CHAPTER XIV

# System Calls

One of the most crucial pieces of the PC puzzle, and sometimes the most often misunderstood, is the set of system calls. The functions that the system calls represent perform practically all the rudimentary operations of the computer—screen and disk handling, keyboard and mouse handling, managing the file system, time of day, and printing are just some of the tasks performed by the system calls.

Collectively, the system calls are often referred to as the BIOS, which stands for Basic Input Output System. In reality, there are several different BIOSes. For example, the motherboard BIOS performs initial hardware detection and system booting; the VGA BIOS (if you have a VGA card) handles all the screen-manipulation functions; the fixed disk BIOS manages the hard drives; and so on. DOS is a software layer that "sits" on top of these lower-level BIOSes and provides a common access to these lower-level BIOSes. What this means is that, in general, there is a DOS system call for just about every kind of system feature you want to initiate. DOS will call one of the lower level BIOSes to actually perform the requested task. Throughout this chapter, you will find that you can call DOS to perform a task, or you can call the lower-level BIOS directly to perform the same task.

# XIV.1: How can environment variable values be retrieved?

## *Answer:*

The ANSI C language standard provides a function called `getenv()` that performs this very task. The `getenv()` function is simple—you hand it a pointer to the environment string you want to search for, and it returns a pointer to the value of that variable. The following example code illustrates how to get the `PATH` environment variable from C:

```
#include <stdlib.h>

main(int argc, char ** argv)
{
    char envValue[129];         /* buffer to store PATH */
    char * envPtr = envValue;   /* pointer to this buffer */

    envPtr = getenv("PATH");    /* get the PATH */

    printf("PATH=%s\n", envPtr);    /* print the PATH */

}
```

If you compile and run this example, you will see the same output that you see when you enter the `PATH` command at the DOS prompt. Basically, you can use `getenv()` to retrieve any of the environment values that you have in your AUTOEXEC.BAT file or that you have typed at the DOS prompt since booting.

Here's a cool trick. When Windows is running, Windows sets a new environment variable called `WINDIR` that contains the full pathname of the Windows directory. Following is sample code for retrieving this string:

```
#include <stdlib.h>

main(int argc, char ** argv)
{
    char envValue[129];
    char * envPtr = envValue;

    envPtr = getenv("windir");

    /* print the Windows directory */
    printf("The Windows Directory is %s\n", envPtr);

}
```

This is also useful for determining whether Windows is running and your DOS program is being run in a DOS shell rather than "real" DOS. Note that the `windir` string is lowercase—this is important because it is case-sensitive. Using `WINDIR` results in a `NULL` string (variable-not-found error) being returned.

It is also possible to *set* environment variables using the `_putenv()` function. Note, however, that this function is not an ANSI standard, and it might not exist by this name or even exist at all in some compilers. You can do many things with the `_putenv()` function. In fact, it is this function that Windows uses to create the `windir` environment variable in the preceding example.

## Cross Reference:

XIV.2: How can I call DOS functions from my program?

XIV.3: How can I call BIOS functions from my program?

# XIV.2: How can I call DOS functions from my program?

## *Answer:*

To be honest, you are calling DOS functions whenever you call `printf()`, `fopen()`, `fclose()`, any function whose name starts with `_dos`, or dozens of other C functions. However, Microsoft and Borland C provide a pair of functions called `int86()` and `int86x()` that enable you to call not only DOS, but other low-level functions as well. Using these functions, you often can save time by bypassing the standard C functions and calling the DOS functions directly. The following example illustrates how to call DOS to get a character from the keyboard and print a character string, instead of calling `getch()` and `printf()` to perform the task. (This code needs to be compiled with the Large memory model.)

```
#include <stdlib.h>
#include <dos.h>

char GetAKey(void);
void OutputString(char *);

main(int argc, char ** argv)
{
    char str[128];
    union REGS regs;
    int ch;

    /* copy argument string; if none, use "Hello World" */
    strcpy(str, (argv[1] == NULL ? "Hello World" : argv[1]));


    while((ch = GetAKey()) != 27){
        OutputString(str);
    }

}

char
GetAKey()
{
    union REGS regs;

    regs.h.ah = 1;    /* function 1 is "get keyboard character" */
    int86(0x21, &regs, &regs);

    return((char)regs.h.al);
}

void
OutputString(char * string)
```

```
{
    union REGS regs;
    struct SREGS segregs;

    /* terminate string for DOS function */
    *(string+strlen(string)) = '$';

    regs.h.ah = 9;    /* function 9 is "print a string" */
    regs.x.dx = FP_OFF(string);
    segregs.ds = FP_SEG(string);

    int86x(0x21, &regs, &regs, &segregs);
}
```

The preceding example code created two functions to replace `getch()` and `printf()`. They are `GetAKey()` and `OutputString()`. In truth, the `GetAKey()` function is actually more analogous to the standard C `getche()` function because it, like `getche()`, prints the key character on-screen. In both cases, DOS itself was called using the `int86()` function (in `GetAKey()`) and the `int86x()` function (in `OutputString()`) to perform the desired tasks.

DOS contains a veritable plethora of callable routines just like these two functions. Although you'll find that many of them are well covered by standard C routines, you'll also find that many are not. DOS also contains many *undocumented* functions that are quite interesting, and useful as well. An excellent example of this is the DOS Busy Flag, also called the `InDos` Flag. DOS function 34 hex returns a pointer to a system memory location that contains the DOS Busy Flag. This flag is set to 1 whenever DOS is busy doing something critical and doesn't want to be called (not even by itself). The flag is cleared (set to zero) when DOS is not busy. The purpose of this flag is to inform DOS when it is executing critical code. However, programmers find this flag useful as well so that they can also know when DOS is busy. Because Microsoft has recently documented this function, it is technically no longer an undocumented routine, but it has been in DOS since version 2.0. Several excellent books on documented and undocumented DOS functions are available for those who are more interested in this topic.

## Cross Reference:

XIV.3: How can I call BIOS functions from my program?

# XIV.3: How can I call BIOS functions from my program?

## *Answer:*

As in the preceding example, you are quite frequently calling BIOS functions when you use standard C library routines such as `_setvideomode()`. In addition, even the DOS functions used previously (`getch()` and `printf()`) ultimately made calls into the BIOS to actually perform the tasks. In such cases, DOS is simply passing on your DOS request to the proper lower-level BIOS function. In fact, the following example code illustrates this fact perfectly. This code performs the same tasks that the preceding example does, except that DOS is bypassed altogether and the BIOS is called directly.

```
#include <stdlib.h>
#include <dos.h>
```

```
char GetAKey(void);
void OutputString(char *);

main(int argc, char ** argv)
{
    char str[128];
    union REGS regs;
    int ch;

    /* copy argument string; if none, use "Hello World" */
    strcpy(str, (argv[1] == NULL ? "Hello World" : argv[1]));


    while((ch = GetAKey()) != 27){
        OutputString(str);
    }

}

char
GetAKey()
{
    union REGS regs;

    regs.h.ah = 0;              /* get character */
    int86(0x16, &regs, &regs);

    return((char)regs.h.al);
}

void
OutputString(char * string)
{
    union REGS regs;

    regs.h.ah = 0x0E;           /* print character */
    regs.h.bh = 0;

    /* loop, printing all characters */
    for(;*string != '\0'; string++){
        regs.h.al = *string;
        int86(0x10, &regs, &regs);
    }
}
```

As you can see, the only changes were in the `GetAKey()` and `OutputString()` functions themselves. The `GetAKey()` function bypasses DOS and calls the keyboard BIOS directly to get the character (note that in this particular call, the key is not echoed to the screen, unlike in the previous example). The `OutputString()` function bypasses DOS and calls the Video BIOS directly to print the string. Note the inefficiency of this particular example—the C code must sit in a loop, printing one character at a time. The Video BIOS does support a print-string function, but because C cannot access all the registers needed to set up the call to print the string, I had to default to printing one character at a time. Sigh. Regardless, you can run this example to produce the same output as is produced in the example before it.

## Cross Reference:

XIV.2: How can I call DOS functions from my program?

# XIV.4: How can I access important DOS memory locations from my program?

## *Answer:*

Like the DOS and BIOS functions, there are dozens of memory locations that contain useful and interesting information about the computer. Need to know the current display mode without using an interrupt? It's in 40:49 hex (that's segment 40, offset 49). Want to find out whether the user is currently pressing the Shift, Ctrl, or Alt keys? That information is in 40:17. Want to write directly to the screen? Monochrome screens start at 0xB000:0, color text mode and 16-color graphics modes (below 640×480 16 colors) start at 0xB800:0, and all other standard graphics modes (including 640×480 16 colors and up) start at 0xA000:0. Refer to FAQ XIV.8 for more information. The following example shows how to print color text mode characters to the screen. Note that this is a slight modification to the example used in the previous few questions.

```
#include <stdlib.h>
#include <dos.h>

char GetAKey(void);
void OutputString(int,int,unsigned int,char *);

main(int argc, char ** argv)
{
    char str[128];
    union REGS regs;
    int ch, tmp;

    /* copy argument string; if none, use "Hello World" */
    strcpy(str, (argv[1] == NULL ? "Hello World" : argv[1]));

    /* print the string in red at top of screen */
    for(tmp=0;((ch = GetAKey()) != 27); tmp+=strlen(str)){
        OutputString(0,tmp,0x400,str);
    }

}

char
GetAKey()
{
    union REGS regs;

    regs.h.ah = 0;              /* get character */
    int86(0x16, &regs, &regs);

    return((char)regs.h.al);
}
```

```
void
OutputString(int row, int col, unsigned int videoAttribute, char *outStr)
{
    unsigned short far * videoPtr;

    videoPtr = (unsigned short far *)(0xB800L << 16);
videoPtr += (row * 80) + col;   /* Move videoPtr to cursor position */
videoAttribute &= 0xFF00;       /* Ensure integrity of attribute */

    /* print string to RAM */
    while(*outStr != '\0'){

        /* If newline was sent, move pointer to next line, column 0 */
        if( (*outStr == '\n') || (*outStr == '\r') ){
            videoPtr += (80 - ( ((int)FP_OFF(videoPtr) / 2) % 80));
            outStr++;
            continue;
        }

        /* If backspace was requested, go back one */
        if(*outStr == 8){
            videoPtr--;
            outStr++;
            continue;
        }

        /* If BELL was requested, don't beep, just print a blank
           and go on */
        if(*outStr == 7){
            videoPtr++;
            outStr++;
            continue;
        }

        /* If TAB was requested, give it eight spaces */
        if(*outStr == 9){
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            *videoPtr++ = videoAttribute | ' ';
            outStr++;
            continue;
        }

        /* If it was a regular character, print it */
        *videoPtr = videoAttribute | (unsigned char)*outStr;
        videoPtr++;
        outStr++;
    }

    return;

}
```

Obviously, printing text characters to the screen is a bit more complicated when you have to do it yourself. I even made shortcuts by ignoring the meaning of the BELL character (to issue a beep) and a couple of other special characters (although I did implement carriage return and linefeed). Regardless, this function performs basically the same task as the previous examples, except that now you have control over the color of the character and its position when printed. This example starts printing from the top of the screen. If you want more examples of using memory locations, refer to FAQs XX.1, XX.12, and XX.17—these all use pointers to low DOS memory to find useful information about the computer.

## Cross Reference:

XIV.5: What is BIOS?

XX.1: How are command-line parameters obtained?

XX.12: How can I pass data from one program to another?

XX.17: Can you disable warm boots (Ctrl-Alt-Delete)?

# XIV.5: What is BIOS?

## *Answer:*

The Basic Input Output System, or BIOS, is the foundation of the personal computer's operation. It is the program that is executed first when the computer is powered up, and it is used by DOS and other programs to access each piece of hardware inside the computer.

The bootup program, however, isn't the only code in the computer that is called BIOS. In fact, the BIOS that executes when the PC is turned on is typically called the *Motherboard BIOS*, because it is located on the motherboard. Until recently, this BIOS was fixed in a ROM chip and could not be reprogrammed to fix bugs and enhance the features. Today, the Motherboard BIOS is in an electronically reprogrammable memory chip called *Flash EPROM*, but it is still the same old BIOS. Anyway, the Motherboard BIOS walks through system memory to find other hardware in the system that also contains code foundational to its use (other BIOS code). For example, your VGA card has its own BIOS physically located on the VGA card itself—it's usually called the Video BIOS or VGA BIOS. Your hard and floppy disk controller has a BIOS that is also executed at boot time. People often will both refer to these collective programs as the BIOS and refer to a specific individual BIOS as the BIOS. Neither reference is incorrect.

With all that said, you should know that BIOS is not DOS—BIOS is the lowest-level software functionality available on the PC. DOS "sits on top of" the BIOS and calls the BIOS regularly to perform routine operations that you might mistakenly attribute to being a "DOS" function. For example, you might use DOS function 40 hex to write data to a file on the hard disk. DOS actually performs this task by ultimately calling the hard disk BIOS's function 03 to actually write the data to the disk.

## Cross Reference:

XIV.6: What are interrupts?

# XIV.6: What are interrupts?

## *Answer:*

First of all, there are *hardware interrupts* and there are *software interrupts*. Interrupts provide a way for different hardware and software "pieces" of the computer to talk to each other. That is their purpose, but *what* are they, and how do they perform this communication?

The CPU (Central Processing Unit), which in the PC's case is an Intel or clone 80x86 processor, has several pins on it that are used to interrupt the CPU from its current task and make it perform some other task. Connected to each interrupt pin is some piece of hardware (a timer, for example) whose purpose is to apply a specific voltage on that CPU interrupt pin. When this event occurs, the processor stops executing the software it is currently executing, saves its current operating state, and "handles" the interrupt. The processor has already been loaded with a table that lists each interrupt number, and the program that is supposed to be executed when that particular interrupt number occurs.

Take the case of the system timer. As part of the many tasks it has to perform, the PC must maintain the time of day. Here's how it works: A hardware timer interrupts the CPU 18 times every second. The CPU stops what it is doing and looks in the interrupt table for the location of the program whose job it is to maintain the system timer data. This program is called an *interrupt handler*, because its job is to handle the interrupt that occurred. In this case, the CPU looks up interrupt eight in the table because that happens to be the system timer interrupt number. The CPU executes that program (which stores the new timer data into system memory) and then continues where it left off. When your program requests the time of day, this data is formatted into the style you requested and is passed on to you. This explanation greatly oversimplifies how the timer interrupt works, but it is an excellent example of a *hardware interrupt*.

The system timer is just one of hundreds of *events* (as interrupts are sometimes called) that occur via the interrupt mechanism. Most of the time, the hardware is not involved in the interrupt process. What I mean by that is that software frequently uses an interrupt to call another piece of software, and hardware doesn't have to be involved. DOS and the BIOS are two prime examples of this. When a program opens a file, reads or writes data to it, writes characters to the screen, gets a character from the keyboard, or even asks for the time of day, a *software interrupt* is necessary to perform each task. You might not know that this is happening because the interrupts are buried inside the innocuous little functions you call (such as `getch()`, `fopen()`, and `ctime()`).

In C, you can generate an interrupt using the `int86()` or `int86x()` functions. The `int86()` and `int86x()` functions require an argument that is the interrupt number you want to generate. When you call one of these functions, the CPU is interrupted as before and checks the interrupt table for the proper program to execute. In these cases, typically a DOS or BIOS program is executed. Table XIV.6 lists many of the common interrupts that you can call to set or retrieve information about the computer. Note that this is not a complete list and that each interrupt you see can service hundreds of different functions.

Table XIV.6. Common PC interrupts.

| *Interrupt (hex)* | *Description* |
|---|---|
| 5 | Print Screen Services |
| 10 | Video Services (MDA, CGA, EGA, VGA) |
| 11 | Get Equipment List |
| 12 | Get Memory Size |
| 13 | Disk Services |
| 14 | Serial Port Services |
| 15 | Miscellaneous Function Services |
| 16 | Keyboard Services |
| 17 | Printer Services |
| 1A | Time of Day Services |
| 21 | DOS Functions |
| 2F | DOS Multiplex Services |
| 33 | Mouse Services |
| 67 | EMS Services |

Now that you know what an interrupt is, you might realize that while your computer is just sitting idle, it is probably processing dozens of interrupts a second, and it is often processing hundreds of interrupts each second when it's working hard. FAQ XX.12 includes an example of a program that enables you to write your own interrupt handlers so that you can have two programs talk to each other through the interrupt. Check it out if you find this stuff fascinating.

## Cross Reference:

XX.12: How can I pass data from one program to another?

# XIV.7: Which method is better, ANSI functions or BIOS functions?

## *Answer:*

Each method has its advantages and disadvantages. What you must do is answer a few questions to find out which method is right for the application you need to create. Do you need to get your application out quickly? Is it just a "proof of concept" or the "real thing"? Does speed matter? Do you need the approval of your inner child? OK, so maybe your inner child doesn't care, but the other questions should be answered before you decide. The following list compares the basic advantages of ANSI versus BIOS functions:

| *ANSI Advantages over BIOS* |
|---|
| It requires only `printf()` statements to perform tasks |
| You can easily change text color and attributes |
| It works on every PC, regardless of configuration |
| You don't need to memorize BIOS functions |

| *BIOS Advantages over ANSI* |
|---|
| It runs faster |
| You can do more things using BIOS |
| It does not require a device driver (ANSI requires ANSI.SYS) |
| You don't need to memorize ANSI commands |

What you will discover is that ANSI is a good place to start, and it will enable you to create some nice programs. However, you will probably find that ANSI begins to "get in your way," and you'll soon want to move on to BIOS functions. Of course, you'll also find that the BIOS gets in your way sometimes, and you'll want to go even faster. For example, FAQ XIV.4 contains an example of avoiding even the BIOS to print text to the screen. You'll probably find that this method is more fun than ANSI or BIOS.

## Cross Reference:

XIV.4: How can I access important DOS memory locations from my program?

# XIV.8: Can you change to a VGA graphics mode using the BIOS?

## *Answer:*

Yes. Interrupt 10 hex, the Video BIOS, handles switching between text and graphics modes (among other things). When you execute a program that changes from text mode to graphics mode and back (even if the program is Microsoft Windows), the Video BIOS is requested to perform the change. Each different setting is called a *display mode*.

To change the display mode, you must call the Video BIOS through the "int10" services. That is, you must make interrupt calls to the interrupt handler at interrupt 10. This is just like making DOS calls (int21), except the interrupt number is different. Following is a piece of sample code that calls Video BIOS function 0 to switch from standard text mode (Mode 3) to a mode number from the command line and back:

```
#include <stdlib.h>
#include <dos.h>

main(int argc, char ** argv)
{
    union REGS regs;
    int mode;

    /* accept Mode number in hex */
    sscanf(argv[1], "%x", &mode);
```

```
    regs.h.ah = 0;           /* AH=0 means "change display mode" */
    regs.h.al = (char)mode;  /* AL=??, where ?? is the Mode number */
    regs.x.bx = 0;           /* Page number, usually zero */

    int86(0x10, &regs, &regs);  /* Call the BIOS (int10) */

    printf("Mode 0x%X now active\n", mode);
    printf("Press any key to return... ");
    getch();

    regs.h.al = 3;              /* return to Mode 3 */
    int86(0x10, &regs, &regs);
}
```

One interesting feature of this particular function that isn't shown here is the capability to change display modes without clearing the screen. This feature can be extremely useful in certain circumstances. To change modes without affecting screen contents, simply OR hex 80 to the display mode value you place into the AL register. For instance, if you want to switch to mode 13 (hex), you put hex 93 in AL. The remaining code stays unchanged.

Today, VGA cards also adhere to the VESA Video BIOS standard in their support of the extended display modes (see the following sidebar for an explanation). However, it requires a new "change display mode" function to support these extended modes. Per the VESA standard, you use function hex 4F rather than function 0 in the preceding example to switch VESA modes. The following example code is a modification of the preceding example to incorporate VESA mode numbers.

```
#include <stdlib.h>
#include <dos.h>

main(int argc, char ** argv)
{
    union REGS regs;
    int mode;

    /* accept Mode number in hex */
    sscanf(argv[1], "%x", &mode);

    regs.x.ax = 0x4F02;      /* change display mode */
    regs.x.bx = (short)mode; /* three-digit mode number */

    int86(0x10, &regs, &regs);  /* Call the BIOS (int10) */

    if(regs.h.al != 0x4F){
        printf("VESA modes NOT supported!\n");
    }
    else{
        printf("Mode 0x%X now active\n", mode);
        printf("Press any key to return... ");
        getch();
}
    regs.h.al = 3;              /* return to Mode 3 */
    int86(0x10, &regs, &regs);
}
```

Note that this now conflicts with that hex 80 "don't clear the screen" value. For VESA, it has simply moved from the high-order bit of the two-digit number to the high-order bit of the three-digit value (all VESA modes are three digits in size—again, see the sidebar for details). Therefore, to change to VESA mode 101, you make the VESA mode number 901, and the screen's contents will be preserved.

### All About Display Modes

IBM created a display mode standard that attempted to define all the display modes that could ever possibly be needed. These included all the possible pixel depths (number of colors) that would ever be needed. So IBM created 19 display modes (numbered from 0 to 13 hex). Table XIV.8a shows the display mode standard.

Table XIV.8a. Standard display modes.

| *Mode* | *Resolution* | *Graphics or Text?* | *Colors* |
|---|---|---|---|
| 0 | 40×25 | Text | Monochrome |
| 1 | 40×25 | Text | 16 |
| 2 | 80×25 | Text | Monochrome |
| 3 | 80×25 | Text | 16 |
| 4 | 320×200 | Graphics | 4 |
| 5 | 320×200 | Graphics | 4 grays |
| 6 | 640×200 | Graphics | Monochrome |
| 7 | 80×25 | Text | Monochrome |
| 8 | 160×200 | Graphics | 16 |
| 9 | 320×200 | Graphics | 16 |
| A | 640×200 | Graphics | 4 |
| B | Reserved for EGA BIOS use | | |
| C | Reserved for EGA BIOS use | | |
| D | 320×200 | Graphics | 16 |
| E | 640×200 | Graphics | 16 |
| F | 640×350 | Graphics | Monochrome |
| 10 | 640×350 | Graphics | 4 |
| 11 | 640×480 | Graphics | Monochrome |
| 12 | 640×480 | Graphics | 16 |
| 13 | 320×200 | Graphics | 256 |

See anything you recognize? Mode 3 is the 80×25 color text mode that you see when you turn on your PC. Mode 12 is what you get when you select "VGA" as your Microsoft Windows 3.*x* driver (you know, the one that comes with Windows). Note the lack of any display mode featuring more

*continues*

than 256 colors, or higher than 640×480 resolution. Modes 4, 9, and D for many years were the popular choice of DOS game makers, featuring a "big" resolution of 320×200 and enough colors (4 or 16) to display decent graphics. Mode 13 is the display mode used for just about all the popular action games, including DOOM (I and II), id Software's new Heretic, Apogee's Rise of the Triad, Interplay's Descent, and many others. In truth, many of these action games perform little tricks on the VGA cards that convert Mode 13 into 320×240, with more memory pages to improve graphics appearance and speed—they call it Mode X.

So where did all the other display modes, the ones you're used to, come from? They were made up by VGA card manufacturers. The other display modes you might be familiar with (800×600, 1024×768, 1280×1024, and even 1600×1200) come from a "melting pot" of sources, but regardless of their origins, the VGA card makers put them on their VGA cards to increase the cards' value. Such modes are usually called *extended display modes*. Thanks to the wonders of competition and capitalism, the card makers migrated toward these higher display modes. Others have been tried (ever heard of 1152×900?) but weren't received as well as these modes.

OK, so what is VESA, and what does it have to do with VGA cards? Although the VGA card makers all chose to support the same display modes (even the extended ones), each implemented these extended modes in its own proprietary way. Game and productivity software makers were stressed to support each proprietary method of each VGA card on the market. A group of manufacturers and other representatives formed a committee to standardize as much as possible the setup and programming of these cards. VESA (Video Electronics Standards Association) is this committee. The VESA committee adopted a standard of the extended display modes so that software could make common BIOS calls that would set up and initialize all VGA cards that adhered to this standard. It probably is safe to say that 100 percent of all VGA cards sold in the United States support the VESA standard in one form or another.

All *VESA modes* (which is what the standardized set of display modes is called) utilize mode numbers that are nine bits wide rather than the eight-bits-wide standard mode. Having a nine-bit mode number allows the VESA modes to be three hex digits long rather than two in the IBM standard (the ones from 0 to 13 hex shown in Table XIV.8a), thereby avoiding a numbering conflict. Therefore, all VESA mode values are above 100 hex. Here's how the VESA modes work: You want to program your VGA card to display 1024×768 at 256 colors, which happens to be VESA mode 105. You make a BIOS call using 105 as the display mode number. The VESA mode number is translated into the internal proprietary number by the Video BIOS (sometimes called the *VESA BIOS*) to actually perform the mode switch. VGA card manufacturers supply a VESA BIOS with each VGA card to perform these translations so that all you have to worry about is the VESA number. Table XIV.8b shows the latest list of VESA display modes (it is an ever-evolving standard).

Table XIV.8b. VESA display modes.

| *Resolution* | *Colors* | *VESA Mode* |
|---|---|---|
| 640×400 | 256 | 100 |
| 640×480 | 256 | 101 |

| *Resolution* | *Colors* | *VESA Mode* |
|---|---|---|
| 640×480 | 32,768 | 110 |
| 640×480 | 65,536 | 111 |
| 640×480 | 16.7 million | 112 |
| 800×600 | 16 | 102 |
| 800×600 | 256 | 103 |
| 800×600 | 32,768 | 113 |
| 800×600 | 65,536 | 114 |
| 800×600 | 16.7 million | 115 |
| 1024×768 | 16 | 104 |
| 1024×768 | 256 | 105 |
| 1024×768 | 32,768 | 116 |
| 1024×768 | 65,536 | 117 |
| 1024×768 | 16.7 million | 118 |
| 1280×1024 | 16 | 106 |
| 1280×1024 | 256 | 107 |
| 1280×1024 | 32,768 | 119 |
| 1280×1024 | 65,536 | 11A |
| 1280×1024 | 16.7 million | 11B |

Notice that these are the modes you are accustomed to seeing, especially if you use Microsoft Windows.

## Cross Reference:

XIV.6: What are interrupts?

# XIV.9: Does operator precedence always work (left to right, right to left)?

## *Answer:*

If you mean "Does a right-to-left precedence operator ever go left to right, and vice versa?" the answer is no. If you mean "Can a lower-order precedence ever be risen above a higher-order precedence?" the answer is yes. Table XIV.9 lists the order of each operator from top to bottom (highest order to lowest) and shows each operator's associativity.

Table XIV.9. Operator precedence.

| *Operator* | *Associativity* |
|---|---|
| `() [] -> .` | Left to right |
| `! ~ ++ -- -` (typecast) `* & sizeof` | Right to left |
| `* / %` | Left to right |
| `+ -` | Left to right |
| `<< >>` | Left to right |
| `< <= > >=` | Left to right |
| `== !=` | Left to right |
| `&` | Left to right |
| `^` | Left to right |
| `\|` | Left to right |
| `&&` | Left to right |
| `\|\|` | Left to right |
| `?:` | Right to left |
| `= += -=` | Right to left |
| `,` | Left to right |

Note in the table that the `!=` operator takes precedence over the `=` operator (in fact, practically everything takes precedence over the `=` operator). The following two source lines illustrate how precedence of one operator over another can get a programmer into trouble:

```
while(ch = getch() != 27)  printf("Got a character\n");
while((ch = getch()) != 27)  printf("Got a character\n");
```

Obviously, the purpose of this code is to get a character from the keyboard and check it against decimal 27 (the Escape key). Unfortunately, in source line one, the `getch()` is compared to the Escape key. The *resulting test* (which will return a `TRUE` or `FALSE`), not the character from the keyboard, is placed into `ch`. This is due to the precedence of the `!=` operator over the `=` operator.

In the second source line, a set of parentheses was added to surround the `ch = getch()` operation. Because parentheses are the highest order of precedence, the keyboard character is placed into `ch`, then checked against the Escape key. This final check will return `TRUE` or `FALSE` to the `while` statement, which is exactly what is desired (while this statement is `TRUE`, print this sentence). As a matter of detail, it should be pointed out that `ch` is not checked against 27; the result of the parenthetical statement `ch = getch()` is checked against 27. It might not make much difference in this case, but parentheses can really change the way code is created and executed. In the case of statements with multiple parenthetical statements, the code is executed from the innermost set of parentheses to the outermost, from left to right.

Note that the associativity in each operator's individual case (left to right or right to left) does not change, but the order of precedence does.

### Cross Reference:

None.

## XIV.10: Should a variable's type be declared within the header of a function or immediately following? Why?

### *Answer:*

The ANSI standard is to declare a variable's type within the header of the function. As you'll discover in Chapter XX, C was originally designed to run on the UNIX operating system back in the '70s. Naturally, this was before any sort of C ANSI standard. The declaration method for the original C compilers was to put the argument's type immediately after the function header.

Today, the ANSI standard dictates that the type be declared within the function's header. The problem with the old method was that it didn't allow for argument function type checking—the compiler could perform only return value checking. Without argument checking, there is no way to see whether the programmer is passing incorrect types to a function. By requiring arguments to be inside the function's header, and by requiring you to prototype your functions (including the argument types), the compiler can check for incorrect parameter passing in your code.

### Cross Reference:

XIV.11: Should programs always include a prototype for `main()`?

## XIV.11: Should programs always include a prototype for *main()*?

### *Answer:*

Sure, why not? Including the prototype is not required, but it is good programming practice. Everyone knows what the argument types are for `main()`, but your program can define the return type. You've probably noticed in the examples in this chapter (and possibly in other chapters) that `main()` is not prototyped, that no explicit return type is shown in the `main()` body itself, and that `main()` does not even contain a `return()` statement. By writing the examples in this way, I have implied a `void` function that returns an int. However, because there is no `return()` statement, a garbage value will be returned. This is not good programming practice. Good programming practice dictates a function prototype even for `main()`, and a proper return value for that prototype.

### Cross Reference:

XIV.12: Should `main()` always return a value?

# XIV.12: Should *main()* always return a value?

## *Answer:*

Your `main()` does not always have to return a value, because the calling function, which is usually COMMAND.COM, does not care much about return values. Occasionally, your program could be in a batch file that checks for a return code in the DOS `errorLevel` symbol. Therefore, return values from `main()` are purely up to you, but it is always good to return a value to the caller just in case.

Your `main()` can return `void` (or have no `return` statement) without problems.

## Cross Reference:

XIV.11: Should programs always include a prototype for `main()`?

# XIV.13: Can I control the mouse using the BIOS?

## *Answer:*

Yes. You can communicate with mouse services by using interrupt 33 hex. Table XIV.13 lists the most common mouse services available at interrupt 33.

Table XIV.13. Mouse interrupt services.

| *Number* | *Description* |
|---|---|
| 0 | Initialize Mouse; Hide If Currently Visible |
| 1 | Show Mouse |
| 2 | Hide Mouse |
| 3 | Get Mouse Position |
| 4 | Set Mouse Position |
| 6 | Check Whether Mouse Buttons Are Down |
| 7 | Set Horizontal Limits on Mouse |
| 8 | Set Vertical Limits on Mouse |
| 9 | Set Graphics Mode Mouse Shape |
| 10 | Set Text Mode Mouse Style |
| 11 | Get Mouse Delta Movement |

The following example code uses a few of the preceding mouse routines to manipulate a text mode mouse:

```
#include <stdlib.h>
#include <dos.h>

main()
```

```
{
    union REGS regs;

    printf("Initializing Mouse...");
    regs.x.ax = 0;
    int86(0x33, &regs, &regs);

    printf("\nShowing Mouse...");
    regs.x.ax = 1;
    int86(0x33, &regs, &regs);

    printf("\nMove mouse around. Press any key to quit...");
    getch();

    printf("\nHiding Mouse...");
    regs.x.ax = 2;
    int86(0x33, &regs, &regs);

    printf("\nDone\n");
}
```

When you run this example, a flashing block cursor that you can move around will appear on-screen. Using function 3, you can query the mouse handler for the position of the mouse at any time. In fact, I wrote an entire set of mouse library routines using the functions in Table XIV.13 that are incorporated into many of my text mode DOS programs.

To access these functions, you must have a mouse driver loaded. Typically, this occurs in AUTOEXEC.BAT, where a DOS mouse driver is loaded. However, it is becoming common to have only a Windows mouse driver loaded at Windows runtime. In that case, you will have to be running in a DOS shell to access the mouse functions.

## Cross Reference:

None.