CZ1106 Problem Solving and Computation II

Programming Lab 3

Computer Game by Direct Screen I/O Technique

15 Feb 2007

---

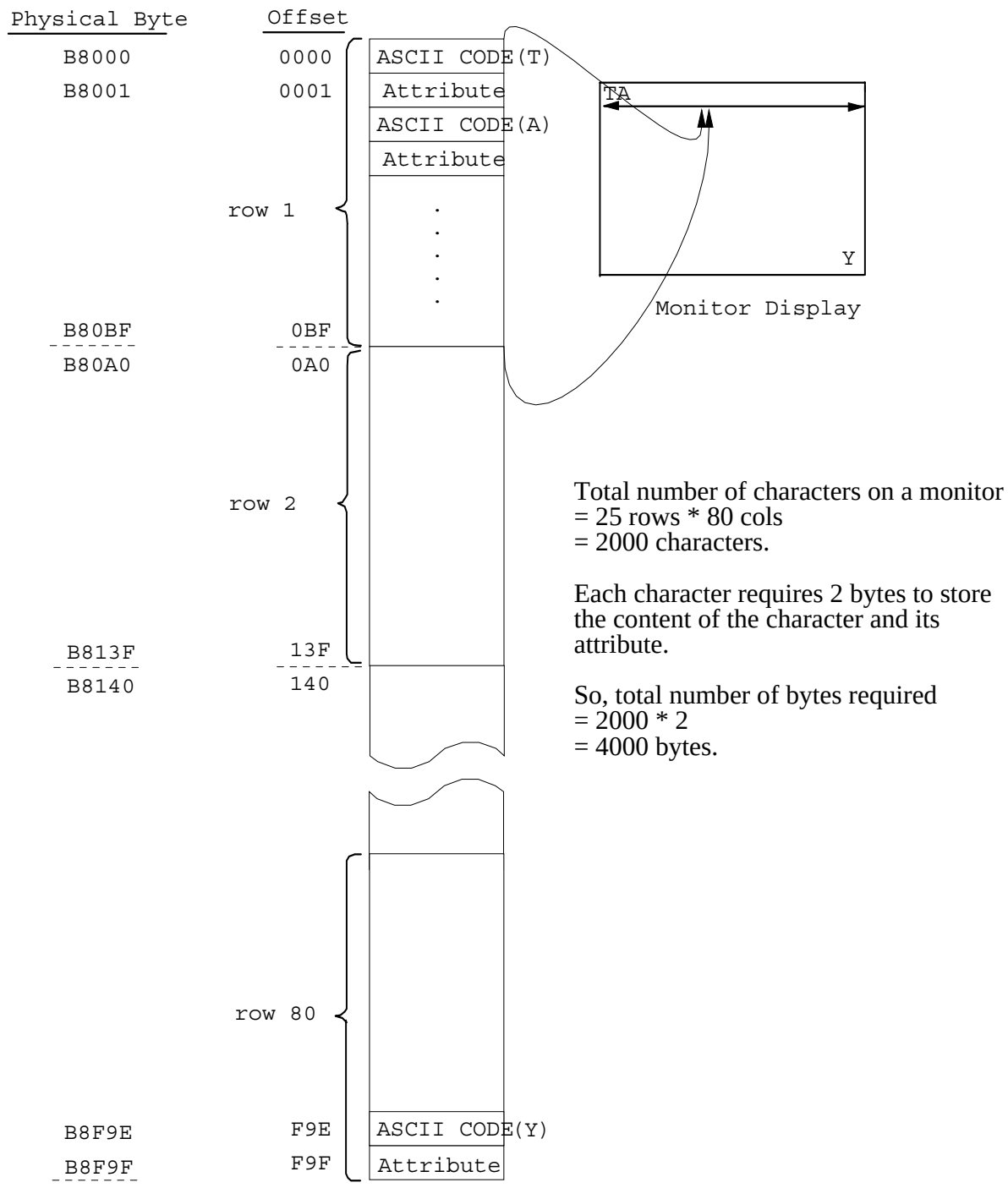
1. Direct Screen I/O

A PC's display consists of two parts: the monitor and the video controller graphic card (adapter). The graphic card serves as an interface between the monitor and the computer. Communication between the adapter and the computer is done in a special section of memory (called display memory), which is physically located on the adapter board and can be accessed both by the microprocessor and by the display screen.

The microprocessor can insert (write) values into this memory (video buffer) and read them out just as it can with ordinary random-access memory (RAM). The display hardware continuously takes value from this video buffer and places the corresponding character on the screen.

The default mode of character display consists of 25 lines and each line contains 80 columns. Altogether 2000 characters may be displayed on a monitor. Each of these characters is associated with a particular address in the display memory. Two bytes in the memory are used for each character: one to hold the extended ASCII character code, a value from 0 to 255 (0 to ff in hex), and one to hold the attribute. Thus 4000 bytes of memory are needed to represent the 2000 characters on the screen. The memory used to store the 2000 characters and their attributes starts at $B8000_{(16)}$.

Each 16-bit word in the video buffer corresponds to a single character. The first byte (8 bits) of the word is the ASCII code for the character displayed, the second byte tells the adapter what color the character is and is known as the attribute byte. In 25 x 80 text mode a whole screen of characters is stored in 2 x 25 x 80 = 4000 bytes.



Total number of characters on a monitor
= 25 rows * 80 cols
= 2000 characters.

Each character requires 2 bytes to store the content of the character and its attribute.

So, total number of bytes required
= 2000 * 2
= 4000 bytes.

Mapping Video Memory to Monitor Display

2. Addressing the Video Buffer

The *offset* byte-address ***from the BEGINNING*** of the video buffer (starting address of display memory) is calculated as follows:

The character "T" is at row 1, column 1 (refer to previous diagram):
offset = 0
(offset refers to the distance from a reference point).

The next character "A" is at row 1, column 2 (refer to previous diagram):
This is the next position and must have a offset of 2, since each character requires 2 bytes for ASCII code and attribute.

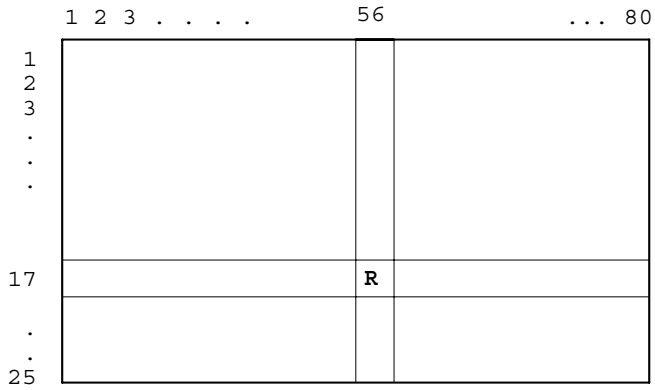
Character "Y" is at row 25, column 80 which is 24 rows and 79 columns away from character "T" (refer to previous diagram):
Since there are 80 characters on a line, "Y" has a offset of $2*(24*80+79) = 3998_{(10)} = F9E_{(16)}$.

In general the offset can be calculated in terms of row number and column number (**take note that the row and column numbers start from 1 in my notation**, in some text books they start from 0) according to :

Video Buffer Byte Offset = 2 * ((row-1)*80 + (column-1))

Example :

Compute the offset and the physical address that store the *character* R on the following screen. Also, what are the offset and the physical address that store the *attribute* of character R?



Solution :

ASCII byte :

Offset = 2 * ((17-1) * 80 + (56-1))
= $2670_{(10)}$
= $A6E_{(16)}$

Physical Address = $8000_{(16)}$ + $A6E_{(16)}$
= $B8A6E_{(16)}$

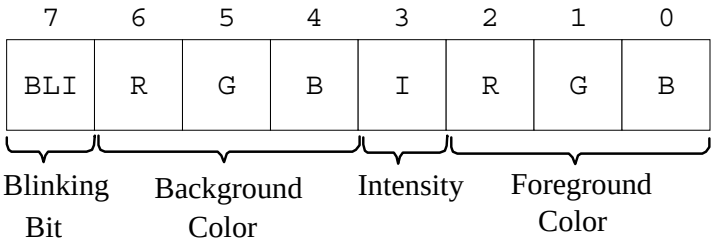
Attribute byte :

Offset = the next byte of character
$= A6E_{(16)} + 1_{(16)}$
$= A6F_{(16)}$

Physical Address = $B8A6F_{(16)}$

3. The Attribute Byte

The attribute byte in color text mode controls the characteristics (blink, foreground color and background color) of each character displayed. Bit patterns represent the display characteristics as follows::



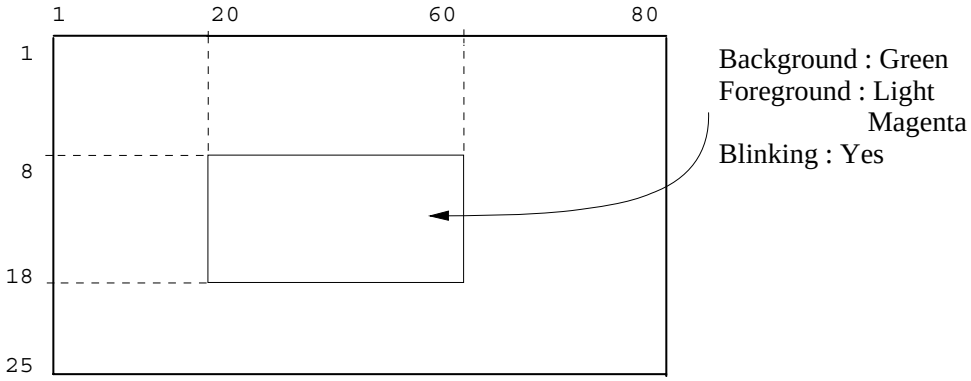
Bit 7 : selects blinking (1) or not blinking (0).
Bits 6 - 4 : control the color of the background behind the character.
Bit 3 : selects high intensity (1) or low intensity (0).
Bits 2 - 0 : control the foreground color, i.e., color of the character itself.

The letters R, G and B denote bit positions for Red, Green and Blue respectively. Note that 000 is black and 111 is white. The following is the corresponding color table for foreground colors :

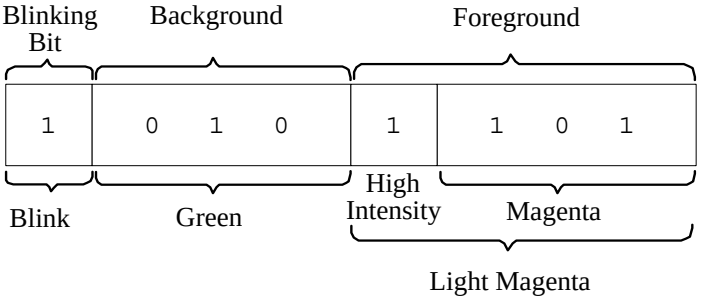
Bits 2 - 0	Bit 3 = 0	Bit 3 = 1
000	Black	Grey
001	Blue	Light Blue
010	Green	Light Green
011	Cyan	Light Cyan
100	Red	Light Red
101	Magenta	Light Magenta
110	Brown	Yellow
111	White (light grey)	High Intensity White

Example

A program used to fill the screen with small rectangle of Green background and Light Magenta foreground, and characters in the rectangle blinking.



New attribute setting :



$= AD_{(16)}$

What we have to do now is to set the attribute bytes of the memory addresses corresponding to the rectangle to $10101101_{(2)}$ and that's all.

Do not touch those memory addresses which do not belong to the rectangle, and do not touch the ASCII bytes.

But we have yet to know how to use pointers to do it.

Pointers

- Pointer address we have learned so far is stored in a 16-bit word, so it can only address from $0_{(16)}$ to $FFFF_{(16)}$. That is, the addresses (I called it house number in CP1311) where the pointer is pointing can be $0_{(16)}$, $1_{(16)}$, $2_{(16)}$, $3_{(16)}$, ... $69_{(16)}$, ... $296B_{(16)}$, ...$76DA_{(16)}$, ... up to $FFFF_{(16)}$.

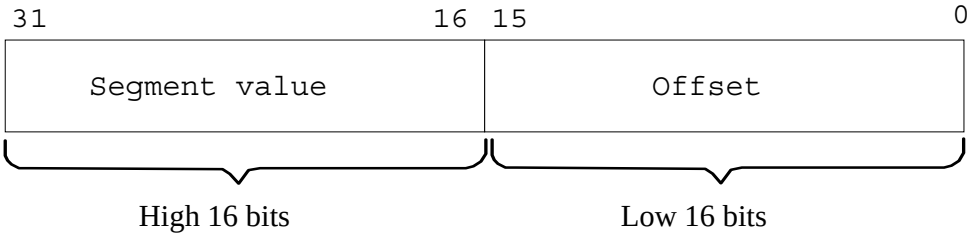
The base address of our video buffer is too far away. $B8000_{(16)}$ is beyond the addressing space of 16 bits. We therefore use a far pointer (or long pointer) to solve this problem.

far Pointer

A **far pointer** contains 32-bit address capacity. It consists of 2 parts: segment value stored in high 16 bits, and offset value stored in low 16 bits.

- 1 segment contains 16 bytes (or $10_{(16)}$ bytes).
- The actual address where a far pointer is pointing to is computed by

Actual Address = Segment * $16_{(10)}$ + Offset
= (high 16 bits) * $10_{(16)}$ + (low 16 bits)

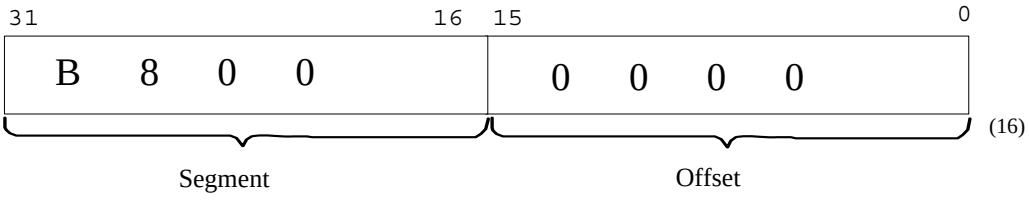


A far Pointer

Example:

```
char far *base;

base = (char far *) 0xB8000000L;  /* 0x means base 16, L means long integer (32 bits) */
```



The address where base is pointing is

$$B800_{(16)} * 10_{(16)} + 0_{(16)} = B8000_{(16)}.$$

*Base is a 32-bit far pointer, but the content where base is pointing is a character, i.e., a **8-bit byte**.*

Now we are ready to study the program !!!

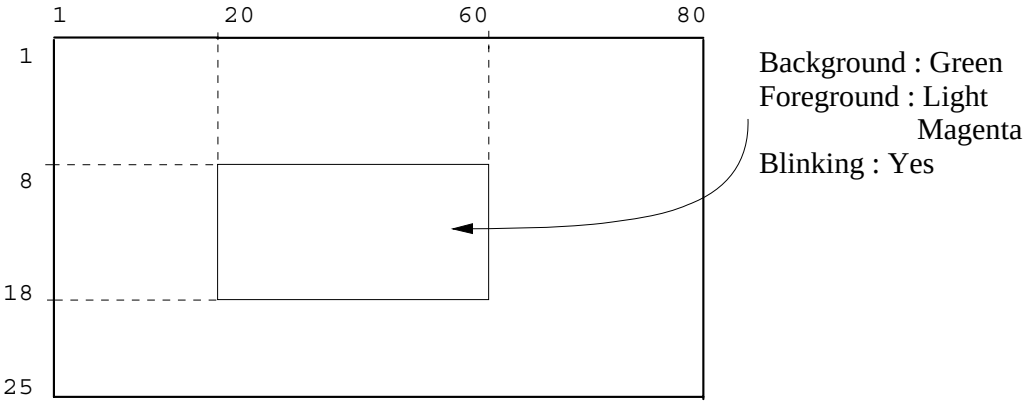
Program

```
/* AC2-8.C */

void main ()
{
  char far *base;
  int row, column;
  char new_attribute;

  base = (char far *)0xB8000000L;
  new_attribute = 0xAD ;

  for (row = 8; row <= 18; row ++)
    for (column = 20; column <=60; column ++)
      *(base + 2*((row-1)*80+column-1) +1 ) = new_attribute;
}
```



Write a program to scan over all the character displayed on the monitor screen and blinks the letters E, A, S, Y.

Program

```
/* AC2-9.c */

# include <conio.h>
# include <stdio.h>

void main ()
{
  char far *base;
  int row, column;

  base = (char far *)0xB8000000L;

  for (row = 1; row <= 25; row ++)
    for (column = 1; column <=80; column ++)
                 /* check character byte */
     if ((*(base + 2*((row-1)*80+column-1)) == 'E')  ||
         (*(base + 2*((row-1)*80+column-1)) == 'A')  ||
         (*(base + 2*((row-1)*80+column-1)) == 'S')  ||
         (*(base + 2*((row-1)*80+column-1)) == 'Y'))

      *(base + 2*((row-1)*80+column-1) +1) |= 0x80;
                              /* set attribute byte */
}
```

Attribute byte :

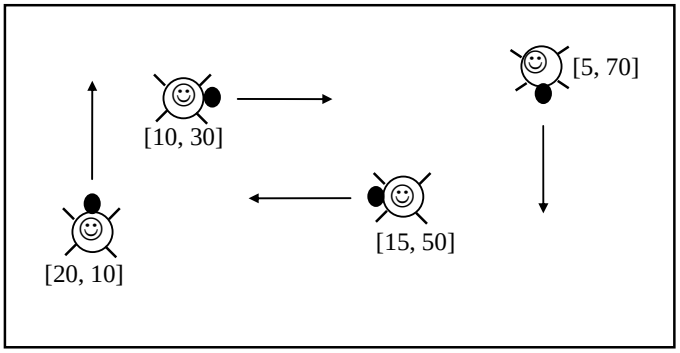
```
        ????????   (existing value)
|       10000000   (80(16))
        1???????   (new attribute)
```

background and foreground remain unchanged, but blink bit is set.

Assignment

Part-I

The moving bugs seen in pacman games can be animated by the erase-and-paint technique. Write a C program to move four bugs on the screen simultaneously.



Coordinates show the respective initial position in [row, column] format. Arrows indicate the respective direction of movement. When a bug hits the boundary of the screen, it will continue its movement from the opposite end (wrapped around visual effect).

Your tasks are organized as follows:

(i) Write a generic *draw* function to be used by other functions. The function header is as follows.

```
void draw (int row, int col, char ch, char attr)
```

This function is to display the character *ch* with the attribute *attr* on the *row*-th row and *col*-th column of the screen.

(ii) Write a function to draw a bug. The function header is as follows.

```
void draw_bug (int row, int col)
```

This function will call the *draw* function in (i) to display a bug, represented by the ASCII code 0x02, at *row*-th row and *col*-th column. The attribute of the ASCII byte is as follows:

- no blink
- background : black
- foreground : cyan

(iii) Write a function to erase a bug. The function header is as follows.

```
void erase_bug (int row, int col)
```

This function is similar to the *draw_bug* function in (ii), except that it paints a black space to erase the bug.

(iv) Write the main function that uses the *draw_bug* and *erase_bug* functions to animate four bugs moving on the screen. The initial positions for the bugs are given in the diagram, and an array may be used to store the coordinates. The following algorithm may be used:

```
call draw_bug 4 times to draw 4 bugs on their initial positions;

while the keyboard is not pressed
{
    repeat 4 times (one time for each bug)
    {
        call erase_bug using the current position;
        advance the bug's position;
        if the bug hits the boundary, adjust the position to
           the other end;
        call draw_bug using the new position;
    }
}
```

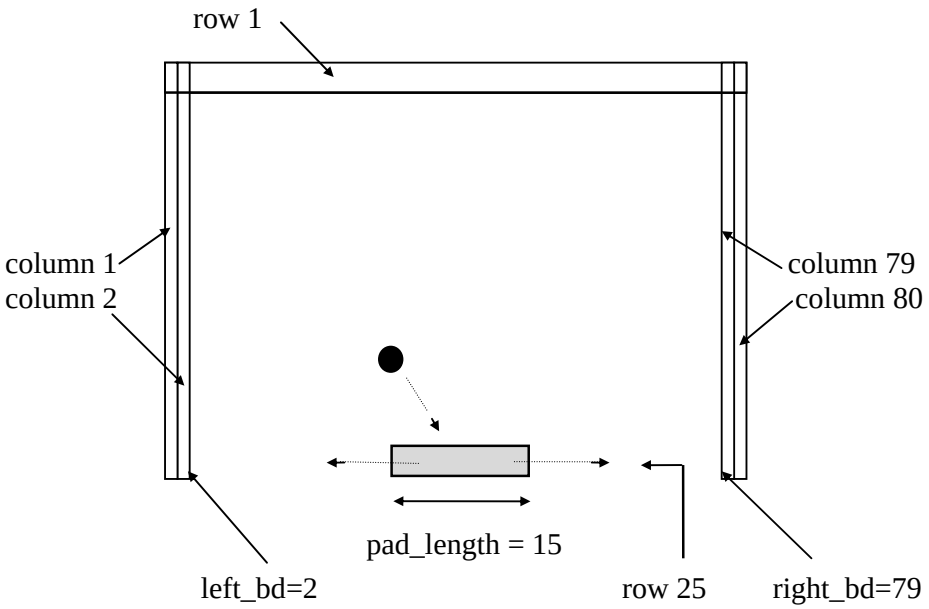
Part II

(Part II is to be handled in. Please send in a printed copy and a soft copy)

Why some super-programmers choose to earn a living by writing computer game programs and not any other things else? There are two reasons. First, writing game programs is always interesting as we all like to play game. Second, the monetary reward is amazingly attractive and the demand is never end. Perhaps the second one is the actual reason. Yes, we are going to write a computer game program now. I hope this practical will serve as a starter for you to earn that big money in your future career.

As a beginner, we will write a simple arcade game: the Pin Ball. First, copy my executable file ***p3.exe*** from the web and play the pin ball for 5 minutes (not more than 5 minutes as you have to hand in your program).

Let us do some serious work now. To animate a pin ball on the computer screen, the simple technique is to draw the ball, erase it and draw it again in a position slightly further away. Doing this rapidly and continually fools the eyes into seeing a moving ball. There are three walls and one pad to bounce the ball and the game is over if the ball miss the pad.



The following steps may be followed totally, partially or none at all depending on individuals. I believe that most of you are capable to organize your program efficiently, and therefore I won't be surprised to see some algorithms even better than what I have here.

1. Generic Draw Function

First, we shall write a generic *draw* function to be used by other functions. The header goes as follows.

```
void draw (int row, int col, char ch, char attr)
```

Its task is to display the character *ch* with the attribute *attr* on the *row*-th row and *col*-th column of the screen.

2. Boundary and Interior

Next, we are ready to draw the three walls, consisting of the first row, first two columns and the last two columns, and draw the interior. As for the walls, you can use the *draw* function to display the vertical rectangles (ASCII code = 219) on the screen. For the interior you can use spaces. Need not follow my color. Choose your own favorite variations. You can include these tasks in a function of the following header.

```
void draw_bound_and_interior()
```

3. Drawing and Erasing Ball

Now, write two generic functions to draw a ball and erase a ball respectively. The function headers are as follows.

```
void draw_ball(int row, int col)
void erase_ball(int row, int col)
```

The first function will call the *draw* function to display a ball, represented by a 'o' character of your favorite attribute at *row*-th row and *col*-th column. The second function is similar, but use the space and the interior's attribute to erase the 'o' character.

4. Drawing and Erasing Pad

Write two generic functions again to draw a pad and erase a pad respectively. The function headers are as follows.

```
void  draw_pad(int left)
void  erase_pad(int left)
```

The input parameter *left* refers to the column number of the left end of the moving pad. Row number is fixed at 25. The task of the first function is to call the draw function to draw 15 colored spaces starting from the *left*-th column on the last row of the screen. The second function does the same thing, but the color of the spaces is changed to that of interior.

5. Moving Ball

The ball moves diagonally. Passing parameters by references is used in the following function so that the contents of the row and column numbers can be updated.

```
void move_ball (int *row, int *col)
```

First, the static increments δrow and δcol are initialized to +1 to indicate that the ball moves downward and to the right. When the boundary conditions are met, the sign of the increment will need to be changed and a "pop" sound will be created to simulate the hit. The *erase_ball* and *draw_ball* functions are used. The "pop" sound can be created by these library routines.

```
# include <dos.h>

sound (frequency);
delay (milliseconds);
nosound();
```

Suggested values are: frequency = 400, milliseconds = 15.

The following logic may be used in this function.

```
erase existing ball;
add δrow to the contents of row number;
add δcol to the contents of column number;

if the ball is at the top row or bottom row
{
  change the sign of δrow;
  add δrow to the contents of row number;
  create "pop" sound;
}

if the ball is at left wall or right wall
{
  change the sign of δcol;
  add δcol to the contents of column number;
  create "pop" sound;
}

draw a ball at new position;
```

Why δrow and δcol have to be static ?

6. The Main Algorithm

The main algorithm is universal in most computer game programs. It is basically an infinite loop and exits on some event occurrence. In our pin ball game the *main* function starts by drawing the three walls, a pad at the center and a ball at left top corner. A speed regulator, suggested value is 4 (never less than 2 unless you are super alert), is used to control the ball speed. The program will have to detect if the user presses the left or the right arrow key to move the pad. Two sliding distances are used. The first one (*slide1*) must be one step, while the second one (*slide2*) can be wider (say 3 steps). The keyboard scan codes (KSC) for the arrow keys are as follows.

Keystroke	→	←
KSC	0, 77	0, 75

Use the *kbhit* function to check if a key stroke is pressed. E.g.,

```
if ( kbhit() )
{
    ......
}
```

The following descriptions contain the main logic.

```
draw walls and interior;
draw ball;
draw pad;
```

```
make noise;

Infinite loop
{
  delay(10 miniseconds);

  if (flag==regulator)
     {
          if ball is one row above pad
          {
            if ball is not on top of pad (check column number)
             {
               make noise;
              game over and exit straight away;
             }
          }
          move the ball;
          set flag to 0;
     }
    else
      increment flag by 1;
if a key is pressed
    {
      get the keyboard scan code;

      if scan code is for a space
        abort the game and exit;
      else
      if scan code is 0
        {
            get next scan code
            if scan code is for a left arrow
              {
                if (left corner pad position > left_bd+1)
                 {
                      erase the existing pad;
                      if (left corner pad position > left_bd+slide2)
                        update left corner pad position to the left by slide2;
                      else
                        update left corner pad position to the left by slide1;
                      draw a new pad;
                 }
                else
                  make “pop” sound;
              }
            else
            if scan code is for a right arrow
              {
                if (left corner pad position < right_bd- pad_length)
                 {
                      erase existing pad;
```

```
                    if (left corner pad position < right_bd- pad_length - slide2)
                      update left corner pad position to the right by slide2;
                    else
                      update left corner pad position to the right by slide1;
                    draw a new pad;
                  }
                 else
                  make “pop” sound;
               }
          }    /* if scan code is 0 */
    }       /* if a key is pressed */
  }         /* infinite loop */
}
```

Can you explain why slide1 must be 1 step ?

You may pair up with another student to do this practical exercise. Hand in only one copy of listing and diskette but write two names on both print-out and diskette label. Name the source file as **game.c**.

Optional:

Later you may like to improve your game program further, such as displaying some instructions and the name of the game creator (your big name) on the first screen, keeping the highest score history and the player’s name in a file and display them before game starts, giving the player an option to adjust the ball speed, displaying “GAME OVER”, etc. Let your friends and your family members play the game.

There are many other games which can be evolved from the pin ball. You may want to write the ping pong game (2 pads opposite), the squash game (1 pad center and 1 pad below), or even with 4 pads on a square. All these new games require only minimal modifications to the existing program and you are encouraged to hand in these games to replace the pin ball, and for bonus of course.

I hope this assignment has fired up your desire to become a computer game creator. All the best.

Use debugger!!