. `geninterrupt`() takes a single argument which is the number of the interrupt to generate. This is a software interrupt (of course!). Each interrupt can be referenced by its interrupt number.

`ESC` is "defined" in `codes.h`. The definition gives the ASCII number for the escape key.

The Keyboard Driver

The keyboard driver will consist of two sets of files that you must write.

- `queue.c/queue.h`
- In this file, you will create the functions for manipulating a queue. The primary functions in this file are `Enqueue()` and `Dequeue()`, but there are other functions as well.
- `keyboard.c/keyboard.h` This file will include the interrupt handler for the keyboard as well as a function to get letters from the queue.

Implementing a Keyboard Buffer

As you type on the keyboard, the characters will be saved in a queue. Hence, you will need to write code for handling a queue. You should write the following functions and put it in queue.c. The queue should have a structure that holds a pointer to the front and rear of a queue. You may also want to have a field such as integer which will be a code for the kind of objects being used in a queue. For example, you might, for this project, set the integer to 0, and this would be a code to indicate that the queue contains characters. For the next project, you might use the integer 1 to indicate the queue holds PCBs (process control blocks). The goal is to write a general purpose queue that can be used to contain many types of objects, so that you can reuse the code.

The elements of the queue should be a structure that contains a next pointer (to the next element in the queue or to NULL) and also a `void *` pointer. The `void *` pointer will be used to point to the elements of a queue. This will allow you to cast objects to void *. For example, suppose you wish to enqueue a character. If *x* is a variable of character type, then `Enqueue( (void *) &x )` will pass a pointer to the character, which has been cast to a `void *`.

You may wish to define the following functions within queue.c. `queuetype *Q_Create(), int Q_Destroy(queuetype *q), int Q_Empty(queuetype q), int Enqueue(queuetype *q, void *item), int Dequeue(queuetype *q, void *item`). The functions for enqueueing and dequeueing should be straightforward. You need to call `Q_Create()` to create a queue which will return a pointer to a structure that contains a pointer to the front and back of a queue, and possibly an integer, as described earlier. Feel free to make changes as you see fit. The above is merely a suggestion. Recall that you want to pass pointers to the objects (called `item` in the parameter list), and cast the pointer as a void *.

The function `Q_Empty` can be used to check if a queue is empty or not. You can return 1 for success and 0 for failure. You may wish to use #define to define constants that stand for 0 and 1.

You will use the malloc routines (use `Safe_malloc()`) from `cmsc412.c` and `cmsc412.h` to create the space for this structure. If `Safe_malloc()` returns 0, then return NULL for `Q_Create()`. You should always test to make sure if Safe_malloc() returns NULL or not, and deal with this case accordingly. You will not usually get NULL, but you should deal with it, in any case.

Since both the keyboard driver and the main program will be accessing and modifying the queue (the keyboard driver will add elements to the queue while main will remove elements), Q_Dequeue() and Q_Enqueue() must run atomically. This means that it should run to completion without being interrupted. To carry this out, you will disable interrupts upon entering these routines, and enable interrupts before returning back. The following code and define's should help you to disable and re-enable interrupts.

```
//Disables interrupts.

asm CLI; // asm is inline assembly
// Restores interrupts.

asm STI;
```

If you fail to re-enable interrupts, you may cause the machine to freeze. To remedy this may require rebooting, although you should attempt to close the window, and reopen a DOS box.

Writing the Keyboard Interrupt Handler

Every time you press *and* release a key, you generate an interrupt. Specifically, interrupt 9h is generated. When an interrupt is generated, an interrupt handler (which is basically a function) is called. You will write a function called `Key_handler()`.

This function will take no arguments. The purpose of the keyboard handler is to read the character that has been typed in from the keyboard and place it in the queue pointed to by `keyboard_buff_ptr`. This variable is globally declared in `main.c`. We will provide you with a sample, `keyboard.c`, which you will need to modify.

If you press a key, interrupt 9h will be called. If you release a key, interrupt 9h will also be called. Hence, pressing and releasing a key causes two interrupts to be called. When the key is held down for a period of time, the interrupts will be called periodically. When an interrupt is called, a scancode is generated. A scancode is an integer between 0and 255. A key that is just pressed will generate a scancode whose value lies between 0 and 127. If a key is released, the scancode generated will like between 128 and 255, inclusive. The scan The scancode is NOT the same as the ASCII code. Hence, you will have a lookup table to converts scancodes to ASCII. This will be provided to you in a file called `scancode`, but you should include this file in `keyboard.c`.

You can read the scancode at port 60h. A port is merely an address used for I/O. The Intel 8086 allows for normal addresses as well as port addresses. To read the value from the port, call the function, `inp`, which takes a single argument which is the address of the port to be read. It returns the value read at that address. You will then need to reset port 61h (see sample code provided) so that port 60h will be ready to read the next character. If you do not reset the port, no new characters will be read in.

You need to be careful about shift keys. When a shift key is pressed, this will NOT be entered in the queue. Instead, the keys that are typed from that point on are modified. In this case, you will enter the ASCII characters as capital letters. When you press the caps lock, you will need to modify the behavior of the keys pressed afterwards. The modifications can start as soon as the caps lock key is depressed. You do not, and should not, wait until the key is released. For combinations of caps lock and shift, you should emulate the behavior of DOS keyboards. The effect of the caps lock being active, and the shift being pressed will generate lower case letters.

Certain scan codes are not in the array provided in `scancode`. The rest of the scancodes are defined in `codes.h` which is also provided to you.

Most of the times, you will convert the scan code into an ASCII character, and enqueue that character. Note that an uppercase letter and a lower case letter produce the *same* scancode. The only way you can tell if a letter is uppercase is by keeping track of the "state" of the caps lock key as well as the shift key. I.e., you need to know if the caps lock is active, and you need to know if the shift key is depressed. You should use variables to keep track of this information.

You will not want to enter all keys into the queue. Specifically, the shift keys and the caps lock key will not be entered into the queue (since they can not be printed). The shift and caps lock keys modify the keys that are pressed afterwards. For example, suppose you press the shift key. Then, you press the character 'b'. When 'b' is pressed, there is a scancode. This scancode is the same for regardless of whether the shift key has been pressed or not. I.e., the scancode is only related to they key being pressed, not to whether it is capitalized or not. This is why you need to keep track in some variable about whether the shift key is pressed and whether the caps lock is active. Based on that information and the scancode, you can decide whether you should enqueue a lower or uppercase character. For the case of numbers, you should rely on the DOS keyboard conventions. For example, Caps Lock will not cause the pressing of the number 8 to produce *.

Implementing Get_char

In `keyboard.c`, you will also want to implement a function called `Get_char`. This function will take a character from the queue, and return that character. The prototype for that function should be char `Get_char()`. Return NULL if there are no characters in the queue. Otherwise, return the character.

The Screen Driver

Unlike the keyboard driver, the screen driver is not interrupt driven. You will write a C function called `Put_char` which will emulate the standard C function, `putch`. The screen driver will write characters to the screen by writing to the video RAM. You must handle scrolling when you are at the end of the screen, as well as backspacing, and tabbing. In addition, you will have to control the cursor. The normal DOS-controlled cursor will have to be disabled for you to do this.

Technical Description

The routine `Put_char` will place characters on the screen by writing the ASCII code of the character directly into the screen controller's video ram. The video ram starts at B000:0000 for monochrome monitors and B800:0000 for color monitors. At power up, memory location 410h is set with a code to determine which type of monitor is active. The following code will read the current configuration, set the appropriate pointer to the base of video RAM, initialize the screen, and turn off the hardware cursor. You should place this code in the file `screen.c`. A skeleton of this file will be provided via the Web site.

To restore the hardware cursor, load AX with 0x0100 and CX with 0x0D0E, and generate interrupt 10h. For clarity, use `#define` instead of hard-coded magic numbers in your code. See `main.c`.

Each screen character is held in two bytes of the video memory. The first byte contains the ASCII code of the character, and the second byte contains the color/attribute. Bits 0-2 select the color of the character (bit 0=blue, bit 1=green, bit 2=red);bit 3, when set, is the bright bit; bits 4-6 are the background color bits and bit 7 is the character blink bit. The character words are arranged in rows. Each row takes 160 bytes (80 characters times two bytes per character). Color attribute 7 should give you the best results for normal characters. You can use attribute 15 for highlight and 112 for reverse video if you want. The following code will put character c at column x and row y with color 7:

```
charbase [(x * 2) + (y * 160)] = c; // Character

charbase [(x * 2) + (y * 160) + 1] = 7; // Attribute
```

You can produce a cursor by changing the background color of the proper spot on the screen. The cursor should have an attribute such that you can see it on a blank screen. The cursor should not occlude characters that it is on top of.(I use a blue background with 'blink' set.)

Handling Special Cases

You must handle the following cases in your screen driver.

- **Space** There are several ways to generate a space if you play around with the attributes. You may want to use more than one method, especially to implement tab.

- **Backspace** If you encounter a backspace, you must delete the current character and move the cursor to the left one space. If you are at the beginning of a line, then stay at the beginning of the line and do nothing. This is basically what happens in a shell (i.e., it is line oriented). Do not back up to the previous line!
  Tab TAB is not a character which you can print (although it will actually be entered in the queue as will backspace, using the ASCII representation. If you encounter a TAB, you will move *four* spaces to the right. You can treat these four spaces as regular spaces for purposes of backspacing.

Caps Lock If the caps lock is pressed and released, it should toggle (invert, flip, switch). That is, if it was in uppercase before, it should be in lowercase now. Note that caps lock only applies to the letters of the alphabet. If you type in numbers, or any other non-alphabetic letter, you should not treat it as if it were being shifted. You should handle a repetition of caps lock in the same way that DOS deals with caps lock being pressed for a long time.
Return When you hit a return key, you should go to the beginning of the next line. If you are on the last line, you will want to scroll.
Wrap around If you type to the end of the line, you will need to go to the next line, and start at the leftmost point. You may need to scroll.
Key repetition If a key is pressed, the scan codes will repeat this character. You should also repeat thison the screen.
Scrolling There are several conditions where you will want to scroll the window. If you hit the return key when you are on the bottommost line on the screen, you will scroll the window. If you type characters beyond the bottom right hand corner, and wrap around, you will need to scroll. You will scroll in the following fashion. Line 2 will be moved to line 1. Line 3 will be moved to line 2,and so on. In general, line n will be moved to line n-1.Line 1 will be removed, and there will be a blank line at the bottom.
You will not need to store an additional array to hold the screen. Anything that is scrolled off the screen is considered lost.

Warning: Writing the screen driver is not very difficult but you do need to keep track of the cursor positions and do some calculations. This can be quite time consuming. Make sure to give yourself enough time to work on it.

What to Hand In

When the project is due, you should submit your 3.5" floppy diskette with the appropriate information on the label. Be sure to include all files necessary to compile your code, including your makefile. Also, make sure to hand in a hardcopy of your code. It is your responsibility to make sure that your disks are not damaged and do not have bad sectors. If this is the case, you may be penalized. Your code must work on Borland C++ 4.5. Please consult the TAs or instructor if this poses a problem. Make sure to include your name, and make sure to keep copies of the disk as backup. You may wish to add a README file if you feel you need to explain anything special to the TAs. This project is worth 4% of your overall grade in this class.

To prevent the entire Borland environment from being linked into your program, exit to DOS (do not merely ‘‘shell out”) before running make. Otherwise your OS might run out of memory.
You do not have to remove any object files from your diskette, but if they don’t fit, they don’t have to be there.
Suggestion: Get started early! Debugging assembly and interrupt routines can be extremely time consuming.