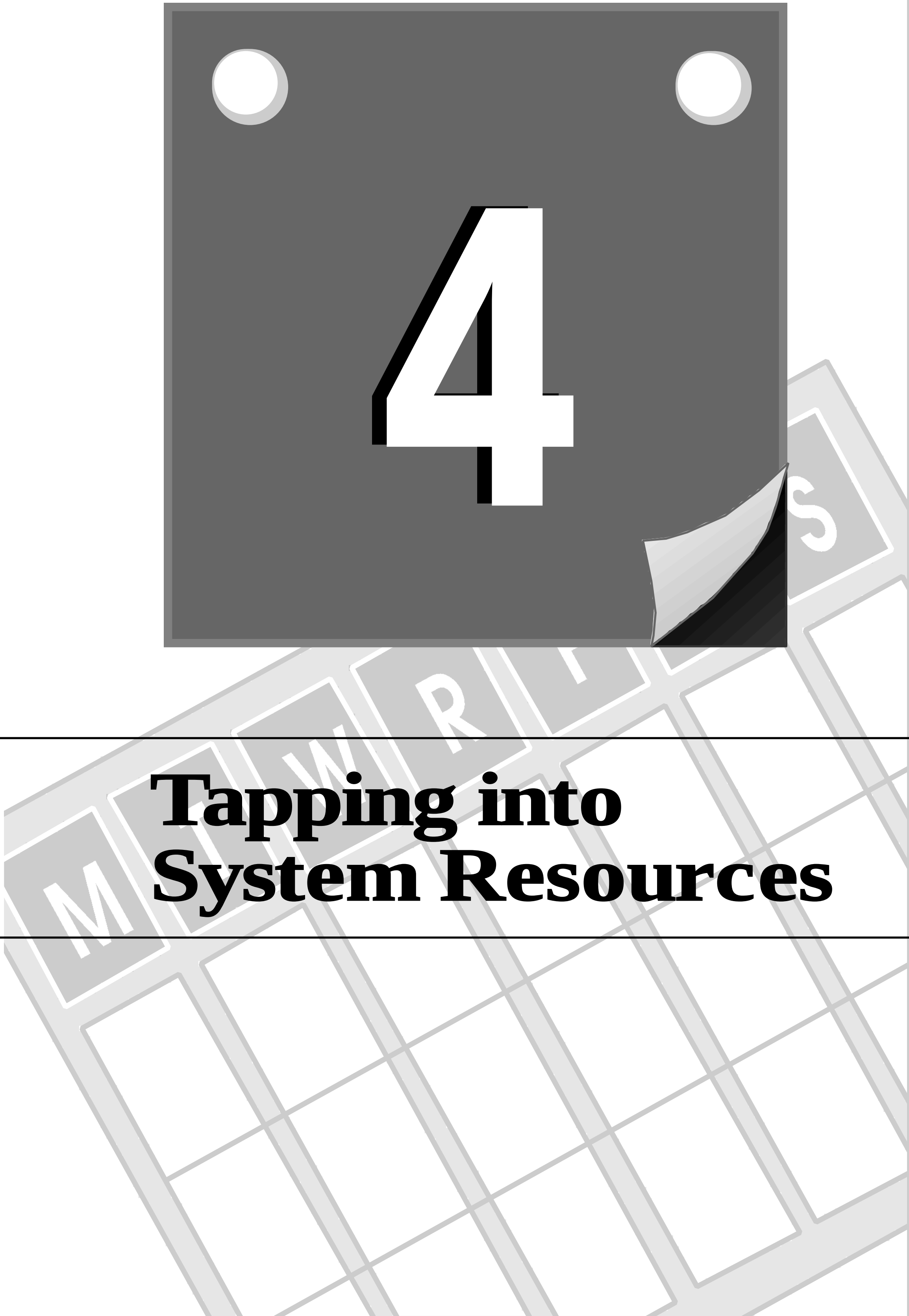


4

Tapping into System Resources

Today you will be introduced to many concepts that are not in most beginning C books. Several of today's topics will help bring your C programs to a new level. In addition, after today's lessons, you will have several useful functions. Today you will learn:

☐ The downside of tapping into system resources.

☐ The various methods for taking advantage of system resources.

☐ How to use the ANSI driver to control the screen.

☐ How to change colors, key values, and the cursor's position.

☐ What direct video memory updating is.

☐ What BIOS is and how it differs from the ANSI functions.

What Are System Resources?

System resources are resources provided by your computer system that add functionality. Such resources help you accomplish many tasks that would otherwise be nearly impossible. These tasks could include working with the video display, loading fonts, accessing disk drives, determining memory size, accessing the keyboard, reading a joystick, and much more.

System resources can't be used without any concerns. The cost of using system resources can vary. The largest concern with using system resources should be with portability. Depending on which system resources you access, you could greatly limit the portability of your programs. The resources that will be presented today can be found on most IBM-compatible machines running MS/DOS or an operating system that supports MS/DOS. A different computer platform, such as a Macintosh, may not be able to run the programs presented.

Working with the Display

One of the characteristics of C is its flexibility. Typically, there are several ways to accomplish similar tasks. Each method has its own pros and cons. This is especially true when working with system resources. There are various system resources that can be used to work with information on a display. Three areas will be examined along with some of their individual pros and cons. These areas are:

- ☐ Using ANSI functions
- ☐ Using direct memory access
- ☐ Using BIOS

The first two areas will be covered today, and the third will be covered in detail on Day 8, “Tapping System Resources via BIOS.”

Using ANSI Functions

ANSI stands for American National Standards Institute. ANSI sets standards for more than just computer languages. ANSI is often mentioned in describing the standards set for the C language. The ANSI standards go beyond just the C programming language. The ANSI committee is devoted to developing standards for any area that will promote the productivity and international competiveness of American enterprises.

The ANSI terminal standards can be used on an IBM-compatible computer that has loaded the ANSI.SYS system driver. The ANSI.SYS driver comes with Microsoft and PC DOS. Once installed, the ANSI system driver enables the computer to use functions that allow for cursor movement, display extended graphics, and redefine key values. To install the ANSI.SYS driver, consult the manuals that came with your computer’s operation system.

The Pros and Cons of Using the ANSI Functions

There are pros and cons to using the ANSI functions. The most obvious benefit is that using the ANSI functions is relatively simple. Once the driver has been installed, the functions are easily called. Later today, you’ll see how easy this is. Another benefit is that the ANSI functions are well documented. Since the ANSI driver generally comes with the computer’s operating system, there is usually an abundance of documentation. The commands that you learn later today can be used in your C programs. They can also be used in other languages, or in native operation system commands such as PROMPT.

Using ANSI functions isn’t without a downside. The biggest problem comes from running on a system that doesn’t support the ANSI functions. If the program doesn’t support the ANSI functions, or if the ANSI.SYS driver has not been loaded, then gibberish may be displayed on the screen. It’s this reliance on the ANSI.SYS driver that causes most programmers to avoid the ANSI functions. That not all operating systems support the ANSI terminal functions should be a factor when considering whether to use them.

In the following sections, several tables list ANSI controls and functions. These functions all require the ANSI.SYS driver to be loaded on your computer to operate properly. Following is an example of the line in a CONFIG.SYS file that loads the ANSI driver.

```
DEVICE=C:\DOS\ANSI.SYS
```

You should consult your computer's operating system manuals before modifying the CONFIG.SYS.

The ANSI Functions

The ANSI functions aren't really functions. Instead they are really escape sequences to control the system's video screen and keyboard. An *escape sequence* is a series of ASCII characters. A complete table of the individual ASCII characters is presented in Appendix B.

The first ASCII character in the escape sequence is either the escape character (value 27 or 1Bh) or the left-bracket character. The characters following the escape or left-bracket character determine what process occurs. The escape sequence may be upper- or lowercase letters depending on what the escape sequence does. This will be demonstrated later.

Note: A variety of functions will be created that use the ANSI functions. Later today and in later days, you'll learn other ways to accomplish the same task using better methods.

Escape Sequences

Several tables follow that contain the codes that will be used in the ANSI escape sequences. Tables 4.1 and 4.2 present the ANSI color codes.

Table 4.1. ANSI foreground colors.

Code	Color
30	Black
31	Red
32	Green

Code	Color
33	Yellow
34	Blue
35	Magenta
36	Cyan
37	White

Table 4.2. ANSI background colors.

Code	Color
40	Black
41	Red
42	Green
43	Yellow
44	Blue
45	Magenta
46	Cyan
47	White

By themselves, the values presented in these tables don't make a great deal of sense; however, later you'll see how to use these codes. Table 4.3 presents the ANSI text control characters that can also be used. These can be used in conjunction with the ANSI colors. Table 4.4 provides video mode controls that can be used to change the width or type of the video mode.

Table 4.3. The ANSI text controls.

Code	Control
0	All attributes off
1	Bold on

continues

Table 4.3. continued

Code	Control
4	Underscore
5	Blink on
7	Reverse video on
8	Concealed on

Table 4.4. Setting video mode.

Control	Resolution	Type
0	40×148×25	monochrome text
1	40×148×25	color text
2	80×148×25	monochrome text
3	80×148×25	color text
4	320×148×200	4-color graphics
5	320×148×200	monochrome graphics
6	640×148×200	monochrome graphics
7	(enables line wrapping)	
13	320×148×200	color graphics
14	640×148×200	color 16-color graphics
15	640×148×350	monochrome 2-color graphics
16	640×148×350	color 16-color graphics
17	640×148×480	monochrome 2-color graphics
18	640×148×480	color 16-color graphics
19	320×148×200	color 256-color graphics

Table 4.5 presents the ANSI character values that can be used.

Table 4.5. ANSI character values.

Key	Code	SHIFT +code	CTRL +code	ALT +code
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;86	0;96	0;106
F4	0;62	0;87	0;97	0;107
F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
F11	0;133	0;135	0;137	0;139
F12	0;134	0;136	0;138	0;140
HOME (num keypad)	0;71	55	0;119	—
UP ARROW (num keypad)	0;72	56	(0;141)	—
PAGE UP (num keypad)	0;73	57	0;132	—
LEFT ARROW (num keypad)	0;75	52	0;115	—
RIGHT ARROW (num keypad)	0;77	54	0;116	—
END (num keypad)	0;79	49	0;117	—
DOWN ARROW (num keypad)	0;80	50	(0;145)	—
PAGE DOWN (num keypad)	0;81	51	0;118	—
INSERT (num keypad)	0;82	48	(0;146)	—
DELETE (num keypad)	0;83	46	(0;147)	—
HOME	(224;71)	(224;71)	(224;119)	(224;151)
UP ARROW	(224;72)	(224;72)	(224;141)	(224;152)

continues

Table 4.5. continued

Key	Code	SHIFT +code	CTRL +code	ALT +code
PAGE UP	(224;73)	(224;73)	(224;132)	(224;153)
LEFT ARROW	(224;75)	(224;75)	(224;115)	(224;155)
RIGHT ARROW	(224;77)	(224;77)	(224;116)	(224;157)
END	(224;79)	(224;79)	(224;117)	(224;159)
DOWN ARROW	(224;80)	(224;80)	(224;145)	(224;154)
PAGE DOWN	(224;81)	(224;81)	(224;118)	(224;161)
INSERT	(224;82)	(224;82)	(224;146)	(224;162)
DELETE	(224;83)	(224;83)	(224;147)	(224;163)
PRINT SCREEN	—	—	0;114	—
PAUSE/BREAK	—	—	0;0	—
BACKSPACE	8	8	127	(0)
ENTER	13	—	10	(0
TAB	9	0;15	(0;148)	(0;165)
NULL	0;3	—	—	—
A	97	65	1	0;30
B	98	66	2	0;48
C	99	66	3	0;46
D	100	68	4	0;32
E	101	69	5	0;18
F	102	70	6	0;33
G	103	71	7	0;34
H	104	72	8	0;35
I	105	73	9	0;23
J	106	74	10	0;36
K	107	75	11	0;37

Key	Code	SHIFT +code	CTRL +code	ALT +code
L	108	76	12	0;38
M	109	77	13	0;50
N	110	78	14	0;49
O	111	79	15	0;24
P	112	80	16	0;25
Q	113	81	17	0;16
R	114	82	18	0;19
S	115	83	19	0;31
T	116	84	20	0;20
U	117	85	21	0;22
V	118	86	22	0;47
W	119	87	23	0;17
X	120	88	24	0;45
Y	121	89	25	0;21
Z	122	90	26	0;44
1	49	33	—	0;120
2	50	64	0	0;121
3	51	35	—	0;122
4	52	36	—	0;123
5	53	37	—	0;124
6	54	94	30	0;125
7	55	38	—	0;126
8	56	42	—	0;126
9	57	40	—	0;127
0	48	41	—	0;129

continues

Table 4.5. continued

Key	Code	SHIFT +code	CTRL +code	ALT +code
-	45	95	31	0;130
=	61	43	—	0;131
[	91	123	27	0;26
]	93	125	29	0;27
	92	124	28	0;43
;	59	58	—	0;39
‘	39	34	—	0;40
,	44	60	—	0;51
.	46	62	—	0;52
/	47	63	—	0;53
‘	96	126	—	(0;41)
ENTER (keypad)	13	—	10	(0;166)
/ (keypad)	47	47	(0;142)	(0;74)
* (keypad)	42	(0;144)	(0;78)	—
- (keypad)	45	45	(0;149)	(0;164)
+ (keypad)	43	43	(0;150)	(0;55)
5 (keypad)	(0;76)	53	(0;143)	—

**Table is from Microsoft DOS manual.

Types of ANSI Sequences

Using the values in the previous tables, you can accomplish a multitude of tasks. Each task, or function, uses a different sequence of characters. For example:

```
1B[r;cH
```

where `r` = a row on the screen

where `c` = a column on the screen

This escape sequence can be used to move the cursor position. The 1B is the value for ESCAPE, hence the phrase escape sequence. The left bracket begins the sequence and the rest of the sequence determines the specific functionality. To use the escape sequence, you simply print it to the screen. This can be done with the `printf()` function. Consider the following C functions in Listing 4.1.

Listing 4.1. CPUT.C. Place the cursor on the screen.

```
void put_cursor( int row, int col )
{
    printf( “\x1B[%d;%dH”, row, col );
}
```

This function moves the cursor to the position on the screen specified by `row` and `col`. The value printed in the `printf()` function is the ANSI escape sequence for moving the cursor. If the escape sequence was used without a row and column value, then the cursor would move to the top, left position (home) on the screen.

Moving the Cursor

Several escape sequences are available to move the cursor. For example, the escape sequence

```
1B[xA
```

moves the cursor up, toward the top of the screen. The cursor moves the number of lines specified by `x`. If the cursor is already at the top, or if the number is larger than the number of lines available to move, then the cursor stops at the top line. You can easily put this sequence into a more usable C function, as shown in Listing 4.2.

Listing 4.2. CUP.C. Move the cursor up.

```
void move_cursor_up( int nbr_rows )
{
    printf( “\x1B[%dA”, nbr_rows );
}
```

Following is an escape sequence that moves the cursor down:

```
1B[xB
```

This operates in the same manner as moving the cursor up. Listing 4.3 shows a function that makes use of this escape sequence.

Type

Listing 4.3. CDOWN.C. Move the cursor down.

```
void move_cursor_down( int nbr_rows )
{
    printf( “\x1B[%dB”, nbr_rows );
}
```

In addition to moving the cursor up and down, you may also want to move the cursor forward and backward. The escape sequence to move the cursor forward is:

```
1B[xC
```

The sequence to move the cursor backward is:

```
1B[xD
```

Each of these sequences attempts to move the cursor. Like the previous functions, if the function attempts to move the cursor beyond the edge of the screen, the cursor stops at the edge. Listings 4.4 and 4.5 provide two C functions that are more usable.

Type

Listing 4.4. CRIGHT.C. Move the cursor right.

```
void move_cursor_right( int nbr_col )
{
    printf( “\x1B[%dC”, nbr_col );
}
```

Type

Listing 4.5. CLEFT.C. Move the cursor left.

```
void move_cursor_left( int nbr_col )
{
    printf( “\x1B[%dD”, nbr_col );
}
```

With these five functions, you have everything you need to control the movement of the cursor. There are two additional cursor functions that may be useful. These are functions to save and restore the cursor.

Use the escape sequence

```
1B[s
```

to save the current cursor position. Only the last position can be saved. If you call on this function more than once, only the last position will be remembered.

You can restore the saved cursor position by using the following escape sequence:

```
1B[u
```

This puts the cursor back to the position saved. Listing 4.6 and Listing 4.7 contain two C functions that make using these escape sequences easier.

Type **Listing 4.6. SAVECURS.C. Save the current cursor position.**

```
void save_cursor_position( void )
{
    printf("\x1B[s");
}
```

Type **Listing 4.7. RSTRCURS.C. Restore the saved cursor position.**

```
void restore_cursor_position( void )
{
    printf("\x1B[u");
}
```

Erasing the Screen

Several escape sequences are available for erasing either part or all of the screen. The types of functions and the escape sequences available to accomplish them follow.

To erase the entire screen:

```
1B[2J
```

This escape sequence clears the entire screen. When completed, the cursor is placed in the top, left position on the screen. Listing 4.8 shows a function for using this escape sequence.

Listing 4.8. CLRSCRN.C. Clear the screen.

```
void clear_screen( void )
{
    printf(“\x1B[2J”);
}
```

The `clear_screen()` function clears the entire screen. Sometimes there is a need to clear only portions of the screen. The following escape sequence clears the characters on a line starting at the position of the cursor:

```
1B[K
```

The following function in Listing 4.9 uses this escape sequence to clear to the end of the line. You should note that the character at the location of the cursor is also cleared.

Listing 4.9. CLEAREOL.C. Clear to the end of the line.

```
void clear_eol( void )
{
    printf( “\x1B[K” );
}
```

All of these cursor functions are easily used. Listings 4.1 and 4.2 pull these functions into files called A_CURSOR.C and A_CURSOR.H respectively. These files contain all the functions shown up to this point. By compiling the A_CURSOR.C file along with the A_CURSOR.H file, you can use them in all of your programs without retyping them. Listing 4.3 is a program that illustrates the use of several of these functions. Notice that this program doesn't include the code for each of the functions used. Instead, it includes the header file with the prototypes, A_CURSOR.H. When you compile Listing 4.3, you'll want to also compile A_CURSOR.C. This can be done as follows:

```
TCC LIST0403.C A_CURSOR.C
```

You should replace TCC with the appropriate compile command for your compiler. If you are using an integrated development environment, you should have both files open.

Note: If you decide to use the ANSI functions in many of your programs, you may want to create a library containing all of them. Instructions on creation and use of a library are provided on Day 7, “Using Libraries.”

Type

Listing 4.10. A_CURSOR.C. The ANSI cursor functions.

```
/* Program: A_CURSOR.c
 * Author:  Bradley L. Jones
 * Purpose: Source file for a multitude of ANSI cursor
 *          functions.
 *=======================================================*/

#include “a_cursor.h”
#include <stdio.h>

 /*-----------------------------*
  *         The Functions       *
  *-----------------------------*/
/*** put the cursor on the screen ***/
void put_cursor( int row, int col )
{
    printf( “\x1B[%d;%dH”, row, col );
}

/*** move the cursor up ***/
void move_cursor_up( int nbr_rows )
{
    printf( “\x1B[%dA”, nbr_rows );
}

/*** move the cursor down ***/
void move_cursor_down( int nbr_rows )
{
    printf( “\x1B[%dB”, nbr_rows );
}

/*** move cursor to the right ***/
void move_cursor_right( int nbr_col )
{
    printf( “\x1B[%dC”, nbr_col );
}

/*** move the cursor to the left ***/
void move_cursor_left( int nbr_col )
{
    printf( “\x1B[%dD”, nbr_col );
}

/*** Save the cursor’s position ***/
void save_cursor_position( void )
{
    printf(“\x1B[s”);
}

```

continues

Listing 4.10. continued

```
/*** Restore the cursor’s position ***/
void restore_cursor_position( void )
{
    printf(“\x1B[u”);
}

/*** clear the screen ***/
void clear_screen( void )
{
    printf(“\x1B[2J”);
}

/*** clear to end of line ***/
void clear_eol( void )
{
    printf( “\x1B[K” );
}
```

Listing 4.11. A_CURSOR.H. The ANSI cursor functions header file.

```
/* Program: A_CURSOR.H
 * Author:  Bradley L. Jones
 * Purpose: Header file for the multitude of ANSI cursor
 *          functions.
 *=======================================================*/

/*-------------------------*
 *   Function prototypes   *
 *-------------------------*/

void put_cursor( int row, int col );
void move_cursor_up( int nbr_rows );
void move_cursor_down( int nbr_rows );
void move_cursor_right( int nbr_col );
void move_cursor_left( int nbr_col );
void save_cursor_position( void );
void restore_cursor_position( void );
void clear_screen( void );
void clear_eol( void );

/*---------- end of file ----------*/
```

Listing 4.12. LIST0403.C. Using the ANSI cursor functions.

```
/* Program: LIST0403.c
 * Author:  Bradley L. Jones
 * Purpose: Demonstrates ANSI cursor escape sequences.
 *=======================================================*/

#include <stdio.h>
#include “a_cursor.h”

/*** Function prototypes ***/
void box( int ul_row, int ul_col, int lr_row, int lr_col,
          unsigned char ch );

void main(void)
{
   int row,
       column,
       x,
       y;

   save_cursor_position();
   clear_screen();

   box( 1, 18, 3, 61, ‘*’ );

   put_cursor( 2, 21 );
   printf( “   THIS IS AT THE TOP OF THE SCREEN   “ );

   box( 15, 20, 19, 60, 1 );
   box( 16, 24, 18, 56, 2 );

   restore_cursor_position();
}

void box( int ul_row, int ul_col, int lr_row, int lr_col,
          unsigned char ch )
{
   int x, y;

   if( (ul_row > lr_row) || (ul_col > lr_col) )
   {
       printf( “Error calling box.” );
   }
   else
   {
      for( x = ul_row; x <= lr_row; x++ )
      {
         put_cursor( x, ul_col);
```

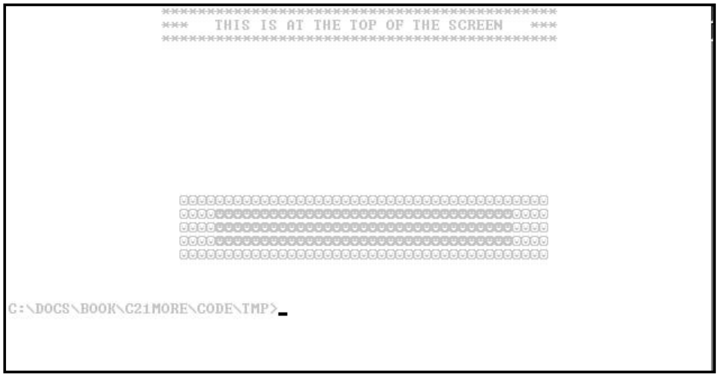
continues

Listing 4.12. continued

```

            for( y = ul_col; y <= lr_col; y++ )
            {
               printf( "%c", ch );
            }
         }
      }
   }
```

Note: As stated earlier, to use the ANSI functions, the ANSI.SYS driver must be loaded, otherwise, you won't get the expected results.

Output



```
          ********************************************
          ***   THIS IS AT THE TOP OF THE SCREEN   ***
          ********************************************







           ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺
           ☺☺☺☺☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☺☺☺☺
           ☺☺☺☺☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☺☺☺☺
           ☺☺☺☺☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☻☺☺☺☺
           ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺



C:\DOCS\BOOK\C21MORE\CODE\TMP>_
```

Analysis

As you can see, working with the ANSI functions gives you a great deal of control over the output. Listing 4.1 is a listing that you'll want to build on as you learn more ANSI functions later today. This listing, A_CURSOR.C, contains all of the ANSI cursor functions. Line 7 includes a header file called A_CURSOR.H which is presented in Listing 4.2. The A_CURSOR.H header file contains function prototypes for all of the functions in Listing 4.1. You'll want to include the A_CURSOR.H header file in any programs that include the ANSI cursor functions. Listing 4.3 is an example of one such program.

Listing 4.3 is a fun program. It uses the ANSI function that you have learned to create a new function called `box()`. Line 10 contains the prototype for the `box()` function. As you can see, it takes several parameters. The parameters tell you where the box is to be located. The upper-left row (`ul_row`), the upper-left column (`ul_col`), the lower-right row (`lr_row`), and the lower-right column (`lr_col`) are all passed along with the character that the box is to be made with.

The `box()` function is defined in lines 33 to 53. This function used two `for` loops to create the box. Each line of the box is drawn one at a time. The cursor is placed at the beginning of each line using the `put_cursor()` functions. A `printf()` call then places each character on the line. Later today, this function will be enhanced to also include color.

There are a few other notables in this program. Line 19 uses the `save_cursor_position()` function to save the cursor position. Line 30 restores the position just before the program ends. Line 20 clears the screen using the `clear_screen()` function. Line 22 calls the `box()` function which creates a box centered in the top of the screen. Line 24 places the cursor in the box using the `put_cursor()` function so line 25 can type a header into the middle of the box. Lines 27 and 28 call the box function two more times. The character with a decimal value of 1, a clear smiley face, is printed in the first call to `box()`. The second call to `box()` prints solid smiley faces in the middle of the original box.

ANSI and Color

You can use the ANSI driver to manipulate the screens colors also. Changing colors is a little more difficult than the functions shown before. Table 4.1 presented the foreground colors that can be used. Foreground colors are used on the information that is presented on the screen. This includes all the text. Table 4.2 presented the background colors that are available through the ANSI driver. This is the color that will be put on the screen behind the text.

Foreground and background colors are often used in conjunction with each other. Typically, you'll want to state what color is in the foreground and what color is in the background. When you use the ANSI colors, once you set them, they apply from that point on.

The ANSI command to set the color is as follows:

```
1B[a;b;...;nm
```

For this escape sequence, you can stack several commands at once. The parameters, *a*, *b*,...,*n* are each replaced with a command from Tables 4.1, 4.2, and/or 4.3. If conflicting commands are given, the final command is used. For instance calling for the foreground color black (30) followed by the foreground color red (31) would result in red. After all, you can't use two foreground colors at once. Don't be confused, you can have more than one foreground color on your screen, but you can't try to write a single character in both black and red at the same time. Listings 4.4 and 4.5

demonstrate the use of the ANSI color attributes. Listing 4.4 is a header file that is included in Listing 4.5. You'll need to remember to use the A_CURSOR source file that was used with Listing 4.3 since a few of the earlier ANSI functions are also used.

Type **Listing 4.13. ANSICLRS.H. The ANSI colors.**

```
/* Program: ANSICLRS.h
 * Author:  Bradley L. Jones
 * Purpose: Header file for ANSI colors
 *=======================================================*/

/*-------------------------*
 *    Foreground colors    *
 *-------------------------*/

#define F_BLACK   30
#define F_RED     31
#define F_GREEN   32
#define F_YELLOW  33
#define F_BLUE    34
#define F_MAGENTA 35
#define F_CYAN    36
#define F_WHITE   37

/*-------------------------*
 *    Background colors    *
 *-------------------------*/

#define B_BLACK   40
#define B_RED     41
#define B_GREEN   42
#define B_YELLOW  43
#define B_BLUE    44
#define B_MAGENTA 45
#define B_CYAN    46
#define B_WHITE   47

/*-------------------------*
 *        Attributes       *
 *-------------------------*/

#define  BOLD       1
#define  UNDERSCORE 4
#define  BLINK      5
#define  REVERSE    7
#define  CONCEAL    8

/*---------- end of file ----------*/
```

Type

Listing 4.14. LIST0405.C. Using the ANSI colors.

```
/* Program: LIST0405.c
 * Author:  Bradley L. Jones
 * Purpose: Demonstrates ANSI colors.
 *======================================================*/

#include <stdio.h>
#include “a_cursor.h”
#include “ansiclrs.h”

/*** Function prototypes ***/

void box( int ul_row, int ul_col,
          int lr_row, int lr_col,
          unsigned char ch,
          int fcolor, int bcolor );

void color_string( char *string, int fcolor, int bcolor );
void set_color( int fore, int back );

void main(void)
{
   int row,
       column,
       x,
       y;

   set_color( F_WHITE, B_MAGENTA );

   clear_screen();

   box( 3, 19, 5, 62, ‘ ‘, F_BLACK, B_BLACK);   /* shadow */
   box( 2, 18, 4, 61, ‘*’, F_YELLOW, B_BLUE );

   put_cursor( 3, 21 );
   color_string( “   THIS IS AT THE TOP OF THE SCREEN   “,
                 F_RED, B_BLUE );

   box( 16, 21, 21, 61, ‘ ‘, F_BLACK, B_BLACK ); /* shadow */
   box( 15, 20, 20, 60, ‘*’, F_RED, B_GREEN );

   set_color( F_WHITE, B_BLACK );
   put_cursor( 23, 0);
}

void box( int ul_row, int ul_col,
          int lr_row, int lr_col,
          unsigned char ch,
          int fcolor, int bcolor )
```

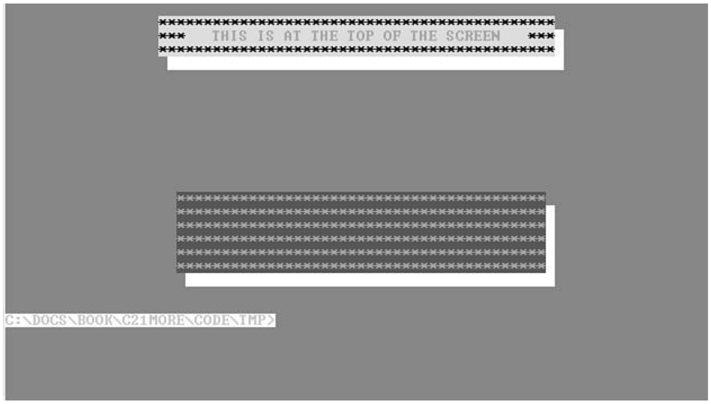
continues

Listing 4.14. continued

```
{
   int x, y;

   if( (ul_row > lr_row) || (ul_col > lr_col) )
   {
       printf( “Error calling box.” );
   }
   else
   {
      set_color( fcolor, bcolor);

      for( x = ul_row; x <= lr_row; x++ )
      {
         put_cursor( x, ul_col);

         for( y = ul_col; y <= lr_col; y++ )
         {
            printf( “%c”, ch );
         }
      }
   }
}

void color_string( char *string, int fcolor, int bcolor )
{
  set_color( fcolor, bcolor );
  printf( string );
}

void set_color( int fore, int back )
{
   printf(“\x1B[%d;%dm”, fore, back );
}
```

Note: Since this isn’t a full-color book, the output printed here isn’t in color. On your monitor (if it is colored), this output should appear in color.

Output



```
********************************************
***   THIS IS AT THE TOP OF THE SCREEN   ***
********************************************




  *****************************************
  *****************************************
  *****************************************
  *****************************************
  *****************************************
  *****************************************


C:\DOCS\BOOK\C21MORE\CODE\TMP>
```

Analysis The set color function could be added to the A_CURSOR.C program. In addition, the header file, A_CURSOR.H, could be set up to include the prototypes for the extra functions, `box()`, `set_color()`, and `color_string()`. You should combine all the ANSI functions together so that they are easily accessible. You may want to consider a better name for the files since there would be more than just cursor functions when the new function is added.

Listing 4.5 is similar to Listing 4.3. The main difference is color has been added to the output. Line 8 includes the ANSICLRS.H header file that is presented in Listing 4.4. This header file contains defined constants for each of the colors and other text attributes. By using defined constants such as these, your programs are much easier to read. It's not always intuitive to know that 31 means foreground red.

Several of the functions in Listing 4.5 are slightly modified from Listing 4.3. The `box()` function has two additional parameters, `fcolor` and `bcolor`. These are used to set the color of the box. Lines 31 and 32 demonstrate the use of the box function with the additional parameters. Lines 31 and 32 also illustrate a programming trick. Line 31 creates a black box. Line 32 then overwrites the black box with a colored box. The original black box becomes a shadow for the second box. This is how most shadows are created.

There is a second trick that has been used. Line 27 sets the color to white on magenta using the new function `set_color()`. Line 29 then calls the `clear_screen()`. You should notice that the screen is cleared to the last background color set, in this case magenta. The last color set also carries on after the program ends. For this reason, line 41 sets the color to something usable—white on black.

Most of the code presented should be easy to follow. The difficult areas were covered when Listing 4.3 was analyzed. The only actual new code is the `set_color()` function in lines 78 through 81. As you can see, the foreground and background colors are passed to a `printf()` statement. As before, an escape sequence is used.

In addition to color and cursor placement, you can change the mode of the monitor also. Table 4.4 displayed the monitor modes that can be used. Be careful using these modes; there is no guarantee that every monitor will support all modes.

Other ANSI Functions

The ANSI functions also give you the ability to redefine the keys on the keyboard. Table 4.5 can be used to accomplish this. There are several reasons to redefine characters. One is to re-map your keyboard to a different keyboard layout. Another reason might be to re-map keys a game uses. You might map the ‘A’ key to be the same as a right arrow, the ‘D’ to be a left arrow, the ‘S’ to be a down arrow, and the ‘W’ to be an up arrow. This would allow a game player an option of which keys to use, the letters—which would be easier for a left-handed person—or the actual arrows.

Type

Listing 4.15. Illustrates re-mapping of keys using the ANSI escape sequences.

```
/* Program: LIST0406.c
 * Author:  Bradley L. Jones
 * Purpose: Demonstrates ANSI keyboard values.
 * Note:    Running this program will set the F1 key to
 *          display “Bradley”. It will set the F2 key to
 *          display “Jones”. Running the program with an
 *          extra command line parameter will reset the
 *          F1 and F2 keys.
 *=======================================================*/

#include <stdio.h>

#define F1  “0;59”
#define F2  “0;60”

int main(int argc)
{
   if( argc < 2 )
   {
      printf(“\x1B[%s;66;114;97;100;108;101;121p”,F1);
      printf(“\x1B[%s;74;111;110;101;115p”,F2);
   }
   else
   {
      printf(“\x1B[%s;%sp”,F1, F1);
26:          printf(“\x1B[%s;%sp”,F2, F2);
27:       }
28:
29:       return;
30:    }
```

Output After this program runs, there is no output.

Analysis This program appears to do nothing when executed because there isn’t any output. However, after the program runs, the F1 and F2 keys will function differently. Run the program and then press the F1 and F2 keys. Your program has now changed their functionality to print “Bradley” and “Jones” instead of performing their normal functions. If you re-execute the program and pass a parameter as follows:

```
LIST0406 X
```

then the program will reset the F1 and F2 keys to their original functions.

As shown in the listing, this isn’t a complex program. Lines 13 and 14 define the F1 and F2 keys to make them easier to work with. Line 16 begins the `main()` function. Since we are going to use a command line parameter to toggle the key values, we need to receive the `argc` variable. The `argc` variable contains the number of items on the command line including the program currently running. If a program runs without any command line parameters, then the `argc` variable will contain 1. Line 18 checks the value. If it is 1—less than 2—then the values of F1 and F2 are re-mapped (lines 20 and 21). The re-mapped values are stacked following the original value to be reassigned. In this case, there are several values being assigned to the F1 and F2 keys. The reassignment values stop when the letter ‘p’ is reached. Lines 25 and 26 are executed when there is an additional value on the command line. In these cases, the values of F1 and F2 are mapped to themselves. This, in effect, resets them to their original values.

In today’s exercises, you’ll be asked to write a program similar to this one. Instead of having the function keys print text such as your name, you’ll assign DOS commands to them.

The Extended ASCII Characters

The ANSI functions are often combined with the extended ASCII character set. The extended character set is considered to be the characters from 128 to 255. These characters include special type characters, line characters, and block characters. Many of these characters are used to create boxes and grids that can be used on the screen. Appendix A contains an ASCII chart that shows all of the different characters available. Later in the book, you'll develop an application that uses many of the extended characters to create a user-friendly screen.

Using Direct Memory Access

Memory is set aside for use by the video display. This memory can be accessed directly to manipulate the graphics or characters that are on the screen. Because this is memory that is directly mapped to the video display, a change can be seen instantly. By updating the video display's memory directly, you can gain the fastest screen updates.

This speed comes at the cost of portability. The memory reserved for the video display isn't always in the same location. In an IBM-compatible computer system, a part of the memory between 640K and 1M is reserved for the video display. Portability is lost because the area reserved isn't always guaranteed to be the same from computer system to system. To use this direct video memory, the system must be 100-percent IBM-compatible with an IBM PC's hardware. It's safe to assume that the same brand of computer with the same type of hardware will have video memory stored in the same location. It's not safe to assume that all other computers will use the same location. In addition, memory for using a CGA monitor isn't always allocated in the same area that memory for a VGA monitor would be.

Note: Borland includes a variable called `directvideo`. If this variable contains a value of 1, then a program's video display activities go directly to the video memory. If the `directvideo` variable contains the value of 0, then BIOS is used. BIOS is converted in the next section. The default for `directvideo` is 1.

Because BIOS functions are more portable, they will be covered in this book. Direct video programming is beyond the scope of this book. If you are interested in direct video programming, consult a graphics book that has been written for your specific compiler.

What Is BIOS?

BIOS stands for Basic Input/Output System. Every MS/PC DOS computer operates with some form of BIOS. The BIOS is a set of service routines that are activated by software interrupts. A *software interrupt* is an interruption that causes the operating system (DOS) to respond. By going through these service routines, and therefore BIOS, you avoid interacting directly with the computer's hardware. This eliminates concerns, such as the possibility of different locations for video memory, because the BIOS determines where and what you need based on the interrupt you cause.

There are BIOS services for a multitude of different input and output activities. This includes being able to manipulate the screen, keyboard, printers, disks, mouse, and more. In addition, there are services available to manipulate the system date and time. On Day 8, tables will be presented that detail many of the available interrupts. For now, it is more important to know that these functions exist.

It's better to use BIOS instead of direct memory video access or the ANSI functions. Direct memory access has a downside that has already been described—you don't know for sure where the video memory will be located. The downside of the ANSI functions is the external device driver; ANSI.SYS must be loaded for the functions to work. If you run Listing 4.3 without the device driver, you get the following result:

Output

```
 [s [2J
[1;18H***************************************_[2;18H*************
***********************_[3;18H***********************************
--[2;21H--THIS IS AT THE TOP OF THE SCREEN--[15;20H------------
----------------[16;20H------------------------------[17;20H---
-------------------------------[18;20H-------------------------
---[19;20H----------------------------------[16;24H------------
----------- [17;24H----------------------------[18;24H---------
---------------[u----------------------------------------------
```

This isn't the desired result. By going through the BIOS, you don't need external device drivers, nor do you have to determine where video memory is located. The BIOS takes care of that for you.

While all of this makes the BIOS calls sound like the perfect answer, there is a downside to using BIOS also. The speed of going through BIOS isn't going to be as good as accessing video memory directly. You should note that these speeds are both extremely fast. In addition, using the BIOS isn't going to be as easy as using the ANSI functions. Neither of these negatives outweighs the additional portability that you gain by using the BIOS functions.

Summary

Today, you were provided with a great deal of information that can be fun. You were shown how to use the ANSI.SYS driver to manipulate the screen. This included learning how to place the cursor, clear the screen, change the colors, and re-map keyboard values. In addition, you were given an overview of writing directly to video memory. The day ended with a high-level discussion on BIOS, which will resume on Day 8. On Day 8, you'll be presented with many examples, along with a list of the many BIOS interrupt functions that are available.

Q&A

Q Are the functions learned today portable to other computer systems?

A The functions covered in today's materials are portable to some computers. The ANSI functions are portable to any computer system that supports the ANSI terminal standards. The BIOS functions are portable to computers that are 100-percent compatible with IBM BIOS. In addition, older versions of BIOS may not support all of the functions. You can consult your DOS manuals and system documentation to determine what interrupts your computer supports.

Q ANSI functions are simple to use. Why are they not recommended?

A If you ran today's ANSI functions without the ANSI.SYS driver loaded, then you already know the answer to this question. By using the ANSI functions, you are reliant upon an external factor—the ANSI driver.

Q What is meant by BIOS functions being portable?

A The portability of BIOS functions isn't necessarily the same as the portability that will be discussed later in this book. The calls to BIOS functions aren't necessarily portable C code—each compiler may call interrupts slightly differently. What is meant by portability is that an executable program (.EXE or .COM) will have a better chance of running on many different computer configurations than if you use ANSI functions or direct memory writes. BIOS function calls are only portable to IBM-compatible machines; however, they can have a multitude of different video monitors, modems, printers, and so on.

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. What does ANSI stand for?
2. Why is it important to know what ANSI is?
3. What is a reason for using ANSI functions?
4. What is a reason not to use ANSI functions?
5. What is the value of red?
6. What is the difference between a foreground color and a background color?
7. When does an ANSI color quit being applied?
8. What is the benefit of using direct video memory updates?
9. What is the downside of using direct video memory updates?
10. What does BIOS stand for?

Exercises

1. What does the following do?

   ```
   "\x1B[5A"
   ```

2. What are the values of the following color sets?

   Black on White

   White on Black

   Yellow on Blue

   Yellow on Red

3. What is the escape sequence for the following colors?

   Yellow on Blue

   Bright Yellow on Red

4. **BUG BUSTER:** What, if anything, is wrong with the following?

   ```
   /* ANSI escape sequence to set the color */

   printf(“\x1B[31;37m”);
   ```

5. **BUG BUSTER:** What, if anything, is wrong with the following?

   ```
   void clear_entire_line( int row, int fcolor, int bcolor)
   {
      put_cursor( row, 1);
      set_color( fcolor, bcolor );
      clear_eol();
   }
   ```

6. Write a function called `put_color_string()` that takes screen coordinates, a foreground color, a background color, and a string as parameters. The function should print the string at the provided coordinates in the given colors.

7. Write a program that re-maps the Shifted function keys. The values you assign should be the most common DOS commands that you use. This makes using these DOS commands just a little easier. (See Listing 4.6 if you need help.)

   Following are some examples of values you might use:

   Shift F1HELP
   Shift F2DIR
   Shift F3CLS
   Shift F4CHKDSK
   Shift F5CD C:\ID\T7G
     * *Or whatever game you most often run.*

8. **ON YOUR OWN:** Write a function that creates a box using the extended characters in the ASCII Table.

9. **ON YOUR OWN:** Write a function that centers a string on a line.

10. **ON YOUR OWN:** Write a function that enables the user to enter values that are then assigned to function keys. (Similar to Exercise 7, only interactive with the user.)