8

Tapping into System Resources via BIOS

WEEK
2

On Day 4, you learned how to take advantage of the ANSI functions. As you saw, the ANSI functions require the use of an external program to operate correctly. Today you'll learn a better way of manipulating the resources of your system by doing the following:

- ☐ Review what was covered on Day 4.
- ☐ Review what BIOS is.
- ☐ Learn what an interrupt is.
- ☐ Get an overview of the multitude of BIOS functions available.
- ☐ See how to use some of the interrupt services to work with your system.

System Resources in Review

System resources are resources provided by your computer system that enable you to gain functionality by calling on them. By using system resources, you can create programs that have much more use. In addition, some tasks would be nearly impossible without them. Tasks that can be accomplished include working with the video display, loading fonts, accessing disk drives, determining memory size, accessing the keyboard, reading a joystick, and much more.

As stated on Day 4, system resources can't be used without any concerns. The cost of using system resources can vary. On Day 4, you learned that the largest concern with using system resources should be portability. Depending on which systems 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, won't be able to run the programs presented today.

Working with the Display

On Day 4, you were told that one of the characteristics of C is that it is flexible. Typically, there are several ways to accomplish similar tasks. Each method that can be implemented has its own pros and cons. Also on Day 4, you were presented with three different methods of accessing or manipulating system resources. In particular, you were shown how to manipulate the computer's video monitor. The three areas that were presented were:

- ☐ The ANSI functions
- ☐ Direct memory access
- ☐ BIOS

A Review of Using ANSI Functions

ANSI stands for American National Standards Institute. There are several different areas of standards created by ANSI. 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 redefining key values. Using the ANSI.SYS driver was presented in detail on Day 4.

As you learned on Day 4, there are both pros and cons to using the ANSI functions. The most obvious benefit is that using the ANSI functions is extremely simple. Once the ANSI.SYS driver has been installed, the functions are easily called. Another benefit of the ANSI functions is that they are well documented. Because the ANSI driver generally comes with the computer's operating system, there is usually an abundance of documentation.

As you may have seen, using ANSI functions isn't without its downside. The most negative impact is when you use an ANSI function on a system that doesn't support the ANSI terminal functions. If the program doesn't support the ANSI functions, or if the ANSI.SYS driver hasn't been loaded, then gibberish may be displayed on the screen. Because of this reliance on the ANSI.SYS driver, most programmers choose to avoid the ANSI functions. The fact that not all operating systems support the ANSI terminal emulation functions could also impact a decision to use ANSI functions.

A Review of Using Direct Memory Access

Direct memory access was also discussed on Day 4. 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 to the memory can be seen instantly on the screen. By updating the video display's memory directly, you can gain the fastest screen updates.

This fast speed comes at the cost of portability. The memory set aside for the video display isn't always in the same locations. While it's safe to assume that the memory will be set aside, it's not safe to assume where. Portability is lost because the area set aside is not always guaranteed to be the same from computer system to computer 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 isn't safe to assume that all other computers will use the same location—not even all IBM-compatible systems. In addition, memory for using a CGA monitor isn't always allocated in the same area that memory for a VGA monitor would be.

What Is BIOS?

The use of BIOS was also mentioned on Day 4 as the third alternative for manipulating system resources. BIOS stands for Basic Input/Output System. Every IBM-compatible MS/PC DOS computer operates with BIOS. *BIOS* is a set of service routines that are activated by software interrupts. A *software interrupt* is an interruption caused by the currently running program 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, the keyboard, printers, disks, mouse, and more. In addition, there are services available to manipulate the system date and time. Tables will be presented that will detail many of the available interrupts. For now, it's more important to know how to use them and why.

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 that the external device driver, ANSI.SYS, must be loaded for the functions to work.

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 fast as accessing video memory directly. 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. The speed difference between BIOS and direct memory is negligible for most applications. In addition, once you create a BIOS function, you can store it in your own library and never have to worry about the underlying code. Another problem with BIOS is the portability of accessing the interrupts. Different compilers use different commands. The difference in these commands is minimal between most compilers.

DO	DON'T
DO understand the differences among ANSI, direct memory, and BIOS functions.	**DON'T** use BIOS functions if you plan to port your code to computers that are not IBM-compatible.

Using BIOS

To work with BIOS, you simply set up any needed information and then call the appropriate interrupts number. When you call an interrupt, you pass information within registers. Don't worry about registers at this time. In addition to passing interrupt numbers, you may also need to pass function numbers. *Function numbers* are more specific instructions to BIOS. Table 8.1, presented later today, lists some of the major interrupts and their functions. For instance, interrupt 16 (0x10h in hex) is an interrupt for video display functions. There are several different video display functions that can be used. Table 8.2, which is also presented later today, lists several of the specific functions. For instance, interrupt 0x10h (video display processes) used with function 0x02h will set the cursor position. In some cases, there are even lower breakdowns of functions into *subfunctions*. For example, the video interrupt 0x10h has a function to work with the color pallet (0x10h), which has several subfunctions. Subfunction 0x01h will set the screen's border color.

Using the interrupt calls requires setting up information first. Following are two structures that can be used to pass information to BIOS. These two structures are followed by a union that combines the structures into one. This set of structures and the related union are given as they appear in both the Borland and Microsoft compilers:


```
struct WORDREGS {
    unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS {
    unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};

union   REGS    {
    struct  WORDREGS x;
    struct  BYTEREGS h;
};
```


```
/* word registers */

struct _WORDREGS {
    unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
    unsigned int cflag;
    };

/* byte registers */

struct _BYTEREGS {
    unsigned char al, ah;
    unsigned char bl, bh;
    unsigned char cl, ch;
    unsigned char dl, dh;
    };

/* general purpose registers union -
 *  overlays the corresponding word and byte registers.
 */

union _REGS {
    struct _WORDREGS x;
    struct _BYTEREGS h;
    };
```

As you can see, regardless of the compiler, these are set up in the same manner. Naming may be a little different so you will want to check your compiler for specific names to use. Within the examples presented in this book, the following code fragment will be included. Even if you have the Borland or Microsoft compilers, you can include the following, or if you wish, you can use the structures declared in your compiler.

Note: The structures shown previously are defined in the DOS.H header file in the include directory of the corresponding compiler. If your compiler is ANSI-compatible, it should have similar declarations in its DOS.H header file.

Type

Listing 8.1. BIOSREGS.H.

```
1:  /* Header:  BIOSREGS.H
2:   * Purpose: Include generic structures for BIOS registers
3:   *-------------------------------------------------------*/
4:  #ifdef __BIOSREGS_H
5:  #define __BIOSREGS_H 1
6:
7:  struct XREG
8:  {
9:      unsigned int ax;
10:     unsigned int bx;
11:     unsigned int cx;
12:     unsigned int dx;
13:     unsigned int si;
14:     unsigned int di;
15:     unsigned int cflag;
16: };
17:
18: struct HREG
19: {
20:     unsigned char al, ah;
21:     unsigned char bl, bh;
22:     unsigned char cl, ch;
23:     unsigned char dl, dh;
24: };
25:
26: union REGS
27: {
28:     struct XREG x;
29:     struct HREG h;
30: };
31:
32: #endif
```

Analysis

This is just a header file so there isn't any output. As you can see, the XREG and HREG structures are virtually identical to what was presented from the Borland and Microsoft compilers earlier. This header file should be compatible with either of those compilers, along with any other DOS-based compiler. The REGS union

declared in lines 26 to 32 will be used when calling BIOS. You should be aware that XREG and HREG are used for different reasons. The use of these will be covered later. For now, remember that REGS is a union, which means that you can use XREG or HREG, but not both at the same time.

At this point, you may feel somewhat lost and confused. To help clear up what has been presented so far, here are a few examples. Listing 8.2 presents an example of using an interrupt to get the current date via an interrupt call to BIOS.

Listing 8.2. LIST0802. Using BIOS interrupt call.

```
1:  /* Program: LIST0802.c
2:   * Author:  Bradley L. Jones
3:   * Purpose: Demonstrates a BIOS function to get current
4:   *          date.
5:   *=========================================================*/
6:
7:  #include “biosregs.h”
8:  #include <dos.h>
9:  #include <stdio.h>
10:
11: /*** Function prototypes ***/
12: void current_date( int *month, int *day, int *year );
13:
14: void main(void)
15: {
16:   int month, day, year;
17:   printf(“\nDetermining the current date...”);
18:
19:   current_date( &month, &day, &year );
20:
21:   printf(“\n\nThe current date is: %d/%d/%d.”, month, day, year);
22: }
23:
24: void current_date( int *month, int *day, int *year)
25: {
26:     union REGS inregs, outregs;
27:     inregs.h.ah = 0x2a;
28:
29:     int86(0x21, &inregs, &outregs);
30:
31:     *month = outregs.h.dh;
32:     *day   = outregs.h.dl;
33:     *year  = outregs.x.cx;
34: }
```

Note: Line 29 uses a non-ANSI-compatible function. This means that some compilers may use a different name for the `int86()` function. Microsoft documents the use of `_int86()`; however, `int86()` works. Borland uses `int86()`. Consult your function reference for specific information on your compiler. For the remainder of this book, the function name of `int86()` will be used; however, you should be able to replace it with your compiler's comparable command. In the case of the Microsoft compiler, simply add the underscore.

Output

```
Determining the current date...

The current date is: 12/22/1993.
```

Analysis

When you run this program, you should end up with the current date on your screen instead of 12/22/1993. The date is received via an interrupt call. In this program, two values are declared to be of type REGS. Remember that REGS is the union that holds the register values (see Listing 8.1). The `inregs` union is used to hold the values going into the interrupt call. The `outregs` variable is used to hold the values being returned.

To get the date, the function number 0x2Ah (33 decimal) is used with a call to BIOS. The function number goes into the `ah` register. The `ah` register is a part of the `h` structure of the `inregs` REGS union. The function 0x2Ah is a function within interrupt 0x21h. Line 27 sets this function number into the `ah` register. Line 29 calls the BIOS interrupt using `int86()`. If you are using a Microsoft compiler, remember that you may need to use `_int86()` instead.

The `int86()` function passes the interrupt, the registers that are going into and coming out from BIOS. Once called, the values in the `outregs` variable can be used. Lines 31, 32, and 33 get the values for the day, month, and year from these `outregs` registers.

As you progress through the rest of this book, several functions will be developed using BIOS interrupt calls. On Day 7, you learned to work with libraries. Many of the BIOS functions that you create will be useful in many of your programs. You should create a library of all of your BIOS functions.

BIOS and the Cursor

On Day 4, you learned to use the ANSI.SYS driver to place the cursor at different locations on the screen. Following are two functions that are similar to two ANSI

functions used on Day 4. These two functions are `cursor()` in Listing 8.3, which places the cursor, and `get_cursor()` in Listing 8.4, which gets the cursor's current location. In addition, Listing 8.5 demonstrates the use of these two functions.

Listing 8.3. PCURSOR.C. Placing the cursor at screen coordinates.

```
1:  /* Program: PCURSOR.C
2:   * Author:  Bradley L. Jones
3:   * Purpose: Demonstrates a BIOS function to position the
4:   *          cursor on the screen.
5:   * Note:    This function places the cursor at a given
6:   *          location on the screen. The upper left position
7:   *          of the screen is considered (0,0) not (1,1)
8:   *========================================================*/
9:
10: #include "biosregs.h"
11: #include <dos.h>
12:
13: void cursor(int row, int column);
14:
15: void cursor( int row, int column)
16: {
17:     union REGS inregs;
18:
19:     inregs.h.ah = 0x02;
20:     inregs.h.bh = 0;
21:     inregs.h.dh = row;
22:     inregs.h.dl = column;
23:
24:     int86(0x10, &inregs, &inregs);
25: }
```

Listing 8.4. GCURSOR.C. Getting the coordinates of the cursor.

```
1:  /* Program: GCURSOR.C
2:   * Author:  Bradley L. Jones
3:   * Purpose: Demonstrates a BIOS function to get the position
4:   *          of the cursor on the screen.
5:   * Note:    This function considers the upper left position
6:   *          of the screen to be (0,0) not (1,1)
7:   *========================================================*/
8:
9:  #include "biosregs.h"
10: #include <dos.h>
11:
```

```
12:  void get_cursor(int *row, int *column);
13:
14:  void get_cursor( int *row, int *column)
15:  {
16:      union REGS inregs;
17:
18:      inregs.h.ah = 0x03;
19:      inregs.h.bh = 0;
20:
21:      int86(0x10, &inregs, &inregs);
22:
23:      *row    = (int) inregs.h.dh;
24:      *column = (int) inregs.h.dl;
25:  }
```

Type

Listing 8.5. LIST0805.C. Using the BIOS cursor functions.

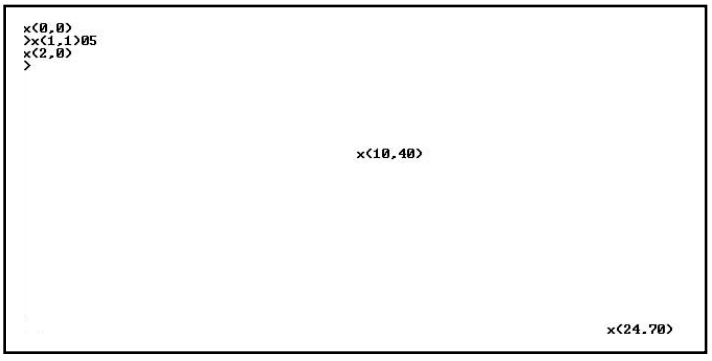
```
1:   /* Program: LIST0805.c
2:    * Author:  Bradley L. Jones
3:    * Purpose: Demonstrates the use of the cursor() and
4:    *          get_cursor() functions.
5:    *=========================================================*/
6:
7:   #include <stdio.h>
8:
9:   /*-----------------------------------*
10:   *      Function Prototypes          *
11:   *-----------------------------------*/
12:
13:  void get_cursor(int *row, int *column);
14:  void cursor(int row, int column);
15:  void main(void);
16:
17:  void main(void)
18:  {
19:     int row, column;
20:
21:     get_cursor( &row, &column);
22:
23:     cursor( 10, 40 );
24:     printf("x(10,40)", 10, 40);
25:
26:     cursor( 0, 0 );
27:     printf( "x(0,0)");
28:
29:     cursor( 1, 1 );
30:     printf( "x(1,1)");
```

continues

Listing 8.5. continued

```
31:
32:     cursor( 24, 70 );
33:     printf( "x(24,70)");
34:
35:     cursor(row, column);
36:     printf("x(%d,%d)", row, column);
37:
38: }
```

Output



Analysis To compile Listing 8.5, you need to include the PCURSOR.C and GCURSOR.C in your compile line. If you are compiling from the command line, this would require entering the following (TCC should be replaced with your compiler's command):

```
TCC LIST0805.C PCURSOR.C GCURSOR.C
```

An alternative to this is to create a library and link it with LIST0805 as shown on Day 7.

The output you receive from running this program may vary just a little. This program doesn't clear the screen before it starts, therefore, any information that was on the screen before the program is executed will remain. In the output shown here, you can see some of the remains of the command to run the program (second line of output).

Notice that the x marks where the cursor was placed. Listing 8.5 is extremely simple. Line 21 starts the program by getting the position of the cursor so that we can put it back when the program ends. Lines 23 to 36 call the `cursor()` function and then print the value of the location. The last call to `cursor()` positions the cursor back where it was when the program started.

The PCURSOR.C listing (Listing 8.3) contains the `cursor()` function. This function uses an interrupt to BIOS to place the cursor. Line 10 includes the registers that were presented earlier today. Line 11 includes the DOS.H header file that contains the function prototype for the interrupt function in line 24. Line 13 contains the prototype for the cursor function. If you are creating a library of these functions, you should create a header file that contains all of the function prototypes for all of the functions in your library. Line 19 sets the `ah` register to the function number 2. The `bh` register is set to 0. If you were working with multiple video pages, then this would be set to the number of the video page. Lines 21 and 22 contain `dh` and `dl`, which are set to the values passed to the `cursor()` function in the `row` and `column` variables. This is the actual row and column where the cursor will be placed. Line 24 wraps this function up by calling the `int86()` function. Interrupt 16 (0x10h) is called with the values that were set. This causes the cursor to be placed.

The `get_cursor()` function in Listing 8.4 is similar to the `cursor()` function. The big difference is that function 3 is placed in the `ah` register instead of function 2. The `bh` register is still set to the video page number. Because we're not doing video paging, this function sets the `bh` register to 0. The `int86()` function is then called in line 21 to finally set the cursor position. Once the BIOS function is called, the `dh` and `dl` registers contain the values for the row and column value of the cursor. These values are placed in the variables that are pointed to by `row` and `column`.

In addition to getting and putting the cursor, you can also change the cursor's shape. Listing 8.6 presents a function that allows you to manipulate the cursor. In addition, Listings 8.7 and 8.8 show this new function in action. Listings 8.7 and 8.8 should be compiled independently. Each should link in the cursor code.

Type

Listing 8.6. `set_cursor_size()`, manipulating the cursor's shape.

```
1:  /* Program: SCURSOR.C
2:   * Authors: Bradley L. Jones
3:   *          Gregory L. Guntle
4:   * Purpose: BIOS function to set the cursor size.
5:   *=======================================================*/
6:
7:  #include <dos.h>
8:
9:  #define SET_CURSOR_SIZE  0x01
10: #define BIOS_VIDEO       0x10
11:
12: void set_cursor_size(int start, int end)
13: {
```

continues

Listing 8.6. continued

```
14:      union REGS inregs;
15:
16:      inregs.h.ah = SET_CURSOR_SIZE;
17:      inregs.h.ch = start;
18:      inregs.h.cl = end;
19:      int86(BIOS_VIDEO, &inregs, &inregs);
20:  }
```

Listing 8.7. BIGCURS.C. Using the `set_cursor_size()` function.

```
1:   /* Program: BIGCURS.c
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: Changes the cursor to a big cursor.
5:    *=========================================================*/
6:
7:   /*** prototype ***/
8:   void set_cursor_size( int, int );
9:
10:  int main(void)
11:  {
12:     set_cursor_size( 1, 8 );
13:     return 0;
14:  }
```

Listing 8.8. SMLCURS.C. Using the `set_cursor_size()` function.

```
1:   /* Program: SMLCURS.c
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: Changes the cursor to a small cursor.
5:    *=========================================================*/
6:
7:   /*** prototype ***/
8:   void set_cursor_size( int, int );
9:
10:  int main(void)
11:  {
12:     set_cursor_size( 7, 8 );
13:     return 0;
14:  }
```

Note: There is no output for these two programs. See the analysis.

Analysis Listings 8.7 and 8.8 both use the `set_cursor_size()` function presented in Listing 8.6 to change the size of the cursor. Listing 8.7, BIGCURS.C, is a program that calls the `set_cursor_size()` function to change the cursor to a large block. After running BIGCURS.C, your cursor will be replaced with a large cursor. The SMLCURS.C program is similar, except that it sets the cursor to a small underscore.

Both of these programs operate by passing two values to the `set_cursor_size()` function. The `set_cursor_size()` function in Listing 8.6 is similar to the other BIOS functions. In line 16, the `ah` register is set with the function number that will be set. This listing differs in that a defined constant helps to make the setting of the function easier to understand. An additional defined constant, `BIOS_VIDEO`, also helps to make the program easier to read. Defined constants such as these can be placed in a header file and included in several of your functions.

The `set_cursor_size()` accepts two parameters, `start` and `end`. These values are used to change the size of the cursor. As you can see in BIGCURS.C, setting the `start` to 1 and the `end` to 8, you get a large cursor. Setting the `start` to 7 and the `end` to 8 provides a more traditionally sized cursor as shown in the SMLCURS.C listing.

Tip: Put all of your related BIOS functions into a library. (This will be an exercise.)

Expert Tip: As you should begin to see, BIOS functions are very powerful. You will be creating several BIOS functions over the next few days. You will see these functions in use during the development of an application in the second half of this book. Most of these functions will be useful long after you are done with this book.

Using BIOS for Other Functions

So far you have seen only a single date function and a few cursor functions. These functions barely scratch the surface of what functions are available by using interrupt routines. The multitude of different interrupts available is dependent upon your system and the systems that you're running the programs on. Table 8.1 lists the common interrupts and the areas they cover. Many of the interrupt functions that should be available using these interrupts are listed in Table 8.2.

Table 8.1. The ROM BIOS interrupts.

Interrupt	Function Types
16 (0x10h)	Video display functions
17 (0x11h)	Computer equipment function
18 (0x12h)	Conventional memory function
19 (0x13h)	Disk functions
20 (0x14h)	Serial communication port functions
21 (0x15h)	I/O subsystem functions
22 (0x16h)	Keyboard functions
23 (0x17h)	Parallel port functions
24 (0x18h)	ROM BASIC function
25 (0x19h)	System reboot function
26 (0x1Ah)	Clock driver functions
51 (0x33h)	Mouse functions

Table 8.2. The ROM BIOS interrupts.

Function (SubFunction)	Number (Hex)
Interrupt 16 (0x10h)	
Set video mode	0 (0x00h)
Set type of cursor	1 (0x01h)

Function (SubFunction)	Number (Hex)	
Set position of cursor	2 (0x02h)	
Get position of cursor	3 (0x03h)	
Get position of light pen	4 (0x04h)	
Set display page	5 (0x05h)	
Scroll window up	6 (0x06h)	
Scroll window down	7 (0x07h)	
Get char/attribute where cursor is located	8 (0x08h)	
Put char/attribute where cursor is located	9 (0x09h)	
Put character where cursor is located	10 (0x0Ah)	
Set background, border, and palette	11 (0x0Bh)	
Write a pixel	12 (0x0Ch)	
Read a pixel	13 (0x0Dh)	
Using teletype mode, write a character	14 (0x0Eh)	
Determine video mode	15 (0x0Fh)	
Set pallet	15	0 (0x00h)
Set border	15	1 (0x01h)
Set both, pallet and border	15	2 (0x02h)
Toggle bit for blink/intensity	15	3 (0x03h)
Determine video mode	15	
Set palette	16 (0x10h)	0 (0x00h)
Set color of border	16	1 (0x01h)
Set both, palette and border	16	2 (0x02h)
Toggle bit for blink/intensity	16	3 (0x03h)
Get palette	16	7 (0x07h)
Get color of border	16	8 (0x08h)

continues

Table 8.2. continued

Function (SubFunction)	Number (Hex)	
Get both, palette and border	16	9 (0x09h)
Set color register	16	16 (0x10h)
Set a block of color registers	16	18 (0x12h)
Set state of color page	16	19 (0x13h)
Get color register	16	21 (0x15h)
Get block of color registers	16	23 (0x17h)
Get state of color page	16	26 (0x1Ah)
Set gray-scale values	16	27 (0x1Bh)
Load a user font	17 (0x11h)	0 (0x00h)
Load ROM 8x14 font	17	1 (0x01h)
Load ROM 8x8 font	17	2 (0x02h)
Set block specifier	17	3 (0x03h)
Load ROM 8x16 font	17	4 (0x04h)
Load user font, reprogram controller	17	16 (0x10h)
Load ROM 8x14 font, reprogram controller	17	17 (0x11h)
Load ROM 8x8 font, reprogram controller	17	18 (0x12h)
Load ROM 8x16 font, reprogram controller	17	20 (0x14h)
Set Interrupt 31 (1Fh) pointer	17	32 (0x20h)
Set Interrupt 67 (43h) for a user's font	17	33 (0x21h)
Set Interrupt 67 (43h) for ROM 8x14 font	17	34 (0x22h)
Set Interrupt 67 (43h) for ROM 8x8 font	17	35 (0x23h)
Set Interrupt 67 (43h) for ROM 8x16 font	17	36 (0x24h)
Get information on font	17	18 (0x30h)
Get information on configuration	18 (0x12h)	16 (0x10h)
Select alternate print screen	18	32 (0x20h)

Function (SubFunction)	Number (Hex)	
Set scan lines	18	48 (0x30h)
Enable or disable loading palette	18	49 (0x31h)
Enable or disable the video	18	50 (0x32h)
Enable or disable gray-scale summing	18	51 (0x33h)
Enable or disable cursor emulation	18	52 (0x34h)
Switch active display	18	53 (0x35h)
Enable or disable screen refresh	18	54 (0x36h)
Write string in teletype mode	19 (0x13h)	
Determine or set display combination code	26 (0x1Ah)	
Get information on state/functionality	27 (0x1Bh)	
Save or restore the video state	28 (0x1Ch)	

Interrupt 17 (0x11h)

Get equipment configuration

Interrupt 18 (0x12h)

Get size of conventional memory

Interrupt 19 (0x13h)

(Disk drive functions)	
Reset the disk system	0 (0x00h)
Get status of disk system	1 (0x01h)
Read a sector	2 (0x02h)
Write a sector	3 (0x03h)
Verify a sector	4 (0x04h)
Format a track	5 (0x05h)
Format a bad track	6 (0x06h)
Format a drive	7 (0x07h)

continues

Table 8.2. continued

Function (SubFunction)	Number (Hex)
Get the drive parameters	8 (0x08h)
Initialize the fixed disk	9 (0x09h)
Read a long sector	10 (0x0Ah)
Write a long sector	11 (0x0Bh)
Do a seek	12 (0x0Ch)
Reset the fixed disk	13 (0x0Dh)
Read sector buffer	14 (0x0Eh)
Write sector buffer	15 (0x0Fh)
Get the drive status	16 (0x10h)
Recalibrate the drive	17 (0x11h)
Controller RAM diagnostic	18 (0x12h)
Controller drive diagnostic	19 (0x13h)
Controller internal diagnostic	20 (0x14h)
Get the type of disk	21 (0x15h)
Get status of disk change	22 (0x16h)
Set the disk type	23 (0x17h)
Set the media type for format	24 (0x18h)
Park drive heads	25 (0x19h)
Format drive (ESDI)	26 (0x1Ah)
Interrupt 20 (0x14h)	
Initialize the serial port	0 (0x00h)
Write a character to the serial port	1 (0x01h)
Read a character from the serial port	2 (0x02h)
Determine serial port status	3 (0x03h)

Function (SubFunction)	Number (Hex)
Extended initialize serial port	4 (0x04h)
Extended serial port control	5 (0x05h)
Interrupt 21 (0x15h)	
Turn cassette motor on	0 (0x00h)
Turn cassette motor off	1 (0x01h)
Read from cassette	2 (0x02h)
Write to cassette	3 (0x03h)
Intercept keyboard	79 (0x4Fh)
Event to wait	131 (0x83h)
Read from joystick	132 (0x84h)
SysReq key press	133 (0x85h)
Pause (delay)	134 (0x86h)
Move an extended memory block	135 (0x87h)
Determine extended memory size	136 (0x88h)
Start protected mode	137 (0x89h)
Have device wait	144 (0x90h)
Get the system environment	192 (0xC0h)
Determine the address of the extended BIOS data area	193 (0xC1h)
Pointing device functions	194 (0xC2h)
Interrupt 22 (0x16h)	
(Keyboard functions)	
Read a character from the keyboard	0 (0x00h)
Get the status of the keyboard	1 (0x01h)
Get keyboard flags	2 (0x02h)

continues

Table 8.2. continued

Function (SubFunction)	Number (Hex)
Set rate for repeat	3 (0x03h)
Set the keyboard to click	4 (0x04h)
Push a character and scan code	5 (0x05h)
Read a character (enhanced keyboard)	16 (0x10h)
Get status of keyboard (enhanced keyboard)	17 (0x11h)
Get keyboard flags (enhanced keyboard)	18 (0x12h)
Interrupt 23 (0x17h)	
(Parallel port functions)	
Write a character to the parallel port	0 (0x00h)
Initialize parallel port	1 (0x01h)
Get status of parallel (print) port	2 (0x02h)
Interrupt 24 (0x18h)	
(ROM BASIC)	
Interrupt 25 (0x19h)	
Re-boot the computer system.	
Interrupt 26 (0x1Ah)	
(CMOS clock driver)	
Determine tick count	1 (0x00h)
Set tick count	2 (0x02h)
Determine the time	3 (0x03h)
Set the time	4 (0x04h)
Determine the date	5 (0x05h)
Set the date	6 (0x06h)
Set the alarm	7 (0x07h)

Function (SubFunction)	Number (Hex)
Reset the alarm	8 (0x08h)
Set the sound source	128 (0x80h)
Interrupt 51 (0x33h)	
(Mouse functions)	
Reset and get mouse status	0 (0x00h)
Display the mouse pointer	1 (0x01h)
Hide the mouse pointer	2 (0x02h)
Determine mouse position and button status	3 (0x03h)
Set the mouse pointer position	4 (0x04h)
Determine the button press information	5 (0x05h)
Determine the button release information	6 (0x06h)
Set the horizontal limits for the pointer	7 (0x07h)
Set the vertical limits for the pointer	8 (0x08h)
Set the shape of the pointer (graphics)	9 (0x09h)
Set the pointer type (text)	10 (0x0Ah)
Read the mouse motion	11 (0x0Bh)
Set a user-defined mouse event handler	12 (0x0Ch)
Light pen emulation on	13 (0x0Dh)
Light pen emulation off	14 (0x0Eh)
Set exclusion area for mouse pointer	16 (0x10h)
Set threshold for double speed	19 (0x13h)
Switch user-defined event handlers for mouse	20 (0x14h)
Determine save state buffer size for mouse	21 (0x15h)
Save the mouse's driver state	22 (0x16h)
Restore the mouse's driver state	23 (0x17h)

continues

Table 8.2. continued

Function (SubFunction)	Number (Hex)
Set an alternate mouse event handler	24 (0x18h)
Determine address of alternate event handler	25 (0x19h)
Set the mouse's sensitivity	26 (0x1Ah)
Get the mouse's sensitivity	27 (0x1Bh)
Set the mouse's interrupt rate	28 (0x1Ch)
Select a pointer page	29 (0x1Dh)
Determine the pointer page	30 (0x1Eh)
Disable the mouse driver	31 (0x1Fh)
Enable the mouse driver	32 (0x20h)
Reset the mouse driver	33 (0x21h)
Set the mouse driver message language	34 (0x22h)
Get the language number	35 (0x23h)
Get information on mouse	36 (0x24h)

It is beyond the scope of this book to provide detailed examples of using each and every interrupt, their functions, and all of their subfunctions. A few of these interrupts will be used in functions presented in the next section. In addition, many of the interrupts will be used as this book progresses.

Note: If you want more information on BIOS interrupts, there are several other books available that go into much more detail. A few to consider are Jack Purdum's *C Programmer's Toolkit* and *DOS Programmer's Reference* both published by Que Corporation. In addition, there is the *C Programmers Guide to NetBios, IPX, and SPX* by SAMS Publishing. Many books on assembly language also talk about the BIOS functions.

Examples of Using BIOS

This section contains a couple of additional BIOS functions that you may find useful. The first function, `keyhit()`, determines if a character has been entered into the keyboard. If a character has not been entered, the program continues on. The `keyhit()` function will be presented in Listing 8.9. The second function clears the keyboard and waits for a keyhit. This function, `kbwait()`, will be useful when you need to wait for a keystroke and you need to remove any characters that a user may have typed ahead. The `kbwait()` function is presented in Listing 8.10. As stated earlier, several additional functions will be presented throughout the rest of this book.

Type

Listing 8.9. `keyhit()`. A function to determine if a keyboard character has been pressed.

```
1:  /* Program: KEYHIT.C
2:   * Authors: Bradley L. Jones
3:   *          Gregory L. Guntle
4:   * Purpose: BIOS function to determine if a key has been
5:   *          hit.
6:   * Return:  0 - key not hit
7:   *          # - key that was hit.
8:   *              If # > 0x100, then key is a scan code.
9:   *=======================================================*/
10:
11: #include <dos.h>
12:
13: int keyhit( void )
14: {
15:     int   flag;
16:     union REGS inregs;
17:
18:     inregs.h.ah = 0x06;
19:     inregs.h.dl = 0xFF;
20:     flag = int86(0x21, &inregs, &inregs);
21:
22:     if(( flag & 0x40 ) == 0 )
23:     {
24:        if( inregs.h.al == 0 )
25:        {
26:           /* extended character, get second half */
27:           inregs.h.ah = 0x06;
28:           inregs.h.dl = 0xFF;
29:           int86(0x21, &inregs, &inregs);
30:           return( inregs.h.al + 0x100 );
31:        }
32:        else
33:        {
```

continues

Listing 8.9. continued

```
34:             return inregs.h.al;  /* the key hit */
35:          }
36:       }
37:       else
38:       {
39:          return 0;  /* key not hit */
40:       }
41:  }
```

Listing 8.10. `kbwait()`. A function to clear the keyboard buffer.

```
1:   /* Program: KBWAIT.C
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: BIOS function to clear the keyboard buffer
5:    *          and wait for a key to be pressed.
6:    *=========================================================*/
7:
8:   #include <dos.h>
9:   #include <stdio.h>
10:
11:  void kbwait( void )
12:  {
13:      union REGS inregs;
14:
15:      inregs.h.ah = 0x0C;
16:      inregs.h.al = 0x08;
17:      int86(0x21, &inregs, &inregs);
18:  }
```

Listing 8.11. LIST0811.C. A program demonstrating the previous two functions.

```
1:   /* Program: list0811.c
2:    * Author:  Bradley L. Jones
3:    * Purpose: Demonstrates the use of the kbwait() and
4:    *          keyhit() functions.
5:    *=========================================================*/
6:
7:   #include <stdio.h>
8:
9:   /*-----------------------------------*
10:   *      Function Prototypes          *
11:   *-----------------------------------*/
```

```
12:
13:  int keyhit(void);
14:  void kbwait(void);
15:
16:  int main(void)
17:  {
18:     int ctr = 65;
19:     char buffer[256];
20:
21:     printf("\n\nClearing the keyboard buffer.");
22:     printf("\nPress any key to continue...");
23:
24:     kbwait();
25:
26:     printf("\nMoving on...");
27:
28:     while(!keyhit())
29:     {
30:       printf("%c", ctr );
31:
32:       if( ctr >= 90 )
33:          ctr = 65;
34:       else
35:          ctr++;
36:     }
37:
38:     printf("DONE");
39:
40:     return 0;
41:  }
```

Analysis This program displays a message and then asks you to press any key to continue. Once you continue, the program begins printing the letters of the alphabet. These letters are printed until a key is pressed. Using a `while` loop in line 28, allows the program to do what is contained in lines 29 to 36 until a key is hit (`keyhit()` returns a key value). After each placement of a letter, the `keyhit()` function is used to determine if a character has been pressed. If not, the program continues to the next letter. If a character has been pressed, then the program prints "Done" and ends. Because `keyhit()` returns the value of the key pressed, this program could be changed to check for a specific key before ending.

Creating Your Own Interrupts

Not all of the interrupts have functions behind them. Because of this, you can create your own interrupt events. This could be an event such as causing the speaker to beep,

or a memory resident program. Unfortunately, writing your own interrupt functions is beyond the scope of this book.

Compiler-Specific BIOS Functions

Many compilers come with several of their own functions that are already set up to perform specific BIOS interrupt tasks. Microsoft comes with several. These include:

`_bios_equiplist`	Uses interrupt 0x11h (17) to perform an equipment checklist.
`_bios_memsize`	Uses interrupt 0x12h (18) to provide information about available memory.
`_bios_disk`	Uses interrupt 0x13h (19) to issue service requests for hard and floppy disks.
`_bios_serialcom`	Uses interrupt 0x14h (20) to perform serial communications services.
`_bios_keybrd`	Uses interrupt 0x16h (22) to provide access to keyboard services.
`_bios_printer`	Uses interrupt 0x17h (23) to perform printer output services.
`_bios_timeofday`	Uses interrupt 0x1Ah (26) to access the system clock.

Note: When using the Microsoft or the Borland predefined BIOS functions, you need to include the BIOS.H header file.

The Borland compilers also have several preincluded BIOS interrupt functions. These include the following:

`biosequip`	Uses interrupt 0x11h (17) to check equipment list.
`_bios_equiplist`	Uses interrupt 0x11h (17) to check equipment list.
`biosmemory`	Uses interrupt 0x12h (18) to determine the size of memory.
`_bios_memsize`	Uses interrupt 0x12h (18) to determine the size of memory.
`biosdisk`	Uses interrupt 0x13h (19) to perform disk drive services.
`_bios_disk`	Uses interrupt 0x13h (19) to perform disk drive services.

`bioscom`	Uses interrupt 0x14h (20) to perform serial communications services.
`_bios_serialcom`	Uses interrupt 0x14h (20) to perform serial communication services.
`bioskey`	Uses interrupt 0x16h (23) to work with the keyboard interface.
`_bios_keybrd`	Uses interrupt 0x16h (23) to work with the keyboard interface.
`biosprint`	Uses interrupt 0x17h (24) to perform printer services.
`_bios_printer`	Uses interrupt 0x17h (24) to perform printer services.
`biostime`	Uses interrupt 0x1Ah (26) to read or set the system clock.
`_bios_timeofday`	Uses interrupt 0x1Ah (26) to read or set the system clock.

You may choose to use these functions, or you may decide to call the interrupts on your own using a more generic `int86()` type of function.

A Note on Compiler Functions Versus BIOS Functions

Now that you've gotten a glimpse of the many functions that can be created using BIOS and have reviewed some compiler-specific functions, it is important to consider why you would ever need to create a new function using BIOS—or any of the other methods presented. There are at least two major reasons for creating your own functions. The first is for the sake of learning. By creating you own functions, you better understand the underlying code. Once you create a function, you can put it into a library and use it everywhere. A second reason is flexibility. The compiler functions are what they are. You cannot change them or modify them to your specific needs. By writing your own functions, you can create them just the way you want them.

As this book continues, you will see several BIOS functions in action. In addition, you will also see several more complex functions that have BIOS functions in their underlying code.

Note: Remember, some compilers use `int86()` and some use `_int86()`. You will need to check your compiler's library reference manual to determine which function is correct for you.

Summary

Today, you were provided with the most powerful way of manipulating your computer system's resources. This was through the use of interrupt calls to BIOS. The acronym, BIOS, stands for Basic Input/Output System. You also were provided with information on why BIOS functions are better to use than ANSI escape sequences and direct memory access. In addition, you were presented with tables containing information on many of the system resources that BIOS can manipulate. This includes placing the cursor, clearing the screen, changing colors, remapping keyboard values, working with modems, reading joystick commands, and much, much more. A few examples of using BIOS functions were presented to give an overall idea of what using them requires. BIOS functions will be revisited off and on throughout the rest of this book.

Q&A

Q Is the BIOSREGS.H header file presented in Listing 8.1 necessary?

A You were shown the values presented by Microsoft and Borland compilers. If you include DOS.H, you may not need to include the BIOSREGS.H header file. Other compilers may or may not need the register structure. Most of the programs from this point on won't include the BIOSREGS.H header file. If your compiler doesn't have the register union and structures, you should include BIOSREGS.H.

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

A The functions covered in today's material are portable to some computers. 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 should consult your DOS manuals and system documentation to determine what interrupts your computer supports.

Q What is meant by BIOS functions being portable?

A The portability of BIOS functions is not necessarily the same as the portability that will be discussed later in this book. The calls to BIOS functions are not 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.

Q Why might someone want to use direct video access instead of BIOS functions?

A If manipulating the video monitor at the highest speeds possible is imperative to an application, then direct video memory access may be the best solution. For graphics-intensive games, it's often necessary to directly manipulate the video memory to get the best results and the smoothest graphical movements.

Q Are all the interrupts presented today always available?

A No! Older versions of the DOS operating system may not support all of the interrupts presented. If you are using an extremely old version of DOS, such as 2.01, or even 3.0, then you may find that several of the functions are not supported. Your DOS manuals may contain a list of supported interrupts.

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 BIOS stand for?
2. What are two reasons to use BIOS functions instead of ANSI functions?
3. What is a reason to use the BIOS function instead of direct memory access when updating the video display?
4. What is an interrupt number?
5. What is an interrupt function?
6. What is an interrupt subfunction?
7. What does interrupt 0x33h function do?
8. What does function 2 of interrupt 51 (0x33h) do?

9. What does interrupt 25 (0x19h) do?
10. Do all computers support all BIOS functions?

Exercises

1. Create a library containing all of the BIOS functions presented today. Call this library TYAC.LIB. This library can be used in future chapters. This library should contain the following functions:

```
current_date()
cursor()
get_cursor()
set_cursor_size()
keyhit()
kbwait()
```

Note: You should also create a header file with the same name as your library. This header file should contain the function prototypes for the functions in your library.

2. Create a function that enables you to scroll the screen up one line at a time. This function can be created via a BIOS interrupt. You should set the following registers:

   `ah` = 0x07
   `al` = 1
   `ch` = 0
   `cl` = 0
   `dh` = 25
   `dl` = col
   `bh` = 0

   The BIOS interrupt is 0x10.

3. Rewrite the function from Exercise 2. This time write the function generically so that it can be used to scroll the screen—or a portion of the screen—up or down:

ah should be set to interrupt 0x07 to scroll up or 0x06 to scroll down.

al should be set to the number of lines to scroll.

ch should be set to the number of the starting row on the screen area to be scrolled (0 is the first line on the screen).

cl should be set to the number of the starting column on the screen area to be scrolled (0 is the first column on the screen).

dh should be set to the width of the scrolling area. (This should be added to the row to only scroll the intended area.)

dl should be set to the height of the scrolling area. (This should be added to the column to only scroll the intended area.)

bh should be used for an attribute. In this case, always set bh to 0.

4. Write a program that uses the function from Exercise 3.
5. Add the previous function to your TYAC.LIB library. You should also add the prototype to the TYAC.H header file along with defined constants for SCROLL_UP and SCROLL_DOWN.
6. **BUG BUSTER:** What, if anything, is wrong with the following?

```
void current_date( int *month, int *day, int *year)
{
    union REGS inregs, outregs;
    inregs.h.ah = 0x2a;

    int86(0x21);

    *month = outregs.h.dh;
    *day   = outregs.h.dl;
    *year  = outregs.x.cx;
}
```