15 Miscellaneous Features

- Enumerated Data Type
  - Uses of Enumerated Data Type
- Renaming Data Types with *typedef*
- Typecasting
- Bit fields
- Pointers to Functions
- Functions Returning Pointers
- Functions with Variable Number of Arguments
- Unions
  - Union of Structures
- Summary
- Exercise

The topics discussed in this chapter were either too large or far too removed from the mainstream C programming for inclusion in the earlier chapters. These topics provide certain useful programming features, and could prove to be of immense help in certain programming strategies. In this chapter we would examine enumerated data types, the typedef keyword, typecasting, bit fields, function pointers, functions with variable number of arguments and unions.

Enumerated Data Type

The enumerated data type gives you an opportunity to invent your own data type and define what values the variable of this data type can take. This can help in making the program listings more readable, which can be an advantage when a program gets complicated or when more than one programmer would be working on it. Using enumerated data type can also help you reduce programming errors.

As an example, one could invent a data type called **mar_status** which can have four possible values—single, married, divorced or widowed. Don't confuse these values with variable names; married for instance has the same relationship to the variable **mar_status** as the number 15 has with an integer variable.

The format of the **enum** definition is similar to that of a structure. Here's how the example stated above can be implemented:

```
enum mar_status
{
    single, married, divorced, widowed
};
enum mar_status  person1, person2 ;
```

Like structures this declaration has two parts:

(a) The first part declares the data type and specifies its possible values. These values are called 'enumerators'.
(b) The second part declares variables of this data type.

Now we can give values to these variables:

```
person1 = married ;
person2 = divorced ;
```

Remember we can't use values that aren't in the original declaration.

Thus, the following expression would cause an error:

```
person1 = unknown ;
```

Internally, the compiler treats the enumerators as integers. Each value on the list of permissible values corresponds to an integer, starting with 0. Thus, in our example, single, is stored as 0, married is stored as 1, divorced as 2 and widowed as 3.

This way of assigning numbers can be overridden by the programmer by initializing the enumerators to different integer values as shown below.

```
enum mar_status
{
    single = 100, married = 200, divorced = 300, widowed = 400
};
enum mar_status  person1, person2 ;
```

Uses of Enumerated Data Type

Enumerated variables are usually used to clarify the operation of a program. For example, if we need to use employee departments in a payroll program, it makes the listing easier to read if we use

values like Assembly, Manufacturing, Accounts rather than the integer values 0, 1, 2, etc. The following program illustrates the point I am trying to make.

```
main( )
{
    enum emp_dept
    {
        assembly, manufacturing, accounts, stores
    };
    struct employee
    {
        char name[30] ;
        int age ;
        float bs ;
        enum emp_dept department ;
    };
    struct employee e ;

    strcpy ( e.name, "Lothar Mattheus" ) ;
    e.age = 28 ;
    e.bs = 5575.50 ;
    e.department = manufacturing ;

    printf ( "\nName = %s", e.name ) ;
    printf ( "\nAge = %d", e.age ) ;
    printf ( "\nBasic salary = %f", e.bs ) ;
    printf ( "\nDept = %d", e.department ) ;

    if ( e.department == accounts )
        printf ( "\n%s is an accounant", e.name ) ;
    else
        printf ( "\n%s is not an accounant", e.name ) ;
}
```

And here is the output of the program...

```
Name = Lothar Mattheus
```

```
Age = 28
Basic salary = 5575.50
Dept = 1
Lothar Mattheus is not an accountant
```

Let us now dissect the program. We first defined the data type **enum emp_dept** and specified the four possible values, namely, assembly, manufacturing, accounts and stores. Then we defined a variable **department** of the type **enum emp_dept** in a structure. The structure **employee** has three other elements containing employee information.

The program first assigns values to the variables in the structure. The statement,

```
e.department = manufacturing ;
```

assigns the value manufacturing to **e.department** variable. This is much more informative to anyone reading the program than a statement like,

```
e.department = 1 ;
```

The next part of the program shows an important weakness of using **enum** variables... there is no way to use the enumerated values directly in input/output functions like **printf( )** and **scanf( )**.

The **printf( )** function is not smart enough to perform the translation; the department is printed out as 1 and not manufacturing. Of course we can write a function to print the correct enumerated values, using a **switch** statement, but that would reduce the clarity of the program. Even with this limitation, however, there are many situations in which enumerated variables are god sent!

Renaming Data types with *typedef*

There is one more technique, which in some situations can help to clarify the source code of a C program. This technique is to make use of the **typedef** declaration. Its purpose is to redefine the name of an existing variable type.

For example, consider the following statement in which the type **unsigned long int** is redefined to be of the type **TWOWORDS**:

```
typedef unsigned long int TWOWORDS ;
```

Now we can declare variables of the type **unsigned long int** by writing,

```
TWOWORDS var1, var2 ;
```

instead of

```
unsigned long int var1, var2 ;
```

Thus, **typedef** provides a short and meaningful way to call a data type. Usually, uppercase letters are used to make it clear that we are dealing with a renamed data type.

While the increase in readability is probably not great in this example, it can be significant when the name of a particular data type is long and unwieldy, as it often is with structure declarations. For example, consider the following structure declaration:

```
struct employee
{
    char name[30] ;
    int age ;
    float bs ;
} ;
```

```
struct employee e ;
```

This structure declaration can be made more handy to use when renamed using **typedef** as shown below:

```
struct employee
{
    char name[30] ;
    int age ;
    float bs ;
} ;
typedef struct employee EMP ;
EMP e1, e2 ;
```

Thus, by reducing the length and apparent complexity of data types, **typedef** can help to clarify source listing and save time and energy spent in understanding a program.

The above typedef can also be written as

```
typedef struct employee
{
    char name[30] ;
    int age ;
    float bs ;
}EMP ;
EMP e1, e2 ;
```

Typecasting

Sometimes we are required to force the compiler to explicitly convert the value of an expression to a particular data type. This would be clear from the following example:

```
main( )
{
    float a ;
```

```
    int x = 6, y = 4 ;

    a = x / y ;
    printf ( "\nValue of a = %f", a ) ;
}
```

And here is the output...

```
Value of a = 1.000000
```

The answer turns out to be 1.000000 and not 1.5. This is because, 6 and 4 are both integers and hence **6 / 4** yields an integer, 1. This 1 when stored in **a** is converted to 1.000000 . But what if we don't want the quotient to be truncated. One solution is to make either **x** or **y** as **float**. Let us say that other requirements of the program does not permit us to do this. In such a case what do we do? Use type casting. The following program illustrates this.

```
main( )
{
    float a ;
    int x = 6, y = 4 ;

    a = ( float ) x / y ;
    printf ( "\nValue of a = %f", a ) ;
}
```

And here is the output...

```
Value of a = 1.500000
```

This program uses type casting. This consists of putting a pair of parentheses around the name of the data type. In this program we said,

```
a = ( float ) x / y ;
```

The expression (**float**) causes the variable **x** to be converted from type **int** to type **float** before being used in the division operation.

Here is another example of type casting:

```
main( )
{
    float  a = 6.35 ;

    printf ( "\nValue of a on type casting = %d", ( int ) a ) ;
    printf ( "\nValue of a = %f", a ) ;
}
```

And here is the output...

```
Value of a on type casting = 6
Value of a = 6.350000
```

Note that the value of **a** doesn't get permanently changed as a result of typecasting. Rather it is the value of the expression that undergoes type conversion whenever the cast appears.

Bit Fields

If in a program a variable is to take only two values 1 and 0, we really need only a single bit to store it. Similarly, if a variable is to take values from 0 to 3, then two bits are sufficient to store these values. And if a variable is to take values from 0 through 7, then three bits will be enough, and so on.

Why waste an entire integer when one or two or three bits will do? Well, for one thing, there aren't any one bit or two bit or three bit data types available in C. However, when there are several variables whose maximum values are small enough to pack into a single memory location, we can use 'bit fields' to store several values in a single integer. To demonstrate how bit fields work, let

us consider an example. Suppose we want to store the following data about an employee. Each employee can:

(a) be male or female
(b) be single, married, divorced or widowed
(c) have one of the eight different hobbies
(d) can choose from any of the fifteen different schemes proposed by the company to pursue his/her hobby.

This means we need one bit to store gender, two to store marital status, three for hobby, and four for scheme (with one value used for those who are not desirous of availing any of the schemes). We need ten bits altogether, which means we can pack all this information into a single integer, since an integer is 16 bits long.

To do this using bit fields, we declare the following structure:

```
struct employee
{
    unsigned  gender : 1 ;
    unsigned  mar_stat : 2 ;
    unsigned  hobby : 3 ;
    unsigned  scheme : 4 ;
} ;
```

The colon in the above declaration tells the compiler that we are talking about bit fields and the number after it tells how many bits to allot for the field.

Once we have established a bit field, we can reference it just like any other structure element, as shown in the program given below:

```
#define MALE 0 ;
#define FEMALE 1 ;
#define SINGLE 0 ;
#define MARRIED 1 ;
#define DIVORCED 2 ;
```

```
#define WIDOWED 3 ;

main( )
{
    struct employee
    {
        unsigned gender : 1 ;
        unsigned mar_stat : 2 ;
        unsigned hobby : 3 ;
        unsigned scheme : 4 ;
    } ;
    struct employee e ;

    e.gender = MALE ;
    e.mar_status = DIVORCED ;
    e.hobby = 5 ;
    e.scheme = 9 ;

    printf ( "\nGender = %d", e.gender ) ;
    printf ( "\nMarital status = %d", e.mar_status ) ;
    printf ( "\nBytes occupied by e = %d", sizeof ( e ) ) ;
}
```

And here is the output...

```
Gender = 0
Marital status = 2
Bytes occupied by e = 2
```

Pointers to Functions

Every type of variable that we have discussed so far, with the exception of register, has an address. We have seen how we can reference variables of the type **char, int, float,** etc through their addresses—that is by using pointers. Pointers can also point to C functions. And why not? C functions have addresses. If we know

the function's address we can point to it, which provides another way to invoke it. Let us see how this can be done.

```
main( )
{
    int display( ) ;

    printf ( "\nAddress of function display is %u", display ) ;
    display( ) ; /* usual way of invoking a function */
}

display( )
{
    puts ( "\nLong live viruses!!" ) ;
}
```

The output of the program would be:

```
Address of function display is 1125
Long live viruses!!
```

Note that to obtain the address of a function all that we have to do is mention the name of the function, as has been done in the **printf( )** statement above. This is similar to mentioning the name of the array to get its base address.

Now let us see how using the address of a function we can manage to invoke it. This is shown in the program given below:

```
/* Invoking a function using a pointer to a function */
main( )
{
    int display( ) ;
    int ( *func_ptr )( ) ;

    func_ptr = display ; /* assign address of function */
    printf ( "\nAddress of function display is %u", func_ptr ) ;
```

```
    ( *func_ptr )( ) ;  /* invokes the function display( ) */
}

int  display( )
{
    puts ( "\nLong live viruses!!" ) ;
}
```

The output of the program would be:

```
Address of function display is 1125
Long live viruses!!
```

In **main( )** we declare the function **display( )** as a function returning an **int**. But what are we to make of the declaration,

```
int ( *func_ptr )( ) ;
```

that comes in the next line? We are obviously declaring something that, like display( ), will return an int, but what is it? And why is *func_ptr enclosed in parentheses?

If we glance down a few lines in our program, we see the statement,

```
func_ptr = display ;
```

so we know that **func_ptr** is being assigned the address of **display( )**. Therefore, **func_ptr** must be a pointer to the function **display( )**. Thus, all that the declaration

```
int ( *func_ptr )( ) ;
```

means is, that **func_ptr** is a pointer to a function, which returns an **int**. And to invoke the function we are just required to write the statement,

```
( *func_ptr )( ) ;
```

Pointers to functions are certainly awkward and offputting. And why use them at all when we can invoke a function in a much simpler manner? What is the possible gain of using this esoteric feature of C? There are two possible uses:

(a) in writing memory resident programs
(b) in writing viruses, or vaccines to remove the viruses

Both these topics form interesting and powerful C applications and would call for separate book on each if full justice is to be given to them. Much as I would have liked to, for want of space I would have to exclude these topics.

Functions Returning Pointers

The way functions return an **int**, a **float**, a **double** or any other data type, it can even return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function definition. The following program illustrates this.

```
main( )
{
    int *p ;
    int *fun( ) ;

    p = fun ( ) ;
}

int *fun( )
{
    int i = 20 ;
    return ( &i ) ;
}
```

This program just indicates how an integer pointer can be returned from a function. Beyond that it doesn't serve any useful purpose. This concept can be put to use while handling strings. For example look at the following program which copies one string into another and returns the pointer to the target string.

```
main( )
{
    char *str ;
    char *copy( ) ;
    char source[ ] = "Jaded" ;
    char target[10] ;

    str = copy ( target, source ) ;
    printf ( "\n%s", str ) ;
}

char *copy ( char *t, char *s )
{
    char *r ;

    r = t ;

    while ( *s != '\0' )
    {
        *t = *s ;
        t++ ;
        s++ ;
    }

    *t = '\0' ;
    return ( r ) ;
}
```

Here we have sent the base addresses of **source** and **target** strings to **copy( )**. In the **copy( )** function the **while** loop copies the characters in the source string into the target string. Since during

copying **t** is continuously incremented, before entering into the loop the initial value of **t** is safely stored in the character pointer **r**. Once copying is over this character pointer **r** is returned to **main( )**.

Functions with Variable Number of Arguments

We have used **printf( )** so often without realizing how it works properly irrespective of how many arguments we pass to it. How do we go about writing such routines that can take variable number of arguments? And what have pointers got to do with it? There are three macros available in the file "stdarg.h" called **va_start**, **va_arg** and **va_list** which allow us to handle this situation. These macros provide a method for accessing the arguments of the function when a function takes a fixed number of arguments followed by a variable number of arguments. The fixed number of arguments are accessed in the normal way, whereas the optional arguments are accessed using the macros **va_start** and **va_arg**. Out of these macros **va_start** is used to initialize a pointer to the beginning of the list of optional arguments. On the other hand the macro **va_arg** is used to advance the pointer to the next argument. Let us put these concepts into action using a program. Suppose we wish to write a function **findmax( )** which would find out the maximum value from a set of values, irrespective of the number of values passed to it.

```
# include "stdarg.h"
main( )
{
    int max ;
    int findmax ( int, ... ) ;

    max = findmax ( 5, 23, 15, 1, 92, 50 ) ;
    printf ( "\nmaximum = %d", max ) ;

    max = findmax ( 3, 100, 300, 29 ) ;
```

```
    printf ( "\nmaximum = %d", max ) ;
}

int findmax ( int tot_num, ... )
{
    int max, count, num ;

    va_list ptr ;

    va_start ( ptr, tot_num ) ;
    max = va_arg ( ptr, int ) ;

    for ( count = 1 ; count < tot_num ; count++ )
    {
        num = va_arg ( ptr, int ) ;
        if ( num > max )
            max = num ;
    }

    return ( max ) ;
}
```

Note how the **findmax( )** function has been declared. The ellipses (...) indicate that the number of arguments after the first argument would be variable.

Here we are making two calls to **findmax( )** first time to find maximum out of 5 values and second time to find maximum out of 3 values. Note that for each call the first argument is the count of arguments that follow the first argument. The value of the first argument passed to **findmax( )** is collected in the variable **tot_num**. **findmax( )** begins with a declaration of a pointer **ptr** of the type **va_list**. Observe the next statement carefully:

```
va_start ( ptr, tot_num ) ;
```

This statement sets up **ptr** such that it points to the first variable argument in the list. If we are considering the first call to **finndmax( ) ptr** would now point to 23. The statement **max = va_arg (ptr, int)** would assign the integer being pointed to by **ptr** to **max**. Thus 23 would be assigned to **max**, and **ptr** would now start pointing to the next argument, i.e 15. The rest of the program is fairly straightforward. We just keep picking up successive numbers in the list and keep comparing them with the latest value in **max**, till all the arguments in the list have been scanned. The final value in **max** is then returned to **main( )**.

How about another program to fix your ideas? This one calls a function **display( )** which is capable of printing any number of arguments of any type.

```
#include "stdarg.h"

main( )
{
    void display ( int, int, ... ) ;

    display ( 1, 2, 5, 6 ) ;
    display ( 2, 4, 'A', 'a', 'b', 'c' ) ;
    display ( 3, 3, 2.5, 299.3, -1.0 ) ;
}

void display ( int type, int num, ... )
{
    int i, j ;
    char c ;
    float f ;
    va_list ptr ;

    va_start ( ptr, num ) ;
    printf ( "\n" ) ;
    switch ( type )
    {
        case 1 :
```

```
            for ( j = 1 ; j <= num ; j++ )
            {
                i = va_arg ( ptr, int ) ;
                printf ( "%d ", i ) ;
            }
            break ;

        case 2 :
            for ( j = 1 ; j <= num ; j++ )
            {
                c = va_arg ( ptr, char ) ;
                printf ( "%c ", c ) ;
            }
            break ;

        case 3 :
            for ( j = 1 ; j <= num ; j++ )
            {
                f = ( float ) va_arg ( ptr, double ) ;
                printf ( "%f ", f ) ;
            }
    }
}
```

Here we pass two fixed arguments to the function **display( )**. The first one indicates the data type of the arguments to be printed and the second indicates the number of such arguments to be printed. Once again through the statement **va_start (ptr, num)** we set up **ptr** such that it points to the first argument in the variable list of arguments. Then depending upon whether the value of type is 1, 2 or 3 we print out the arguments as **ints**, **chars** or **floats**.

In all calls to **display( )** the first argument indicated how many values are we trying to print. Contrast this with **printf( )**. To it we never pass an argument indicating how many value are we trying to print. Then how does **printf( )** figure this out? Simple. It scans

the format string and counts the number of format specifiers that we have used in it to decide how many values are being printed.

Unions

Unions are derived data types, the way structures are. But Unions have the same relationship to structures that you might have with a distant cousin who resembled you but turned out to be smuggling contraband in Mexico. That is, unions and structures look alike, but are engaged in totally different enterprises.

Both structures and unions are used to group a number of different variables together. But while a structure enables us treat a number of different variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables. That is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type on another occasion.

You might wonder why it would be necessary to do such a thing, but we will be seeing several very practical applications of unions soon. First, let us take a look at a simple example:

```
/* Demo of union at work */
main( )
{
    union a
    {
        int i ;
        char ch[2] ;
    } ;
    union a key ;

    key.i = 512 ;
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
```

```
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;
}
```

And here is the output...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

As you can see, first we declared a data type of the type **union a**, and then a variable **key** to be of the type **union a**. This is similar to the way we first declare the structure type and then the structure variables. Also, the union elements are accessed exactly the same way in which the structure elements are accessed, using a '.' operator. However, the similarity ends here. To illustrate this let us compare the following data types:

```
struct a
{
    int i ;
    char ch[2] ;
} ;
struct a key ;
```

This data type would occupy 4 bytes in memory, 2 for **key.i** and one each for **key.ch[0]** and **key.ch[1]**, as shown in Figure 15.1.

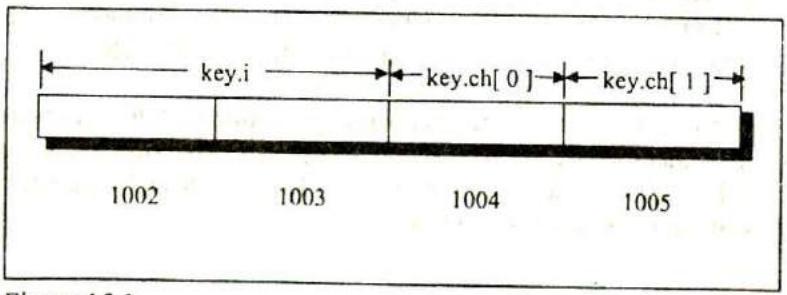

Figure 15.1

Now we declare a similar data type, but instead of using a structure we use a union.

```
union a
{
    int i ;
    char ch[2] ;
} ;
union a key ;
```

Representation of this data type in memory is shown in Figure 15.2.

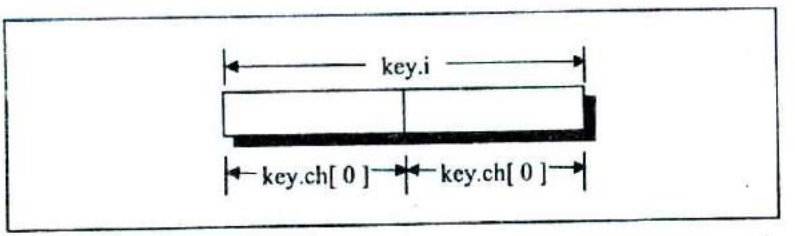

Figure 15.2

As shown in Figure 15.2, the union occupies only 2 bytes in memory. Note that the same memory locations which are used for **key.i** are also being used by **key.ch[0]** and **key.ch[1]**. It means that the memory locations used by **key.i** can also be accessed using **key.ch[0]** and **key.ch[1]**. What purpose does this serve? Well, now we can access the two bytes simultaneously (by using **key.i**) or the same two bytes individually (using **key.ch[0]** and **key.ch[1]**).

This is a frequent requirement while interacting with the hardware. i.e. sometimes we are required to access two bytes simultaneously and sometimes each byte individually. Faced with such a situation, using union is the answer, usually.

Perhaps you would be able to understand the union data type more thoroughly if we take a fresh look at the output of the above program. Here it is...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

Let us understand this output in detail. 512 is an integer, a 2 byte number. It's binary equivalent will be 0000 0010 0000 0000. We would expect that this binary number when stored in memory would look as shown below.

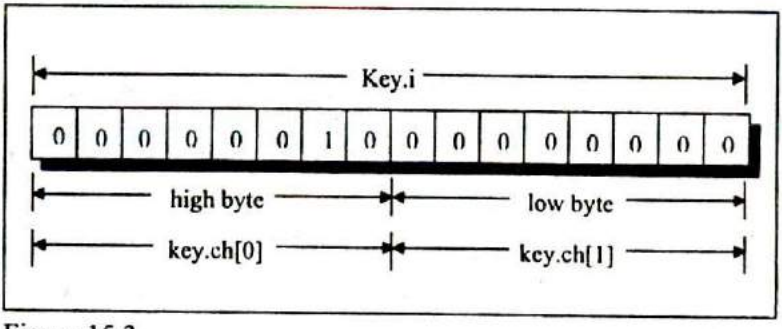

Figure 15.3

If the number is stored in this manner then, the output of **key.ch[0]** and **key.ch[1]** should have been 2 and 0. But, if you look at the output of the program written above, it is exactly the opposite. Why is it so? Because, when a two-byte number is stored in memory, the low byte is stored before the high byte. It means, actually 512 would be stored in memory as shown in Figure 15.4.

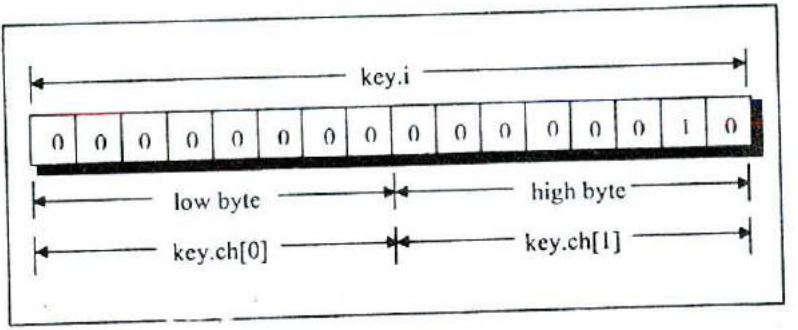

Figure 15.4

Now, we can see why value of **key.ch[0]** is printed as 0 and value of **key.ch[1]** is printed as 2.

One last thing. We can't assign different values to the different union elements at the same time. That is, if we assign a value to **key.i**, it gets automatically assigned to **key.ch[0]** and **key.ch[1]**. Vice versa, if we assign a value to **key.ch[0]** or **key.ch[1]**, it is bound to get assigned to **key.i**. Here is a program that illustrates this fact.

```
main( )
{
    union a
    {
        int i ;
        char ch[2] ;
    } ;
    union a key ;

    key.i = 512 ;
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;

    key.ch[0] = 50 ; /* assign a new value to key.ch[0] */
```

```
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;
}
```

And here is the output...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
key.i= 562
key.ch[0] = 50
key.ch[1] = 2
```

Before we move on to the next section, let us reiterate that a union provides a way to look at the same data in several different ways. For example, there can exist a union as shown below.

```
union b
{
    double d ;
    float f[2] ;
    int i[4] ;
    char ch[8] ;
} ;
union b data ;
```

In what different ways can the data be accessed from it? Sometimes as a complete set of eight bytes (**data.d**), sometimes as two sets of 4 bytes each (**data.f[0]** and **data.f[1]**), sometimes as four sets of 2 bytes each (**data.i[0]**, **data.i[1]**, **data.i[2]** and **data.[3]**) and sometimes as eight individual bytes (**data.ch[0]**, **data.ch[1]... data.ch[7]**).

Union of Structures

Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union. Here is an example of stuctures nested in a union.

```
main( )
{
    struct a
    {
        int i ;
        char c[2] ;
    } ;
    struct b
    {
        int j ;
        char d[2] ;
    } ;
    union z
    {
        struct a key ;
        struct b data ;
    } ;
    union z strange ;

    strange.key.i = 512 ;
    strange.data.d[0] = 0 ;
    strange.data.d[1] = 32 ;

    printf ( "\n%d", strange.key.i ) ;
    printf ( "\n%d", strange.data.j ) ;
    printf ( "\n%d", strange.key.c[0] ) ;
    printf ( "\n%d", strange.data.d[0] ) ;
    printf ( "\n%d", strange.key.c[1] ) ;
    printf ( "\n%d", strange.data.d[1] ) ;
}
```

And here is the output...

```
512
512
0
0
32
32
```

Just as we do with nested structures, we access the elements of the union in this program using the '.' operator twice. Thus,

```
strange.key.i
```

refers to the variable **i** in the structure **key** in the union **strange**. Analysis of the output of the above program is left to the reader.

Summary

(a) The enumerated data type and the **typedef** declaration add to the clarity of the program.
(b) Typecasting makes the data type conversions for specific operations.
(c) When the information to be stored can be represented using a few bits of a byte we can use bit fields to pack more information in a byte.
(d) Every C function has an address that can be stored in a pointer to a function. Pointers to functions provide one more way to call functions.
(e) We can write a function that receives a variable number of arguments.
(f) Unions permit access to same memory locations in multiple ways.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    enum status { pass, fail, atkt } ;
    enum status  stud1, stud2, stud3 ;
    stud1 = pass ;
    stud2 = fail ;
    stud3 = atkt ;
    printf ( "\n%d %d %d", stud1, stud2, stud3 ) ;
}
```

(b)
```
main( )
{
    printf ( "%f", ( float ) ( ( int ) 3.5 / 2 ) ) ;
}
```

(c)
```
main( )
{
    float  i, j ;
    i = ( float ) 3 / 2 ;
    j = i * 3 ;
    printf ( "\n%d", ( int ) j ) ;
}
```

[B] Point out the error, if any, in the following programs:

(a)
```
main( )
{
    typedef struct patient
    {
        char  name[20] ;
        int  age ;
        int  systolic_bp ;
        int  diastolic_bp ;
    } ptt ;
    ptt p1 = { "anil", 23, 110, 220 } ;
    printf ( "\n%s %d", p1.name, p1.age ) ;
    printf ( "\n%d %d", p1.systolic_bp, p1.diastolic_bp );
```

```
}
```

(b)
```
main( )
{
    void show( ) ;
    void ( *s )( ) ;
    s = show ;
    ( *s )( ) ;
}
void show( )
{
    printf ( "\ndon't show off. It won't pay in the long run"
}
```

(c)
```
main( )
{
    int show( ) ;
    int ( *s )( ) ;
    s = show( );
    ( *s )( ) ;
}
float show( )
{
    printf ( "\nControl did reach here" ) ;
    return ( 3.33 ) ;
}
```

(d)
```
main( )
{
    void show( int, float ) ;
    void ( *s )( int, float ) ;
    s = show ;
    ( *s )( 10, 3.14 ) ;
}
show ( int i, float f )
{
    printf ( "\n %d %f", i, f ) ;
}
```

[C] Attempt the following:

(a) Create an array of four function pointers. Each pointer should point to a different function. Each of these functions should receive two integers and return a float. Using a loop call each of these functions using the addresses present in the array.

(b) Write a function that receives variable number of arguments, where the arguments are the coordinates of a point. Based on the number of arguments received, the function displays, type of shape like a point, line, triangle, etc. that can be drawn.

(c) Write a program, which stores information about a date in a structure containing three members—day, month and year. Using bit fields the day number should get stored in first 5 bits of day, the month number in 4 bits of month and year in 12 bits of year. Write a program to read date of joining of 10 employees and display them in ascending order of year.

(d) Write a program to read and store information about insurance policy holder. The information contains details like gender, whether the holder is minor/major, policy name and duration of the policy. Make use of bit-fields to store this information.

16 C Under Windows

- Which Windows...
- Integers
- The Use of typedef
- Pointers in the 32-bit World
  - Memory Management
  - Device Access
- DOS Programming Model
- Windows Programming Model
  - Event Driven Model
- Windows Programming, a Closer Look
- The First Windows Program
- Hungarian Notation
- Summary
- Exercise

So far we have learnt every single keyword, operator and instruction available in C. Thus we are through with the language elements that were there to learn. We did all this learning by compiling our programs using a 16-bit compiler like Turbo C/C++. Now it is time to move on to more serious stuff. To make a beginning one has to take a very important decision—should we attempt to build programs that are targeted towards 16-bit environments like MS-DOS or 32-bit environments like Windows/Linux. Obviously we should choose the 32-bit platform because that is what is in commercial use today and would remain so until 64-bit environment takes over in future. That raises a very important question—is it futile to learn C programming using 16-bit compiler like Turbo C/C++? Absolutely not! The typical 32-bit environment offers so many features that the beginner is likely to feel lost. Contrasted with this, 16-bit compilers offer a very simple environment that a novice can master quickly.

Now that the C fundamentals are out of the way and you are confident about the language features it is time for us to delve into the modern 32-bit operating environments. In today's commercial world 16-bit operating environments like DOS are more or less dead. More and more software is being created for 32-bit environments like Windows and Linux. In this chapter we would explore how C programming is done under Windows. Chapters 20 & 21 are devoted to exploring C under Linux.

Which Windows...

To a common user the differences amongst various versions of Windows like Windows 95,98, ME, NT, 2000, XP, Server 2003 is limited to only visual appearances—things like color of the title bar, shape of the buttons, desktop, task bar, programs menu etc. But the truth is much farther than that. Architecturally there are huge differences amongst them. So many are the differences that Microsoft categorizes the different versions under two major heads—Consumer Windows and Windows NT Family. Windows

95, 98, ME fall under the Consumer Windows, whereas Windows NT, 2000, XP, Server 2003 fall under the Windows NT Family. Consumer Windows was targeted towards the home or small office users, whereas NT family was targeted towards business users. Microsoft no longer provides support for Consumer Windows. Hence in this book we would concentrate only on NT Family Windows. So in the rest of this book whenever I refer to Windows I mean Windows NT family, unless explicitly specified.

Before we start writing C programs under Windows let us first see some of the changes that have happened under Windows environment.

Integers

Under 16-bit environment the size of integer is of 2 bytes. As against this, under 32-bit environment an integer is of 4 bytes. Hence its range is -2147483648 to +2147483647. Thus there is no difference between an **int** and a **long int**. But what if we wish to store the age of a person in an integer? It would be improper to sacrifice a 4-byte integer when we know that the number to be stored in it is hardly going to exceed hundred. In such as case it would be more sensible to use a **short int** since it is only 2 bytes long.

The Use of *typedef*

Take a look at the following declarations:

```
COLORREF color ;
HANDLE h ;
WPARAM w ;
LPARAM l ;
BOOL b ;
```

Are COLORREF, HANDLE, etc. new datatypes that have been added in C under Windows compiler? Not at all. They are merely typedef's of the normal integer datatype.

A typical C under Windows program would contain several such **typedefs**. There are two reasons why Windows-based C programs heavily make use of **typedefs**. These are:

(a) A typical Windows program is required to perform several complex tasks. For example a program may print documents, send mails, perform file I/O, manage multiple threads of execution, draw in a window, play sound files, perform operations over the network apart from normal data processing tasks. Naturally a program that carries out so many tasks would be very big in size. In such a program if we start using the normal integer data type to represent variables that hold different entities we would soon lose track of what that integer value actually represents. This can be overcome by suitably typedefining the integer as shown above.

(b) At several places in Windows programming we are required to gather and work with dissimilar but inter-related data. This can be done using a structure. But when we define any structure variable we are required to precede it with the keyword **struct**. This can be avoided by using **typedef** as shown below:

```
struct rect
{
    int top ;
    int left ;
    int right ;
    int bottom ;
} ;

typedef struct rect RECT ;
typedef struct rect* PRECT ;
```

```
RECT r ;
PRECT pr ;
```

What have we achieved out of this? It makes user-defined data types like structures look, act and behave similar to standard data types like integers, floats, etc. You would agree that the following declarations

```
RECT r ;
int i ;
```

are more logical than

```
struct RECT r ;
int i ;
```

Imagine a situation where each programmer **typedef**s the integer to represent a color in different ways. Some of these could be as follows:

```
typedef int COL ;
typedef int COLOR ;
typedef int COLOUR ;
typedef int COLORREF ;
```

To avoid this chaos Microsoft has done several **typedef**s for commonly required entities in Windows programming. All these have been stored in header files. These header files are provided as part of 32-bit compiler like Visual C++.

Pointers in the 32-bit World

In a 16-bit world (like MS-DOS) we could run only one application at a time. If we were to run another program we were required to terminate the first one before launching the second. As only one program (task) could run at a time this environment was

called single-tasking environment. Since only one program could run at any given time entire resources of the machine like memory and hardware devices were accessible to this program. Under 32-bit environment like Windows several programs reside and work in memory at the same time. Hence it is known as a multi-tasking environment. But the moment there are multiple programs running in memory there is a possibility of conflict if two programs simultaneously access the machine resources. To prevent this, Windows does not permit any application direct access to any machine resource. To channelize the access without resulting into conflict between applications several new mechanisms were created in the Microprocessor & OS. This had a direct bearing on the way the application programs are created. This is not a Windows OS book. So we would restrict our discussion about the new mechanisms that have been introduced in Windows to topics that are related, to C programming. These topics are 'Memory Management and Device Access'.

Memory Management

Since users have become more demanding, modern day applications have to contend with these demands and provide several features in them. To add to this, under Windows several such applications run in memory simultaneously. The maximum allowable memory—1 MB—that was used in 16-bit environment was just too small for this. Hence Windows had to evolve a new memory management model. Since Windows runs on 32-bit microprocessors each CPU register is 32-bit long. Whenever we store a value at a memory location the address of this memory location has to be stored in the CPU register at some point in time. Thus a 32-bit address can be stored in these registers. This means that we can store 2^{32} unique addresses in the registers at different times. As a result, we can access 4 GB of memory locations using 32-bit registers. As pointers store addresses, every pointer under 32-bit environment also became a 4-byte entity.

However, if we decide to install 4 GB memory it would cost a lot. Hence Windows uses a memory model which makes use of as much of physical memory (say 128 MB) as has been installed and simulates the balance amount of memory (4 GB – 128 MB) on the hard disk. Be aware that this balance memory is simulated as and when the need to do so arises. Thus memory management is demand based.

Note that programs cannot execute straight-away from hard disk. They have to be first brought into physical memory before they can get executed. Suppose there are multiple programs already in memory and a new program starts executing. If this new program needs more memory than what is available right now, then some of the existing programs (or their parts) would be transferred to the disk in order to free the physical memory to accommodate the new program. This operation is often called page-out operation. Here page stands for a block of memory (usually of size 4096 bytes). When that part of the program that was paged out is needed it is brought back into memory (called page-in operation) and some other programs (or their parts) are paged out. This keeps on happening without a common user's knowledge all the time while working with Windows. A few more facts that you must note about paging are as follows:

(a) Part of the program that is currently executing might also be paged out to the disk.

(b) When the program is paged in (from disk to memory) there is no guarantee that it would be brought back to the same physical location where it was before it was paged out.

Now imagine how the paging operations would affect our programming. Suppose we have a pointer pointing to some data present in a page. If this page gets paged out and is later paged in to a different physical location then the pointer would obviously have a wrong address. Hence under Windows the pointer never holds the physical address of any memory location. It always holds a virtual address of that location. What is this virtual address? At

its name suggests it is certainly not a real address. It is a number, which contains three parts. These parts when used in conjunction with a CPU register called CR3 and contents of two tables called Page Directory Table and Page Table leads to the actual physical address. This is shown in Figure 16.1.

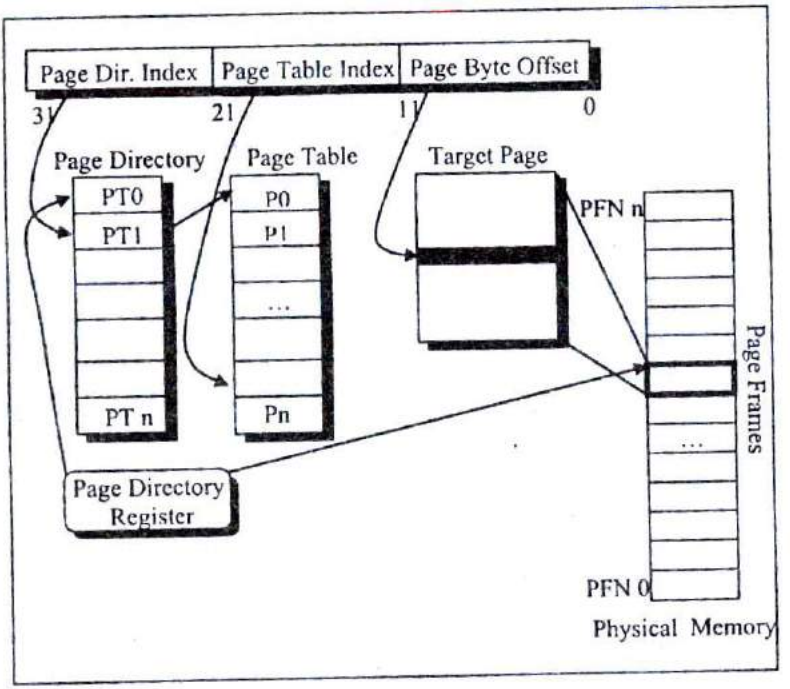

Figure 16.1

The CR3 register holds the physical location of Page Directory Table. The left part of the 32-bit virtual address holds the index into the Page Directory Table. The value present at this index is the starting address of the Page Table. The middle part of the 32-bit virtual address holds the index into the Page Table. The value present at this index is the starting address of the physical page in memory. The right part of the 32-bit virtual address holds the byte

offset (from the start of the page) of the physical memory location to be accessed.

Note that the CR3 register is not accessible from an application. Hence an application can never directly reach a physical address. Also, as the paging activity is going on the OS would suitably keep updating the values in the two tables.

Device Access

All devices under Windows are shared amongst all the running programs. Hence no program is permitted a direct access to any of the devices. The access to a device is routed through a device driver program, which finally accesses the device. There is a standard way in which an application can communicate with the device driver. It is device driver's responsibility to ensure that multiple requests coming from different applications are handled without causing any conflict. This standard way of communication is discussed in detail in Chapter 17.

DOS Programming Model

Typical 16-bit environments like DOS use a sequential programming model. In this model programs are executed from top to bottom in an orderly fashion. The path along which the control flows from start to finish may vary during each execution depending on the input that the program receives or the conditions under which it is run. However, the path remains fairly predictable. C programs written in this model begin execution with **main( )** (often called entry point) and then call other functions present in the program. If you assume some input data you can easily walk through the program from beginning to end. In this programming model it is the program and not the operating system that determines which function gets called and when. The operating system simply loads and executes the program and then waits for it to finish. If the program wishes it can take help of the OS to carry

out jobs like console I/O, file I/O, printing, etc. For other operations like generating graphics, carrying out serial communication, etc. the program has to call another set of functions called ROM-BIOS functions.

Unfortunately the DOS functions and the BIOS functions do not have any names. Hence to call them the program had to use a mechanism called interrupts. This is a messy affair since the programmer has to remember interrupt numbers for calling different functions. Moreover, communication with these functions has to be done using CPU registers. This lead to lot of difficulties since different functions use different registers for communication. To an extent these difficulties are reduced by providing library functions that in turn call the DOS/BIOS functions using interrupts. But the library doesn't have a parallel function for every DOS/BIOS function. DOS functions either call BIOS functions or directly access the hardware.

At times the programs are needed to directly interact with the hardware. This has to be done because either there are no DOS/BIOS functions to do this, or if they are there their reach is limited.

Figure 16.2 captures the essence of the DOS programming model.

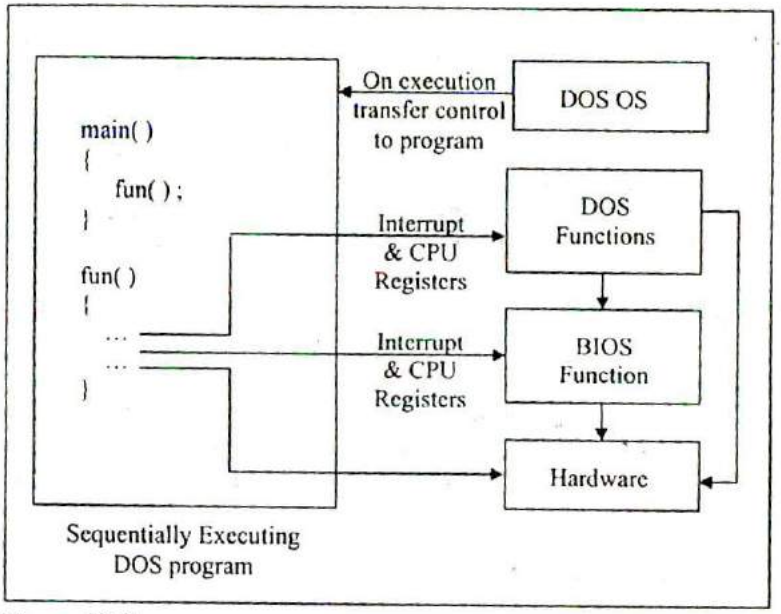

Figure 16.2

From the above discussion you can gather that there are several limitations in the DOS programming model. These have been listed below:

No True Reuse

The library functions that are called from each program become part of the executable file (.EXE) for that program. Thus the same functions get replicated in several EXE files, thereby wasting precious disk space.

Inconsistent Look and Feel

Every DOS program has a different user interface that the user has to get used to before he can start getting work out of the program. For example, successful DOS-based software like Lotus 1-2-3, Foxpro, Wordstar offered different types of menus. This happened because DOS/BIOS doesn't provide any functions for creating user interface elements like menus. As the look and feel of all DOS based programs is different, the user takes a lot of time in learning how to interact with the program

Messy Calling Mechanism

It is difficult to remember interrupt numbers and the registers that are to be used for communication with DOS/BIOS functions. For example, if we are to position the cursor on the screen using a BIOS function we are required to remember the following details:

Interrupt number – 16
CPU Registers to be used:
AH – 2 (service number)
DH – Row number where cursor is to be positioned
DL – Column number where cursor is to be positioned

While using these interrupt numbers and registers there is always a chance of error.

Hardware Dependency

DOS programs are always required to bother about the details of the hardware on which they are running. This is because for every new piece of hardware introduced there are new interrupt numbers and new register details. Hence DOS programmers are under the constant fear that if the hardware on which the programs are running changes then the program may crash.

Moreover the DOS programmer has to write lot of code to detect the hardware on which his program is running and suitably make use of the relevant interrupts and registers. Not only does this make the program lengthy, the programmer has to understand a lot of technical details of the hardware. As a result the programmer has to spend more time in understanding the hardware than in the actual application programming.

Windows Programming Model

From the perspective of the user the shift from MS-DOS to Windows OS involves switching over to a Graphical User Interface from the typical Text Interface that MS-DOS offers. Another change that the user may feel and appreciate is the ability of Windows OS to execute several programs simultaneously, switching effortlessly from one to another by pointing at windows and clicking them with the mouse. Mastering this new GUI environment and getting comfortable with the multitasking feature is at the most a matter of a week or so. However, from the programmer's point of view programming for Windows is a whole new ball game!

Windows programming model is designed with a view to:

(a) Eliminate the messy calling mechanism of DOS
(b) Permit true reuse of commonly used functions
(c) Provide consistent look and feel for all applications
(d) Eliminate hardware dependency

Let us discuss how Windows programming model achieves this.

Better Calling Mechanism

Instead of calling functions using Interrupt numbers and registers Windows provides functions within itself which can be called using names. These functions are called API (Application Programming Interface) functions. There are literally hundreds of

API functions available. They help an application to perform various tasks such as creating a window, drawing a line, performing file input/output, etc.

True Reuse

A C under Windows program calls several API functions during course of its execution. Imagine how much disk space would have been wasted had each of these functions become part of the EXE file of each program. To avoid this, the API functions are stored in special files that have an extension .DLL.

DLL stands for Dynamic Link Libraries. A DLL is a binary file that provides a library of functions. The functions present in DLLs can be linked during execution. These functions can also be shared between several applications running in Windows. Since linking is done dynamically the functions do not become part of the executable file. As a result, the size of EXE files does not go out of hand. It is also possible to create your own DLLs. You would like to do this for two reasons:

(a) Sharing common code between different executable files.

(b) Breaking an application into component parts to provide a way to easily upgrade application's components.

The Windows API functions come in three DLL files. Figure 16.3 lists these filenames along with purpose of each.

DLL	Description
USER32.DLL	Contains functions that are responsible for window management, including menus, cursors, communications, timer etc.
GDI32.DLL	Contains functions for graphics drawing and painting
KERNEL32.DLL	Contains functions to handle memory management, threading, etc.

Figure 16.3

Consistent Look and Feel

Consistent look and feel means that each program offers a consistent and similar user interface. As a result, user doesn't have to spend long periods of time mastering a new program. Every program occupies a window—a rectangular area on the screen. A window is identified by a title bar. Most program functions are initiated through the program's menu. The display of information too large to fit on a single screen can be viewed using scroll bars. Some menu items invoke dialog boxes, into which the user enters additional information. One dialog box is found in almost every Windows program. It opens a file. This dialog box looks the same (or very similar) in many different Windows programs, and it is almost always invoked from the same menu option.

Once you know how to use one Windows program, you're in a good position to easily learn another. The menus and dialog boxes allow user to experiment with a new program and explore its features. Most Windows programs have both a keyboard interface and a mouse interface. Although most functions of Windows programs can be controlled through the keyboard, using the mouse is often easier for many chores.

From the programmer's perspective, the consistent user interface results from using the Windows API functions for constructing menus and dialog boxes. All menus have the same keyboard and mouse interfaces because Windows—rather than the application program—handles this job.

Hardware Independent Programming

As we saw earlier a Windows program can always call Windows API functions. Thus an application can easily communicate with OS. What is new in Windows is that the OS can also communicate with application. Let us understand why it does so with the help of an example.

Suppose we have written a program that contains a menu item, which on selection is supposed to display a string "Hello World" in the window. The menu item can be selected either using the keyboard or using the mouse. On executing this program it will perform the initializations and then wait for the user input. Sooner or later the user would press the key or click the mouse to select the menu-item. This key-press or mouse-click is known as an 'event'. The occurrence of this event is sensed by the keyboard or mouse device driver. The device driver would now inform Windows about it. Windows would in turn notify the application about the occurrence of this event. This notification is known as a 'message'. Thus the OS has communicated with the application. When the application receives the message it communicates back with the OS by calling a Windows API function to display the string "Hello World" in the window. This API function in turn communicates with the device driver of the graphics card (that drives the screen) to display the string. Thus there is a two-way communication between the OS and the application. This is shown in Figure 16.4.

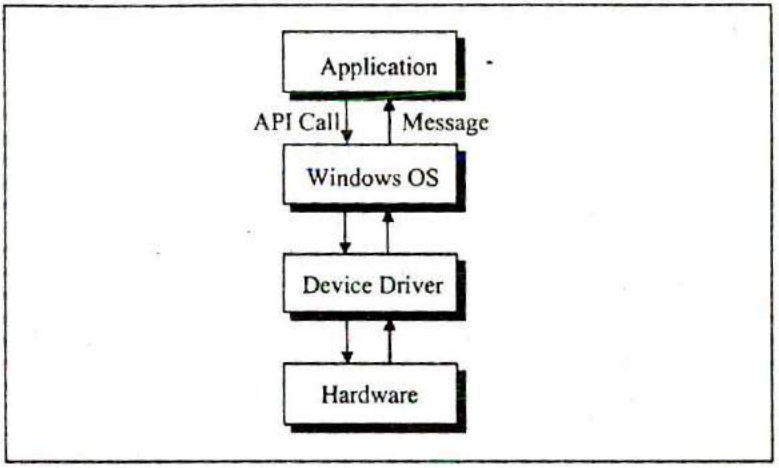

Figure 16.4

Suppose the keyboard and the mouse are now replaced with a new keyboard and mouse. Doing so would not affect the application at all. This is because at no time does the application carry out any direct communication with the devices. Any differences that may be there in the new set of mouse and keyboard would be handled the device driver and not by the application program. Similarly, if the screen or the graphics card is replaced no change would be required in the program. In short hardware independence at work! At times a change of device may necessitate a change in the device driver program, but never a change in the application.

Event Driven Model

When a user interacts with a Windows program a lot of events occur. For each event a message is sent to the program and the program reacts to it. Since the order in which the user would interact with the user-interface elements of the program cannot be predicted the order of occurrence of events, and hence the order of messages, also becomes unpredictable. As a result, the order of

calling the functions in the program (that react to different messages) is dictated by the order of occurrence of events. Hence this programming model is called 'Event Driven Programming Model'.

That's really all that is there to event-driven programming. Your job is to anticipate what users are likely to do with your application's user interface objects and have a function waiting, ready to execute at the appropriate time. Just when that time is, no one except the user can really say.

Windows Programming, a Closer Look

There can be hundreds of ways in which the user may interact with an application. In addition to this some events may occur without any user interaction. For example, events occur when we create a window, when the window's contents are to be drawn, etc. Not only this, occurrence of one event may trigger a few more events. Thus literally hundreds of messages may be sent to an application thereby creating a chaos. Naturally, a question comes—in which order would these messages get processed by the application. Order is brought to this chaos by putting all the messages that reach the application into a 'Queue'. The messages in the queue are processed in First In First Out (FIFO) order.

In fact the OS maintains several such queues. There is one queue, which is common for all applications. This queue is known as 'System Message Queue'. In addition there is one queue per application. Such queues are called 'Application Message Queues'. Let us understand the need for maintaining so many queues.

When we click a mouse and an event occurs the device driver posts a message into the System Message Queue. The OS retrieves this message finds out with regard to which application the message has been sent. Next it posts a message into the

Application Message Queue of the application in which the mouse was clicked. Refer Figure 16.5.

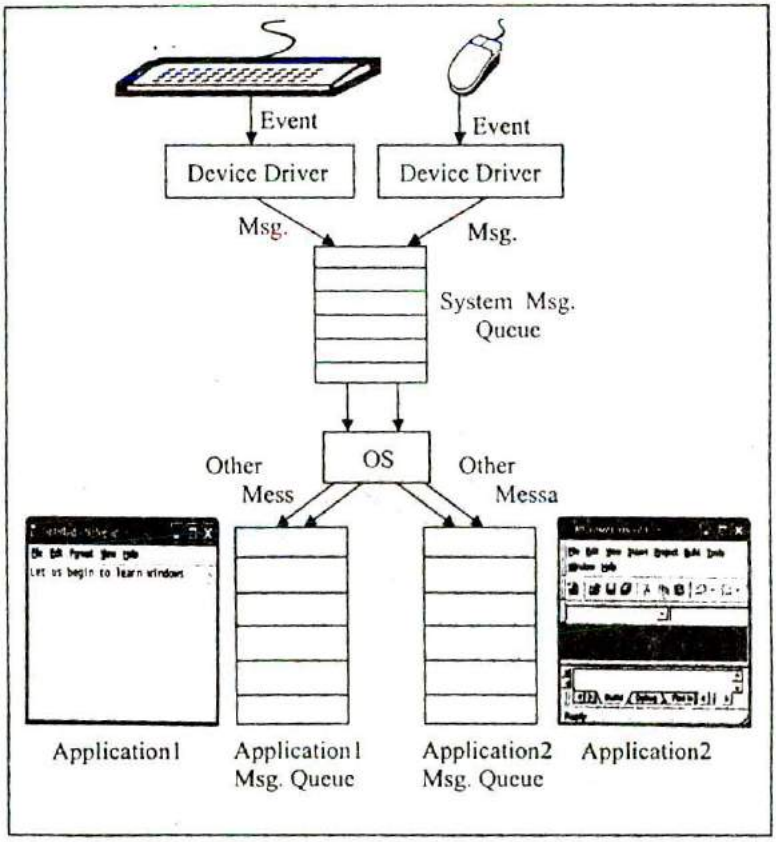

Figure 16.5

I think now we have covered enough ground to be able to actually start C under Windows programming. Here we go...

The First Windows Program

To keep things simple we would begin with a program that merely displays a "Hello" message in a message box. Here is the program...

```
#include <windows.h>
int _stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdline, int nCmdShow )
{
    MessageBox ( 0, "Hello!", "Title", 0 ) ;
    return ( 0 ) ;
}
```

Naturally a question would come to your mind—how do I create and run this program and what output does it produce. Firstly take a look at the output that it produces. Here it is...

Figure 16.6

Let us now look at the steps that one needs to carry to create and execute this program:

(a) Start VC++ from 'Start | Programs | Microsoft Visual C++ 6.0'. The VC++ IDE window will get displayed.
(b) From the File | New menu, select 'Win32 Application', and give a project name, say, 'sample1'. Click on OK.
(c) From the File | New menu, select 'C++ Source File', and give a suitable file name, say, 'sample1'. Click on OK.
(d) The 'Win32 Application-Step 1 of 1' window will appear. Select 'An empty project' option and click 'Finish' button.

(e) A 'New Project Information' dialog will appear. Close it by clicking on OK.
(f) Again select 'File | New | C++ Source File'. Give the file name as 'sample1.c'. Click on OK.
(g) Type the program in the 'sample1.c' file that gets opened in the VC++ IDE.
(h) Save this file using 'Save' option from the File menu.

To execute the program follow the steps mentioned below:

(a) From the Build menu, select 'Build sample1.exe'.
(b) Assuming that no errors were reported in the program, select 'Execute sample1.exe' from the Build menu.

Let us now try to understand the program. The way every C under DOS program begins its execution with **main( )**, every C under Windows program begins its execution with **WinMain( )**. Thus **WinMain( )** becomes the entry point for a Windows program. A typical **WinMain( )** looks like this:

```
int __stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow )
```

Note the **__stdcall** before **WinMain( )**. It indicates the calling convention used by the **WinMain( )** function. Calling Conventions indicate two things:

(a) The order (left to right or right to left) in which the arguments are pushed onto the stack when a function call is made.

(b) Whether the caller function or called function removes the arguments from the stack at the end of the call.

Out of the different calling conventions available most commonly used conventions are **__cdecl** and **__stdcall** . Both these calling conventions pass arguments to functions from right to left. In **__cdecl** the stack is cleaned up by the calling function, whereas in case of **__stdcall** the stack is cleaned up by the called function. All

API functions use **__stdcall** calling convention. If not mentioned, **__cdecl** calling convention is assumed by the compiler.

HINSTANCE and LPSTR are nothing but **typedefs**. The first is an **unsigned int** and the second is a **pointer** to a **char**. These macros are defined in 'windows.h'. This header file must always be included while writing a C program under Windows. **hInstance**, **hPrevInstance**, **lpszCmdLine** and **nCmdShow** are simple variable names. In place of these we can use **i**, **j**, **k** and **l** respectively. Let us now understand the meaning of these parameters as well as the rest of the program.

- **WinMain( )** receives four parameters which are as under:

  hInstance: This is the 'instance handle' for the running application. Windows creates this ID number when the application starts. We will use this value in many Windows functions to identify an application's data.

  A handle is simply a 32-bit number that refers to an entity. The entity could be an application, a window, an icon, a brush, a cursor, a bitmap, a file, a device or any such entity. The actual value of the handle is unimportant to your programs, but the Windows module that gives your program the handle knows how to use it to refer to an entity. What is important is that there is a unique handle for each entity and we can refer and reach the entity only using its handle.

  hPrevInstance: This parameter is a remnant of earlier versions of Windows and is no longer significant. Now it always contains a value 0. It is being persisted with only to ensure backward compatibility.

  lpszCmdLine: This is a pointer to a character string containing the command line arguments passed to the program. This is similar to the **argv**, **argc** parameters passed to **main( )** in a DOS program.

nCmdShow: This is an integer value that is passed to the function. This integer tells the program whether the window that it creates should appear minimized, as an icon, normal, or maximized when it is displayed for the first time.

- The **MessageBox( )** function pops up a message box whose title is 'Title' and which contains a message 'Hello!'.
- Returning **0** from **WinMain( )** indicates success, whereas, returning a nonzero value indicates failure.
- Instead of printing 'Hello!' in the message box we can print the command line arguments that the user may supply while executing the program. The command line arguments can be supplied to the program by executing it from Start | Run as shown in Figure 16.7.

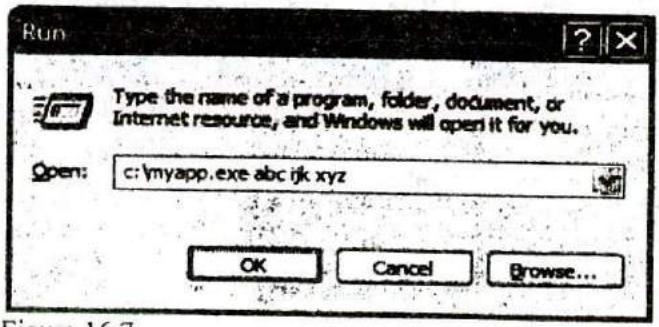

Figure 16.7

Note from Figure 16.7 that 'myapp.exe' is the name of our application, whereas, 'abc ijk xyz' represents command line arguments. The parameter **lpszCmdline** points to the string "abc ijk xyz". This string can be printed using the following statement:

```
MessageBox ( 0, lpszCmdline, "Title", 0 ) ;
```

If the entire command line including the filename is to be retrieved we can use the **GetCommandLine( )** function.

Hungarian Notation

Hungarian Notation is a variable-naming convention so called in the honor of the legendary Microsoft programmer Charles Simonyi. According to this convention the variable name begins with a lower case letter or letters that denotes the data type of the variable. For example, the **sz** prefix in **szCmdLine** stands for 'string terminated by zero'; the prefix **h** in **hInstance** stands for 'handle'; the prefix **n** in **nCmdShow** stands for **int**. Prefixes are often combined to form other prefixes, as **lpsz** in **lpszCmdLine** stands for 'long pointer to a zero terminated string'. Though basically this notation is a good idea nowadays its usage is discouraged. This is because when a transition happens from say a 16-bit code to 32-bit code then a whole lot of variable names have to be changed. For example, suppose the 16-bit code used 2-byte and 4-byte integer variables called **wParam** and **lParam**, where **w** indicated a 16-bit integer (word) and a 32-bit integer (long) respectively. When this code is ported to a 32-bit environment **wParam** had to be changed to **lParam** since in this environment every integer is 4 bytes long. You would agree that if we follow the Hungarian notation then we would have to make a whole lot of changes in the variable names when we port the code to a 32-bit or a 64-bit environment. Hence the usage of this convention is nowadays discouraged.

Summary

(a) Under Windows an integer is four bytes long. To use a two-byte integer pre-qualify it with **short**.
(b) Under Windows a pointer is four bytes long.
(c) Windows programming involves a heavy usage of **typedefs**.
(d) DOS uses a Sequential Programming Model, whereas, Windows uses an Event Driven Programming Model.
(e) Entry point of every Windows program is a function called **WinMain( )**.

(f) Windows does not permit direct access to memory or hardware devices.
(g) Windows uses a Demand-based Virtual Memory Model to manage memory.
(h) Under Windows there is two-way communication between the program and the OS.
(i) Windows maintains a system message queue common for all applications.
(j) Windows maintains an application message queue per running application.
(k) Calling convention decides the order in which the parameters are passed to a function and whether the calling function o. the called function clears the stack.
(l) Commonly used calling conventions are **__cdecl** and **__stdcall**.
(m) Hungarian notation though good its usage is not recommended any more.

Exercise

[A] State True or False:

(a) MS-DOS uses a procedural programming model.
(b) A Windows program can directly call a device driver program for a device.
(c) API functions under Windows do not have names.
(d) DOS functions are called using an interrupt mechanism.
(e) Windows uses a 4 GB virtual memory space.
(f) Size of a pointer under Windows depends upon whether it is **near** or **far**.
(g) Under Windows the address stored in a pointer is a virtual address and not a physical address.
(h) One of the parameters of **WinMain( )** called **hPrevInstance** is no longer relevant.

[B] Answer the following:

(a) Why is Event-driven Programming Model better than the Sequential Programming Model?
(b) What is the meaning of different parts of the address stored in a pointer under Windows environment?
(c) Why Windows does not permit direct access to hardware?
(d) What is the difference between an event and a message?
(e) Why Windows maintains a different message queue for each application?
(f) In which different situations messages get posted into an application message queue?

[C] Attempt the following:

(a) Write a program that prints the value of **hInstance** in a message box.
(b) Write a program that displays three buttons 'Yes', 'No' 'Cancel' in the message box.
(c) Write a program that receives a number as a command line argument and prints its factorial value in a message box.
(d) Write a program that displays command line arguments including file name in a message box.

17 Windows Programming

- The Role of a Message Box
- Here comes the window...
- More Windows
- A Real-World Window
  - Creation and Displaying of Window
  - Interaction with Window
  - Reacting to Messages
- Program Instances
- Summary
- Exercise

Event driven programming requires a change in mind set. I hope Chapter 16 has been able to bring about this change. However this change would be bolstered by writing event driven programs. This is what this chapter intends to do. I am hopeful that by the time you reach the end of this chapter you would be so comfortable with it as if you have been using it all your life.

The Role of a Message Box

Often we are required to display certain results on the screen during the course of execution of a program. We do this to ascertain whether we are getting the results as per our expectations. In a sequential DOS based program we can easily achieve this using **printf( )** statements. Under Windows screen is a shared resource. So you can imagine what chaos would it create if all running applications are permitted to write to the screen. You would not be able to make out which output is of what application. Hence no Windows program is permitted to write anything directly to the screen. That's where a message box enters the scene. Using it we can display intermediate results during the course of execution of a program. It can be dismissed either by clicking the 'close button' in its title bar or by clicking the OK button present in it. There are numerous variations that you can try with the **MessageBox( )**. Some of these are given below

```
MessageBox ( 0, "Are you sure", "Caption", MB_YESNO ) ;
MessageBox ( 0, "Print to the Printer", "Caption", MB_YESNO CANCEL) ;
MessageBox ( 0, "icon is all about style", "Caption", MB_OK |
                MB_ICONINFORMATION ) ;
```

You can put the above statements within **WinMain( )** and see the results for yourself. Though the above message boxes give you flexibility in displaying results, button, icons, there is a limit to which you can stretch them. What if we want to draw a free hand drawing or display an image, etc. in the message box. This would

not be possible. To achieve this we need to create a full-fledged window. The next section discusses how this can be done.

Here Comes the window...

Before we proceed with the actual creation of a window it would be a good idea to identify the various elements of it. These are shown in Figure 17.1.

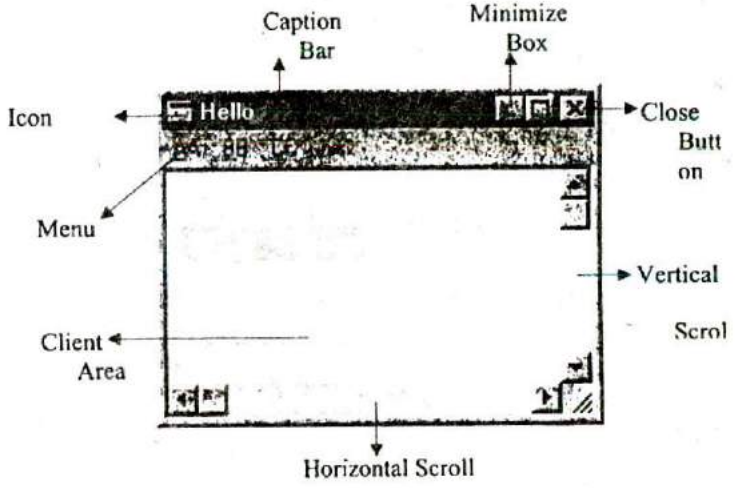

Figure 17.1

Note that every window drawn on the screen need not necessarily have every element shown in the above figure. For example, a window may not contain the minimize box, the maximize box, the scroll bars and the menu.

Let us now create a simple program that creates a window on the screen. Here is the program...

```
#include <windows.h>
```

```
int _stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdLine, int nCmdShow )
{
    HWND h ;

    h = CreateWindow ( "BUTTON", "Hit Me", WS_OVERLAPPEDWINDOW,
                       10, 10, 150, 100, 0, 0, i, 0 ) ;
    ShowWindow ( h, nCmdShow ) ;
    MessageBox ( 0, "Hi!", "Waiting", MB_OK ) ;
    return 0 ;
}
```

Here is the output of the program...

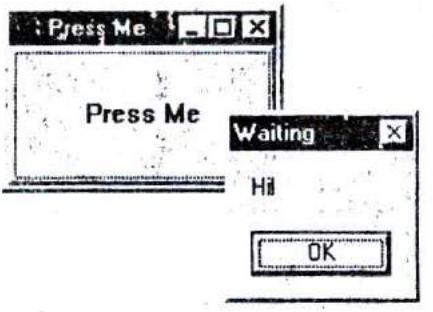

Figure 17.2

Let us now understand the program. Every window enjoys certain properties—background color, shape of cursor, shape of icon, etc. All these properties taken together are known as 'window class'. The meaning of 'class' here is 'type'. Windows insists that a window class should be registered with it before we attempt to create windows of that type. Once a window class is registered we can create several windows of that type. Each of these windows would enjoy the same properties that have been registered through the window class. There are several predefined window classes. Some of these are BUTTON, EDIT, LISTBOX, etc. Our program has created one such window using the predefined BUTTON class.

To actually create a window we need to call the API function **CreateWindow( )**. This function requires several parameters starting with the window class. The second parameter indicates the text that is going to appear on the button surface. The third parameter specifies the window style. WS_OVERLAPPEDWINDOW is a commonly used style. The next four parameters specify the window's initial position and size—the x and y screen coordinates of the window's top left corner and the window's width and height in pixels. The next three parameters specify the handles to the parent window, the menu and the application instance respectively. The last parameter is the pointer to the window-creation data.

We can easily devote a section of this book to **CreateWindow( )** and its parameters. But don't get scared of it. Nobody is supposed to remember all the parameters, their meaning and their order. You can always use MSDN (Microsoft Developer Network) help to understand the minute details of each parameter. This help is available as part of VC++ 6.0 product. It is also available on the net at http://www.msdn.microsoft.com/library.

Note that **CreateWindow( )** merely creates the window in memory. We still are to display it on the screen. This can be done using the **ShowWindow( )** API function. **CreateWindow( )** returns handle of the created window. Our program uses this handle to refer to the window while calling **ShowWindow( )**. The second parameter passed to **ShowWindow( )** signifies whether the window would appear minimized, maximized or normal. If the value of this parameter is **SW_SHOWNORMAL** we get a normal sized window, if it is **SW_SHOWMINIMIZED** we get a minimized window and if it is **SW_SHOWMINIMIZED** we get a maximized window. We have passed **nCmdShow** as the second parameter. This variable contains **SW_SHOWNORMAL** by default. Hence our program displays a normal sized window.

The WS_OVERLAPPEDWINDOW style is a collection of the following styles:

WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX

As you can make out from these macros they essentially control the look and feel of the window being created. All these macros are **#defined** in the 'Windows.h' header file.

On executing this program a window and a message box appears on the screen as shown in the Figure 17.2. The window and the message box disappear as soon as we click on OK. This is because on doing so execution of **WinMain( )** comes to an end and moreover we have made no provision to interact with the window.

You can try to remove the call to **MessageBox( )** and see the result. You would observe that no sooner does the window appear it disappears. Thus a call to **MessageBox( )** serves the similar purpose as **getch( )** does in sequential programming.

More Windows

Now that we know how to create a window let us create several windows on the screen. The program to do this is given below.

```
#include <windows.h>

int _stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdLine, int nCmdShow )
{
    HWND h[10] ;
    int x ;

    for ( x = 0 ; x <= 9 ; x++ )
    {
```

```
        h[x] = CreateWindow ( "BUTTON", "Press Me",
                              WS_OVERLAPPEDWINDOW, x * 20,
                              x * 20, 150, 100, 0, 0, i, 0 ) ;
        ShowWindow ( h[x], I ) ;
    }

    MessageBox ( 0, "Hi!", "Waiting", 0 ) ;
    return 0 ;
}
```

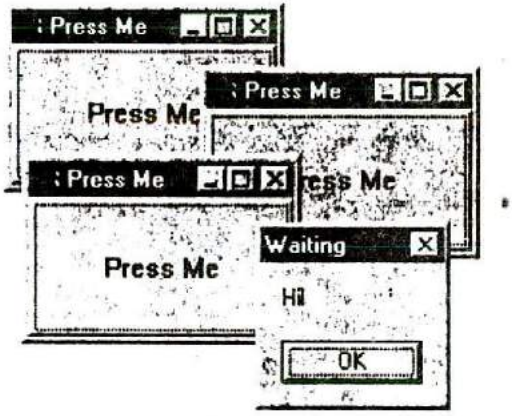

Figure 17.3

Note that each window created in this program is assigned a different handle. You may experiment a bit by changing the name of the window class to EDIT and see the result.

A Real-World Window

Suppose we wish to create a window and draw a few shapes in it. For creating such a window there is no standard window class available. Hence we would have to create our own window class, register it with Windows OS and then create a window on the basis of it. Instead of straightway jumping to a program that draws

shapes in a window let us first write a program that creates a window using our window class and lets us interact with it. Here is the program...

```
#include <windows.h>
#include "helper.h"

void OnDestroy ( HWND ) ;

int __stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdline, int nCmdShow )
{
    MSG m ;

    /* perform application initialization */
    InitInstance  ( hInstance, nCmdShow, "title" ) ;

    /* message loop */
    while ( GetMessage ( &m, 0, 0, 0 ) )
        DispatchMessage ( &m ) ;

    return 0 ;
}


LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_DESTROY :
            OnDestroy ( hWnd ) ;
            break ;
        default :
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
    }
    return 0 ;
}
```

```
void OnDestroy ( HWND hWnd )
{
    PostQuitMessage ( 0 ) ;
}
```

On execution of this program the window shown in Figure 17.4 appears on the screen. We can use minimize and the maximize button it its title bar to minimize and maximize the window. We can stretch its size by dragging its boundaries. Finally, we can close the window by clicking on the close window button in the title bar.

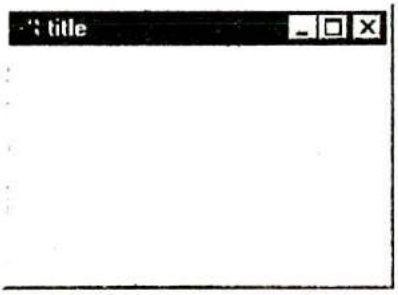

Figure 17.4

Let us now try to understand this program step by step.

Creation and Displaying of Window

Creating and displaying a window on the screen is a 4-step process. These steps are:

(a) Creation of a window class.
(b) Registering the window class with the OS.
(c) Creation of a window based on the registered class.
(d) Displaying the window on the screen.

Creation of a window class involves setting up of elements of a structure called **WNDCLASSEX**. This structure contains several

elements. They govern the properties of the window. Registration of a window class, creation of a window and displaying of a window involves calling of API functions **RegisterClassEx( )**, **CreateWindow( )** and **ShowWindow( )** respectively. Since all the 4 steps mentioned above would be required in almost every program in this chapter I have written this code in a user-defined function called **InitInstance( )** in the file 'helper.h'.

Though writing code in a header file goes against the convention I have still done so to achieve simplicity. The complete listing of 'helper.h' file is available in Appendix F. Alternatively you can download it from the following link:

www.kicit.com/books/letusc/sourcecode/helper.h

As expected **WinMain( )** starts off by calling the function **InitInstance( )** present in 'helper.h' file. This file has been **#included** at the beginning of the program. Remember to copy this file to your project directory—the directory in which you are going to create this program.

Once the window has been created and displayed let us see how we can interact with it.

Interaction with Window

As and when the user interacts with the window—by stretching its boundaries or clicking the buttons in the title bar, etc. a suitable message is posted into the message queue of our application. Our application should now pick them up from the message queue and process them.

A message contains a message id and some other additional information about the message. For example, a mouse click message would contain additional information like handle to the window with which the user has interacted, the coordinates of

mouse cursor and the status of mouse buttons. Since it is difficult to memorize the message ids they have been suitably **#defined** in 'windows.h'. The message id and the additional information are stored in a structure called MSG.

In **WinMain( )** this MSG structure is retrieved from the message queue by calling the API function **GetMessage( )**. The first parameter passed to this function is the address of the MSG structure variable. **GetMessage( )** would pick the message info from the message queue and place it in the structure variable passed to it. Don't bother about the other parameters right now.

After picking up the message from the message queue we need to process it. This is done by calling the **DispatchMessage( )** API function. This function does several activities. These are as follows:

(a) From the **MSG** structure that we pass to it, **DisplayMessage( )** extracts the handle of the window for which this message is meant for.
(b) From the handle it figures out the window class based on which the window has been created.
(c) From the window class structure it obtains the address of a function called **WndProc( )** (short for window procedure). Well I didn't tell you earlier that in **InitInstance( )** while filling the **WNDCLASSEX** structure one of the elements has been set up with the address of a user-defined function called **WndProc( )**.
(d) Using this address it calls the function **WndProc( )**.

Since several messages get posted into the message queue picking of the message and processing it should be done repeatedly. Hence calls to **GetMesage( )** and **DispatchMessage( )** have been made in a **while** loop in **WinMain( )**. When **GetMessage( )** encounters a message with id **WM_QUIT** it returns a **0**. Now the control comes out of the loop and **WinMain( )** comes to an end.

Reacting to Messages

As we saw in the previous section, for every message picked up from the message queue the control is transferred to the **WndProc( )** function. This function is shown below:

```
LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
```

This function always receives four parameters. The first parameter is the handle to the window for which the message has been received. The second parameter is the message id, whereas, the third and fourth parameters contain additional information about the message.

LRESULT is a **typedef** of a **long int** and represents the return value of this function. **CALLBACK** is a **typedef** of **__stdcall**. This **typedef** has been done in 'windows.h'. **CALLBACK** indicates that the **WndProc** function has been registered with Windows (through **WNDCLASSEX** structure in **InitInstance( )**) with an intention that Windows would call this back (through **DispatchMessage( )** function).

In the **WndProc( )** function we have checked the message id using a **switch**. If the id is **WM_DESTROY** then we have called the function **OnDestroy( )**. This message is posted to the message queue when the user clicks on the 'Close Window' button in the title bar. In **OnDestroy( )** function we have called the API function **PostQuitMessage( )**. This function posts a **WM_QUIT** message into the message queue. As we saw earlier, when this message is picked up the message loop and **WinMain( )** is terminated.

For all messages other than **WM_DESTROY** the control lands in the **default** clause of **switch**. Here we have simply made a call to **DefWindowProc( )** API function. This function does the default

processing of the message that we have decided not to tackle. The default processing for different message would be different. For example on double clicking the title bar **DefWindowProc( )** maximizes the window.

Actually speaking when we close the window a **WM_CLOSE** message is posted into the message queue. Since we have not handled this message the **DefWindowProc( )** function gets called to tackle this message. The **DefWindowProc( )** function destroys the window and places a **WM_DESTROY** message in the message queue. As discussed earlier, in **WndProc( )** we have made the provision to terminate the application on encountering **WM_DESTROY**.

That brings us to the end of a lonnngggg explanation! You can now heave a sigh of relief. I would urge you to go through the above explanation till the time you are absolutely sure that you have understood every detail of it. A very clear understanding of it would help you make a good Windows programmer. For your convenience I have given a flowchart of the entire working in Figure 17.5.

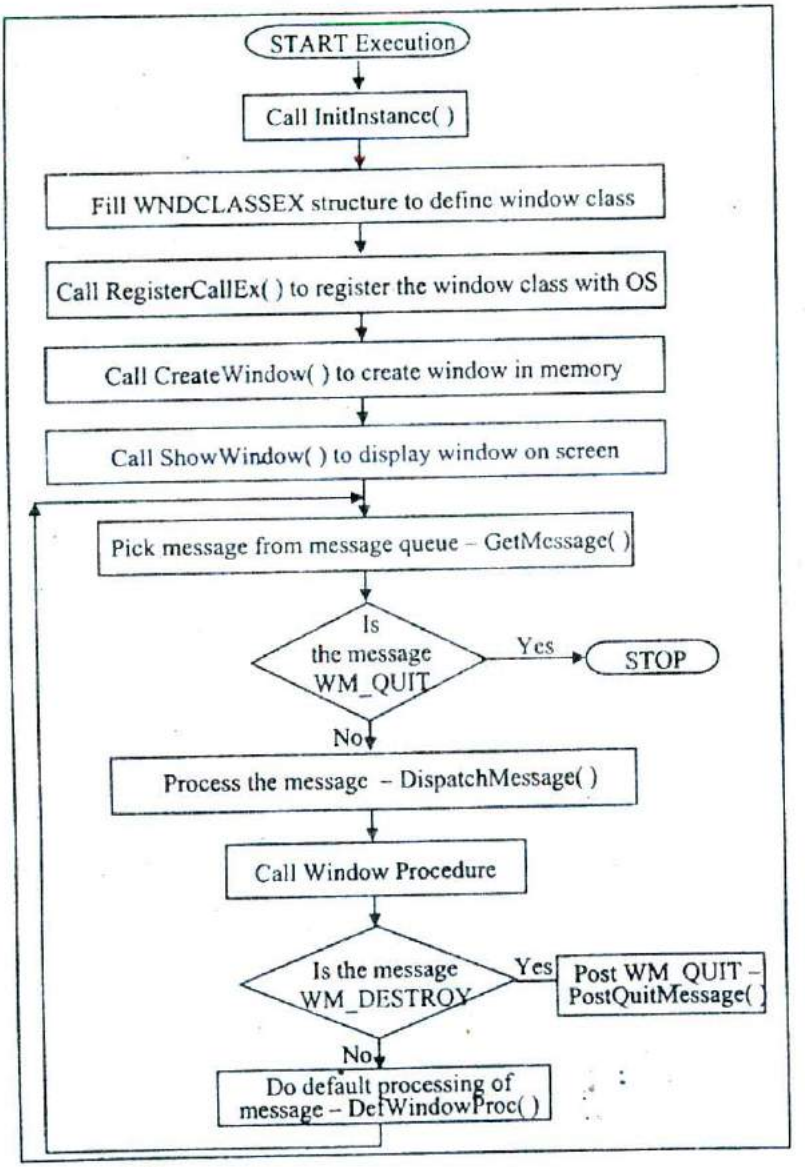
START Execution
Call InitInstance( )
Fill WNDCLASSEX structure to define window class
Call RegisterCallEx( ) to register the window class with OS
Call CreateWindow( ) to create window in memory
Call ShowWindow( ) to display window on screen
Pick message from message queue – GetMessage( )
Is the message WM_QUIT
Yes
STOP
No
Process the message – DispatchMessage( )
Call Window Procedure
Is the message WM_DESTROY
Yes
Post WM_QUIT – PostQuitMessage( )
No
Do default processing of message – DefWindowProc( )

Figure 17.5

Program Instances

Windows allows you to run more than one copy of a program at a time. This is handy for cutting and pasting between two copies of Notepad or when running more than one terminal session with a terminal emulator program. Each running copy of a program is called a 'program instance'.

Windows performs an interesting memory optimization trick. It shares a single copy of the program's code between all running instances. For example, if you get three instances of Notepad running, there will only be one copy of Notepad's code in memory. All three instances share the same code, but will have separate memory areas to hold the text data being edited. The difference between handling of the code and the data is logical, as each instance of Notepad might edit a different file, so the data must be unique to each instance. The program logic to edit the files is the same for every instance, so there is no reason why a single copy of Notepad's code cannot be shared.

Summary

(a) A message box can be displayed by calling the **MessageBox( )** API function.
(b) Message boxes are often used to ascertain the flow of a program.
(c) Appearance of a message box can be customized.
(d) The **CreateWindow( )** API function creates the window in memory.
(e) The window that is created in memory is displayed using the **ShowWindow( )** API function.
(f) A 'window class' specifies various properties of the window that we are creating.
(g) The header file 'Windows.h' contains declaration of several macros used in Windows programming.

(h) When the user clicks in a window, or moves mouse pointer on the window, messages are generated and posted in the application message queue.
(i) A message contains the message id and additional information about the message.
(j) The **GetMessage( )-DispatchMessage( )** loop breaks when **GetMessage( )** encounters the **WM_QUIT** message.
(k) If we don't handle a message received by our application then the **DefWindowProc( )** function is called to do the default processing.

Exercise

[A] State True or False:

(a) **MessageBox( )** is an API function.
(b) Calling the **MessageBox( )** function displays the specified string in console window.
(c) The **CreateWindow( )** function creates and displays the window on the screen.
(d) The **ShowWindow( )** function can display only the maximized window.
(e) Every window has to be created using pre-registered window class.
(f) Window classes are similar to classes in C++.
(g) We can use the pre-defined window classes but cannot create our own.
(h) The style WS_OVERLAPPED | WS_CAPTION | WS_MINIMIZEBOX will create a window with caption bar and minimize box only.
(i) To be able to interact with a window it is necessary to implement the message loop.

[B] Answer the following:

(a) Outline the steps that a typical Windows program follows during execution.

(b) Run any Windows based program and see whether you can identify all the elements of the application window.

(c) How would you minimize a window programmatically?

(d) What would happen if we do not place WM_QUIT message in the message queue when the user tries to close the window.

(e) Explain the need of **RegisterClassEx( )** function.

(f) What is the difference between **GetMessage( )** and **DispatchMessage( )** function?

(g) Write a program, which receives an integer as a command line argument, creates a button window, and based on the value of the integer displays button window as maximized / minimized / normal.

(h) Try to display a window with different combinations of window styles and observer the results.

18 Graphics Under Windows

- Graphics as of Now
- Device Independent Drawing
- Hello Windows
- Drawing Shapes
- Types of Pens
- Types of Brushes
  Code and Resources
- Freehand Drawing, the Paintbrush Style
- Capturing the mouse
- Device Context, A Closer Look
- Displaying a Bitmap
- Animation at Work
  WM_CREATE and *OnCreate( )*
  WM_TIMER and *OnTimer( )*
  A Few More Points...
- Windows, the Endless World...
- Summary
- Exercise

Since times immemorial colors and shapes have fascinated mankind like nothing else. Otherwise people would have still been using the character oriented interfaces of MS-DOS or Unix. In fact the graphical ability of Windows has played a very important role in its success story. Once you get a hang of how to draw inside a window it would open up immense possibilities that you never thought were possible.

Graphics as of Now

World has progressed much beyond 16 colors and 640 x 480 resolution graphics that Turbo C/C++ compilers offered under MS-DOS environment. Today we are living in a world of 1024 x 768 resolution offering 16.7 million colors. Graphical menus, icons, colored cursors, bitmaps, wave files and animations are the order of the day. So much so that a 16-color graphics program built using Turbo C working on a poor resolution almost hurts the eye. Moreover, with the whole lot of Windows API functions to support graphics activity there is so much that can be achieved in a graphics program under Windows. I am sure that this chapter will help you understand and appreciate these new capabilities.

Device Independent Drawing

Windows allow programmers to write programs to display text or graphics on the screen without concern over the specifics of the video hardware. A Windows program that works on a VGA display will work without modification on an SVGA or on a XGA display that Windows supports.

The key to this 'device independence' is Windows' use of a 'device context'. We will explore how the device context can be used for both text and graphics output, and how using the device context keeps our programs from interfering with each other on the screen.

During the original design of Windows, one of the goals was to provide 'device independence'. Device independence means that the same program should be able to work using different screens, keyboards and printers without modification to the program. Windows takes care of the hardware, allowing the programmer to concentrate on the program itself. If you have ever had to update the code of an MS-DOS program for the latest printer, plotter, video display, or keyboard, you will recognize device independence as a huge advantage for the developer.

Windows programs do not send data directly to the screen or printer. A Windows program knows where (screen/printer) its output is being sent. However, it does not know how it would be sent there, neither does it need to bother to know this. This is because Windows uses a standard and consistent way to send the output to screen/printer. This standard way uses an entity called Device Context, or simply a DC. Different DC's are associated with different devices. For example, a screen DC is associated with a screen, a printer DC is associated with a printer, etc. Any drawing that we do using the screen DC is directed to the screen. Similarly, any drawing done using the printer DC is directed to the printer. Thus, the only thing that changes from drawing to screen and drawing to printer is the DC that is used.

A windows program obtains a handle (ID value) for the screen or printer's DC. The output data is sent to the screen/printer using its DC, and then Windows and the Device Driver for the device takes care of sending it to the real hardware. The advantage of using the DC is that the graphics and text commands that we send using the DC are always the same, regardless of where the physical output is showing up.

The part of Windows that converts the Windows graphics function calls to the actual commands sent to the hardware is the GDI, or Graphics Device Interface. The GDI is a program file called GDI32.DLL and is stored in the Windows System directory. The

Windows environment loads GDI32.DLL into memory when it is needed for graphical output. Windows also loads a 'device driver' program if the hardware conversions are not part of GDI32.DLL. Common examples are VGA.SYS for VGA video screen and HPPLC.SYS for the HP LaserJet printer. Drivers are just programs that assist the GDI in converting Windows graphics commands to hardware commands.

Thus GDI provides all the basic drawing functionality for Windows; the device context represents the device providing a layer of abstraction that insulates your applications from the trouble of drawing directly to the hardware. The GDI provides this insulation by calling the appropriate device driver in response to windows graphics function calls.

Hello Windows

We would begin our tryst with graphics programming under windows by displaying a message "Hello Windows" in different fonts. Note that though we are displaying text under Windows even text gets drawn graphically in the window. First take a look at the program given below before we set out to understand it.

```
# include <windows.h>
# include "helper.h"

void OnPaint ( HWND ) ;
void OnDestroy ( HWND ) ;

int __stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpszCmdline, int nCmdShow )
{
    MSG m ;

    /* Perform application initialization */
    InitInstance ( hInstance, nCmdShow, "Text" ) ;
```

```
    /* Main message loop */
    while ( GetMessage ( &m, NULL, 0, 0 ) )
        DispatchMessage(&m);

    return 0 ;
}

LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_DESTROY :
            OnDestroy ( hWnd ) ;
            break ;
        case WM_PAINT :
            OnPaint ( hWnd ) ;
            break ;
        default :
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
  }
  return 0 ;
}

void OnDestroy ( HWND hWnd )
{
    PostQuitMessage ( 0 ) ;
}

void OnPaint ( HWND hWnd )
{
    HDC hdc ;
    PAINTSTRUCT ps ;
    HFONT hfont ;
    LOGFONT f = { 0 } ;
    HGDIOBJ holdfont ;
    char *fonts[ ] = { "Arial", "Times New Roman", "Comic Sans MS" } ;
    int i ;
```

```
    hdc = BeginPaint ( hWnd, &ps ) ;

    for ( i = 0 ; i < 3 ; i++ )
    {
        strcpy ( f.lfFaceName, fonts[ i ] ) ;  /* copy font name */
        f.lfHeight = 40 * ( i + 1 ) ;  /* font height */
        f.lfItalic = 1 ;  /* italic */

        hfont = CreateFontIndirect ( &f ) ;
        holdfont = SelectObject ( hdc, hfont ) ;

        SetTextColor ( hdc, RGB ( 0, 0, 255 ) ) ;

        TextOut ( hdc, 10, 70 * i, "Hello Windows", 13 ) ;

        SelectObject ( hdc, holdfont ) ;
        DeleteObject ( hfont ) ;
    }

    EndPaint ( hWnd, &ps ) ;
}
```

On execution of this program the window shown in Figure 18.1 appears.

Figure 18.1

Drawing to a window involves handling the **WM_PAINT** message. This message is generated whenever the client area of the window needs to be redrawn. This redrawing would be required in the following situations:

(a) When the Window is displayed for the first time.
(b) When the window is minimized and then maximized.
(c) When some portion of the window is overlapped by another window and the overlapped window is dismissed.
(d) When the size of the window changes on stretching its boundaries.
(e) When the window is dragged out of the screen and then brought back into the screen.

Would a **WM_PAINT** message be generated when the cursor is dragged in the window? No. In this case the window saves the area overlapped by the cursor and restores it when the cursor moves to another position.

When the **switch-case** structure inside **WndProc( )** finds that the message ID passed to **WndProc( )** is **WM_PAINT**, it calls the function **OnPaint( )**. Within **OnPaint( )** we have called the API function **BeginPaint( )**. This function obtains a handle to the device context. Additionally it also fills the PAINTSTRUCT structure with information about the area of the window which needs to be repainted. Lastly it removes **WM_PAINT** from the message queue. After obtaining the device context handle, the control enters a loop.

Inside the loop we have displayed "Hello Windows" in three different fonts. Each time through the loop we have setup a **LOGFONT** structure **f**. This structure is used to indicate the font properties like font name, font height, italic or normal, etc. Note that in addition to these there are other font properties that may be setup. The properties that we have not setup in the loop are all initialized to **0**. Once the font properties have been setup we have called the **CreateFontIndirect( )** API function to create the font.

This function loads the relevant font file. Then using the information in the font file and the font properties setup in the **LOGFONT** structure it creates a font in memory. **CreateFontIndirect( )** returns the handle to the font created in memory. This handle is then passed to the **SelectObject( )** API function to get the font into the DC. This function returns the handle to the existing font in the DC, which is preserved in **holdfont** variable. Next we have used the **SetTextColor( )** API function to set the color of the text to be displayed through **TextOut( )**. The **RGB( )** macro uses the red, green and blue component values to generate a 32-bit color value. Note that each color component can take a value from **0** to **255**. To **TextOut( )** we have to pass the handle to the DC, position where the text is to be displayed, the text to be displayed and its length.

With **hfont** only one font can be associated at a time. Hence before associating another font with it we have deleted the existing font using the **DeleteObject( )** API function. Once outside the loop we have called the **EndPaint( )** API function to release the DC handle. If not released we would be wasting precious memory, because the device context structure would remain in memory but we would not be able access it.

In place of **TextOut( )** we can also use the **DrawText( )** API function. This function permits finer control over the way the text is displayed. You can explore this function on your own.

Drawing Shapes

If text is so near can graphics be far behind? Now that we know how to draw text in a window let us now create a simple program that displays different shapes in a window. Instead of showing the entire program given below is the listing of **OnPaint( )**. The rest of the program is same as in the previous section. Here onwards I would be showing only the **OnPaint( )** handler unless otherwise required.

```
void OnPaint ( HWND hWnd )
{
    HDC hdc ;
    PAINTSTRUCT ps ;
    HBRUSH hbr ;
    HGDIOBJ holdbr ;
    POINT pt[5] = { 250, 150, 250, 300, 300, 350, 400, 300, 320, 190 } ;

    hdc = BeginPain  hWnd, &ps ) ;

    hbr = CreateSolidBrush ( RGB ( 255, 0, 0 ) ) ;
    holdbr = SelectObject ( hdc, hbr ) ;

    MoveToEx ( hdc, 10, 10, NULL ) ;
    LineTo ( hdc, 200, 10 ) ;

    Rectangle ( hdc, 10, 20, 200, 100 ) ;

    RoundRect ( hdc, 10, 120, 200, 220, 20, 20 ) ;

    Ellipse ( hdc, 10, 240, 200, 340 ) ;

    Pie ( hdc, 250, 10, 350, 110, 350, 110, 350, 10 ) ;

    Polygon ( hdc, pt, 5 ) ;

    SelectObject ( hdc, holdbr ) ;
    DeleteObject ( hbr ) ;

    EndPaint ( hWnd, &ps ) ;
}
```

On execution of this program the window shown in Figure 18.2 appears.

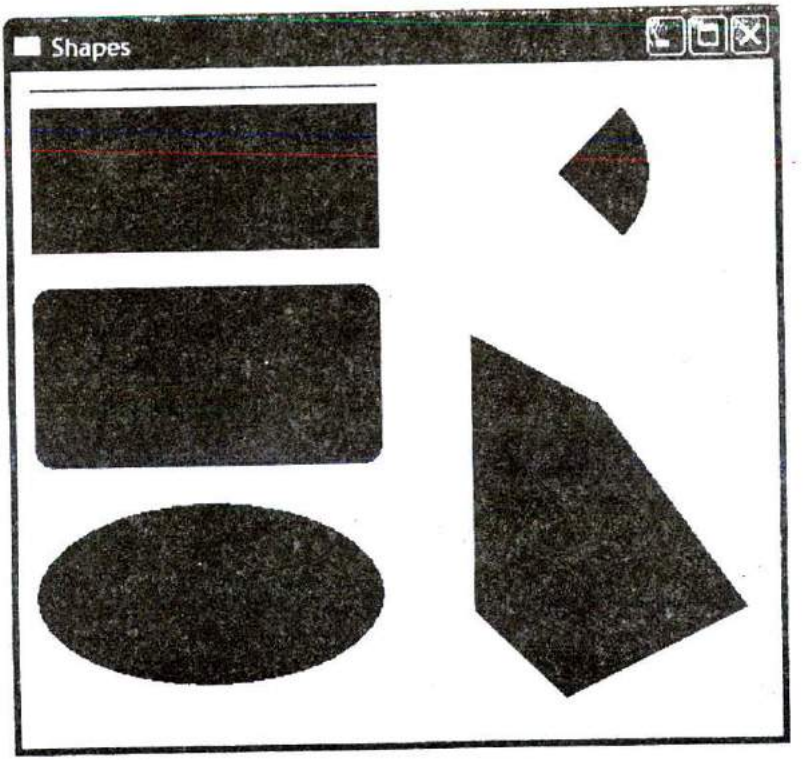

Figure 18.2

For drawing any shape we need a pen to draw its boundary and a brush to paint the area enclosed by it. The DC contains a default pen and brush. The default pen is a solid pen of black color and the default brush is white in color. In this program we have used the default pen and a blue colored solid brush for drawing the shapes.

As before, we begin by obtaining a handle to the DC using **BeginPaint( )** function. For creating a solid colored brush we need to call the **CreateSolidBrush( )** API function. The second parameter of this function specifies the color of the brush.. The function returns the handle of the brush which we have preserved

in the **hbr** variable. Next we have selected this brush in the DC. The handle of the default brush in DC is collected in the **holdbr** variable.

Once we have selected the brush into the DC we are ready to draw the shapes. For drawing the line we have used **MoveToEx( )** and **LineTo( )** API functions. Similarly for drawing a rectangle we have used the **Rectangle( )** function.

The **RoundRect( )** function draws a rectangle with rounded corners. In **RoundRect (x1, y1, x2, y2, x3, y3), x1, y1** represents the *x* and *y*-coordinates of the upper-left corner of the rectangle. Likewise, **x2, y2** represent coordinates of the bottom right corner of the rectangle. **x3, y3** specify the width and height of the ellipse used to draw the rounded corners.

Note that rectangle and the rounded rectangle are drawn from **x1, y1** up to **x2-1, y2-1**.

Parameters of **Ellipse( )** specify coordinates of bounding rectangle of the ellipse.

The **Pie( )** function draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by **x1, y1** and **x2, y2**. In **Pie(x1, y1, x2, y2, x3, y3, x4, y4), x1, y1** and **x2, y2** specify the *x* and *y*-coordinates of the upper left corner and bottom right corner respectively, of the bounding rectangle. **x3, y3** and **x4, y4** specify the *x* and *y*-coordinates of the arc's starting point and ending point respectively.

In **Polygon (lpPoints, nCount), lpPoints** points to an array of points that specifies the vertices of the polygon. Each point in th array is a **POINT** structure. **nCount** specifies the number vertices stored in the array. The system closes the polyg automatically, if necessary, by drawing a line from the last ver to the first.

Once we are through with drawing the shapes the old brush is selected back in the DC and then the brush created by us is deleted using **DeleteObject( )** function.

Types of Pens

In the previous program we have used the default solid black pen of thickness 1 pixel. We can create pens of different style, color and thickness to do our drawing. The following **OnPaint( )** handler shows how this can be achieved.

```
void OnPaint ( HWND hWnd )
{
    HDC hdc ;
    PAINTSTRUCT ps ;
    HPEN hpen ;
    HGDIOBJ holdpen ;

    hdc = BeginPaint ( hWnd, &ps ) ;

    hpen = CreatePen ( PS_DASH, 1, RGB ( 255, 0, 0 ) ) ;
    holdpen = SelectObject ( hdc, hpen ) ;

    MoveToEx ( hdc, 10, 10, NULL ) ;
    LineTo ( hdc, 500, 10 ) ;

    SelectObject ( hdc, holdpen ) ;
    DeleteObject ( hpen ) ;

    hpen = CreatePen ( PS_DOT, 1, RGB ( 255, 0, 0 ) ) ;
    holdpen = SelectObject ( hdc, hpen ) ;

    MoveToEx ( hdc, 10, 60, NULL ) ;
    LineTo ( hdc, 500, 60 ) ;

    SelectObject ( hdc, holdpen ) ;
    DeleteObject ( hpen ) ;
```

```
    hpen = CreatePen ( PS_DASHDOT, 1, RGB ( 255, 0, 0 ) ) ;
    holdpen = SelectObject ( hdc, hpen ) ;

    MoveToEx ( hdc, 10, 110, NULL ) ;
    LineTo ( hdc, 500, 110 ) ;

    SelectObject ( hdc, holdpen ) ;
    DeleteObject ( hpen ) ;

    hpen = CreatePen ( PS_DASHDOTDOT, 1, RGB ( 255, 0, 0 ) ) ;
    holdpen = SelectObject ( hdc, hpen ) ;

    MoveToEx ( hdc, 10, 160, NULL ) ;
    LineTo ( hdc, 500, 160 ) ;

    SelectObject ( hdc, holdpen ) ;
    DeleteObject ( hpen ) ;

    hpen = CreatePen ( PS_SOLID, 10, RGB ( 255, 0, 0 ) ) ;
    holdpen = SelectObject ( hdc, hpen ) ;

    MoveToEx ( hdc, 10, 210, NULL ) ;
    LineTo ( hdc, 500, 210 ) ;

    SelectObject ( hdc, holdpen ) ;
    DeleteObject ( hpen ) ;

    EndPaint ( hWnd, &ps ) ;
}
```

On execution of this program the window shown in Figure 18.3 appears.

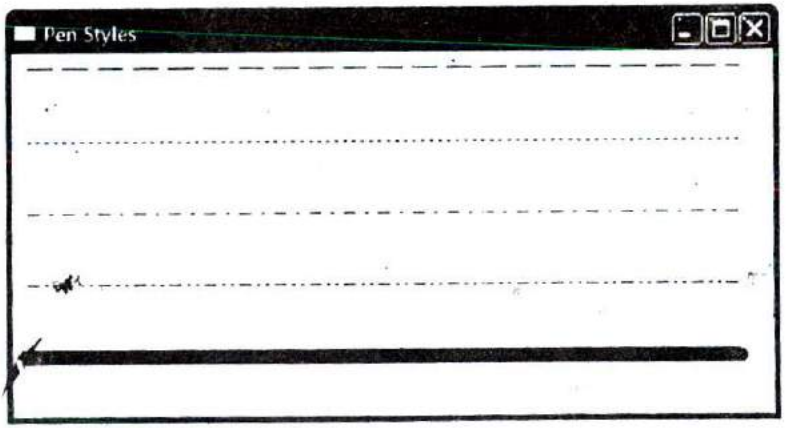

Figure 18.3

A new pen can be created using the **CreatePen( )** API function. This function needs three parameters—pen style, pen thickness and pen color. Different macros like PS_SOLID, PS_DOT, etc. have been defined in 'windows.h' to represent different pen styles. Note that for pen styles other than PS_SOLID the pen thickness has to be 1 pixel.

Types of Brushes

The way we can create different types of pens, we can also create three different types of brushes. These are—solid brush, hatch brush and pattern brush. Let us now write a program that shows how to build these brushes and then use them to fill rectangles. Here is the **OnPaint( )** handler which achieves this.

```
void OnPaint ( HWND hWnd )
{
    HDC hdc ;
    PAINTSTRUCT ps ;
    HBRUSH hbr ;
```

```
    HGDIOBJ holdbr ;
    HBITMAP hbmp ;

    hdc = BeginPaint ( hWnd, &ps ) ;

    hbr = CreateSolidBrush ( RGB (255, 0, 0 ) ) ;
    holdbr = SelectObject ( hdc, hbr ) ;

    Rectangle ( hdc, 5, 5, 105, 100 ) ;

    SelectObject ( hdc, holdbr ) ;
    DeleteObject ( hbr ) ;

    hbr = CreateHatchBrush ( HS_CROSS, RGB ( 255, 0, 0 ) ) ;
    holdbr = SelectObject ( hdc, hbr ) ;

    Rectangle ( hdc, 125, 5, 225, 100 ) ;

    SelectObject ( hdc, holdbr ) ;
    DeleteObject ( hbr ) ;

    hbmp = LoadBitmap ( hInst, MAKEINTRESOURCE ( IDB_BITMAP1 ) ) ;

    hbr = CreatePatternBrush ( hbmp ) ;
    holdbr = SelectObject ( hdc, hbr ) ;

    Rectangle ( hdc, 245, 5, 345, 100 ) ;

    SelectObject ( hdc, holdbr ) ;
    DeleteObject ( hbr ) ;
    DeleteObject ( hbmp ) ;

    EndPaint ( hWnd, &ps ) ;

    DeleteObject ( hbr ) ;
}
```

On execution of this program the window shown in Figure 18.4 appears.

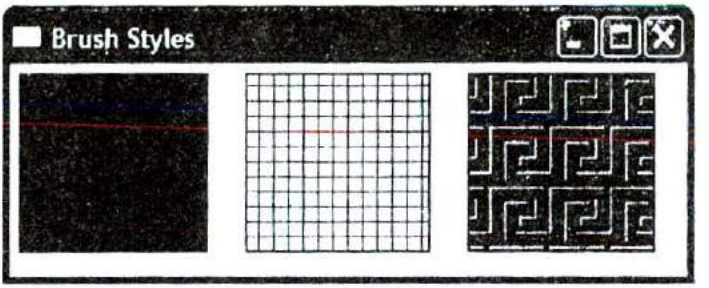

Figure 18.4

In the **OnPaint( )** handler we have drawn three rectangles—first using a solid brush, second using a hatched brush and third using a pattern brush. Creating and using a solid brush and hatched brush is simple. We simply have to make calls to **CreateSolidBrush( )** and **CreateHatchBrush( )** respectively. For the hatch brush we have used the style HS_CROSS. There are several other styles defined in 'windows.h' that you can experiment with.

For creating a pattern brush we need to first create a bitmap (pattern). Instead of creating this pattern, we have used a readymade bitmap file. You can use any other bitmap file present on your hard disk.

Bitmaps, menus, icons, cursors that a Windows program may use are its resources. When the compile such a program we usually want these resources to become a part of our EXE file. If so done we do not have to ship these resources separately. To be able to use a resource (bitmap file in our case) it is not enough to just copy it in the project directory. Instead we need to carry out the steps mentioned below to add a bitmap file to the project.

(a) From the 'Insert' menu option of VC++ 6.0 select the 'Resource' option.

(b) From the dialog that pops up select 'bitmap' followed by the import button.
(c) Select the suitable .bmp file.
(d) From the 'File' menu select the save option to save the generated resource script file (Script1.rc). When we select 'Save' one more file called 'resource.h' also gets created.
(e) Add the 'Script1.rc' file to the project using the Project | Add to Project | Files option.

While using the bitmap in the program it is always referred using an id. The id is **#defined** in the file 'resource.h'. Somewhere information has to be stored linking the id with the actual .bmp file on the disk. This is done in the 'Script1.rc' file. We need to include the 'resource.h' file in the program.

To create the pattern brush we first need to load the bitmap in memory. We have done this using the **LoadBitmap( )** API function. The first parameter passed to this function is the handle to the instance of the program. When **InitInstance( )** function is called from **WinMain( )** it stores the instance handle in a global variable **hInst**. We have passed this **hInst** to **LoadBitmap( )**. The second parameter passed to it is a string representing the bitmap. This string is created from the resource id using the **MAKEINTRESOURCE** macro. The **LoadBitmap( )** function returns the handle to the bitmap. This handle is then passed to the **CreatePatternBrush( )** function. This brush is then selected into the DC and then a rectangle is drawn using it.

Note that if the size of the bitmap is bigger than the rectangle being drawn then the bitmap is suitably clipped. On the other hand if the bitmap is smaller than the rectangle it is suitably replicated.

While doing the clean up firstly the brush is deleted followed by the bitmap.

Code and Resources

A program consists of both instructions and static data. Static data is that portion of the program which is not executed as machine instructions and which does not change as the program executes. Static data are character strings, data to create fonts, bitmaps, etc. The designers of Windows wisely decided that static data should be handled separately from the program code. The Windows term for static data is 'Resource data', or simply 'Resources'. By separating static data from the program code the creators of Windows were able to use a standard C/C++ compiler to create the code portion of the finished Windows program, and they only had to write a 'Resource compiler' to create the resources that Windows programs use. Separating the code from the resource data has other advantages like reducing memory demands and making programs more portable. It also means that a programmer can work on a program's logic, while a designer works on how the program looks.

Freehand Drawing, the Paintbrush Style

Even if you are knee high in computers I am sure you must have used PaintBrush. It provides a facility to draw a freehand drawing using mouse. Let us see if we too can achieve this. We can indicate where the freehand drawing begins by clicking the left mouse button. Then as we move the mouse on the table with the left mouse button depressed the freehand drawing should get drawn in the window. This drawing should continue till we do not release the left mouse button.

The mouse input comes in the form of messages. For free hand drawing we need to tackle three mouse messages—**WM_LBUTTONDOWN** for left button click, **WM_MOUSEMOVE** for mouse movement and **WM_LBUTTONUP** for releasing the left mouse button. Let us now see how these messag : are tackled for drawing freehand. The

WndProc( ) function and the message handlers that perform this task are given below

```
int x1, y1, x2, y2 ;

LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_DESTROY :
            OnDestroy ( hWnd ) ;
            break ;

        case WM_LBUTTONDOWN :
            OnLButtonDown ( hWnd, LOWORD ( lParam ),
                                  HIWORD ( lParam ) ) ;
            break ;

        case WM_LBUTTONUP :
            OnLButtonUp( ) ;
            break ;

        case WM_MOUSEMOVE :
            OnMouseMove ( hWnd, wParam, LOWORD ( lParam ),
                                        HIWORD ( lParam ) ) ;
            break ;

        default:
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
    }
    return 0 ;
}

void OnLButtonDown ( HWND hWnd, int x, int y )
{
    SetCapture ( hWnd ) ;
    x1 = x ;
```

```
    y1 = y ;
}

void OnMouseMove ( HWND hWnd, int flags, int x, int y )
{
    HDC hdc ;
    if ( flags == MK_LBUTTON ) /* is left mouse button depressed */
    {
        hdc = GetDC ( hWnd ) ;
        x2 = x ;
        y2 = y ;
        MoveToEx ( hdc, x1, y1, NULL ) ;
        LineTo ( hdc, x2, y2 ) ;

        ReleaseDC ( hWnd, hdc ) ;

        x1 = x2 ;
        y1 = y2 ;
    }
}

void OnLButtonUp( )
{
    ReleaseCapture( ) ;
}
```

On execution of this program the window shown in Figure 18.5 appears. We can now click the left mouse button with mouse pointer placed anywhere in the window. We can then drag the mouse on the table to draw the freehand. The freehand drawing would continue till we do not release the left mouse button.

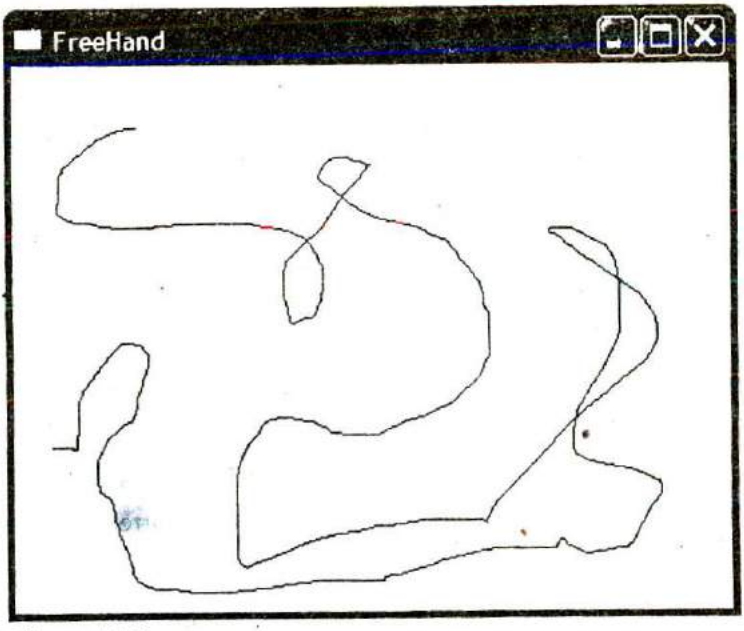

Figure 18.5

It appears that for drawing the freehand we should simply receive the mouse coordinates as it is moved and then highlight the pixels at these coordinates using the **SetPixel( )** API function. However, if we do so the freehand would be broken at several places. This is because usually the mouse is dragged pretty fast whereas the mouse move messages won't arrive so fast. A solution to this problem is to construct the freehand using small little line segments. This is what has been done in our program. These lines are so small is size that you would not even recognize that the freehand has been drawn by connecting these small lines.

Let us now discuss each mouse handler. When the **WM_LBUTTONDOWN** message arrives the **WndProc( )** function calls the handler **OnLButtonDown( )**. While doing so, we have passed the mouse coordinates where the click occurred. These coordinates are obtained in **lParam** in **WndProc( )**. In **lParam** the low order 16 bits contain the current x - coordinate of the mouse whereas the high order 16 bits contain the y - coordinate. The **LOWORD** and **HIWORD** macros have been used to separate out these x and y - coordinates from **lParam**.

In **OnLButtonDown( )** we have preserved the starting point of freehand in global variables **x1** and **y1**.

When **OnMouseMove( )** gets called it checks whether the left mouse button stands depressed. If it stands depressed then the **flags** variable contains **MK_LBUTTON**. If it does, then the current mouse coordinates are set up in the global variables **x2, y2**. A line is then drawn between **x1, y1** and **x2, y2** using the functions **MoveToEx( )** and **LineTo( )**. Next time around **x2, y2** should become the starting of the next line. Hence the current values of **x2, y2** are stored in **x1, y1**.

Note that here we have obtained the DC handle using the API function **GetDC( )**. This is because we are carrying out the drawing activity in reaction to a message other than **WM_PAINT**. Also, the handle obtained using **GetDC( )** should be released using a call to **ReleaseDC( )** function.

You can try using **BeginPaint( ) / EndPaint( )** in mouse handlers and **GetDC( ) / ReleaseDC( )** in **OnPaint( )**. Can you draw any conclusions?

Capturing the Mouse

If in the process of drawing the freehand the mouse cursor goes outside the client area then the window below our window would

start getting mouse messages. So our window would not receive any messages. If this has to be avoided then we should ensure that our window continues to receive mouse messages even when the cursor goes out of the client area of our window. The process of doing this is known as mouse capturing.

We have captured the mouse in **OnLButtonDown( )** handler by calling the API function **SetCapture( )**. As a result, the program continues to respond to mouse events during freehand drawing even if the mouse is moved outside the client area. In the **OnLButtonUp( )** handler we have released the captured mouse by calling the **ReleaseCapture( )** API function.

Device Context, a Closer Look

Now that we have written a few programs and are comfortable with idea of selecting objects like font, pen and brush into the DC, it is time for us to understand how Windows achieves the device independent drawing using the concept of DC. In fact a DC is nothing but a structure that holds handles of various drawing objects like font, pen, brush, etc. A screen DC and its working is shown in Figure 18.6.

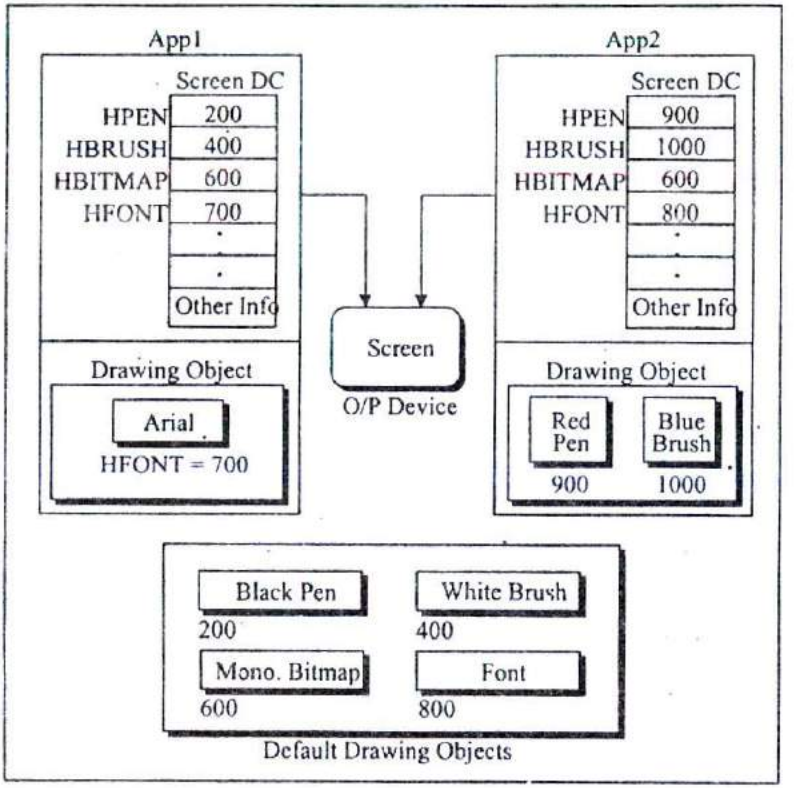

Figure 18.6

You can make following observations from Figure 18.6:

(a) The DC doesn't hold the drawing objects like pen, brush, etc. It merely holds their handles.

(b) With each DC a default monochrome bitmap of size 1 pixel x 1 pixel is associated.

(c) Default objects like black pen, white brush, etc. are shared by different DCs in same or different applications.

(d) The drawing objects that an application explicitly creates can be shared within DCs of the same application, but is never shared between different applications.

(e) Two different applications would need two different DCs even though both would be used to draw to the same screen. In other words with one screen multiple DCs can exist.

(f) A common Device Driver would serve the drawing requests coming from different applications. (Truly speaking the request comes from GDI functions that our application calls).

Screen and printer DC is OK, but what purpose would a memory DC serve? Well, that is what the next program would explain.

Displaying a Bitmap

We are familiar with drawing normal shapes on screen using a device context. How about drawing images on the screen? Windows does not permit displaying a bitmap image directly using a screen DC. This is because there might be color variations in the screen on which the bitmap was created and the screen on which it is being displayed. To account for such possibilities while displaying a bitmap Windows uses a different mechanism—a 'Memory DC'

The way anything drawn using a screen DC goes to screen, anything drawn using a printer DC goes to a printer, similarly anything drawn using a memory DC goes to memory (RAM). But where in RAM—in the 1 x 1 pixel bitmap whose handle is present in memory DC. (Note that this handle was of little use In case of screen/printer DC). Thus if we attempt to draw a line using a memory DC it would end up on the 1 x 1 pixel bitmap. You would agree 1 x 1 is too small a place to draw even a small line. Hence we need to expand the size and color capability of this bitmap. How can this be done? Simple, just replace the handle of the 1 x 1 bitmap with the handle of a bigger and colored bitmap object. This is shown in Figure 18.7.

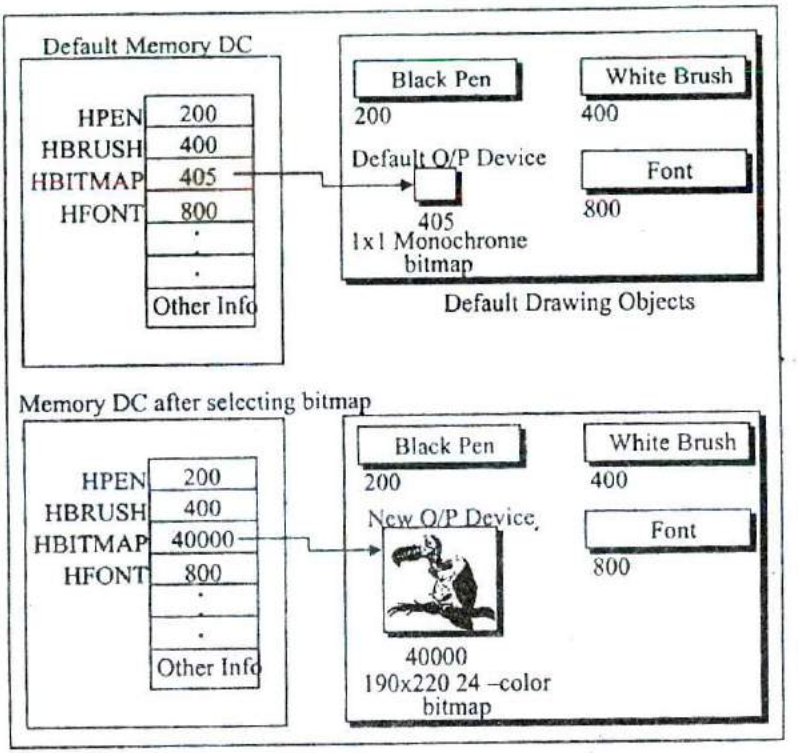

Figure 18.7

What purpose would just increasing the bitmap size/color would serve? Whatever we draw here would get drawn on the bitmap but would still not be visible. We can make it visible by simply copying the bitmap image (including what has been drawn on it) to the screen DC by using the API function **BitBlt( )**.

Before transferring the image to the screen DC we need to make the memory DC compatible with the screen DC. Here making compatible means making certain adjustments in the contents of the memory DC structure. Looking at these values the screen device driver would suitably adjust the colors when the pixels in

the bitmap of memory DC is transferred to screen DC using **BitBlt( )** function.

Let us now take a look at the program that puts all these concepts in action. The program merely displays the image of a vulture in a window. Here is the code...

```
void OnPaint ( HWND hWnd )
{
    HDC hdc ;
    HBITMAP hbmp ;
    HDC hmemdc ;
    HGDIOBJ holdbmp ;
    PAINTSTRUCT ps ;

    hdc = BeginPaint ( hWnd, &ps ) ;

    hbmp = LoadBitmap ( hInst, MAKEINTRESOURCE ( IDB_BITMAP1 ) ) ;

    hmemdc = CreateCompatibleDC ( hdc ) ;
    holdbmp = SelectObject ( hmemdc, hbmp ) ;

    BitBlt ( hdc, 10, 20, 190, 220, hmemdc, 0, 0, SRCCOPY ) ;

    EndPaint ( hWnd, &ps ) ;

    SelectObject ( hmemdc, holdbmp ) ;
    DeleteObject ( hbmp ) ;
    DeleteDC ( hmemdc ) ;
}
```

On executing the program we get the window shown in Figure 18.7.

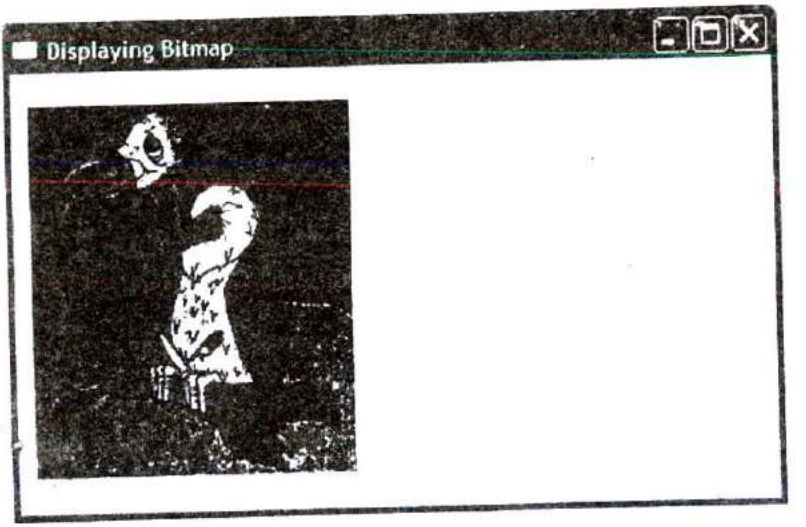

Figure 18.7

As usual we begin our drawing activity in **OnPaint( )** by first getting the screen DC using the **BeginPaint( )** function. Next we have loaded the vulture bitmap image in memory by calling the **LoadBitmap( )** function. Its usage is similar to what we saw while creating a pattern brush in an earlier section of this chapter. Then we have created a memory device context and made its properties compatible with that of the screen DC. To do this we have called the API function **CreateCompatibleDC( )**. Note that we have passed the handle to the screen DC to this function. The function in turn returns the handle to the memory DC. After this we have selected the loaded bitmap into the memory DC. Lastly, we have performed a bit block transfer (a bit by bit copy) from memory DC to screen DC using the function **BitBlt( )**. As a result of this the vulture now appears in the window.

We have made the call to **BitBlt( )** as shown below:

```
BitBlt ( hdc, 10, 20, 190, 220, hmemdc, 0, 0, SRCCOPY ) ;
```

Let us now understand its parameters. These are as under:

hdc – Handle to target DC where the bitmap is to be blitted

10, 20 – Position where the bitmap is to be blitted

190, 220 – Width and height of bitmap being blitted

0, 0 – Top left corner of the source image. If we give 10, 20 then the image from 10, 20 to bottom right corner of the bitmap would get blitted.

SRCCOPY – Specifies one of the raster-operation codes. These codes define how the color data for the source rectangle is to be combined with the color data for the destination rectangle to achieve the final color. SRCCOPY means that the pixel color of source should be copied onto the destination pixel of the target.

Animation at Work

Speed is the essence of life. So having the ability to display a bitmap in a window is fine, but if we can add movement and sound to it then nothing like it. So let us now see how to achieve this animation and sound effect.

If we are to animate an object in the window we need to carry out the following steps:

(a) Create an image that is to be animated as a resource.
(b) Prepare the image for later display.
(c) Repeatedly display this prepared image at suitable places in the window taking care that when the next image is displayed the previous image is erased.
(d) Check for collisions while displaying the prepared image.

Let us now write a program that on execution makes a red colored ball move in the window. As the ball strikes the walls of the

window a noise occurs. Note that the width and height of the red-colored ball is 22 pixels. Given below is the **WndProc( )** function and the various message handlers that help achieve animation and sound effect.

```
HBITMAP hbmp ;
int x, y ;
HDC hmemdc ;
HGDIOBJ holdbmp ;

LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_DESTROY :
            OnDestroy ( hWnd ) ;
            break ;
        case WM_CREATE :
            OnCreate ( hWnd ) ;
            break ;
        case WM_TIMER :
            OnTimer ( hWnd ) ;
            break ;
        default :
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
    }
    return 0 ;
}

void OnCreate ( HWND hWnd )
{
    RECT r ;
    HDC hdc ;

    hbmp = LoadBitmap ( hInst, MAKEINTRESOURCE ( IDB_BITMAP1 ) ) ;

    hdc = GetDC ( hWnd ) ;
```

```
    hmemdc = CreateCompatibleDC ( hdc ) ;
    holdbmp = SelectObject ( hmemdc, hbmp ) ;

    ReleaseDC ( hWnd, hdc ) ;

    srand ( time ( NULL ) ) ;
    GetClientRect ( hWnd, &r ) ;
    x = rand( ) % r.right - 22 ;
    y = rand( ) % r.bottom - 22 ;

    SetTimer ( hWnd, 1, 50, NULL ) ;
}

void OnDestroy ( HWND hWnd )
{
    KillTimer ( hWnd, 1 ) ;
    SelectObject ( hmemdc, holdbmp ) ;
    DeleteDC ( hmemdc ) ;
    DeleteObject ( hbmp ) ;
    PostQuitMessage ( 0 ) ;
}

void OnTimer ( HWND hWnd )
{
    HDC hdc ;
    RECT r ;
    const int wd = 22, ht = 22 ;
    static int dx = 10, dy = 10 ;

    hdc = GetDC ( hWnd ) ;
    BitBlt ( hdc, x, y, wd, ht, hmemdc, 0, 0, WHITENESS ) ;
    GetClientRect ( hWnd, &r ) ;

    x += dx ;
    if ( x < 0 )
    {
        x = 0 ;
        dx = 10 ;
```

```
        PlaySound ("chord.wav", NULL, SND_FILENAME | SND_ASYNC ) ;
    }
    else if ( x > ( r.right - wd ) )
    {
        x = r.right - wd ;
        dx = -10 ;
        PlaySound ("chord.wav", NULL, SND_FILENAME | SND_ASYNC ) ;
    }

    y += dy ;
    if ( y < 0 )
    {
        y = 0 ;
        dy = 10 ;
        PlaySound ("chord.wav", NULL, SND_FILENAME | SND_ASYNC ) ;
    }
    else if ( y > ( r.bottom - ht ) )
    {
        y = r.bottom - ht ;
        dy = -10 ;
        PlaySound ( "chord.wav", NULL, SND_FILENAME | SND_ASYNC );
    }

    BitBlt ( hdc, x, y, wd, ht, hmemdc, 0, 0, SRCCOPY ) ;
    ReleaseDC ( hWnd, hdc ) ;
}
```

From the **WndProc( )** function you can observe that we have handled two new messages here—**WM_CREATE** and **WM_TIMER**. For these messages we have called the handlers **OnCreate( )** and **OnTimer( )** respectively. Let us now understand these handlers one by one

WM_CREATE and *OnCreate( )*

The **WM_CREATE** message arrives whenever a new window is created. Since usually a window is created only once, the one-time

activity that is to be carried out in a program is usually done in **OnCreate( )** handler. In our program to make the ball move we need to display it at different places at different times. To do this it would be necessary to blit the ball image several times. However, we need to load the image only once. As this is a one-time activity it has been done in the handler function **OnCreate( )**.

You are already familiar with the steps involved in preparing the image for blitting—loading the bitmap, creating a memory DC, making it compatible with screen DC and selecting the bitmap in the memory DC.

Apart from preparing the image for blitting we have also done some intialisations like setting up values in some variables to indicate the initial position of the ball. We have also called the **SetTimer( )** function. This function tells Windows to post a message **WM_TIMER** into the message queue of our application every 50 milliseconds.

WM_TIMER and *OnTimer( )*

If we are to perform an activity at regular intervals we have two choices:

(a) Use a loop and monitor within the loop when is it time to perform that activity.
(b) Use a Windows mechanism of timer. This mechanism when used posts a WM_TIMER message at regular intervals to our application.

The first method would seriously hamper the responsiveness of the program. If the control is within the loop and a new message arrives the message would not get processed unless the control goes out of the loop. The second choice is better because it makes the program event driven. That is, whenever **WM_TIMER** arrives that time its handler does the job that we want to get executed

periodically. At other times the application is free to handle other messages that come to its queue.

All that we have done in the **OnTimer( )** handler is erase the ball from previous position and draw it at a new position. We have also checked if the ball has hit the boundaries of the window. If so we have played a sound file using the **PlaySound( )** API function and then changed the direction of the ball.

A Few More Points...

A few more points worth noting before we close our discussion on animation...

(a) One application can set up multiple timers to do different jobs at different intervals. Hence we need to pass the id of the timer that we want to set up to the **SetTimer( )** function. In our case we have specified the id as 1.

(b) For multiple timers Windows would post multiple **WM_TIMER** messages. Each time it would pass the timer id as additional information about the message.

(c) For drawing as well as erasing the ball we have used the same function—**BitBlt( )**. While erasing we have used the raster operation code **WHITENESS**. When we use this code the color values of the source pixels get ignored. Thus red colored pixels of ball would get ignored leading to erasure of the ball in the window.

(d) The size of client area of the window can be obtained using the **GetClientRect( )** API function.

(e) We want that every time we run the application the initial position of the ball should be different. To ensure this we have generated its initial x, y coordinates using the standard library function **rand( )**. However, this function doesn't generate true random numbers. To ensure that we do get true

random numbers, somehow we need to tie the random number generation with time, as time of each execution of our program would be different. This has been achieved by making the call

```
srand ( time ( NULL ) ) ;
```

Here **time( )** is function that returns the time. We have further passed this time to the **srand( )** function.

(f) To be able to use **rand( )** and **srand( )** functions include the file 'stdlib.h'. Similarly for **time( )** function to work include the file 'time.h'.

(g) In the call to the **PlaySound( )** function the first parameter is the name of the wave file that is to be played. If first parameter is filename then the second has to be NULL. The third parameter is a set of flags. **SND_FILENAME** indicates that the first parameter is the filename, whereas **SND_ASYNC** indicates that the sound should be played in the background.

(h) To be able to use the **PlaySound( )** function we need to link the library 'winmm.lib'. This is done by using 'Project | Settings' menu item. On selection of this item a dialog pops up. In the 'Link' tab of this dialog mention the name 'winmm.lib' in the 'Object / Library modules' edit box.

(i) When the application terminates we have to instruct Windows not to send **WM_TIMER** messages to our application any more. For this we have called the **KillTimer( )** API function passing to it the ID of the timer.

Windows, the Endless World...

The biggest hurdle in Windows programming is a sound understanding of its programming model. In this chapter and in the last two I have tried to catch the essence of Windows' Event

Driven Programming model. Once you have understood it thoroughly rest is just a matter of understanding and calling the suitable API functions to get your job done. Windows API is truly an endless world. It covers areas like Networking, Internet programming, Telephony, Drawing and Printing, Device I/O, Imaging, Messaging, Multimedia, Windowing, Database programming, Shell programming, to name a few. The programs that we have written have merely scratched the surface. No matter how many programs that we write under Windows, several still remain to be written. The intention of this chapter was to unveil before you, to give you the first glimpse of what is possible under Windows. The intention all along was not to catch fish for you but to show you how to catch fish so that you can do fishing all your life. Having made a sound beginning, rest is for you to explore. Good luck and happy fishing!

Summary

(a) In DOS, programmers had to write separate graphics code for every new video adapter. In Windows, the code once written works on any video adapter.

(b) A Windows program cannot draw directly on an output device like screen or printer. Instead, it draws to the logical display surface using device context.

(c) When the window is displayed for the first time, or when it is moved or resized **OnPaint( )** handler gets called.

(d) It is necessary to obtain the device context before drawing text or graphics in the client area.

(j) A device context is a structure containing information required to draw on a display surface. The information includes color of pen and brush, screen resolution, color palettes, etc.

(e) To draw using a new pen or brush it is necessary to select them into the device context.

(f) If we don't select any brush or pen into the device context then the drawing drawn in the client area would be drawn

with the default pen (black pen) and default brush (white brush).

(g) RGB is a macro representing the Red, Green and Blue elements of a color. RGB (0, 0, 0) gives black color, whereas, RGB (255, 255, 255) gives white color.

(h) Animation involves repeatedly drawing the same image at successive positions.

Exercise

[A] State True or Faise:

(a) Device independence means the same program is able to work using different screens, keyboards and printers without modifications to the program.

(b) The WM_PAINT message is generated whenever the client area of the window needs to be redrawn.

(c) The API function **EndPaint( )** is used to release the DC.

(d) The default pen in the DC is a solid pen of white color.

(e) The pen thickness for the pen style other than PS_SOLID has to be 1 pixel.

(f) **BeginPaint( )** and **GetDC( )** can be used interchangeably.

(g) If we drag the mouse from (10, 10) to (110, 100), 100 WM_MOUSEMOVE messages would be posted into the message queue.

(h) WM_PAINT message is raised when the window contents are scrolled.

(i) With each DC a default monochrome bitmap of size 1 pixel x 1 pixel is associated.

(j) The WM_CREATE message arrives whenever a window is displayed.

[B] Answer the following:

(a) What is meant by Device Independent Drawing and how it is achieved?

(b) Explain the significance of WM_PAINT message.

(c) How Windows manages the code and various resources of a program?

(d) Explain the Windows mechanism of timer.

(e) What do you mean by capturing a mouse?

(f) Write down the steps that need to be carried out to animate an object.

[C] Attempt the following:

(a) Write a program, which displays "hello" at any place in the window where you click the left mouse button. If you click the right mouse button the color of subsequent hellos should change.

(b) Write a program that would draw a line by joining the new point where you have clicked the left mouse button with the last point where you clicked the left mouse button.

(c) Write a program to gradient fill the entire client area with shades of blue color.

(d) Write a program to create chessboard like boxes (8 X 8) in the client area. If the window is resized the boxes should also get resized so that all the 64 boxes are visible at all times.

(e) Write a program that displays only the upper half of a bitmap of size 40 x 40.

(f) Write a program that displays different text in different colors and fonts at different places after every 10 seconds.

19 Interaction With Hardware

- Hardware Interaction
- Hardware Interaction, DOS Perspective
- Hardware Interaction, Windows Perspective
- Communication with Storage Devices
  - The *ReadSector( )* Function
- Accessing Other Storage Devices
- Communication with Keyboard
  - Dynamic Linking
  - Windows Hooks
- Caps Locked, Permanently
- Did You Press It TTwwiiccee....
- Mangling Keys
- KeyLogger
- Where is This Leading
- Summary
- Exercise

There are two types of Windows programmers those who are happy in knowing the things the way they are under Windows and those who wish to know why the things are the way they are. This chapter is for the second breed of programmers. They are the real power users of Windows. Because it is they who first understand the default working of different mechanisms that Windows uses and then are able to make those mechanisms work to their advantage. The focus here would be restricted to mechanisms that are involved in interaction with the hardware under the Windows world. Read on and I am sure you would be on your path to become a powerful Windows programmer.

Hardware Interaction

Primarily interaction with hardware suggests interaction with peripheral devices. However, its reach is not limited to interaction with peripherals. The interaction may also involve communicating with chips present on the motherboard. Thus more correctly, interaction with hardware would mean interaction with any chip other than the microprocessor. During this interaction one or more of the following activities may be performed:

(a) Reacting to events that occur because of user's interaction with the hardware. For example, if the user presses a key or clicks the mouse button then our program may do something.

(b) Reacting to events that do not need explicit user's interaction. For example, on ticking of a timer our program may want to do something.

(c) Explicit communication from a program without the occurrence of an event. For example, a program may want to send a character to the printer, or a program may want to read/write the contents of a sector from the hard disk.

Let us now see how this interaction is done under different platforms.

Hardware Interaction, DOS Perspective

Under DOS whenever an external event (like pressing a key or ticking of timer) occurs a signal called hardware interrupt gets generated. For different events there are different interrupts. As a reaction to the occurrence of an interrupt a table called Interrupt Vector Table (IVT) is looked up. IVT is present in memory. It is populated with addresses of different BIOS routines during booting. Depending upon which interrupt has occurred the Microprocessor picks the address of the appropriate BIOS routine from IVT and transfers execution control to it. Once the control reaches the BIOS routine, the code in the BIOS routine interacts with the hardware. Naturally, for different interrupts different BIOS routines are called. Since these routines serve the interrupts they are often called 'Interrupt Service Routines' or simply ISRs.

Refer Figure 19.1 to understand this mechanism.

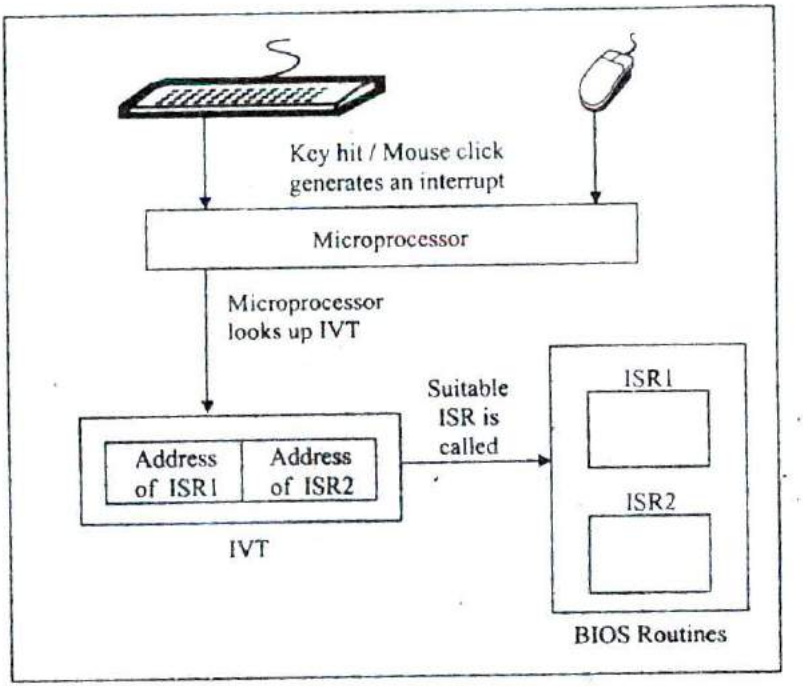

Figure 19.1

If we want that instead of the default ISR our routine should get called then it is necessary to store the address of this routine in IVT. Once this is done whenever a hardware interrupt occurs our routine's address from IVT is picked up and the control is transferred to our routine. For example, we may register our ISR in IVT to gain finer control over the way key-hits from the keyboard are tackled. This finer control may involve changing codes of keys or handling hitting of multiple keys simultaneously.

Explicit communication with the hardware can be done in four different ways. These are shown in Figure 19.2.

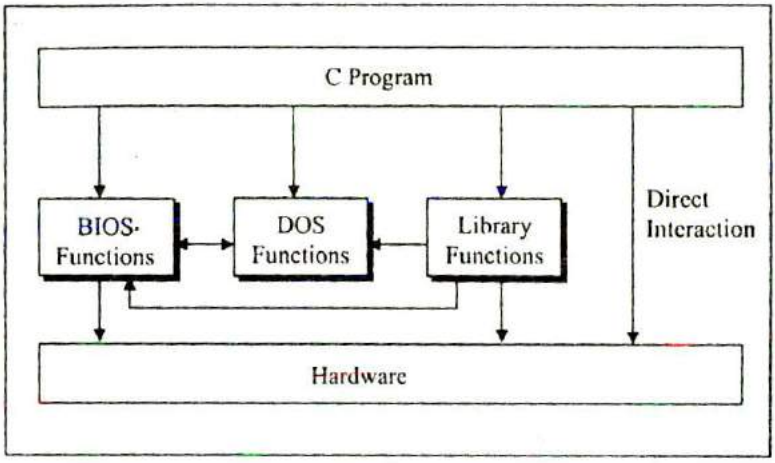

Figure 19.2

Let us now discuss the pros and cons of using these different methods to interact with the hardware.

(a) Calling DOS Functions

To interact with the hardware a program can call DOS functions. These functions can either directly interact with the hardware or they may call BIOS functions which in turn interact with the hardware. As a result, the programmer is not required to know all the hardware details to be able to interact with it. However, since DOS functions do not have names they have to be called through the mechanism of interrupts. This is difficult since the programmer has to remember interrupt service numbers for calling different DOS functions. Moreover, communication with these functions has to be done using CPU registers. This leads to lot of difficulties since different functions use different registers for communication. So one has to know details of different CPU registers, how to use them, which one to use when, etc.

(b) Calling BIOS Functions

DOS functions can carry out jobs like console I/O, file I/O, printing, etc. For other operations like generating graphics, carrying out serial communication, etc. the program has to call another set of functions called ROM-BIOS functions. Note that there are some functions in ROM-BIOS that do same jobs as equivalent DOS functions. BIOS functions suffer from the same difficulty as DOS functions—they do not have names. Hence they have to be called using interrupts and involve heavy usage of registers.

(c) Calling Library Functions

We can call library functions which in turn can call DOS/BIOS functions to carry out the interaction with hardware. Good examples of these functions are **printf( ) / scanf( ) / getch( )** for interaction with console, **absread( ) / abswrite( )** for interaction with disk, **bioscom( )** for interaction with serial port, etc. But the library doesn't have a parallel function for every DOS/BIOS function. Hence at some point of time one has to learn how to call DOS/BIOS functions.

(d) Directly interacting with the hardware

At times the programs are needed to directly interact with the hardware. This has to be done because either there are no library functions or DOS/BIOS functions to do this, or if they are there their reach is limited. For example, while writing good video games one is required to watch the status of multiple keys simultaneously. The library functions as well as the DOS/BIOS functions are unable to do this. At such times we have to interact with the keyboard controller chip directly.

However, direct interaction with the hardware is difficult because one has to have good knowledge of technical details of the chip to be able to do so. Moreover, not every technical detail about how the hardware from a particular manufacturer works is well documented.

Hardware Interaction, Windows Perspective

Like DOS, under Windows too a hardware interrupt gets generated whenever an external event occurs. As a reaction to this signal a table called Interrupt Descriptor Table (IDT) is looked up and a corresponding routine for the interrupt gets called. Unlike DOS the IDT contains addresses of various kernel routines (instead of BIOS routines). These routines are part of the Windows OS itself. When the kernel routine is called, it in turn calls the ISR present in the appropriate device driver. This ISR interacts with the hardware. Two questions may now occur to you:

(a) Why the kernel routine does not interact with the hardware directly?
(b) Why the ISR of the device driver not registered directly in the IDT?

Let us find answer to the first question. Every piece of hardware works differently than the other. As new pieces of hardware come into existence new code has to be written to be able to interact with them. If this code is written in the kernel then the kernel would have to be rewritten and recompiled every time a new hardware comes into existence. This is practically impossible. Hence the new code to interact with the device is written in a separate program called device driver. With every new piece of hardware a new device driver is provided. This device driver is an extension of the OS itself.

Let us now answer the second question. Out of the several components of Windows OS a component called kernel is tightly integrated with the processor architecture. If the processor architecture changes then the kernel is bound to change. One of goals of Windows NT family was to keep the other components of OS and the device drivers portable across different microprocessor architectures. All processor architectures may not use IDT for the registration and lookup mechanism. So, had registration of the device driver's ISR in IDT been allowed, then the mechanism

would fail on processors which do not use IDT, thereby compromising portability of device drivers.

Refer Figure 19.3 for understanding the interrupt handling mechanism under Windows.

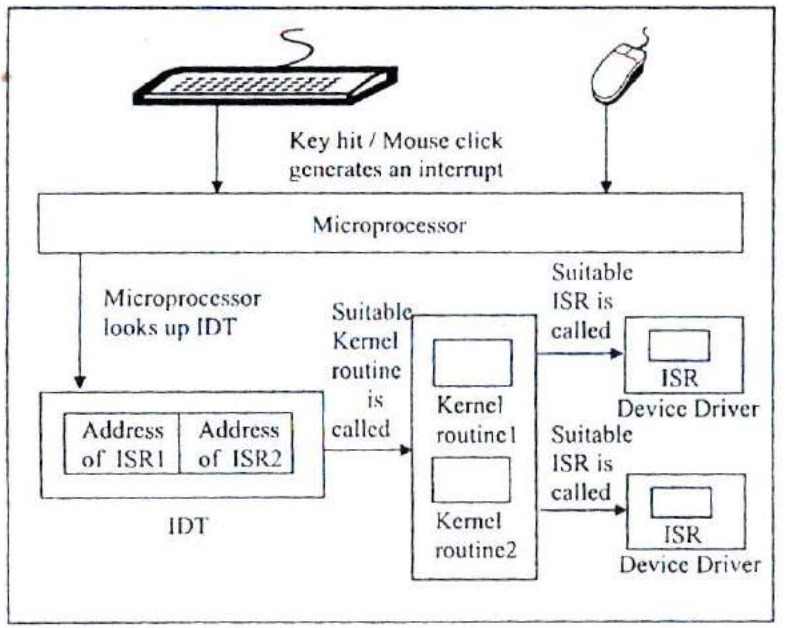

Figure 19.3

If we are to gain finer control while reacting to interrupts we would be required to write a device driver containing a new ISR to do so.

Under Windows explicit communication with hardware is much different than the way it was done under DOS. This is primarily because under Windows every device is shared amongst multiple applications running in memory. To avoid conflict between different programs accessing the same device simultaneously

Windows does not permit an application program to directly access any of the devices. Instead it provides several API functions to carry out the interaction. These functions have names so calling them is much easier than calling DOS/BIOS functions. When we call an API function to interact with a device, it in turn accesses the device driver program for the device. It is the device driver program that finally accesses the device. There is a standard way in which an application can communicate with the device driver. It is device driver's responsibility to ensure that multiple requests coming from different applications are handled without causing any conflict. In the sections to follow we would see how to communicate with the device driver to be able to interact with the hardware.

One last question—won't the API change if a new device comes into existence? No it won't. That is the beauty of the Windows architecture. All that would change is the device driver program for the new device. The API functions that we would need to interact with this new device driver would remain same. This is shown in Figure 19.4

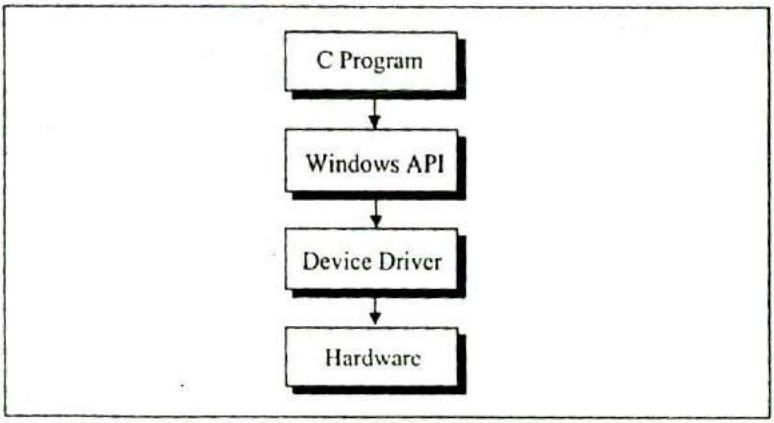

Figure 19.4

Communication with Storage Devices

Since DOS is commercially dead the rest of the chapter would focus on communication with the devices under Windows platform. We would illustrate this with the help of several programs.

Let us begin with the one that interacts with the simplest storage device, namely the floppy disk. Rather than the physical structure of the floppy disk it is the way the stored information is laid out and managed that concerns programmers most. Let us understand how the information is laid out on a floppy disk. Each floppy disk consists of four logical parts—Boot Sector, File Allocation Table (FAT), Directory and Data space. Of these, the Boot Sector contains information about how the disk is organized. That is, how many sides does it contain, how many tracks are there on each side, how many sectors are there per track, how many bytes are there per sector, etc. The files and the directories are stored in the Data Space. The Directory contains information about the files like its attributes, name, size, etc. The FAT contains information about where the files and directories are stored in the data space. Figure 19.5 shows the four logical parts of a 1.44 MB disk.

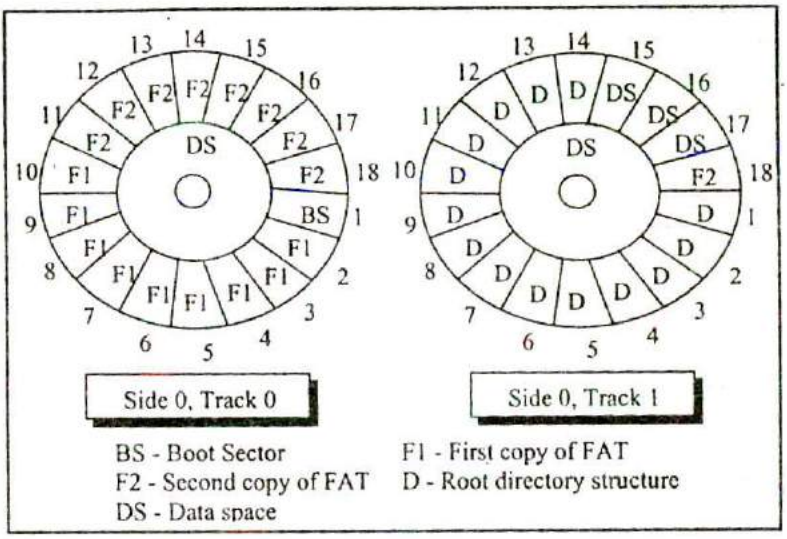

Figure 19.5

With the logical structure of the floppy disk behind us let us now write a program that reads the boot sector of a floppy disk and displays its contents on the screen. But why on earth would we ever like to do this? Well, that's what all Windows-based Anti-viral softwares do when they scan for boot sector viruses. A good enough reason for us to add the capability to read a boot sector to our knowledge! Here is the program...

```
# include <stdafx.h>
# include <windows.h>
# include <stdio.h>
# include <conio.h>

# pragma pack ( 1 )
struct boot
{
    BYTE jump [ 3 ];
```

```
    char bsOemName [ 8 ] ;
    WORD bytesperSector ;
    BYTE sectorspercluster ;
    WORD sectorsreservedarea ;
    BYTE copiesFAT ;
    WORD maxrootdirentries ;
    WORD totalSectors ;
    BYTE mediaDescriptor ;
    WORD sectorsperFAT ;
    WORD sectorsperTrack ;
    WORD sides ;
    WORD hiddenSectors ;
    char reserve [ 480 ] ;
} ;

void ReadSector ( char*src, int ss, int num, void* buff ) ;

void main( )
{
    struct boot b ;
    ReadSector ( "\\\\.\\A:", 0, 1, &b ) ;

    printf ( "Boot Sector name: %s\n", b.id ) ;
    printf ( "Bytes per Sector: %d\n", b.bps ) ;
    printf ( "Sectors per Cluster: %d\n", b.spc ) ;
    /* rest of the statements can be written by referring Figure 19.6
       and Appendix G*/
}

void ReadSector ( char *src, int ss, int num, void* buff )
{
    HANDLE h ;
    unsigned int br ;
    h = CreateFile  ( src, GENERIC_READ,
                      FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0 ) ;
    SetFilePointer ( h, ( ss * 512 ), NULL, FILE_BEGIN ) ;
    ReadFile ( h, buff, 512 * num, &br, NULL ) )
    CloseHandle ( h ) ;
```

}

The boot sector contains two parts—'Boot Parameters' and 'Disk Bootstrap Program'. The Boot Parameters are useful while performing read/write operations on the disk. Figure 19.6 shows the break up of the boot parameters for a floppy disk.

Description	Length	Typical Values
Jump instruction	3	EB3C90
OEM name	8	MSWIN4.1
Bytes per sector	2	512
Sectors per cluster	1	1
Reserved sectors	2	1
Number of FAT copies	1	2
Max. Root directory entries	2	224
Total sectors	2	2880
Media descriptor	1	F0
Sectors per FAT	2	9
Sectors per track	2	18
No. Of sides	2	2
Hidden sectors	4	0
Huge sectors	4	0
BIOS drive number	1	0
Reserved sectors	1	0
Boot signature	1	41
Volume ID	4	349778522
Volume label	11	ICIT
File system type	8	FAT12

Figure 19.6

Using the breakup of bytes shown in Figure 19.6 our program has first created a structure called **boot**. Notice the usage of **#pragma pack** to ensure that all elements of the structure are aligned on a 1-byte boundary, rather than the default 4-byte boundary. Then comes the surprise—there is no **WinMain( )** in the program. This is because we want to display the boot sector contents on the screen rather than in a window. This has been done only for the sake of simplicity. Remember that our aim is to interact with the floppy, and not in drawing and painting in a window. If you wish you can of course adapt this program to display the same contents in a window. So the program is still a Windows application. Only difference is that it is built as a 'Win32 Console Application' using VC++. A console application always begins with **main( )** rather than **WinMain( )**.

To actually read the contents of boot sector of the floppy disk the program makes a call to a user-defined function called **ReadSector( )**. The **ReadSector( )** function is quite similar to the **absread( )** library function available in Turbo C/C++ under DOS.

The first parameter passed to **ReadSector( )** is a string that indicates the storage device from where the reading has to take place. The syntax for this string is **\\machine-name\storage-device name**. In **\\.\A:**, we have used '.' for the machine name. A '.' means the same machine on which the program is executing. Needless to say, **A:** refers to the floppy drive. The second parameter is the logical sector number. We have specified this as 0 which means the boot sector in case of a floppy disk. The third parameter is the number of sectors that we wish to read. This parameter is specified as 1 since the boot sector occupies only a single sector. The last parameter is the address of a buffer/variable that would collect the data that is read from the floppy. Here we have passed the address of the boot structure variable **b**. As a result, the structure variable would be setup with the contents of the boot sector data at the end of the function call.

Once the contents of the boot sector have been read into the structure variable **b** we have displayed the first few of them on the screen using **printf( )**. If you wish you can print the rest of the contents as well.

The *ReadSector( )* Function

With the preliminaries over let us now concentrate on the real stuff in this program, i.e. the **ReadSector( )** function. This function begins by making a call to the **CreateFile( )** API function as shown below:

```
h = CreateFile ( src, GENERIC_READ,
                 FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0 ) ;
```

The **CreateFile( )** API function is very versatile. Anytime we are to communicate with a device we have to firstly call this API function. The **CreateFile( )** function opens the specified device as a file. Windows treats all devices just like files on disk. Reading from this file means reading from the device.

The **CreateFile( )** API function takes several parameters. The first parameter is the string specifying the device to be opened. The second parameter is a set of flags that are used to specify the desired access to the file (representing the device) about to be opened. By specifying the **GENERIC_READ** flag we have indicated that we just wish to read from the file (device). The third parameter specifies the sharing access for the file (device). Since floppy drive is a shared resource across all the running applications we have specified the **FILE_SHARE_READ** flag. In general while interacting with any hardware the sharing flag for the file (device) must always be set to this value since every piece of hardware is shared amongst all the running applications. The fourth parameter indicates security access for the file (device). Since we are not concerned with security here we have specified the value as **0**. The fifth parameter specifies what action to take if

the file already exists. When using **CreateFile( )** for device access we must always specify this parameter as **OPEN_EXISTING**. Since the floppy disk file was already opened by the OS a long time back during the booting. The remaining two parameters are not used when using **CreateFile( )** API function for device access. Hence we have passed a **0** value for them. If the call to **CreateFile( )** succeeds then we obtain a handle to the file (device).

The device file mechanism allows us to read from the file (device) by setting the file pointer using the **SetFilePointer( )** API function and then reading the file using the **ReadFile( )** API function. Since every sector is **512** bytes long, to read from the n^{th} sector we need to set the file pointer to the **512 * n** bytes from the start of the file. The first parameter to **SetFilePointer( )** is the handle of the device file that we obtained by calling the **CreateFile( )** function. The second parameter is the byte offset from where the reading is to begin. This second parameter is relative to the third parameter. We have specified the third parameter as **FILE_BEGIN** which means the byte offset is relative to the start of the file.

To actually read from the device file we have made a call to the **ReadFile( )** API function. The **ReadFile( )** function is very easy to use. The first parameter is the handle of the file (device), the second parameter is the address of a buffer where the read contents should be dumped. The third parameter is the count of bytes that have to be read. We have specified the value as **512 * num** so as to read **num** sectors. The fourth parameter to **ReadFile( )** is the address of an **unsigned int** variable which is set up with the count of bytes that the function was successfully able to read. Lastly, once our work with the device is over we should close the file (device) using the **CloseHandle( )** API function.

Though **ReadSector( )** doesn't need it, there does exist a counterpart of the **ReadFile( )** function. Its name is **WriteFile( )**. This API function can be used to write to the file (device). The parameters of **WriteFile( )** are same as that of **ReadFile( )**. Note

that when **WriteFile( )** is to be used we need to specify the **GENERIC_WRITE** flag in the call to **CreateFile( )** API function. Given below is the code of **WriteSector( )** function that works exactly opposite to the **ReadSector( )** function.

```
void WriteSector ( char *src, int ss, int num, void* buff )
{
    HANDLE h ;
    unsigned int br ;
    h = CreateFile  ( src, GENERIC_WRITE,
                      FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0 ) ;
    SetFilePointer ( h, ( ss * 512 ), NULL, FILE_BEGIN ) ;
    WriteFile ( h, buff, 512 * num, &br, NULL ) )
    CloseHandle ( h ) ;
}
```

Accessing Other Storage Devices

Note that the mechanism of reading from or writing to any device remains standard under Windows. We simply need to change the string that specifies the device. Here are some sample calls for reading/writing from/to various devices:

```
ReadSector ( "\\\\.\\a:", 0, 1, &b ) ;  /* reading from 2nd floppy drive */
ReadSector ( "\\\\.\\d:", 0, 1, buffer ) ;  /* reading from a CD-ROM drive */
WriteSector ( "\\\\.\\c:", 0, 1, &b ) ;  /* writing to a hard disk */
ReadSector ( "\\\\.\\physicaldrive0", 0, 1, &b ) ;  /* reading partition table */
```

Here are a few interesting points that you must note.

(a) If we are to read from the second floppy drive we should replace **A:** with **B:** while calling **ReadSector( )**.

(b) To read from storage devices like hard disk drive or CD-ROM or ZIP drive, etc. use the string with appropriate drive letter. The string can be in the range \\.\C: to \\.\Z:.

(c) To read from the CD-ROM just specify the drive letter of the drive. Note that CD-ROMs follow a different storage organization known as CD File System (CDFS).

(d) The hard disk is often divided into multiple partitions. Details like the place at which each partition begins and ends, the size of each partition, whether it is a bootable partition or not, etc. are stored in a table on the disk. This table is often called 'Partition Table'. If we are to read the partition table contents we can do so by using the string **\\.\physicaldrive0**.

(e) Using **\\.\physicaldrive0** we can also read contents of any other parts of the disk. Here **0** represents the first hard disk in the system. If we are to read from the second hard disk we need to use **1** in place of **0**.

Communication with Keyboard

Like mouse messages there also exist messages for keyboard. These are **WM_KEYDOWN**, **WM_KEYUP** and **WM_CHAR**. Of these, **WM_KEYDOWN** and **WM_KEYUP** are sent to an application (which has the input focus) whenever the key is pressed and released respectively. The additional information in case of these messages is the code of the key being pressed or released. When we tackle **WM_KEYDOWN** or **WM_KEYUP** we need to ourselves check the status of toggle keys like NumLock and CapsLock and shift keys like Ctrl, Alt and Shift. If we wish to avoid all this checking we can tackle the **WM_CHAR** message instead.

What is mentioned above is the normal procedure followed by most Windows applications. However, if we wish to go a step further and deal with the keyboard we need to tackle it differently. For example, suppose we are to perform one of the following jobs:

(a) Once you hit any key CapsLock should become on. Once it becomes on it should remain permanently on.

(b) If we hit a key once it should appear twice on the screen.
(c) If we hit a key A then B should appear on the screen, if we hit a B then C should occur and so on.

Note that all these effects should work on a system-wide basis for all Win32 applications. To be able to achieve these effect we need understand two important mechanisms—'Dynamic Linking' and 'Windows Hooks'. Let us understand these mechanisms one by one.

Dynamic Linking

As we saw in Chapter 16, Windows permits linking of libraries stored in a .DLL file during execution. A .DLL file is a binary file that cannot execute on its own. It contains functions that can be shared between several applications running in memory.

Windows Hooks

As the name suggests, the hook mechanism permits us to intercept and alter the flow of messages in the OS before they reach the application. Since hooks are used to alter the messaging mechanism on a system-wide basis the code for hooking has to be written in a DLL. The hooking mechanism involves writing a hook procedure in a DLL file and registering this procedure with the OS. Since the DLL cannot execute on its own we need a separate program that would load and execute the DLL.

For different messages there are different types of hooks. For example, for keyboard messages there is a keyboard hook, for mouse messages there is mouse hook, etc. You can refer MSDN for nearly a dozen more types of hooks. Here we would restrict our discussion only to the keyboard hook.

Before we proceed to write our own hook procedure let us understand the normal working of the keyboard messages. This is illustrated in Figure 19.7.

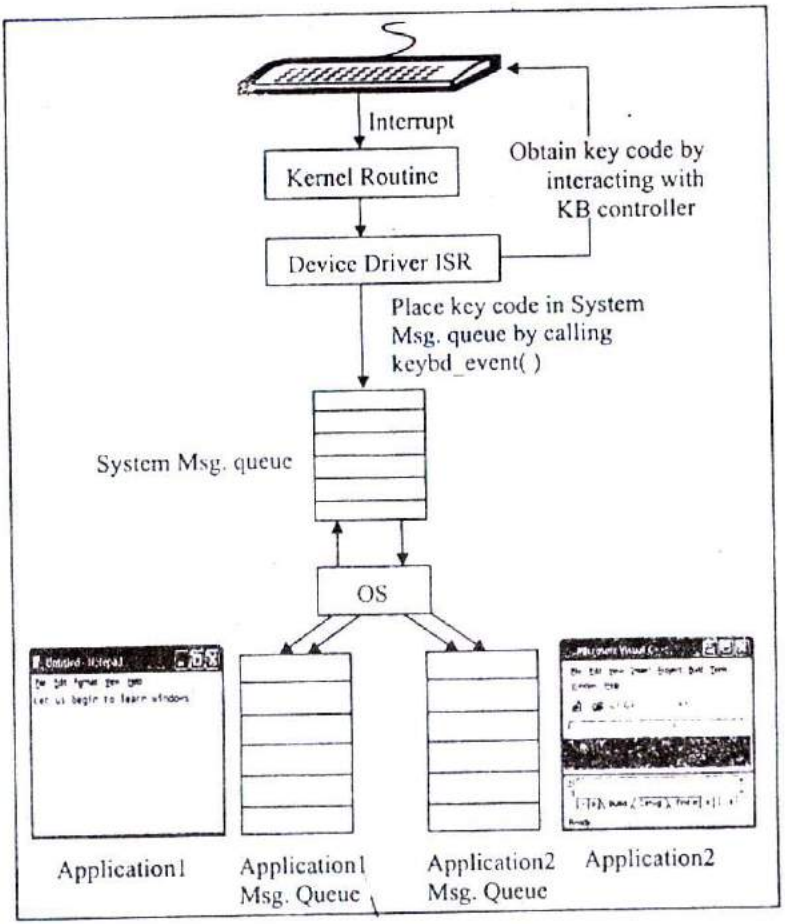

Figure 19.7

With reference to Figure 19.7 here is a list of steps that are carried out when we press a key from the keyboard:

(a) On pressing a key an interrupt occurs and the corresponding kernel routine gets called.
(b) The kernel routine calls the ISR of the keyboard device driver.
(c) The ISR communicates with the keyboard controller and obtains the code of the key pressed.
(d) The ISR calls a OS function **keybd_event( )** to post the key code to the System Message Queue.
(e) The OS retrieves the message from the System Message Queue and posts it into the message queue of the application with regard to which the key has been pressed.

Let us now see what needs to be done if we are to alter this procedure. We simply need to register our hook procedure with the OS. As a result, our hook procedure would receive the message before it is dispatched to the appropriate Application Message Queue. Since our hook procedure gets a first shot at the message it can now alter the working in the following three ways:

(a) It can suppress the message altogether
(b) It can change the message
(c) It can post more messages into the System Message Queue using the **keybd_event( )** function.

Let us now put all this theory into practice by writing a few programs.

aps Locked, Permanently

Let us now write a program that keeps the CapsLock permanently on. This effect would come into being when the first key is hit subsequent to the execution of our program. In fact there would be two programs:

(a) A DLL containing a hook procedure that achieves the CapsLock effect.
(b) An application EXE which loads the DLL in memory.

Given below is the source code of the DLL program.

```
/* hook.c */

# include <windows.h>

static HHOOK hkb = NULL ;
HANDLE h ;

BOOL __stdcall DllMain ( HANDLE hModule, DWORD ul_reason_for_call,
                         LPVOID lpReserved )
{
    h = hModule ;
    return TRUE ;
}

BOOL __declspec ( dllexport ) installhook( )
{
    hkb = SetWindowsHookEx ( WH_KEYBOARD,
                    ( HOOKPROC ) KeyboardProc, ( HINSTANCE ) h, 0 ) ;
    if ( hkb == NULL )
        return FALSE ;

    return TRUE ;
}

LRESULT __declspec ( dllexport ) __stdcall KeyboardProc ( int nCode,
                                  WPARAM wParam, LPARAM lParam )
{
    short int state ;

    if ( nCode < 0 )
        return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;

    if ( ( nCode == HC_ACTION ) &&
                    ( ( DWORD ) lParam & 0x40000000 ) )
    {
        state = GetKeyState ( VK_CAPITAL ) ;
        if ( (state & 1 )== 0)  /* if off */
```

```
        {
            keybd_event ( VK_CAPITAL , 0,
                        KEYEVENTF_EXTENDEDKEY, 0 ) ;
            keybd_event ( VK_CAPITAL , 0,
            KEYEVENTF_EXTENDEDKEY | KEYEVENTF_KEYUP, 0 ) ;
        }
    }
    return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;
}

BOOL __declspec ( dllexport ) removehook( )
{
    return UnhookWindowsHookEx ( hkb ) ;
}
```

Follow the steps mentioned below to create this program:

(a) Select 'File | New' option to start a new project in VC++.
(b) From the 'Project' tab select 'Win32 Dynamic-Link Library' and click on the 'Next' button.
(c) In the 'Win32 Dynamic-link Library Step 1 of 1' select "An empty DLL project" and click on the 'Finish' button.
(d) Select 'File | New' option.
(e) From the 'File' tab select 'C++ source file' and give the file name as 'hook.c'. Type the code listed above in this file.
(f) Compile the program to generate the .DLL file.

Note that this program doesn't contain **WinMain( )** since the program on compilation should not execute on its own. It has been replaced by a function called **DllMain( )**. This function acts as entry point of the DLL program. It gets called when the DLL is loaded or unloaded.

When the application loads the DLL the **DllMain( )** function would be called. In this function we have merely stored the handle to the DLL that has been loaded in memory into a global variable **h** for later use.

Those functions in a DLL that can be called from outside it are called exported functions. Our DLL contains three such functions—**installhook( )**, **removehook( )** and **KeyboardProc( )**. To indicate to the compiler that a function in a DLL is an exported function we have to pre-qualify it with **__declspec (dllexport)**. These functions would be called from the second program. This second program is a normal GUI application created in the same way that we did applications in Chapters 17 and 18. The handlers for messages **WM_CREATE** and **WM_DESTROY** are given below:

```
/* capslocked.c */

HINSTANCE h ;

void OnCreate ( HWND hWnd )
{
    BOOL ( CALLBACK *p )( ) ;

    h = LoadLibrary ( "hook.dll" ) ;
    if ( h != NULL )
    {
        p = GetProcAddress ( h, "installhook" ) ;
        ( *p )( ) ; /* calls installhoook( ) function */
    }
}

void OnDestroy ( HWND hWnd )
{
    BOOL ( CALLBACK *p )( ) ;

    p = GetProcAddress ( h, "removehook" ) ;
    ( *p )( ) ; /* calls removehoook( ) function */

    FreeLibrary ( h ) ;
    PostQuitMessage ( 0 ) ;
}
```

As we know, the **OnCreate( )** and **OnDestroy( )** handlers would be called when the **WM_CREATE** and **WM_DESTROY** messages arrive respectively. In **OnCreate( )** we have loaded the DLL containing the hook procedure. To do this we have called the **LoadLibrary( )** API function. Once the DLL is loaded we have obtained the address of the exported function **installhook( )** using the **GetProcAddress( )** API function. The returned address is stored in **p**, where **p** is a pointer to the **installhook( )** function. Using this pointer we have then called the **installhook( )** function.

In the **installhook( )** function we have called the API function **SetWindowsHookEx( )** to register our hook procedure with the OS as shown below:

```
hkb = SetWindowsHookEx ( WH_KEYBOARD,
                         ( HOOKPROC ) KeyboardProc, ( HINSTANCE ) h, 0 ) ;
```

Here the first parameter is the type of hook that we wish to register, whereas the second parameter is the address of our hook procedure **KeyboardProc( )**. **hkb** stores the handle of the hook installed.

From now on whenever a keyboard message is retrieved by the OS from the System Message Queue the message is firstly passed to our hook procedure, i.e. to **KeyboardProc( )** function. Inside this function we have written code to ensure that the CapsLock always remains on. To begin with we have checked whether **nCode** parameter is less than **0**. If it so then it necessary to call the next hook procedure. The MSDN documentation suggests that "if code is less than zero, the hook procedure must pass the message to the **CallNextHookEx( )** function without further processing and should return the value returned by **CallNextHookEx( )**".

Note that there can be several hook procedures installed by different programs, thus forming a chain of hook procedures. These hook procedures always get called in an order that is

opposite to their order of installation. This means the last hook procedure installed is the first one to get called.

If the **nCode** parameter contains a value **HC_ACTION** it means that the message that was just removed form the system message queue was a keyboard message. If it is so, then we have checked the previous state of the key before the message was sent. If the state of the key was 'depressed' (30^{th} bit of **lParam** is 1) then we have obtained the state of the CapsLock key by calling the **GetKeyState( )** API function. If it is off (0^{th} bit of **state** variable is 0) then we have turned on the CapsLock by simulating a keypress. For this simulation we have called the function **keybd_event( )** twice—first call is for pressing the CapsLock and second is for releasing it. Note that **keybd_event( )** creates a keyboard message from the parameters that we pass to it and posts it into the system message queue. The parameter VK_CAPITAL represents the code for the CapsLock key.

A word of caution! When we use **keybd_event( )** to post keyboard message for a simulated CapsLock keypress, once again our hook procedure would be called when these messages are retrieved from the system message queue. But this time the CapsLock would be on so we would end up passing control to the next hook procedure through a call to **CallNextHookEx( )**.

When we close the application window as usual the **OnDestroy( )** would be called. In this handler we have obtained the address of the **removehook( )** exported function and called it. In the **removehook( )** function we have unregistered our hook procedure by calling the **UnhookWindowsHookEx( )** API function. Note that to this function we have passed the handle to our hook. As a result our hook procedure is now removed from the hook chain. Hereafter the CapsLock would behave normally. Having unhooked our hook procedure the control would return to **OnDestroy( )** handler where we have promptly unload the DLL from memory by calling the **FreeLibrary( )** API function.

One last point about this program—the 'hook.dll' file should be copied into the directory of the application's EXE before executing the EXE.

Did You Press It TTwwiiccee....

With the power of windows hooks below your belt you are into the league of power programmers of Windows. So how about tasting the power some bit more. How about writing a program that would make every key pressed in any Windows application appear twice. Here is the code for the hook procedure.

```
LRESULT __declspec ( dllexport ) __stdcall KeyboardProc ( int nCode,
                                    WPARAM wParam, LPARAM lParam )
{
    static BYTE key ;
    static BOOL flag = FALSE ;

    if ( nCode < 0 )
        return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;

    if ( ( nCode == HC_ACTION ) &&
        ( ( DWORD ) lParam & 0x80000000 ) == 0 )
    {
        if ( flag == FALSE )
        {
            key = wParam ;
            keybd_event ( key , 0, KEYEVENTF_EXTENDEDKEY, 0 ) ;
            flag = TRUE ;
        }
        else
        {
            if ( key == ( BYTE ) wParam )
                flag = FALSE ;
        }
    }
    return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;
```

}

In this hook procedure once again we have checked if the **nCode** parameter contains a value **HC_ACTION**. If it does then we have checked the present state of the key in question. If the present state of the key is 'pressed' (31^{th} bit of **lParam** is 0) then we have posted the message for the same key into the system message queue by calling the **keybd_event( )**. However, this may lead to a serious problem. Can you imagine which? The message that we post, once retrieved, would again bring the control to our hook procedure. Once again the conditions would become true and we would post the same message again. This would go on and on. This can be prevented by using a using a simple **flag** variable as shown in the code.

Note that the rest of the functions in the DLL file are exactly same as in the previous program. So also is the application program.

Mangling Keys

How about one more program to bolster your confidence? Let us try one that would mangle every key that is pressed. That is, convert an A to a B, B to C, C to D, etc. This would be fairly straight-forward. We simply have to increment the key code before posting it into the system message queue. Also, further processing of key has to be prevented. This can be achieved by simply returning a non-zero value from the hook procedure (thus bypassing the call to **CallNextHookEx( )**). This is shown in the following hook procedure.

```
LRESULT __declspec ( dllexport ) __stdcall KeyboardProc ( int nCode,
                                    WPARAM wParam, LPARAM lParam )
{
    static BYTE key ;
    static BOOL flag = FALSE ;
```

```
    if ( nCode < 0 )
        return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;

    if ( ( nCode == HC_ACTION ) &&
       ( ( DWORD ) lParam & 0x80000000 ) == 0 )
    {
        if ( flag == FALSE )
        {
            key = wParam ;
            key ++ ;
            keybd_event ( key , 0, KEYEVENTF_EXTENDEDKEY, 0 ) ;
            flag = TRUE ;
            return 1 ;
        }
        else
        {
            if ( key == ( BYTE ) wParam )
                flag = FALSE ;
        }
    }
    return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;
}
```

KeyLogger

There are several malicious programs that are floating on the net that steal away your passwords. These programs keep a log of every key that is pressed while entering passwords or credit card numbers. These programs make use of windows hooks to trap every key that is pressed. With the knowledge that you have gained from the past three programs this may not be a big deal.

However, such key logger programs deviate from the ones that we developed in three fundamental ways:

(a) They do not pop any window on the screen; otherwise the program's presence would get detected.

(b) These programs also hide themselves from the Task Manager so that the user cannot terminate them.

(c) The logged keys are secretly sent over the net to the malicious users who write such programs. Once the logged keys are known it would be possible to break into the system.

Where is This Leading

Even for a moment do not create an impression in you mind that Windows Hooks are only for notorious activities. There are many good things that they can be put to use for. These activities include:

(a) Multimedia keyboards have special key like Cut, Copy, Paste, etc. Such keyboards also come with special programs which when installed know how to tackle these special keys. On pressing these keys these programs use the hook mechanism to place the simulated keys in the system message queue.

(b) Many demo programs once executed automatically move the mouse pointer to a menu or a toolbar or any such item to demonstrate some feature of the software. To manage these actions a windows hook called Journal hook is used.

(c) For physically impaired persons a keyboard can be simulated on the screen and the mouse clicks on this keyboard can be communicated to Windows as actual key hits. This again can be achieved using mouse and keyboard hook.

There can be many more such examples. But the above three I believe would be ample to prove to you the constructive side of the powerful mechanism called Windows Hooks.

Summary

(a) Hardware interaction can happen in two ways: (1) When the user interacts with the hardware and the program reacts to it. (2) When the program interacts with the hardware without any user intervention.

(b) In DOS when the user interacts with the hardware an ISR gets called which interacts with the hardware. In Windows the same thing is done by the device driver's ISR.

(c) In DOS when the program has to interact with the hardware it can do so by using library functions, DOS/BIOS routines or by directly interacting with the hardware. In Windows the same thing can be done by using API functions.

(d) Under Windows to gain finer control over the hardware we are required to write a device driver program.

(e) Interaction with the any device can be done using API functions like **CreateFile( )**, **ReadFile( )**, **WriteFile( )** and **CloseHandle( )**.

(f) Different strings have to be passed to the **CreateFile( )** functions for interacting with different devices.

(g) Windows provides a powerful mechanism called hooks that can alter the flow of messages before they reach the application.

(h) Windows hook procedures should be written in a DLL since they work on a system wide basis.

(i) Windows hooks can be put to many good uses.

Exercise

[A] State True or False:

(a) In MS-DOS on occurrence of an interrupt values from IDT are used to call the appropriate kernel routine.

(b) Under Windows on occurrence of an interrupt the kernel routine calls the appropriate device driver's ISR.

(c) Under Windows an application can interact with the hardware by directly calling its device driver's routines.

(d) Under Windows we can write device drivers to extend the OS itself.

(e) **ReadSector( )** and **WriteSector( )** are API functions.

(f) While reading a sector from the disk the **CreateFile( )** function creates a file on the disk.

(g) The Windows API function to stop communication with a device is **CloseFile( )**.

(h) The **ReadFile( )** and **WriteFile( )** API functions can only perform reading or writing from/to a disk file.

[B] Answer the following:

(a) How is hardware interaction under Windows different that that under DOS?

(b) What is the advantage of writing code in a DLL?

(c) Explain the Windows hooks mechanism.

(d) What is the standard way of communicating with a device under Windows?

(e) Write a program to read the contents of Boot Sector of a 32-bit FAT file system and print them on the screen. Refer Appendix G for details about the contents of the boot sector.

(f) Write a program that ensures that the key 'A' is completely disabled across all applications.

(g) Write a program that closes any window just by placing the cursor on the 'Close' button in the title bar of it.

20 C Under Linux

- What is Linux
- C Programming Under Linux
- The 'Hello Linux' Program
- Processes
- Parent and Child Processes
- More Processes
- Zombies and Orphans
- One Interesting Fact
- Summary
- Exercise

Today the programming world is divided into two major camps—the Windows world and the Linux world. Since its humble beginning about a decade ago, Linux has steadily drawn the attention of programmers across the globe and has successfully created a community of its own. How big and committed is this community is one of the hottest debates that is raging in all parts of the world. You can look at the hot discussions and the flame wars on this issue on numerous sites on the internet. Before you decide to join the Windows or the Linux camp you should first get familiar with both of them. The last 4 chapters concentrated on Windows programming. This and the next one would deal with Linux programming. Without any further discussions let us now set out on the Linux voyage. I hope you find the journey interesting and exciting.

What is Linux

Linux is a clone of the Unix operating system. Its kernel was written from scratch by Linus Torvalds with assistance from a loosely-knit team of programmers across the world on Internet. It has all the features you would expect in a modern OS. Moreover, unlike Windows or Unix, Linux is available completely free of cost. The kernel of Linux is available in source code form. Anybody is free to change it to suit his requirement, with a precondition that the changed kernel can be distributed only in the source code form. Several programs, frameworks, utilities have been built around the Linux kernel. A common user may not want the headaches of downloading the kernel, going through the complicated compilation process, then downloading the frameworks, programs and utilities. Hence many organizations have come forward to make this job easy. They distribute the precompiled kernel, programs, utilities and frameworks on a common media. Moreover, they also provide installation scripts for easy installations of the Linux OS and applications. Some of the popular distributions are RedHat, SUSE, Caldera, Debian, Mandrake, Slackware, etc. Each of them contain the same kernel

but may contain different application programs, libraries, frameworks, installation scripts, utilities, etc. Which one is better than the other is only a matter of taste.

Linux was first developed for x86-based PCs (386 or higher). These days it also runs on Compaq Alpha AXP, Sun SPARC, Motorola 68000 machines (like Atari ST and Amiga), MIPS, PowerPC, ARM, Intel Itanium, SuperH, etc. Thus Linux works on literally every conceivable microprocessor architecture.

Under Linux one is faced with simply too many choices of Linux distributions, graphical shells and managers, editors, compilers, linkers, debuggers, etc. For simplicity (in my opinion) I have chosen the following combination:

Linux Distribution - Red Hat Linux 9.0
Console Shell - BASH
Graphical Shell - KDE 3.1-10
Editor - KWrite
Compiler - GNU C and C++ compiler (gcc)

We would be using and discussing these in the sections to follow.

C Programming Under Linux

How is C under Linux any different than C under DOS or C under Windows? Well, it is same as well as different. It is same to the extent of using language elements like data types, control instructions and the overall syntax. The usage of standard library functions is also same even though the implementation of each might be different under different OS. For example, a **printf( )** would work under all OSs, but the way it is defined is likely to be different for different OSs. The programmer however doesn't suffer because of this since he can continue to call **printf( )** the same way no matter how it is implemented.

But there the similarity ends. If we are to build programs that utilize the features offered by the OS then things are bound to be different across OSs. For example, if we are to write a C program that would create a Window and display a message "hello" at the point where the user clicks the left mouse button. The architecture of this program would be very closely tied with the OS under which it is being built. This is because the mechanisms for creating a window, reporting a mouse click, handling a mouse click, displaying the message, closing the window, etc. are very closely tied with the OS for which the program is being built. In short the programming architecture (better known as programming model) for each OS is different. Hence naturally the program that achieves the same task under different OS would have to be different.

The 'Hello Linux' Program

As with any new platform we would begin our journey in the Linux world by creating a 'hello world' program. Here is the source code....

```
int main( )
{
    printf ( "Hello Linux\n" ) ;
    return 0 ;
}
```

The program is exactly same as compared to a console program under DOS/Windows. It begins with **main( )** and uses **printf( )** standard library function to produce its output. So what is the difference? The difference is in the way programs are typed, compiled and executed. The steps for typing, compiling and executing the program are discussed below.

The first hurdle to cross is the typing of this program. Though any editor can be used to do so, we have preferred to use the editor called 'KWrite'. This is because it is a very simple yet elegant

editor compared to other editors like 'vi' or 'emacs'. Note that KWrite is a text editor and is a part of K Desktop environment (KDE). Installation of Linux and KDE is discussed in Appendix H. Once KDE is started select the following command from the desktop panel to start KWrite:

K Menu | Accessories | More Accessories | KWrite

If you face any difficulty in starting the KWrite editor please refer Appendix H. Assuming that you have been able to start KWrite successfully, carry out the following steps:

(a) Type the program and save it under the name 'hello.c'.
(b) At the command prompt switch to the directory containing 'hello.c' using the **cd** command.
(c) Now compile the program using the **gcc** compiler as shown below:

```
# gcc hello.c
```

(d) On successful compilation **gcc** produces a file named 'a.out'. This file contains the machine code of the program which can now be executed.
(e) Execute the program using the following command.

```
# ./a.out
```

(f) Now you should be able to see the output 'Hello Linux' on the screen.

Having created a Hello Linux program and gone through the edit-compile-execute cycle once let us now turn our attention to Linux specific programming. We will begin with processes.

Processes

Gone are the days when only one job (task) could be executed in memory at any time. Today the modern OSs like Windows and

Linux permit execution of several tasks simultaneously. Hence these OSs are aptly called 'Multitasking' OSs.

In Linux each running task is known as a 'process'. Even though it may appear that several processes are being executed by the microprocessor simultaneously, in actuality it is not so. What happens is that the microprocessor divides the execution time equally among all the running processes. Thus each process gets the microprocessor's attention in a round robin manner. Once the time-slice allocated for a process expires the operation that it is currently executing is put on hold and the microprocessor now directs its attention to the next process. Thus at any given moment if we take the snapshot of memory only one process is being executed by the microprocessor. The switching of processes happens so fast that we get a false impression that the processor is executing several processes simultaneously.

The scheduling of processes is done by a program called 'Scheduler' which is a vital component of the Linux OS. This scheduler program is fairly complex. Before switching over to the next thread it stores the information about the current process. This includes current values of CPU registers, contents of System Stack and Application Stack, etc. When this process again gets the time slot these values are restored. This process of shifting over from one thread to another is often called a Context Switch. Note that Linux uses preemptive scheduling, meaning thereby that the context switch is performed as soon as the time slot allocated to the process is over, no matter whether the process has completed its job or not.

Kernel assigns each process running in memory a unique ID to distinguish it from other running processes. This ID is often known as processes ID or simply PID. It is very simple to print the PID of a running process programmatically. Here is the program that achieves this...

```
int main( )
{
    printf ( "Process ID = %d", getpid( ) ) ;
}
```

Here **getpid( )** is a library function which returns the process ID of the calling process. When the execution of the program comes to an end the process stands terminated. Every time we run the program a new process is created. Hence the kernel assigns a new ID to the process each time. This can be verified by executing the program several times—each time it would produce a different output.

Parent and Child Processes

As we know, our running program is a process. From this process we can create another process. There is a parent-child relationship between the two processes. The way to achieve this is by using a library function called **fork( )**. This function splits the running process into two processes, the existing one is known as parent and the new process is known as child. Here is a program that demonstrates this...

```
# include <sys/types.h>
int main( )
{
    printf ( "Before Forking\n" ) ;
    fork( ) ;
    printf ( "After Forking\n" ) ;
}
```

Here is the output of the program...

```
Before Forking
After Forking
After Forking
```

Watch the output of the program. You can notice that all the statements after the **fork( )** are executed twice—once by the parent process and second time by the child process. In other words **fork( )** has managed to split our process into two.

But why on earth would we like to do this? At times we want our program to perform two jobs simultaneously. Since these jobs may be inter-related we may not want to create two different programs to perform them. Let me give you an example. Suppose we want perform two jobs—copy contents of source file to target file and display an animated GIF file indicating that the file copy is in progress. The GIF file should continue to play till file copy is taking place. Once the copying is over the playing of the GIF file should be stopped. Since both these jobs are inter-related they cannot be performed in two different programs. Also, they cannot be performed one after another. Both jobs should be performed simultaneously.

At such times we would want to use **fork( )** to create a child process and then write the program in such a manner that file copy is done by the parent and displaying of animated GIF file is done by the child process. The following program shows how this can be achieved. Note that the issue here is to show how to perform two different but inter-related jobs simultaneously. Hence I have skipped the actual code for file copying and playing the animated GIF file.

```
# include <sys/types.h>

int main( )
{
    int pid ;
    pid = fork( ) ;
    if ( pid == 0 )
    {
        printf ( "In child process\n" ) ;
        /* code to play animated GIF file */
```

```
    }
    else
    {
        printf ( "In parent process\n" ) ;
        /* code to copy file */
    }
}
```

As we know, **fork( )** creates a child process and duplicates the code of the parent process in the child process. There onwards the execution of the **fork( )** function continues in both the processes. Thus the duplication code inside **fork( )** is executed once, whereas the remaining code inside it is executed in both the parent as well as the child process. Hence control would come back from **fork( )** twice, even though it is actually called only once. When control returns from **fork( )** of the parent process it returns the PID of the child process, whereas when control returns from **fork( )** of the child process it always returns a 0. This can be exploited by our program to segregate the code that we want to execute in the parent process from the code that we want to execute in the child process. We have done this in our program using an **if** statement. In the parent process the 'else block' would get executed, whereas in the child process the 'if block' would get executed.

Let us now write one more program. This program would use the **fork( )** call to create a child process. In the child process we would print the PID of child and its parent, whereas in the parent process we would print the PID of the parent and its child. Here is the program...

```
# include <sys/types.h>
int main( )
{
    int pid ;
    pid = fork( ) ;

    if ( pid == 0 )
```

```
    {
        printf ( "Child : Hello I am the child process\n" ) ;
        printf ( "Child : Child's PID: %d\n", getpid( ) ) ;
        printf ( "Child : Parent's PID: %d\n", getppid( ) ) ;
    }
    else
    {
        printf ( "Parent : Hello I am the parent process\n" ) ;
        printf ( "Parent : Parent's PID: %d\n", getpid( ) ) ;
        printf ( "Parent : Child's PID: %d\n", pid ) ;
    }
}
```

Given below is the output of the program:

```
Child : Hello I am the child process
Child : Child's PID: 4706
Child : Parent's PID: 4705
Parent : Hello I am the Parent process
Parent : Parent's PID: 4705
Parent : Child's PID: 4706
```

In addition to **getpid( )** there is another related function that we have used in this program—**getppid( )**. As the name suggests, this function returns the PID of the parent of the calling process.

You can tally the PIDs from the output and convince yourself that you have understood the **fork( )** function well. A lot of things that follow use the **fork( )** function. So make sure that you understand it thoroughly.

Note that even Linux internally uses **fork( )** to create new child processes. Thus there is a inverted tree like structure of all the processes running in memory. The father of all these processes is a process called **init**. If we want to get a list of all the running processes in memory we can do so using the **ps** command as shown below.

```
# ps –A
```

Here the switch –A indicates that we want to list all the running processes.

More Processes

Suppose we want to execute a program on the disk as part of a child process. For this first we should create a child process using **fork( )** and then from within the child process we should call an **exec** function to execute the program on the disk as part of a child process. Note that there is a family of **exec** library functions, each basically does the same job but with a minor variation. For example, **execl( )** function permits us to pass a list of command line arguments to the program to be executed. **execv( )** also does the same job as **execl( )** except that the command line arguments can be passed to it in the form of an array of pointers to strings. There also exist other variations like **execle( )** and **execvp( )**.

Let us now see a program that uses **execl( )** to run a new program in the child process.

```
# include <unistd.h>
int main( )
{
    int pid ;
    pid = fork( ) ;
    if ( pid == 0 )
    {
        execl ( "/bin/ls","-al", "/etc", NULL ) ;
        printf ( "Child: After exec( )\n") ;
    }
    else
        printf ( "Parent process\n" ) ;
}
```

After forking a child process we have called the **execl( )** function. This function accepts variable number of arguments. The first parameter to **execl( )** is the absolute path of the program to be executed. The remaining parameters describe the command line arguments for the program to be executed. The last parameter is an end of argument marker which must always be **NULL**. Thus in our case the we have called upon the **execl( )** function to execute the **ls** program as shown below

```
ls -al /etc
```

As a result, all the contents of the **/etc** directory are listed on the screen. Note that the **printf( )** below the call to **execl( )** function is not executed. This is because the **exec** family functions overwrite the image of the calling process with the code and data of the program that is to be executed. In our case the child process's memory was overwritten by the code and data of the **ls** program. Hence the call to **printf( )** did not materialize.

It would make little sense in calling **execl( )** before **fork( )**. This is because a child would not get created and **execl( )** would simply overwrite the main process itself. As a result, no statement beyond the call to **execl( )** would ever get executed. Hence **fork( )** and **execl( )** usually go hand in hand.

Zombies and Orphans

We know that the **ps –A** command lists all the running processes. But from where does the **ps** program get this information? Well, Linux maintains a table containing information about all the processes. This table is called 'Process Table'. Apart from other information the process table contains an entry of 'exit code' of the process. This integer value indicates the reason why the process was terminated. Even though the process comes to an end its entry would remain in the process table until such time that the parent of the terminated process queries the exit code. This act of querying

deletes the entry of the terminated process from the process table and returns the exit code to the parent that raised the query.

When we fork a new child process and the parent and the child continue to execute there are two possibilities—either the child process ends first or the parent process ends first. Let us discuss both these possibilities.

(a) Child terminates earlier than the parent

In this case till the time parent does not query the exit code of the terminated child the entry of the child process would continue to exist. Such a process in Linux terminology is known as a 'Zombie' process. Zombie means ghost, or in plain simple Hindi a 'Bhoot'. Moral is, a parent process should query the process table immediately after the child process has terminated. This would prevent a zombie.

What if the parent terminates without querying. In such a case the zombie child process is treated as an 'Orphan' process. Immediately, the father of all processes—**init**—adopts the orphaned process. Next, as a responsible parent **init** queries the process table as a result of which the child process entry is eliminated from the process table.

(b) Parent terminates earlier than the child

Since every parent process is launched from the Linux shell, the parent of the parent is the **shell** process. When our parent process terminates, the **shell** queries the process table. Thus a proper cleanup happens for the parent process. However, the child process which is still running is left orphaned. Immediately the **init** process would adopt it and when its execution is over **init** would query the process table to clean up the entry for the child process. Note that in this case the child process does not become a zombie.

Thus, when a zombie or an orphan gets created the OS takes over and ensures that a proper cleanup of the relevant process table

entry happens. However, as a good programming practice our program should get the exit code of the terminated process and thereby ensure a proper cleanup. Note that here cleanup is important (it happens anyway). Why is it important to get the exit code of the terminated process. It is because, it is the exit code that would give indication about whether the job assigned to the process was completed successfully or not. The following program shows how this can be done.

```
# include <unistd.h>
# include <sys/types.h>
int main( )
{
    unsigned int i = 0 ;
    int pid, status ;
    pid = fork( ) ;
    if ( pid == 0 )
    {
        while ( i < 4294967295U )
            i++ ;
        printf ( "The child is now terminating\n" ) ;
    }
    else
    {
        waitpid ( pid, &status, 0 ) ;
        if ( WIFEXITED ( status ) )
            printf ( "Parent: Child terminated normally\n" ) ;
        else
            printf ( "Parent: Child terminated abnormally\n" ) ;
    }
    return 0 ;
}
```

In this program we have applied a big loop in the child process. This loop ensures that the child does not terminate immediately. From within the parent process we have made a call to the **waitpid( )** function. This function makes the parent process wait

till the time the execution of the child process does not come to an end. This ensures that the child process never becomes orphaned. Once the child process, terminates the **waitpid( )** function queries its exit code and returns back to the parent. As a result of querying, the child process does not become a zombie.

The first parameter of **waitpid( )** function is the pid of the child process for which the wait has to be performed. The second parameter is the address of an integer variable which is set up with the exit status code of the child process. The third parameter is used to specify some options to control the behavior of the wait operation. We have not used this parameter and hence we have passed a **0**. Next we have made use of the **WIFEXITED( )** macro to test if the child process exited normally or not. This macro takes the status value as a parameter and returns a non-zero value if the process terminated normally. Using this macro the parent suitably prints a message to report the status (normal/abnormal) termination of its child process.

One Interesting Fact

When we use **fork( )** to create a child process the child process does not contain the entire data and code of the parent process. Then does it mean that the child process contains the data and code below the **fork( )** call. Even this is not so. In actuality the code never gets duplicated. Linux internally manages to intelligently share it. As against this, some data is shared, some is not. Till the time both the processes do not change the value of the variables they keep getting shared. However, if any of the processes (either child or parent) attempt to change the value of a variable it is no longer shared. Instead a new copy of the variable is made for the process that is attempting to change it. This not only ensures data integrity but also saves precious memory.

Summary

(a) Linux is a free OS whose kernel was built by Linus Trovalds and friends.
(b) A Linux distribution consists of the kernel with source code along with a large collection of applications, libraries, scripts, etc.
(c) C programs under Linux can be compiled using the popular **gcc** compiler.
(d) Basic scheduling unit in Linux is a 'Process'. Processes are scheduled by a special program called 'Scheduler'.
(e) **fork( )** library function can be used to create child processes.
(f) **Init** process is the father of all processes.
(g) **execl( )** library function is used to execute another program from within a running program,.
(h) **execl( )** function overwrites the image (code and data) of the calling process.
(i) **execl( )** and **fork( )** usually go hand in hand.
(j) **ps** command can be used to get a list of all processes.
(k) **kill** command can be used to terminate a process.
(l) A 'Zombie' is a child process that has terminated but its parent is running and has not called a function to get the exit code of the child process.
(m) An 'Orphan' is a child process whose parent has terminated.
(n) Orphaned processes are adopted by **init** process automatically.
(o) A parent process can avoid creation of a Zombie and Orphan processes using **waitpid( )** function.

Exercise

[A] State True or False:

(a) We can modify the kernel of Linux OS.
(b) All distributions of Linux contain the same collection of applications, libraries and installation scripts.
(c) Basic scheduling unit in Linux is a file.

(d) **execl( )** library function can be used to create a new child process.
(e) The scheduler process is the father of all processes.
(f) A family of **fork( )** and **exec( )** functions are available, each doing basically the same job but with minor variations.
(g) **fork( )** completely duplicates the code and data of the parent process into the child process.
(h) **fork( )** overwrites the image (code and data) of the calling process.
(i) **fork( )** is called twice but returns once.
(j) Every zombie process is essentially an orphan process.
(k) Every orphan process is essentially an orphan process.

[B] Answer the following:

(a) If a program contains four calls to **fork( )** one after the other how many total processes would get created?

(b) What is the difference between a zombie process and an orphan process?

(c) Write a program that prints the command line arguments that it receives. What would be the output of the program if the command line argument is * ?

(d) What purpose do the functions **getpid( )**, **getppid( )**, **getpppid( )** serve?

(e) Rewrite the program in the section 'Zombies and Orphans' replacing the **while** loop with a call to the **sleep( )** function. Do you observe any change in the output of the program?

(f) How does **waitpid( )** prevent creation of Zombie or Orphan processes?

21 More Linux Programming

- Communication using Signals
- Handling Multiple Signals
- Registering a Common Handler
- Blocking Signals
- Event driven programming
- Where Do You Go From Here
- Summary
- Exercise

Communication is the essence of all progress. This is true in real life as well as in programming. In today's world a program that runs in isolation is of little use. A worthwhile program has to communicate with the outside world in general and with the OS in particular. In Chapters 16 and 17 we saw how a Windows based program communicates with Windows. In this chapter let us explore how this communication happens under Linux.

Communication using Signals

In the last chapter we used **fork( )** and **exec( )** library function to create a child process and to execute a new program respectively. These library functions got the job done by communication with the Linux OS. Thus the direction of communication was from the program to the OS. The reverse communication—from the OS to the program—is achieved using a mechanism called 'Signal'. Let us now write a simple program that would help you experience the signal mechanism.

```
int main( )
{
    while ( 1 )
        printf ( "Pogram Running\n" ) ;
    return 0 ;
}
```

The program is fairly straightforward. All that we have done here is we have used an infinite **while** loop to print the message "Program Running" on the screen. When the program is running we can terminate it by pressing the Ctrl + C. When we press Ctrl + C the keyboard device driver informs the Linux kernel about pressing of this special key combination. The kernel reacts to this by sending a signal to our program. Since we have done nothing to handle this signal the default signal handler gets called. In this

default signal handler there is code to terminate the program. Hence on pressing Ctrl + C the program gets terminated.

But how on earth would the default signal handler get called. Well, it is simple. There are several signals that can be sent to a program. A unique number is associated with each signal. To avoid remembering these numbers, they have been defined as macros like **SIGINT**, **SIGKILL**, **SIGCONT**, etc. in the file 'signal.h'. Every process contains several 'signal ID - function pointer' pairs indicating for which signal which function should be called. If we do not decide to handle a signal then against that signal ID the address of the default signal handler function is present. It is precisely this default signal handler for **SIGINT** that got called when we pressed Ctrl + C when the above program was executed. INT in **SIGINT** stands for interrupt.

Let us know see how can we prevent the termination of our program even after hitting Ctrl + C. This is shown in the following program:

```
# include <signal.h>

void sighandler ( int signum )
{
    printf ( "SIGINT received. Inside sighandler\n" ) ;
}

int main( )
{
    signal ( SIGINT, ( void* ) sighandler ) ;
    while ( 1 )
        printf ( "Program Running\n" ) ;
    return 0 ;
}
```

In this program we have registered a signal handler for the SIGINT signal by using the **signal( )** library function. The first parameter

of this function specifies the ID of the signal that we wish to register. The second parameter is the address of a function that should get called whenever the signal is received by our program. This address has to be typecasted to a **void *** before passing it to the **signal( )** function.

Now when we press Ctrl + C the registered handler, namely, **sighandler( )** would get called. This function would display the message 'SIGINT received. Inside sighandler' and return the control back to **main( )**. Note that unlike the default handler, our handler does not terminate the execution of our program. So only way to terminate it is to kill the running process from a different terminal. For this we need to open a new instance of command prompt (terminal). How to start a new instace of command prompt is discussed in Appendix H. Next do a **ps –a** to obtain the list of processes running at all the command prompts that we have launched. Note down the process id of **a.out**. Finally kill 'a.out' process by saying

```
# kill 3276
```

In my case the terminal on which I executed **a.out** was **tty1** and its process id turned out to be **3276**. In your case the terminal name and the process id might be a different number.

If we wish we can abort the execution of the program in the signal handler itself by using the **exit (0)** beyond the **printf( )**.

Note that signals work asynchronously. That is, when a signal is received no matter what our program is doing, the signal handler would immediately get called. Once the execution of the signal handler is over the execution of the program is resumed from the point where it left off when the signal was received.

Handling Multiple Signals

Now that we know how to handle one signal, let us try to handle multiple signals. Here is the program to do this...

```
# include <unistd.h>
# include <sys/types.h>
# include <signal.h>

void inthandler ( int signum )
{
    printf ( "\nSIGINT Received\n" ) ;
}

void termhandler ( int signum )
{
    printf ( "\nSIGTERM Received\n" ) ;
}

void conthandler ( int signum )
{
    printf ( "\nSIGCONT Received\n" ) ;
}

int main( )
{
    signal ( SIGINT, inthandler ) ;
    signal ( SIGTERM, termhandler ) ;
    signal ( SIGCONT, conthandler ) ;

    while ( 1 )
        printf ( "\rProgram Running" ) ;

    return 0 ;
}
```

In this program apart from **SIGINT** we have additionally registered two new signals, namely, **SIGTERM** and **SIGCONT**. The **signal( )** function is called thrice to register a different handler for each of the three signals. After registering the signals we enter a infinite **while** loop to print the 'Program running' message on the screen.

As in the previous program, here too, when we press Ctrl + C the handler for the SIGINT i.e. **inthandler( )** is called. However, when we try to kill the program from the second terminal using the **kill** command the program does not terminate. This is because when the **kill** command is used it sends the running program a **SIGTERM** signal. The default handler for the message terminates the program. Since we have handled this signal ourselves, the handler for **SIGTERM** i.e. **termhandler( )** gets called. As a result the **printf( )** statement in the **termhandler( )** function gets executed and the message 'SIGTERM Received' gets displayed on the screen. Once the execution of **termhandler( )** function is over the program resumes its execution and continues to print 'Program Running'. Then how are we supposed to terminate the program? Simple. Use the following command from the another terminal:

```
kill -SIGKILL 3276
```

As the command indicates, we are trying to send a **SIGKILL** signal to our program. A **SIGKILL** signal terminates the program.

Most signals may be caught by the process, but there are a few signals that the process cannot catch, and they cause the process to terminate. Such signals are often known as un-catchable signals. The **SIGKILL** signal is an un-catchable signal that forcibly terminates the execution of a process.

Note that even if a process attempts to handle the **SIGKILL** signal by registering a handler for it still the control would always land in the default **SIGKILL** handler which would terminate the program.

The **SIGKILL** signal is to be used as a last resort to terminate a program that gets out of control. One such process that makes uses of this signal is a system shutdown process. It first sends a **SIGTERM** signal to all processes, waits for a while, thus giving a 'grace period' to all the running processes. However, after the grace period is over it forcibly terminates all the remaining processes using the **SIGKILL** signal.

That leaves only one question—when does a process receive the **SIGCONT** signal? Let me try to answer this question.

A process under Linux can be suspended using the Ctrl + Z command. The process is stopped but is not terminated, i.e. it is suspended. This gives rise to the un-catchable **SIGSTOP** signal. To resume the execution of the suspended process we can make use of the **fg** (foreground) command. As a result of which the suspended program resumes its execution and receives the **SIGCONT** signal (CONT means continue execution).

Registering a Common Handler

Instead of registering a separate handler for each signal we may decide to handle all signals using a common signal handler. This is shown in the following program:

```
# include <unistd.h>
# include <sys/types.h>
# include <signal.h>

void sighandler ( int signum )
{
    switch ( signum )
    {
        case SIGINT :
```

```
            printf ( "SIGINT Received\n" ) ;
            break ;

    case SIGTERM :
            printf ( "SIGTERM Received\n" ) ;
            break ;

    case SIGCONT :
            printf ( "SIGCONT Received\n" ) ;
            break ;
    }
}

int main( )
{
    signal ( SIGINT, sighandler ) ;
    signal ( SIGTERM, sighandler ) ;
    signal ( SIGCONT, sighandler ) ;

    while ( 1 )
        printf ( "\rProgram running" ) ;

    return 0 ;
}
```

In this program during each call to the **signal( )** function we have specified the address of a common signal handler named **sighandler( )**. Thus the same signal handler function would get called when one of the three signals are received. This does not lead to a problem since the **sighandler( )** we can figure out inside the signal ID using the first parameter of the function. In our program we have made use of the **switch-case** construct to print a different message for each of the three signals.

Note that we can easily afford to mix the two methods of registering signals in a program. That is, we can register separate signal handlers for some of the signals and a common handler for

some other signals. Registering a common handler makes sense if we want to react to different signals in exactly the same way.

Blocking Signals

Sometimes we may want that flow of execution of a critical/time-critical portion of the program should not be hampered by the occurrence of one or more signals. In such a case we may decide to block the signal. Once we are through with the critical/time-critical code we can unblock the signals(s). Note that if a signal arrives when it is blocked it is simply queued into a signal queue. When the signals are unblocked the process immediately receives all the pending signals one after another. Thus blocking of signals defers the delivery of signals to a process till the execution of some critical/time-critical code is over. Instead of completely ignoring the signals or letting the signals interrupt the execution, it is preferable to block the signals for the moment and deliver them some time later. Let us now write a program to understand signal blocking. Here is the program...

```
# include <unistd.h>
# include <sys/types.h>
# include <signal.h>
# include <stdio.h>

void sighandler ( int signum )
{
    switch ( signum )
    {
        case SIGTERM :
            printf ( "SIGTERM Received\n" ) ;
            break ;

        case SIGINT :
            printf ( "SIGINT Received\n" ) ;
            break ;
```

```
        case SIGCONT :
            printf ( "SIGCONT Received\n" ) ;
            break ;
    }
}

int main( )
{
    char buffer [ 80 ] = "\0" ;
    sigset_t block ;

    signal ( SIGTERM, sighandler ) ;
    signal ( SIGINT,  sighandler ) ;
    signal ( SIGCONT, sighandler ) ;

    sigemptyset ( &block ) ;
    sigaddset ( &block, SIGTERM ) ;
    sigaddset ( &block, SIGINT ) ;

    sigprocmask ( SIG_BLOCK, &block, NULL ) ;

    while ( strcmp ( buffer,"n" ) != 0 )
    {
        printf ( "Enter a String: " ) ;
        gets ( buffer ) ;
        puts ( buffer ) ;
    }

    sigprocmask ( SIG_UNBLOCK, &block, NULL ) ;

    while ( 1 )
        printf ( "\rProgram Running" ) ;

    return 0 ;
}
```

In this program we have registered a common handler for the **SIGINT, SIGTERM** and **SIGCONT** signals. Next we want to

repeatedly accept strings in a buffer and display them on the screen till the time the user does not enter an 'n' from the keyboard. Additionally, we want that this activity of receiving input should not be interrupted by the **SIGINT** or the **SIGTERM** signals. However, a **SIGCONT** should be permitted. So before we proceed with the loop we must block the **SIGINT** and **SIGTERM** signals. Once we are through with the loop we must unblock these signals. This blocking and unblocking of signals can be achieved using the **sigprocmask( )** library function.

The first parameter of the **sigprocmask( )** function specifies whether we want to block/unblock a set of signals. The next parameter is the address of a structure (**typedefed** as **sigset_t**) that describes a set of signals that we want to block/unblock. The last parameter can be either NULL or the address of **sigset_t** type variable which would be set up with the existing set of signals before blocking/unblocking signals.

There are library functions that help us to populate the **sigset_t** structure. The **sigemptyset( )** empties a **sigset_t** variable so that it does not refer to any signals. The only parameter that this function accepts is the address of the **sigset_t** variable. We have used this function to quickly initialize the **sigset_t** variable block to a known empty state. To block the **SIGINT** and **SIGTERM** we have to add the signals to the empty set of signals. This can be achieved using the **sigaddset( )** library function. The first parameter of **sigaddset( )** is the address of the **sigset_t** variable and the second parameter is the ID of the signal that we wish to add to the existing set of signals.

After the loop we have also used an infinite **while** loop to print the 'Program running' message. This is done so that we can easily check that till the time the loop that receives input is not over the program cannot be terminated using Ctrl + C or **kill** command since the signals are blocked. Once the user enters 'n' from the keyboard the execution comes out of the **while** loop and unblocks

the signals. As a result, pending signals, if any, are immediately delivered to the program. So if we press Ctrl + C or use the **kill** command when the execution of the loop that receives input is not over these signals would be kept pending. Once we are through with the loop the signal handlers would be called.

Event Driven programming

Having understood the mechanism of signal processing let us now see how signaling is used by Linux – based libraries to create event driven GUI programs. As you know, in a GUI program events occur typically when we click on the window, type a character, close the window, repaint the window, etc. We have chosen the GTK library version 2.0 to create the GUI applications. Here, GTK stands for Gimp's Tool Kit. Refer Appendix H for installation of this toolkit. Given below is the first program that uses this toolkit to create a window on the screen.

```
/* mywindow.c */
# include <gtk/gtk.h>

int main ( int  argc, char *argv[ ] )
{
    GtkWidget *p ;

    gtk_init ( &argc, &argv ) ;
    p = gtk_window_new ( GTK_WINDOW_TOPLEVEL ) ;
    gtk_window_set_title ( p , "Sample Window" ) ;
    g_signal_connect ( p, "destroy", gtk_main_quit, NULL ) ;
    gtk_widget_set_size_request ( p, 300, 300 ) ;
    gtk_widget_show ( p ) ;
    gtk_main( ) ;

    return 0 ;
}
```

We need to compile this program as follows:

```
gcc mywindow.c `pkg-config gtk+-2.0 - -cflags - -libs`
```

Here we are compiling the program 'mywindow.c' and then linking it with the necessary libraries from GTK toolkit. Note the quotes that we have used in the command.

Here is the output of the program...

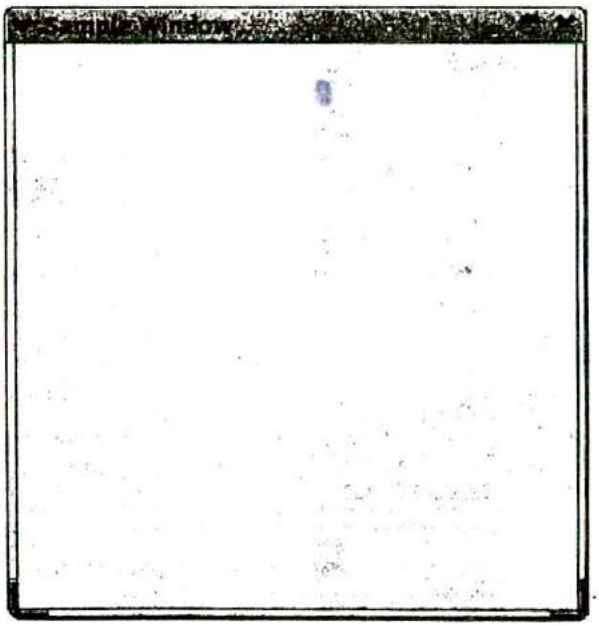

Figure 21.1

The GTK library provides a large number of functions that makes it very easy for us to create GUI programs. Every window under GTK is known as a widget. To create a simple window we have to carry out the following steps:

(a) Initialize the GTK library with a call to **gtk_init( )** function. This function requires the addresses of the command line arguments received in **main( )**.

(b) Next, call the **gtk_window_new( )** function to create a top level window. The only parameter this function takes is the type of windows to be created. A top level window can be created by specifying the **GTK_WINDOW_TOPLEVEL** value. This call creates a window in memory and returns a pointer to the widget object. The widget object is a structure (**GtkWidget**) variable that stores lots of information including the attributes of window it represents. We have collected this pointer in a **GtkWidget** structure pointer called **p**.

(c) Set the title for the window by making a call to **gtk_window_set_title( )** function. The first parameter of this function is a pointer to the **GtkWidget** structure representing the window for which the title has to be set. The second parameter is a string describing the text to be displayed in the title of the window.

(d) Register a signal handler for the destroy signal. The **destroy** signal is received whenever we try to close the window. The handler for the **destroy** signal should perform clean up activities and then shutdown the application. GTK provides a ready-made function called **gtk_main_quit( )** that does this job. We only need to associate this function with the destroy signal. This can be achieved using the **g_signal_connect( )** function. The first parameter of this function is the pointer to the widget for which destroy signal handler has to be registered. The second parameter is a string that specifies the name of the signal. The third parameter is the address of the signal handler routine. We have not used the fourth parameter.

(e) Resize the window to the desired size using the **gtk_widget_set_size_request( )** function. The second and the

third parameters specify the height and the width of the window respectively.

(f) Display the window on the screen using the function **gtk_widget_show( )**.

(g) Wait in a loop to receive events for the window. This can be accomplished using the **gtk_main( )** function.

How about another program that draws a few shapes in the window? Here is the program...

```
/* myshapes.c */
# include <gtk/gtk.h>

int expose_event ( GtkWidget *widget, GdkEventExpose *event )
{
    GdkGC* p ;
    GdkPoint arr [ 5] = { 250, 150, 250, 300, 300, 350, 400, 300, 320, 190 } ;

    p  = gdk_gc_new ( widget -> window ) ;
    gdk_draw_line ( widget -> window, p, 10, 10, 200, 10 ) ;
    gdk_draw_rectangle ( widget -> window, p, TRUE, 10, 20, 200, 100 ) ;
    gdk_draw_arc ( widget -> window, p, TRUE, 200, 10, 200, 200,
                   2880,  -2880*2 ) ;
    gdk_draw_polygon ( widget -> window, p, TRUE , arr, 5 ) ;
    gdk_gc_unref ( p ) ;

    return TRUE ;
}

int main( int  argc, char *argv[ ] )
{
    GtkWidget *p ;

    gtk_init ( &argc, &argv ) ;
```

```
    p = gtk_window_new ( GTK_WINDOW_TOPLEVEL ) ;
    gtk_window_set_title ( p, "Sample Window" ) ;
    g_signal_connect ( p, "destroy", gtk_main_quit, NULL ) ;
    g_signal_connect ( p , "expose_event", expose_event, NULL ) ;
    gtk_widget_set_size_request ( p, 500, 500 ) ;
    gtk_widget_show ( p ) ;
    gtk_main( ) ;

    return 0 ;
}
```

Given below is the output of the program.

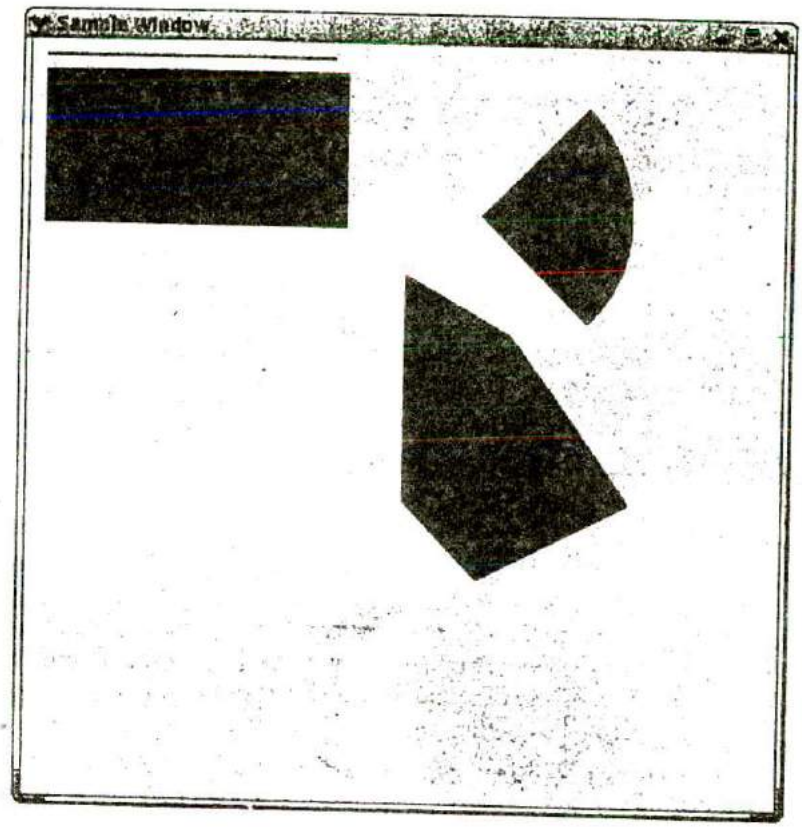

Figure 21.2

This program is similar to the first one. The only difference is that in addition to the destroy signal we have registered a signal handler for the **expose_event** using the **g_signal_connect( )** function. This signal is sent to our process whenever the window needs to be redrawn. By writing the code for drawing shapes in the handler for this signal we are assured that the drawing would never vanish if the windows is dragged outside the screen and then brought back in, or some other window uncovers a portion of our window which was previously overlapped, and so on. This is

because a **expose_event** signal would be sent to our application which would immediately redraw the shapes in our window.

The way in Windows we have a device context, under Linux we have a graphics context. In order to draw in the window we need to obtain a graphics context for the window using the **gdk_gc_new( )** function. This function returns a pointer to the graphics context structure. This pointer must be passed to the drawing functions like **gdk_draw_line( )**, **gdk_draw_rectangle()**, **gdk_draw_arc( )**, **gdk_draw_polygon( )**, etc. Once we are through with drawing we should release the graphics context using the **gdk_gc_unref( )** function.

Where Do You Go From Here

You have now understood signal processing, the heart of programming under Linux. With that knowledge under your belt you are now capable of exploring the vast world of Linux on your own. Complete Linux programming deserves a book on its own. Idea here was to raise the hood and show you what lies underneath it. I am sure that if you have taken a good look at it you can try the rest yourselves. Good luck!

Summary

(a) Programs can communicate with the Linux OS using library functions.
(b) The Linux OS communicates with a program by means of signals.
(c) The interrupt signal (**SIGINT**) is sent by the kernel to our program when we press Ctrl + C.
(d) A term signal (**SIGTERM**) is sent to the program when we use the **kill** command.
(e) A process cannot handle an un-catchable signal.
(f) The **kill –SIGKILL** variation of the **kill** command generates an un-catchable **SIGKILL** signal that terminates a process.

(g) A process can block a signal or a set of signals using the **sigprocmask( )** function.
(h) Blocked signals are delivered to the process when the signals are unblocked.
(i) A **SIGSTOP** signal is generated when we press Ctrl + Z.
(j) A **SIGSTOP** signal is un-catchable signal.
(k) A suspended process can be resumed using the **fg** command.
(l) A process receives the **SIGCONT** signal when it resumes execution.
(m) In GTK, the **g_signal_connect( )** function can be used to connect a function with an event.

Exercise

[A] State True or False:

(a) All signals registered signals must have a separate signal handler.
(b) Blocked signals are ignored by a process.
(c) Only one signal can be blocked at a time.
(d) Blocked signals are ignored once the signals are unblocked.
(e) If our signal handler gets called the default signal handler automatically gets called.
(f) **gtk_main( )** function makes uses of a loop to prevent the termination of the program.
(g) Multiple signals can be registered at a time using a single call to **signal( )** function.
(h) The **sigprocmask( )** function can block as well as unblock signals.

[B] Answer the following:

(a) How does the Linux OS know if we have registered a signal or not?

(b) What happens when we register a handler for a signal?

(c) Write a program to verify that **SIGSTOP** and **SIGKILL** signals are un-catchable signals.

(d) Write a program to handle the **SIGINT** and **SIGTERM** signals. From inside the handler for **SIGINT** signal write an infinite loop to print the message 'Processing Signal'. Run the program and make use of Ctrl + C more than once. Run the program once again and press Ctrl + C once then use the **kill** command. What are your observations?

(e) Write a program that blocks the **SIGTERM** signal during execution of the **SIGINT** signal.