Building a Routine Library

WEEK
2

On Day 10, you were presented with a bunch of new functions. You'll use many of these throughout the rest of the book. In addition to the functions presented on Day 10, you'll also need to have several other functions available. Today you will:

- ☐ Learn about several functions that are important when creating applications.
- ☐ Create several functions, among them:
  - ☐ Hide the cursor
  - ☐ Clear the screen (in color)
  - ☐ Display a grid (great for shadows)
- ☐ Learn how to overwrite and restore a screen.

Some Important Issues

When you are creating applications, there are several functions that are often not thought of. Many of these functions can be easy to create. Today, you'll be presented with a few of these functions so that you can add them to your TYAC.LIB library. The first two files presented are `cursor_on()` and `cursor_off()`.

The *cursor_off()* Function

The `cursor_off()` function enables you to hide the cursor. There are several times in running an application when the user is not entering data. During these times, the cursor can become a nuisance. The `cursor_off()` function does just what its name implies. It turns the cursor off.

Listing 11.1. CURSOFF.C. The `cursor_off()` function.

```
1:  /* ----------------------------------------------------------
2:   * Program: CURSOFF.C
3:   * Authors: Bradley L. Jones
4:   *          Gregory L. Guntle
5:   *
6:   * Purpose: Turns the cursor off.
7:   *
8:   * Enter with: N/A
9:   *
10:  * Returns:
11:  *
12:  * ---------------------------------------------------------- */
13:
```

```
14:  #include <dos.h>
15:  #include "tyac.h"
16:
17:
18:  void cursor_off()
19:  {
20:     union REGS inreg, outreg;          /* Assembly Registers */
21:     inreg.h.ah = 1;                    /* int 10h function 1 */
22:     inreg.x.cx = 0x0F00;               /* Wrap cursor around to
23:                                           turn it off */
24:     int86(BIOS_VIDEO, &inreg, &outreg);  /* BIOS Call */
25:  }
```

Analysis As you can see, Listing 11.1 presents the `cursor_off()` function, which is very short. This is a BIOS function similar to those that you have seen before. In this function, the `ah` register is set to function 1 (line 21). The BIOS video interrupt is then called. The BIOS interrupt is interrupt 0x10h, which is defined in the TYAC.H header file.

The *cursor_on()* Function

Once you turn the cursor off, it remains off until you turn it on again. You'll need the cursor back on when you are ready to have the user input data. You'll also want to make sure that if the cursor is turned off, you turn in back on before you exit your program. Listing 11.2 presents `cursor_on()`, which is a counter function to the `cursor_off()` function.

Warning: If you turn the cursor off and then exit the program, the cursor may remain off.

Type **Listing 11.2. CURSON.C. The `cursor_on()` function.**

```
1:   /* ------------------------------------------------------
2:    * Program: CURSON.C
3:    * Authors: Bradley L. Jones
4:    *          Gregory L. Guntle
5:    *
6:    * Purpose: Turns the cursor on.
7:    *
8:    * Enter with:
```

continues

Listing 11.2. continued

```
 9:    *
10:    * Returns:
11:    *
12:    * ------------------------------------------------------*/
13:
14:   #include <dos.h>
15:   #include "tyac.h"
16:
17:
18:   void cursor_on()
19:   {
20:      union REGS inreg, outreg;
21:      inreg.h.ah = 1;               /* int 10h function 1 */
22:      inreg.x.cx = 0x0607;          /* Cursor size for CGA/VGA */
23:      int86(BIOS_VIDEO, &inreg, &outreg);  /* BIOS call */
24:   }
```

Analysis This function is almost identical to the `cursor_off()` function. The difference is that the value in the `x.cx` register is set to 0x0607. This turns on a cursor that is appropriate for most monitors.

Now that you have functions that can turn a cursor off and on, you are probably interested in seeing them in action. Listing 11.3 presents a small program that uses the TYAC.LIB library. You should go ahead and compile the cursor functions and add them to your library. You should also include prototypes in the TYAC.H header file.

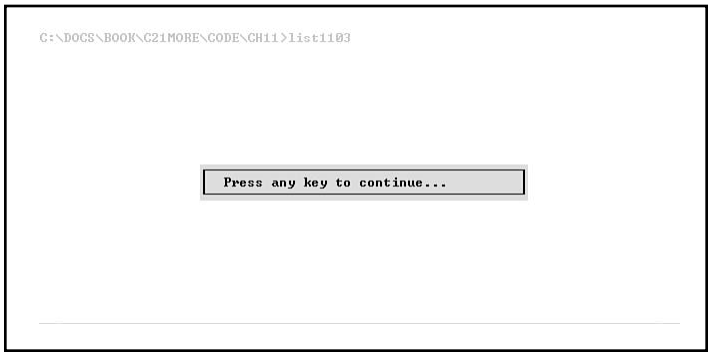
Type

Listing 11.3. LIST1103.C. Using the cursor functions.

```
1:    /*=========================================================
2:     * Filename: LIST1103.c
3:     *
4:     * Author:   Bradley L. Jones
5:     *           Gregory L. Guntle
6:     *
7:     * Purpose:  Demonstrate the cursor on and off functions.
8:     *=========================================================*/
9:
10:   #include <stdio.h>
11:   #include <conio.h>       /* not an ANSI header, for getch() */
12:   #include "tyac.h"
13:
14:   void main(void)
```

```
15:  {
16:     cursor_off();
17:
18:     box(12, 14, 20, 60, SINGLE_BOX, FILL_BOX, YELLOW, BLUE );
19:
20:     write_string( "Press any key to continue...",
21:                   YELLOW, BLUE, 13, 23 );
22:
23:     getch();
24:     cursor_on();
25:  }
```

Output



Analysis This program displays a box on the screen with a message in it. Line 18 uses the `box()` function followed in line 20 by the `write_string()` function. Before setting up this box, line 16 calls the `cursor_off()` function. This function could have been called at any time before line 23. In line 23, the program pauses with a call to `getch()`, which waits for any character to be entered. When a key is pressed, the `cursor_on()` function returns the cursor.

If the cursor had not been turned off, it would be seen flashing on the screen when this box is displayed. You can see this by commenting out line 16. Another good experiment to try is to uncomment line 16 and then comment out line 24. This will cause the cursor to not be turned back on when the program exits. By observing each of these scenarios, you should begin to understand the importance of making sure you know the condition of the cursor—either on or off.

Clearing the Screen

Clearing the screen can be very important. Most programs that work with the screen will start by clearing the screen. You can never be sure about what is on the screen when

you first start a program. Listing 11.4 presents a function that will effectively clear the screen. In addition to removing everything on the screen, this function enables you to state what colors the screen should be cleared to.

Type

Listing 11.4. CLEARSCN.C. The `clear_screen()` function.

```
1:  /* Program: CLRSCRN.C
2:   * Author:  Bradley L. Jones
3:   *          Gregory L. Guntle
4:   * Purpose: Function to clear the entire screen
5:   *          Borland offers a clrscr() function.
6:   *-------------------------------------------------------
7:   * Parameters:
8:   *          fcolor,
9:   *          bcolor         colors for clearing screen
10:  *=======================================================*/
11:
12: #include <dos.h>
13: #include "tyac.h"
14:
15: void clear_screen(int fcolor, int bcolor)
16: {
17:     union REGS irreg;
18:
19:     ireg.h.ah = SCROLL_UP;
20:     ireg.h.al = 0;       /* Clear entire screen area */
21:     ireg.h.ch = 0;
22:     ireg.h.cl = 0;
23:     ireg.h.dh = 24;
24:     ireg.h.dl = 79;
25:     ireg.h.bh = (bcolor <<4) | fcolor;
26:
27:     int86( BIOS_VIDEO, &ireg, &ireg );
28: }
```

Analysis

This function provides a means to clear the screen. This is done by scrolling the information off of the screen; this is a common practice. As you can see, this function enables you to set the foreground and background colors.

Borland

If you are using the Borland compiler, you have the option of using a different function. Borland provides a function called `clrscr()`, which clears the screen

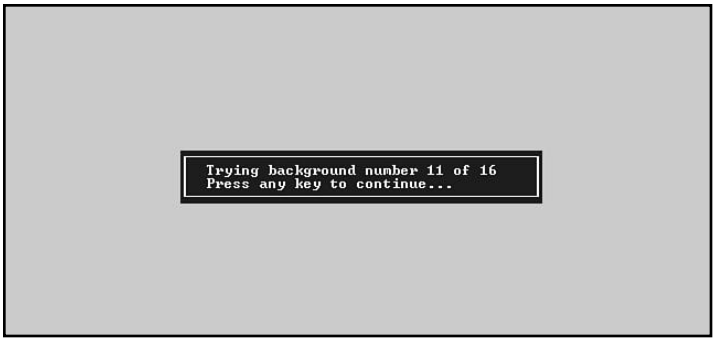
without the option of setting the colors. You should remember that this is a Borland-specific function and, as a result, may not be portable.

Listing 11.5 demonstrates clearing the screen. This listing enables you to clear the screen several times before exiting.

Listing 11.5. TESTCLR.C. Test the `clear_screen()` function.

```
1:  /* Program:  testclr.c
2:   * Author:   Bradley L. Jones
3:   *           Gregory L. Guntle
4:   * Purpose:  Demonstrate the clear_screen function.
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:  #include <conio.h>       /* not an ANSI header, for getch() */
9:  #include "tyac.h"
10:
11: void main(void)
12: {
13:    int  ctr = 0;
14:    char buffer[40];
15:
16:    cursor_off();
17:
18:    for( ctr = 0; ctr < 16; ctr++ )
19:    {
20:       clear_screen( GREEN, ctr );
21:
22:       box(11, 14, 20, 60, SINGLE_BOX, FILL_BOX, YELLOW, BLUE);
23:
24:       sprintf( buffer, "Trying background number %d of 16",
25:                                                        ctr+1);
26:       write_string( buffer, YELLOW, BLUE, 12, 23 );
27:       write_string( "Press any key to continue...",
28:                               YELLOW, BLUE, 13, 23 );
29:
30:       getch();
31:    }
32:
33:    clear_screen( GREEN, BLACK );
34:    cursor_on();
35: }
```

Output



Analysis

This program enables you to see the different colors that the background can be cleared to. As you can see, the `cursor_on()` and `cursor_off()` functions are used to turn the cursor off at the beginning of the listing and then back on at the end of the listing.

The `for` loop, which makes up the bulk of this program (lines 18 to 31), displays the counter number, `ctr`, in a message box. Line 24 formats this counter number into a descriptive message using `sprintf()`. The `sprintf()` function is a standard ANSI function that enables you to format information into a string. Before formatting `buffer`, line 22 displays a box similar to the box in Listing 11.3. Once the box is displayed with its message, the program pauses and waits for the user to enter a key. When a character is pressed, the `for` loop cycles through to the next counter value. This continues for 16 iterations. Line 33 clears the screen one last time before restoring the cursor and exiting.

The *grid()* Function

There are times when you'll want to clear the screen to a textured background. There are also times when you'll want to create a shadow that is somewhat different than just a box. Listing 11.6 presents a function called `grid()` that enables you to display a box created with one of the ASCII grid characters.

Note: The ASCII grid values are:

∞ 176

± 177

≤ 178

Type

Listing 11.6. GRID.C. The `grid()` function.

```
1:  /* Program: GRID.C
2:   * Authors: Bradley L. Jones
3:   *          Gregory L. Guntle
4:   *
5:   * Purpose: When passed the parameter list it displays
6:   *          a grid background using BIOS.
7:   *
8:   * Enter with: start_row, end_row (0-24)
9:   *             start_col, end_col (0-79)
10:  *             fcolor, bcolor
11:  *             gtype
12:  * ------------------------------------------------------------*/
13:
14:  #include <dos.h>
15:  #include "tyac.h"
16:
17:  void grid( int start_row, int end_row,
18:             int start_col, int end_col,
19:             int fcolor,    int bcolor, int gtype)
20:  {
21:     int row, col;
22:
23:     /*   grid types  */
24:     static unsigned char GRID_1[1] = "∞"; /* ASCII value 176 */
25:     static unsigned char GRID_2[1] = "±"; /* ASCII value 177 */
26:     static unsigned char GRID_3[1] = "≤"; /* ASCII value 178 */
27:     static unsigned char *GRID;
28:
29:     switch (gtype)
30:     {
31:       case 1:   GRID = GRID_1;
32:                 break;
33:
34:       case 2:   GRID = GRID_2;
35:                 break;
36:
37:       case 3:   GRID = GRID_3;
38:                 break;
39:     }
40:
41:     for (row=start_row; row < end_row+1; row++)
42:     {
43:         for ( col=start_col; col < end_col+1; col++)
44:         {
45:            cursor(row,col);
46:            write_char( (char)*GRID, fcolor, bcolor );
47:         }
48:     }
49:  }
```

Analysis The grid function can be added to your TYAC.LIB library along with all the other functions. In addition, you should add an appropriate prototype to your TYAC.H header file.

Line 17 begins the `grid()` function. As you can see, there are several parameters being passed. The starting row and column, along with the ending row and column, are passed in the same manner as they were in the `box()` function. The foreground and background colors are also passed so that the color of the grid is able to be customized. The final parameter, `gtype`, is used to determine which ASCII grid character is used.

Line 21 declares two temporary variables, `row` and `col`, that will be used later in the function to display the grid characters. Lines 24 to 27 set up the three different grid characters. Line 28 contains a pointer that will be used in lines 29 to 39 to point to the appropriate character based on the value passed in `gtype`. If `gtype` contains a 1, then the `GRID_1` character will be used. If `gtype` contains a 2 or 3, then `GRID_2` or `GRID_3` will be used. The pointer from line 28 will be set to point at the appropriate character.

Lines 41 to 49 contain two nested `for` loops that draw the grid character on the screen. Using the `for` loops, the row and column values are incremented and a character is written using `write_char()`. When the `for` loops are completed, the grid is drawn.

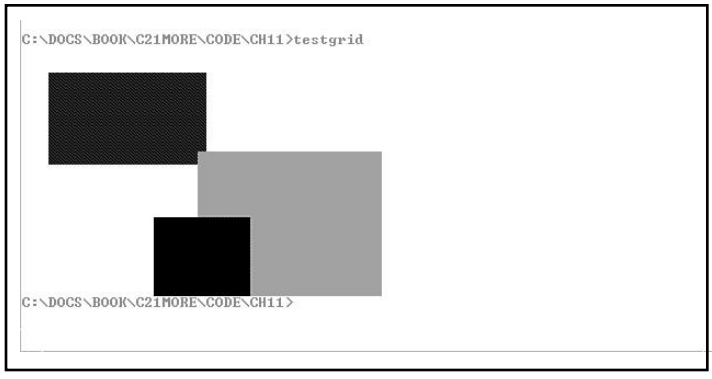
Listing 11.3 uses each of the three grid types. The `grid()` function is called three times displaying boxes with each of the different grids.

Type

Listing 11.7. TESTGRID.C. Testing the `grid()` function.

```
1:  /* Program:  testgrid.c
2:   * Author:   Bradley L. Jones
3:   *           Gregory L. Guntle
4:   * Purpose:  Demonstrate the grid function.
5:   *=========================================================*/
6:
7:  #include "tyac.h"
8:
9:  int main( void )
10: {
11:     grid(4,10,3,20,RED,GREEN,1);
12:     grid(10,20,20,40,WHITE, BLUE,2);
13:     grid(15,20,15,25,BRIGHTWHITE,CYAN,3);
14:
15:     return 0;
16: }
```

Output



Analysis

This listing is as straightforward as they can come. The `grid()` function is called three times. You should notice that different parameters are passed that modify the grids displayed. In addition to different grid styles, the location, size, and colors also vary in each of the three calls.

11

Saving and Restoring the Screen

It's quite useful when you can place items on the screen and then remove them without overwriting the underlying information. For example, if you use the `box()` function to place a box with a message on the screen, you overwrite what was underneath. You must redraw the screen to restore the lost information. By saving off a copy of the screen, or a copy of the portion of the screen that will be overwritten, you can then simply restore it when you are done.

To help you understand this, an example will be presented in Listing 11.10, but you first need to see the `save_screen_area()` function in Listing 11.8 and the `restore_screen_area()` function in Listing 11.9.

Type

Listing 11.8. SAVESCRN.C. Saving a portion of the screen.

```
1:   /* Program: SAVESCRN.C
2:    *
3:    * Authors: Bradley L. Jones
4:    *          Gregory L. Guntle
5:    *
```

continues

Listing 11.8. continued

```
6:    * Purpose: Saves the information that is on the screen
7:    *          which is defined within a row/col area.
8:    *
9:    * Function: save_screen_area(int start_row, int end_row,
10:   *                            int start_col, int end_col)
11:   *
12:   * Enter with: start_row  (0-24)
13:   *             end_row    (0-24)
14:   *             start_col  (0-79)
15:   *             end_col    (0-79)
16:   *
17:   * Returns:  Address to the memory location where screen info
18:   *           has been saved
19:   * ------------------------------------------------------------ */
20:
21:  #include <dos.h>
22:  #include “tyac.h”
23:  #include <stdlib.h>
24:  #include <stdio.h>
25:
26:  #define READ_CHAR_ATTR  0x08
27:
28:  char *save_screen_area(int start_row, int end_row,
29:                         int start_col, int end_col)
30:  {
31:     char *screen_buffer;
32:     union REGS inregs, outregs;
33:
34:     int total_space;      /* Holds space requirements */
35:     int row = start_row;  /* Used to loop through row/cols */
36:     int col;
37:     int ctr;              /* offset ctr for info in buffer */
38:     int trow, tcol, page, cur_st, cur_end;  /* Hold curs info */
39:
40:         /* Page is critical */
41:     get_cursor( &trow, &tcol, &page, &cur_st, &cur_end);
42:
43:         /* Determine amount of space for holding the area */
44:     total_space =
45:          (((end_row-start_row+1)*(end_col-start_col+1)) * 2);
46:
47:     total_space+=5;  /* Hold row/col/page info at beginning */
48:
49:     screen_buffer = (char *)malloc(total_space);
50:     if (!screen_buffer)
51:     {
52:       printf(“Unable to allocate memory!\n”);
53:       exit(1);
54:     }
```

```
55:
56:     /* Save screen area position */
57:     *(screen_buffer+0) = (char) start_row;
58:     *(screen_buffer+1) = (char) end_row;
59:     *(screen_buffer+2) = (char) start_col;
60:     *(screen_buffer+3) = (char) end_col;
61:     *(screen_buffer+4) = (char) page;
62:
63:     /* Save the current info row by row */
64:     ctr = 5;
65:
66:     while (row <= end_row)
67:     {
68:       col = start_col;            /* Reset col pos */
69:       while (col <= end_col)
70:       {
71:          /* Position cursor */
72:          cursor( row, col );
73:          inregs.h.ah = READ_CHAR_ATTR;
74:          inregs.h.bh = page;
75:          int86(BIOS_VIDEO, &inregs, &outregs);
76:
77:              /* Save character */
78:          *(screen_buffer+ctr++) = (char )outregs.h.al;
79:              /* Save attribute */
80:          *(screen_buffer+ctr++) = (char )outregs.h.ah;
81:
82:          col++;  /* next col */
83:       }
84:       row++;    /* Next row */
85:     }
86:
87:     /* Address where screen area saved */
88:     return(screen_buffer);
89:  }
```

Analysis The first thing you should notice about this function is the comments, which describe what is going to happen in the `save_screen_area()`. The parameters that are received in line 28 are discussed in lines 12 to 15 of the comments. These parameters are the starting row, `start_row`, the ending row, `end_row`, the starting column, `start_col`, and the ending column, `end_col`. These define a rectangular area on the screen that will be saved. This can be the entire screen or any portion of it.

Just as important as the parameters is the return value, which returns a character pointer. This pointer will be the address of the buffer used to save the screen information. This returned pointer will be needed to restore the screen information.

Warning: If you choose not to restore a saved portion of the screen, you'll need to use the `free()` function on the returned character pointer. This will free the memory that the `save_screen_area()` function allocated.

Before the function starts working to save the screen area, information on the cursor is retrieved with the `get_cursor()` function to determine the video page. When saving the screen area, you'll need to know the current page.

The `save_screen_area()` function saves the area of screen into a buffer. The first five bytes of this buffer are set aside to hold information on the area that is being saved. Figure 11.1 is a representation of this buffer.

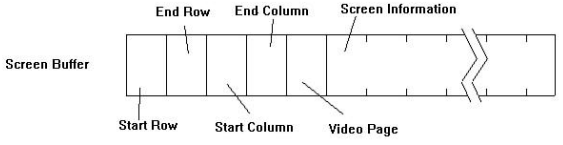


Figure 11.1. *Representation of the saved screen buffer.*

Line 44 determines the amount of space that will be needed to save the screen area. This is done by determining the difference between the starting and ending rows and columns and then multiplying them together. This is then multiplied by two because two bytes will be needed for each position on the screen. One for the actual character displayed, the other for the attributes or color of the character. Line 47 then adds five to this calculated number for the overhead bytes presented in Figure 11.1.

Lines 49 to 54 work to allocate the amount of space that was calculated. If the space is not allocated, an error message is printed and the program ends. This isn't the cleanest exit for a memory allocation error; however, it is acceptable.

Lines 57 to 61 fill in the first five bytes of the screen buffer with the rows, columns, and page of the screen area being saved. While this is done by dereferencing offsets, it could also have been done by using the following:

```
screen_buffer[0] = (char) start_row;
```

Tip: Because the `screen_buffer` is a pointer, it is more consistent to use dereferencing.

In line 64, the `ctr` variable, the offset into the `screen_buffer`, is set to five. It is from this point that you are ready to begin saving screen information. Line 66 begins a `while` loop that cycles through each column. Line 69 begins a second `while` loop that cycles through each row. The result is that each row is read within each column until the entire block is read.

For each position read in the `while` loops, several things occur. In line 72, the `cursor()` function is used to set the cursor to the current row and column position within the block. Line 73 sets the `ah` register to the appropriate BIOS function number for reading a character and its attribute. The `bh` register also needs to be set to the page number that was determined by using the `get_cursor()` function earlier. Once the registers are set, line 75 calls the `BIOS_VIDEO` function. This function returns the character in the `al` register and the attributes in the `ah` register. These values are placed in the `screen_buffer`. At the time the values are placed in the buffer, the offset pointer, `ctr`, is incremented to the next position.

This process cycles through the entire area to be saved. Once the entire screen area is saved, line 88 returns the pointer to the `screen_buffer` to the calling program. The calling program will use this pointer to restore the screen with the `restore_screen_area()` function. This function is presented in Listing 11.9.

Type

Listing 11.9. RESSCRN.C. Restoring the saved portion of the screen.

```
1:  /* Program: RESSCRN.C
2:   *
3:   * Authors: Bradley L. Jones
4:   *          Gregory L. Guntle
5:   *
6:   * Purpose: Restores information from the screen_buffer area
7:   *          that was saved using the   save_screen_area
8:   *          function.
9:   *
10:  * Function: restore_screen_area()
11:  *
12:  * Enter with: Address of area containing data from last
13:  *             save_screen_area call.
14:  *-------------------------------------------------------*/
15:
16:  #include <dos.h>
17:  #include <stdlib.h>
18:  #include "tyac.h"
19:
20:  void restore_screen_area(char *screen_buffer)
```

continues

Listing 11.9. continued

```
21:  {
22:     union REGS inregs, outregs;
23:     int   start_row, start_col, end_row, end_col, video_page;
24:     int   ctr=5;
25:     int   row;
26:     int   col;
27:
28:     start_row  = (int)*(screen_buffer+0);
29:     end_row    = (int)*(screen_buffer+1);
30:     start_col  = (int)*(screen_buffer+2);
31:     end_col    = (int)*(screen_buffer+3);
32:     video_page = (int)*(screen_buffer+4);
33:     row        = start_row;
34:
35:     while (row <= end_row)
36:     {
37:       col = start_col;            /* Start col at beginning */
38:       while (col <= end_col)
39:       {
40:          /* Position cursor */
41:          cursor( row, col );
42:
43:          inregs.h.ah = WRITE_CHAR;
44:          inregs.h.bh = video_page;
45:
46:          /* Get character */
47:          inregs.h.al = *(screen_buffer+ctr++);
48:          /* Get attribute */
49:          inregs.h.bl = *(screen_buffer+ctr++);
50:          inregs.x.cx = 1;
51:
52:          int86(BIOS_VIDEO, &inregs, &outregs);
53:
54:          col++;   /* next col */
55:       }
56:       row++;      /* Next row */
57:     }
58:
59:     free(screen_buffer);        /* Free memory */
60:  }
```

Analysis After seeing the `save_screen_area()` function, you should be able to follow this listing. This listing works almost backwards from the way the `save_screen_area()` function worked. In lines 28 to 33, the values that had been saved off for the rows, columns, and page are taken out of the screen buffer and placed in variables.

Note: Because the values for the buffer location were included in the saved buffer, there was no need to tell the `restore_screen_area()` anything more than where the `screen_buffer` array was.

Lines 35 to 57 include the two `while` statements that are used to loop through the screen buffer. Like the `save_screen_area()` function, the cursor is placed in the appropriate location, the BIOS registers are set up, a BIOS function is called, values are incremented, and the offset into the buffer is incremented. While the process is nearly identical, characters are being written instead of read. In line 43, you see that the `WRITE_CHAR` value is placed in the `ah` register. In lines 47 and 49, the character and attributes are taken from the `screen_buffer` array and placed in the `al` and `bl` registers before calling the `BIOS_VIDEO` interrupt.

The last code line of this function is very important. Line 59 frees the `screen_buffer`. Once freed, this buffer can no longer be used. If you don't free the buffer, the memory will still be allocated.

Saving and Restoring in Practice

You can now save and restore areas of the screen. Listing 11.10 is a program that shows `save_screen_area()` and `restore_screen_area()` in action.

Listing 11.10. LIST1110.C. Demonstration of saving and restoring the screen.

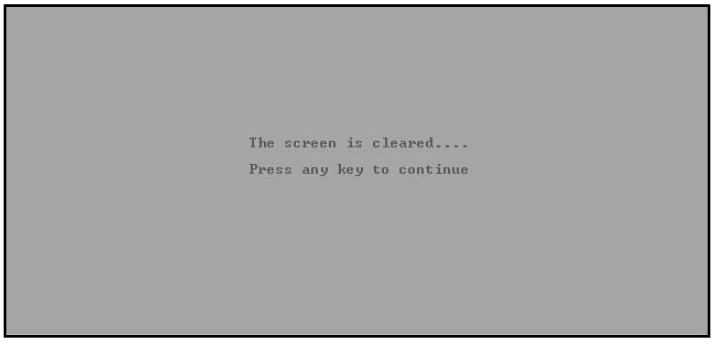
```
1:  /* Program:  LIST1110.C
2:   * Author:   Bradley L. Jones
3:   *           Gregory L. Guntle
4:   * Purpose:  Use the screen saving/restoring functions.
5:   *===========================================================*/
6:
7:  #include <stdio.h>
8:  #include <conio.h>     /* for getch() prototype */
9:  #include "tyac.h"
10:
11: int main(void)
12: {
13:     int crow, ccol, cpage, c_st, c_end; /* for cursor */
14:     char *screen_buffer;
15:
```

continues

Listing 11.10. continued

```
16:      screen_buffer = save_screen_area(0,24,0,79);
17:      get_cursor( &crow, &ccol, &cpage, &c_st, &c_end);
18:
19:      clear_screen( LIGHTBLUE, RED );
20:
21:      write_string("The screen is cleared....",
22:                      LIGHTBLUE, RED, 10, 28);
23:      write_string("Press any key to continue",
24:                      LIGHTBLUE, RED, 12, 28);
25:
26:      getch();
27:
28:      restore_screen_area(screen_buffer);
29:      cursor( crow, ccol );
30:
31:      return 0;
32:  }
```

Output



Analysis

This listing should be enlightening if you have ever run a program that causes the screen to look like it was before you started. This program saves the screen when it starts and then restores it when it is complete. The `save_screen_area()` function and the `get_cursor()` function are called when the program starts. The `save_screen_area()` is used to save the entire screen by passing the standard height and widths. The `get_cursor()` function saves the cursor information.

Once the information is saved, no matter what the program does, you can return the screen to its original look at the end of the listing. In this program, the screen is cleared and a message is displayed. Even though all the information on the screen has been wiped out, you have retained a copy in `screen_buffer`. Line 28 restores the screen with `restore_screen_area()`. The cursor is then placed back to its location before the program ends.

DO	DON'T

DON'T forget to turn the cursor back on before exiting your program if you turned it off.

DO hide the cursor if there aren't any entry fields on your screen (such as when you ask the user to press any key to continue).

DO call `restore_screen_area()` or `free()` if you call `save_screen_area()` so that you release the memory allocated to save the screen.

Summary

This chapter contained several functions that you will find useful. The first functions presented are used to hide and show the cursor. These functions are valuable when you are displaying screens that don't have any enterable fields. While some compilers have functions to clear the screen, not all do. A function was presented that enables you to clear the screen. This `clear_screen()` function includes the capability to state what colors you want the screen cleared to. A grid function was also presented, which enables you to place grid boxes in your applications. The final functions presented are used to save and restore areas of the screen. These functions help you create overlapping items on your screens without loosing the underlying information.

Q&A

Q **What are some uses for the `grid()` function?**

A There are two main uses that you will see in the remainder of this book. One use is to give an application a textured background screen. The second is to give texture to shadows on boxes.

Q **What will happen if I use the `save_screen_area()` function, but never call the `restore_screen_area()` function?**

A The `save_screen_area()` function allocates memory dynamically. This memory must be freed at some point. The `restore_screen_area()` function does the freeing when it redraws the screen area. You may need the memory elsewhere; however, it will remain unavailable until it is freed. If you decide you don't want to restore the screen area, you can use the `free()` function to free the screen buffer area that was allocated.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Why would you want to hide the cursor?
2. What happens if you call the `cursor_off()` function and then exit your program? Will the cursor automatically come back on?
3. Why do you need a `clear_screen()` function?
4. What is the difference between Borland's `clrscr()` function and your `clear_screen()` function?
5. How many different ASCII grid patterns are there?
6. What are the numerical values of the ASCII grid characters?
7. What is the benefit of saving the screen?
8. What happens if you don't restore a saved screen?
9. Why are the row and column positions stored in the `screen_buffer` along with the screen data in the `save_screen_area()` function?
10. Why are two bytes allocated for each position on the screen instead of just one?

Exercises

1. Add the new functions that you created today to your TYAC.LIB library. In addition, add the prototypes for these functions to the TYAC.H header. The new functions from today are as follows:

```
cursor_off()
cursor_on()
clear_screen()
grid()
save_screen_area()
restore_screen_area()
```

2. In the analysis of Listing 11.3, you were asked to comment out various lines of the listing. Try commenting out the lines presented in each of the following scenarios to see what happens.

   a. Comment out the `cursor_on()` line.

   b. Comment out the `cursor_off()` line.

   c. Comment out both the `cursor_on()` and `cursor_off()` lines.

3. Use the save and restore screen area functions in a program.

4. What happens if you keep calling the save screen function and then never restore the screen? Write a program that calls the `save_screen_area()` function over and over without restoring or freeing the allocated area.

5. **ON YOUR OWN:** Write a function that takes a character string as a parameter. Display this message in a box in the center of the screen. The box should have a grid shadow. In addition, the user should be asked to press a key. Once the key is pressed, the box should be removed and the screen should appear as it was before the message and box were displayed.