Chapter 1

Windows Programming

Operating Systems History and Overview

Operating System is a software package which tells the computer how to function. It is essentially the body of the computer. Every general-purpose computer requires some type of operating system that tells the computer how to operate and how to utilize other software and hardware that is installed onto the computer.

GUI - Graphical User Interface operating systems are operating systems that have the capability of using a mouse and are graphical. To establish a point of reference, all computers must have an OS. The OS controls input and output; makes reasonable effort to control peripherals; and in short acts as the interface between you the user, the software, and the hardware.

About MS-DOS

Microsoft DOS (Disk Operating System) is a command line user interface. MS-DOS 1.0 was released in 1981 for IBM computers and the latest version of MS-DOS is MS-DOS 6.22 released in 1994. While MS-DOS is not commonly used by itself today, it still can be accessed from Windows 95, Windows 98 or Windows ME by clicking Start / Run and typing **command** or **CMD** in Windows NT, 2000 or XP.

MS-DOS Programs

DOS programs generally expect themselves to be the only program running on the computer, so they will directly manipulate the hardware, such as writing to the disk or displaying graphics on the screen. They may also be dependent on timing, since the computer won't be doing anything else to slow them down. Many games fall into this category.

Features of DOS programming

- It ”owns” the system
- Provides direct device access
- Non-portability across machines
- Status polling
- No multitasking
- No multithreading- Single path of execution
- DOS launches User Application; when done, control returned to DOS
- Assumes the current program is in total control of the hardware
- Supports the File Allocation Table (FAT) file system
- “Real-Mode” OS that is limited to the 8086 Address Space of 1 MB

Windows History

On November 10, 1983, Microsoft announced Microsoft Windows, an extension of the MS-DOS® operating system that would provide a graphical operating environment for PC users. Microsoft called Windows 1.0 a new software environment for developing and running applications that use bitmap displays and mouse pointing devices. With Windows, the graphical user interface (**GUI**) era at Microsoft had begun.

The release of Windows XP in 2001 marked a major milestone in the Windows desktop operating system family, by bringing together the two previously separate lines of Windows desktop operating systems.

Features of Windows Programming

- Resource sharing
- Device independent programming
- Message driven operating system
- GDI (Graphics Device interface)
- Multitasking
- Multithreading

Difference between MS-DOS and Windows programming

In 32-bit windows programming, we are freed from the curse of 64k segments, far and near pointers, 16-bit integers and general limitations.

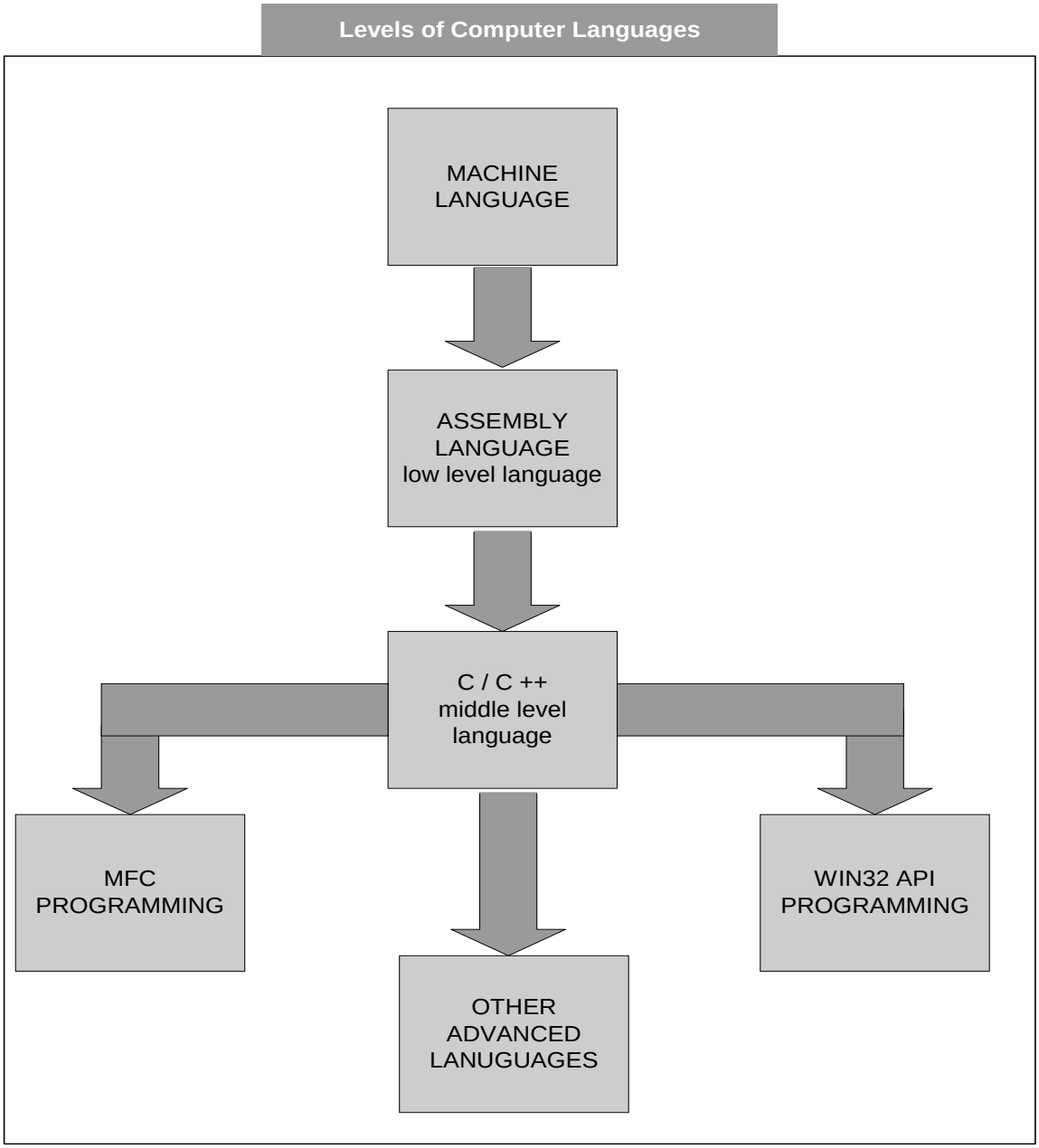
With this power, though, comes responsibility: we no longer have exclusive control over the machine. In fact, we don't have direct access to anything: no interrupts, no video ports, and no direct memory access.

Ultimately, the difference between these two types of programming term is who has control over the system. Moreover, by taking into account the message driven operating system, you would be better able to know what happens behind the scenes and how the system interacts with the internal and external messages.

Levels of Computer Languages

Low level languages are more close to machine language or computer understandable language while the high level language are more close to human understandable language.

Note that from one of the middle level language i.e. C / C++, two programming languages have emerged, MFC programming and Win32 API programming. Both of these programming languages have got their basis from C / C++.



Tips

- During programming, take into account which operating system you are using so that you can make use of all the available resources in the best possible way.
- Windows programs are considered to be more secure and reliable as no direct access to the hardware is available.

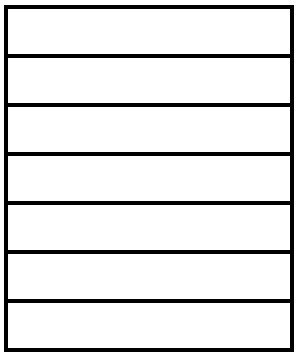
Summary

In this section, we have discussed a brief overview of MS-DOS and Windows operating systems. We have also pointed out the main features of DOS and Windows Programming. Only one DOS program can be executed at a given time and these programs owns the system resources. While in Windows, we can execute several different programs simultaneously. Windows operating system don’t give us the direct access to interrupts, video ports and memory etc.

Chapter 2

Basic C Language Concepts

Random access memory (RAM)



RAM (random access memory) is the place in a computer where the operating system, application programs, and data in current use are kept so that they can be quickly reached by the computer's processor. RAM is much faster to read from and write to than other kinds of storage in a computer, i.e. the hard disk, floppy disk, and CD-ROM. However, the data in RAM stays there only as long as your computer is running. When you turn the computer off, RAM loses its data. When you turn your computer on again, your operating system and other files are once again loaded into RAM, usually from your hard disk.

RAM is the best known form of computer memory. RAM is considered "random access" because you can access any memory cell directly if you know the row and column that intersect at that cell.

Every byte in Ram has an address.

00000000 00000000
00000000 00000001

00000000 00000010
. . .
. . .
. . .
. . .
11111111 11111111

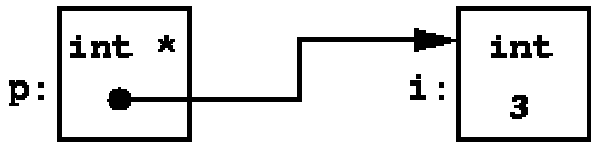
Pointer Definition

Essentially, the computer's memory is made up of bytes. Each byte has a number, an address, associated with it. “Pointer is a kind of variable whose value is a memory address, typically of another variable”. Think of it as an address holder, or directions to get to a variable.

How to assign a value to the pointer?

```
int *p;
int i = 3;
p = &i;
```

- read & as "address of"



In this piece of code, we have taken a pointer to integer denoted by "*p". In second statement, an integer is declared and initialized by '3'. The next step is the most important one. Here we are passing the "Address" of the integer "i" in the pointer. Since pointers hold the variable addresses; so now the pointer "p" contains the address of the integer "i" which has a value of 3.

Pointers Arithmetic

- Pointer Arithmetic deals with performing addition and subtraction operations on pointer variables.
- increment a pointer (++)
- decrement a pointer (--)
- Address in pointer is incremented or decremented by the size of the object it points to (char = 1 byte, int = 2 bytes, ...)

Example: pointers increment and decrement

```
char x = 'A';   // variable declaration and initialization
int  y = 32;

char *xPtr = &x;
int *yPtr = &y; // pointer declaration and initialization

...
xPtr--;      //Since char takes 1 byte, and if xPtr has
                    // a value of 108 now it would have a value of
                    // address 107

xPtr++;     // pointer would have a value of address 108
```

Here, we have explained that if we add 1 to a pointer to integer, then that pointer will point to an address two bytes ahead of its current location. Similarly, when we incremented the xPtr, the address it contained is incremented to one value since xPrt is a pointer to integer.

Note:

Value added or subtracted from pointer is first multiplied by size of object it points to

Example: pointers addition and subtraction

```
...
yPtr-=3;     // Since int takes 2 byte, and assume that yPtr was
                    //pointing to address of 109, now it points to address of
                    // 103 as 109 - (3 * 2) = 103


yPtr+=1;     // now yPtr points to address of 105
```

This means that in the above statement when we will add 1 to the yPtr, where yPtr is a pointer to integer, then the pointer will skip two bytes once and will point to an address of 105 instead of 103.

Example: pointers comparison (>, <, ==)

```
...
if (xPtr == zPtr)

   cout << "Pointers point to the same location";

else

   cout << "Pointers point to different locations";
```

Arrays as Pointers

An array name is actually a pointer to the first element of the array. For example, the following is legal.

int b[100]; // b is an array of 100 integers.

int* p; // p is a pointer to an int**.**

p = b; // Assigns address of first element of b to p.

p = &b[0]; // Exactly the same assignment as above.

In this piece of code, we have used the name of an array as pointer to first address of array. In fact the name of the array is a constant pointer i.e. b is a constant pointer whereas p is a variable pointer. We can change the contents of variable pointer but not or constant pointer. In the last statement, we are assigning the address of b, i.e. the constant pointer to the variable pointer i.e. p.

Array name is a *const* pointer

As we have already discussed above that when you declare an array, the name is a pointer. You cannot alter the value of this pointer. In the previous example, you could never make this assignment.

```
b = p;   // ILLEGAL because b is a constant pointer.
```

A pointer can point to element of an array

```
float x[15];
float *y = &x[0];
float *z = x;
```

- y is a pointer to x[0]
- z is also a pointer to x[0]
- y+1 is pointer to x[1]
- thus *(y+1) and x[1] access the same object
- y[1] is same as *(y+1)
- integer add, subtract and relational operators are allowed on pointers

Example: Pointer De-referencing

```
int *ptr;
int j = 10;

ptr = &j;
printf ("%d\n", *ptr);

*ptr = 15;
printf ("%d %d\n", *ptr, j);

if (ptr != 0)
{ printf ("Pointer ptr points at %d\n", *ptr);
}
```

- *ptr de-references pointer to access object pointed at
- *ptr can be used on either side of assignment operator
- if ptr is equal to 0, then pointer is pointing at nothing and is called a null pointer
- dereferencing a null pointer causes a core dump

Example: Pointer arithmetic.

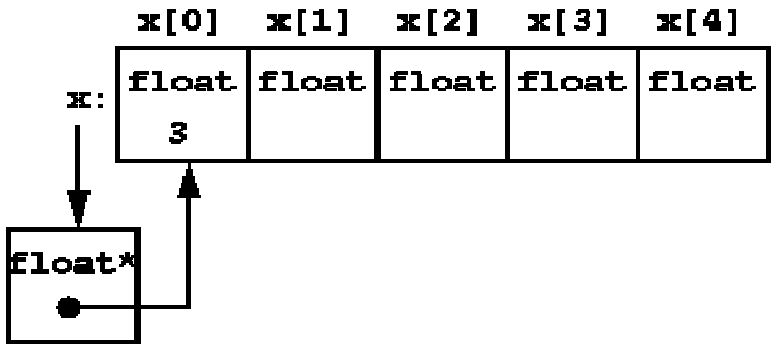
```
double d;
double *ptr_d;
char c;
```

```
char *ptr_c;
ptr_d = &d;
ptr_c = &c;
//This operation will skips 8 bytes in memory because ptr_d is a pointer to double.
ptr_d = ptr_d + 1;
//This operation will skips 1 byte in memory because ptr_c is a pointer to character.
ptr_c = ptr_c +1;
```

Example:

```
float x[5];
```

Our memory model is



- x is a pointer to the first element
- *x and x[0] are the same
- x and &x[0] are the same
- elements of an array can be accessed either way
- x is an array object, not a pointer object

Pointer Advantages

- This allows a function to "return" more than a single value (we are not really returning more than a single variable, but we are able to directly modify the values that are in main, from within a function).
- This allows us to refer to larger data structures with just a single pointer. This cuts back on creating multiple copies of a structure, which consumes both memory and time.
- This also opens the door to dynamic memory allocation.

Pointer Disadvantages

- The syntax may be confusing initially.
- Harder to debug, have to follow pointers to make sure they are doing what is expected.

- More Segmentation faults / Bus errors

What's wrong here?

```
float x[15];
float* y, z;
y = x;    /* Right */
z = x;    /* Wrong */
```

- Y is a pointer to float so it can contain the starting address of array x
- z is a float and not a float pointer

Tips

- Use pointers when you want efficient results.
- To develop plug-ins of existing software uses pointers as much as you can.
- Take extreme care while manipulating arrays with pointers.
- Many bugs in large programs arise due to pointers so only use pointers when necessary.
- Make sure to initialize pointers with some valid value.
- Don’t try to modify the contents of constant pointers.
- Be sure that the data types of pointer variable and the pointed variable are same.
- Do not assign system area addresses to pointers

Summary

In this lecture we started our discussion by revising our basic concepts like RAM. Then we have discussed pointers, how pointers are initialized, what is meant by Pointer Arithmetic. Pointers are very important and useful as with the help of them we can access a very large data structure, similarly other advantages and a few disadvantages of pointers have also been discussed.

Chapter 3

Arrays and Pointers

Arrays

An array is a collection of elements of same type. An array stores many values in memory using only one name. "Array" in programming means approximately the same thing as array, matrix, or vector does in math. Unlike math, you must declare the array and allocate a fixed amount of memory for it. Subscripts are enclosed in square brackets []. Arrays are essentially sequential areas of memory (i.e. a group of memory addresses). However, we do not keep track of the whole array at once. This is because we only have a limited size of data to work with.

Subscripts start at zero

Subscript ranges always start at zero.

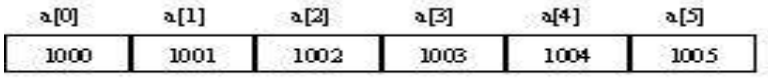
float x[100];

- first element of array is x[0]
- last element of array is x[99]

Array variables as parameters

When an array is passed as a parameter, only the memory address of the array is passed (not all the values). An array as a parameter is declared similarly to an array as a variable, but no bounds are specified. The function doesn't know how much space is allocated for an array.

One important point to remember is that array indexes start from 0. Let’s say our array is of 10 integers, its first element will be a[0] while the last one will be a[9]. Other languages like Fortran carry out 1-based indexing. Due to this 0 based indexing for arrays in C language, programmers prefer to start loops from 0.



Arrays can also be multi-dimensional. In C language, arrays are stored in row major order that a row is stored at the end of the previous row. Because of this storage methodology, if we want to access the first element of the second row then we have to jump as many numbers as the number of columns in the first row. This fact becomes important when we are passing arrays to functions. In the receiving function parameters, we have to write all the dimensions of the array except the extreme-left one. When passing arrays to

functions, it is always call by reference by default; it is not call by value as in the default behavior of ordinary variables.

Operator Precedence

C contains many operators, and because of operator precedence, the interactions between multiple operators can become confusing. Operator precedence describes the order in which C evaluates expressions

For example, ( ) operator has higher precedence then [ ] operator.

The following table shows the precedence of operators in C. Where a statement involves the use of several operators, those with the lowest number in the table will be applied first.

Initializing array elements

```
float x[3] = {1.1, 2.2, 3.3};
float y[] = {1.1, 2.2, 3.3, 4.4};
```

Initializing an array can be taken place in many ways. In the first line of code, we are declaring and initializing an array having three elements in it. In the second array, we are initializing and declaring an array of 4 elements. Note that we have not specified the size of the array on the left side of assignment operator. In this case, the compiler will itself calculate the dimension or the size of the array after counting the initializer given in parenthesis, and will declare an array of corresponding size.

Multi-dimensional arrays

The number of dimensions an array may have is almost endless. To add more dimensions to an array simply add more subscripts to the array declaration. The example below will show how this can be done.

int provinces [50];
// This will declare an one dimensional array of 50 provinces.

int provinces[50][500];
// This will declare a two dimensional array of 50 provinces each including 500 cities.

int provinces[50][500][1000];
// This will declare a three dimensional array.

When using this n-dimensional array, each item would be having a unique position in memory. For example to access the person in the second province, third city, eighth home and the first person in the home the syntax would be:

provinces [2][3][8][1] = variable;

The size of the array would be very large. You can calculate the amount of memory required by multiplying each subscript together, and then multiplying by the size of each element. The size for this array would be 50 x 500 x 1000 x 4 x 2 bytes = 200,000,000 bytes of memory, or 190.73 megabytes, which would be unacceptable on today's computers.

Array of C-strings

An array of C-strings is an array of arrays. Here is an example.

char* days[] = {"Mon", "Tue", "Wed", "Thu", "Fri"};

In the above days array each element is a pointer to a string, and they don't have to be of the same size. For example,

char * greetings[] = {"Hello", "Goodbye", "See you later"};

	Description	Represented By
1	Parenthesis	() []
1	Structure Access	. ->
2	Unary	! ~ ++ -- - * &
3	Mutiply, Divide, Modulus	* / %
4	Add, Subtract	+ -
5	Shift Right, Left	>> <<
6	Greater, Less Than, etc	> < =
7	Equal, Not Equal	== !=
8	Bitwise AND	&
9	Bitwise Exclusive OR	^
10	Bitwise OR	|
11	Logical AND	&&
12	Logical OR	||
13	Conditional Expression	?:
14	Assignment	= += -= etc
15	Comma	,

Function Pointers

Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else

than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.

int (*f1)(void); // *Pointer* to function f1 returning int

Define a Function Pointer

As a function pointer is nothing else than a variable, it must be defined as usual. In the following example we define two function pointers named ptr2Function. It points to a function, which take one float and two char and return an int.

```
// define a function pointer

int (*pt2Function)      (float, char, char);
```

Tips

- Arrays should be used carefully since there is no bound checking done by the compiler.
- Arrays are always passed by “reference” to some function
- The name of the array itself is a constant pointer
- Use function pointer only where it is required.

Summary

The importance and use of Arrays should be very clear till now. Arrays are basically a data structure that is used to so store homogenous or same type of data in it. A very useful thing which we have analyzed here is that when we pass the name of the array as an argument to some function, then only the memory address of array is passed as parameters to the function and not all the values of an array are passed. In the end we have mentioned what the function pointers are? That is such pointers which points to the address of the function.

Chapter 4

Structures and Unions

User Defined or Custom Data types

In addition to the simple data types (int, char, double, ...) there are composite data types which combine more than one data element. The Custom data types include:

1. Structures
2. Unions
3. Enumerations
4. Typedefs

Arrays are used to store many data elements of the same type. An element is accessed by subscript, eg, a[i]. Similarly, structures (also called records) group elements which don't need to all be the same type. They are accessed using the "." operator, eg, r.name.

1. Structures

“A structure is a collection of variables under a single name. These variables can be of different types, and each has a name that is used to select it from the structure”

Let's take an example:

```
struct Person {

  char name[20];
  float height;

  int age;

};
```

This defines a new type, Person. The order of the fields is generally not important. Don't forget the semicolon after the right brace. The convention is to capitalize the first letter in any new type name.

Declaring struct variables

The new struct type can now be used to declare variables. For example,
Person abc;
Structures are syntactically defined with the word struct. So struct is another keyword that cannot be used as variable name followed by the name of the structure. The data, contained in the structure, is defined in the curly braces. All the variables that we have been using can be part of structure. For example:

```
struct Person{
        char name[20];
```

```
        float height;
        int age;
};
```

Here we have declared a structure, ‘person’ containing different elements. The name member element of this structure is declared as char array. For the address, we have declared an array of hundred characters. To store the height, we defined it as float variable type. The variables which are part of structure are called data members i.e. name, height and age are data members of person. Now this is a new data type which can be written as:

```
        person p1, p2;
```

Here p1 and p2 are variables of type person l language and their extensibility. Moreover, it means that we can create new data types depending upon the requirements.

Structures may also be defined at the time of declaration in the following manner:

```
struct person{
        char name[20];
        float height
int age;
}p1, p2;
```

We can give the variable names after the closing curly brace of structure declaration. These variables are in a comma-separated list.

Structures can also contain pointers which also fall under the category of data type. So we can have a pointer to something as a part of a structure. We can’t have the same structure within itself but can have other structures. Let’s say we have a structure of an address. It contains street address like 34 Muslim Town, city like Sukhar, Rawalpindi, etc and country like Pakistan. It can be written in C language as:

```
        struct address{
                char streetAddress[100];
                char city[50];
                char country[50];
        }
```

Now the structure address can be a part of person structure. We can rewrite person structure as under:

```
struct person{
        char name[20];
        address personAdd;
        float height;
```

```
int age;
};
```

Here personAdd is a variable of type Address and a part of person structure. So we can have pointers and other structures in a structure. We can also have pointers to a structure in a structure. We know that pointer hold the memory address of the variable. If we have a pointer to an array, it will contain the memory address of the first element of the array. Similarly, the pointer to the structure points to the starting point where the data of the structure is stored.
The pointers to structure can be defined in the following manner i.e.

```
person *pPtr;
```

Here pptr is a pointer to a data type of structure person. Briefly speaking, we have defined a new data type. Using structures we can declare:

- Simple variables of new structure
- Pointers to structure
- Arrays of structure

Initializing Structures

We have so far learnt how to define a structure and declare its variables. Let's see how we can put the values in its data members. The following example can help us understand the phenomenon further.

```
struct person{
        char name[64];
        int age;
        float height;
};

person p1, p2, p3;
```

Once the structure is defined, the variables of that structure type can be declared. Initialization may take place at the time of declaration i.e.

```
person p1 = {"Ali", 19, 5.5 };
```

In the above statement, we have declared a variable p1 of data type person structure and initialize its data member. The values of data members of p1 are comma separated in curly braces. "Ali" will be assigned to name, 19 to age and 5.5 to height. So far we have not touched these data members directly.
To access the data members of structure, dot operator (.) is used. Therefore while manipulating name of p1, we will say p1.name. This is a way of referring to a data member of a structure. This may be written as:

```
p1.age = 20;
```

p1.height = 6.2;

Similarly, to get the output of data members on the screen, we use dot operator. To display the name of p1 we can write it as:

cout << “The name of p1 = “ << p1.name;

Other data members can be displayed on the screen in the same fashion.

Remember the difference between the access mechanism of structure while using the simple variable and pointer.

- While accessing through a simple variable, use dot operator i.e. p1.name
- While accessing through the pointer to structure, use arrow operator i.e. pPtr->name;

Arrays of structures

Let’s discuss the arrays of structure. The declaration is similar as used to deal with the simple variables. The declaration of array of hundred students is as follows:

students[100];

In the above statement, s is an array of type student structure. The size of the array is hundred and the index will be from 0 to 99. If we have to access the name of first student, the first element of the array will be as under:

s[0].name;

Here s is the array so the index belongs to s. Therefore the first student is s[0], the 2nd student is s[1] and so on. To access the data members of the structure, the dot operator is used. Remember that the array index is used with the array name and not with the data member of the structure.

Accessing the fields of a structure

The fields of a structure are accessed by using the "." operator followed by the name of the field.

abc.name = “Name”;
abc.height = 5.5;
abc.age = 23;

Operations on structures

A struct variable can be assigned to/from, passed as a parameter, returned by function, used as an element in an array. You may not compare structs, but must compare individual fields. The arithmetic operators also don't work with structs. And the I/O operators >> and << do not work for structs; you must read/write the fields individually.

2. Unions

A **union** is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

Using a Union

A C++ union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public.

A C union type can contain only data members.

In C, you must use the union keyword to declare a union variable. In C++, the union keyword is unnecessary:

```
union DATATYPE var2;   // C declaration of a union variable
DATATYPE var3;         // C++ declaration of a union variable
```

A variable of a union type can hold one value of any type declared in the union. Use the member-selection operator (**.**) to access a member of a union:

```
var1.i = 6;          // Use variable as integer
var2.d = 5.327;      // Use variable as double
```

You can declare and initialize a union in the same statement by assigning an expression enclosed in curly braces. The expression is evaluated and assigned to the first field of the union.

Example

// using_a_union.cpp

```
#include <stdio.h>

union NumericType
{
```

```
    int        iValue;
    long       lValue;
    double     dValue;
  };

  int main()
  {
    union NumericType Values = { 10 };   // iValue = 10
    printf("%d\n", Values.iValue);
    Values.dValue = 3.1416;
    printf("%f\n", Values.dValue);
    return 0;
  }
```

Output

```
10
3.141600
```

3. Enumeration

C++ uses the enum statement to assign sequential integer values to names and provide a type name for declaration.

```
enum TrafficLightColor {RED, YELLOW, GREEN};
. . .
int y;
TrafficLightColor x;
. . .
y = 1;
x = YELLOW;
```

The enum declaration creates a new integer type. By convention the first letter of an enum type should be in uppercase. The list of values follows, where the first name is assigned zero, the second 1, etc.

Advantages and Disadvantages of Enumerations

Some advantages of enumerations are that the numeric values are automatically assigned, that a debugger may be able to display the symbolic values when enumeration variables are examined, and that they obey block scope. (A compiler may also generate nonfatal warnings when enumerations and integers are indiscriminately mixed, since doing so can still be considered bad style even though it is not strictly illegal.) A disadvantage is that the programmer has little control over those nonfatal warnings; some programmers also resent not having control over the sizes of enumeration variables.

4. Typedef

Typedef is creating a synonym "new_name" for "data_type". Its syntax is:

typedef data_type new_name;

Advantages of typedef

- Long chain of keyword in declarations can be shortened.
- Actual definition of the data type can be changed.

What's the difference between these two declarations?

```
struct x1 { ... };
typedef struct { ... } x2;
```

The first form declares a "structure tag"; the second declares a "typedef". The main difference is that the second declaration is of a slightly more abstract type -- its users don't necessarily know that it is a structure, and the keyword struct is

Tips

- Use structures when you have to deal with heterogeneous data types like different attributes of an object.
- Take extreme care in accessing the members of a structure.
- Keep in your mind that just declaring the structure does not occupy any space in memory unless and until you define a variable of type struct.
- While using unions, do remember that at one time only one member is contained in it.
- By default the values assigned to enumeration values starts at zero, but if required, we can start assigning integer values from any integer.
- The use of typedefs makes the code simpler by introducing short names for the long data type names.

Summary

The custom or user defined data types are those data types which we create by our own selves according to the requirements or the given situation. The most commonly used custom data types include structures, unions etc. Unlike arrays, structures can store data members of different data types. Unions are also a very important custom data type that at a given time contains only one element from its member list. Enum declarations create a new integer type. The integer type is chosen to represent the values of an enumeration type. Thus, a variable declared as enum is an int. Similarly, typedefs are also used for making a synonym or provides way to shorten the long chain of keywords in declarations.

Chapter 5
Preprocessor Directives

Preprocessor

The preprocessor is a program that runs prior to compilation and potentially modifies a source code file. It may add code in response to the #include directive, conditionally include code in response to #if, #ifdef, #ifndef directives or define constants using the #define directive.
As defined by the ANSI standard, the C preprocessor contains the following directives:

#if #ifdef #ifndef #else #elif #include #define #undef #line #error #pragma

Preprocessor directives: #ifdef and #ifndef

The #ifdef (if defined) and #ifndef (if not defined) preprocessor commands are used to test if a preprocessor variable has been "defined".

Prevent multiple definitions in header files

When there are definitions in a header file that can not be made twice, the code below should be used. A header file may be included twice because more than one other “include file” includes it, or an included file includes it and the source file includes it again.

To prevent bad effects from a double include, it is common to surround the body in the include file with the following:

```
#ifndef MYHEADERFILE_H
#define MYHEADERFILE_H
. . .      // This will be seen by the compiler only once
#endif /* MYHEADERFILE_H */
```

Turning debugging code off and on

Debugging code is necessary in programs; however, it is not usually appropriate to leave it in the delivered code. The preprocessor #ifdef command can surround the debugging code. If DEBUG is defined as below (probably in an include file) all debugging statement surrounded by the #ifdef DEBUG statement will be active. However, if it isn't defined, none of the statements will make it through the preprocessor.

```
#define DEBUG
. . .
#ifdef DEBUG
  . . . // debugging output
#endif
```

Some Preprocessor directives

- **#define**

#define defines an identifier (the macro name) and a string (the macro substitution) which will be substituted for the identifier each time the identifier is encountered in the source file. Once a macro name has been defined, it may be used as part of the definition of other macro names.

If the string is longer than one line, it may be continued by placing a backslash on the end of the first line. By convention, C programmers use uppercase for defined identifiers.

Example: macro #defines

```
#define TRUE 1
#define FALSE 0
```

The macro name may have arguments, in which case every time the macro name is encountered; the arguments associated with it are replaced by the actual arguments found in the program, as in:

```
#define ABS(a) (a)<0 ? -(a) : (a)
...
printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));
```

Such macro substitutions in place of real functions increase the speed of the code at the price of increased program size.

- **#error**

#error forces the compiler to stop compilation. It is used primarily for debugging. The general form is:

```
#error  error_message
```

When the directive is encountered, the error message is displayed, possibly along with other information (depending on the compiler).

- **#include**

#include instructs the compiler to read another source file, which must be included between double quotes or angle brackets. Examples are:

```
#include "stdio.h"
#include <stdio.h>
```

Both of these directives instruct the compiler to read and compile the named header file. If a file name is enclosed in angle brackets, the file is searched for as specified by the creator of the compiler. If the name is enclosed in double quotes, the file is searched for

in an implementation-defined manner, which generally means searching the current directory. (If the file is not found, the search is repeated as if the name had been enclosed in angle brackets.)

Conditional Compilation - #if, #else, #elif, and #endif

Several directives control the selective compilation of portions of the program code, viz, #if, #else, #elif, and #endif.

The general form of #if is:

```
#if constant_expression
statement sequence
#endif
```

#else works much like the C keyword else. #elif means "else if" and establishes an if-else-if compilation chain.

Amongst other things, #if provides an alternative method of "commenting out" code. For example, in

```
#if 0
printf("#d",total);
#endif
```

the compiler will ignore printf("#d",total);.

- **#ifdef and #ifndef**

#ifdef means "if defined", and is terminated by an #endif. #ifndef means "if not defined".

- **#undef**

#undef removes a previously defined definition.

- **#line**

line changes the contents of __LINE__ (which contains the line number of the currently compiled code) and __FILE__ (which is a string which contains the name of the source file being compiled), both of which are predefined identifiers in the compiler.

- **#pragma**

The #pragma directive is an implementation-defined directive which allows various instructions to be given to the compiler i.e. it allows a directive to be defined.
The #pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. Three forms of this directive (commonly known as pragmas) are specified by the 1999 C standard. A C compiler is free to attach any meaning it likes to other pragmas.

The # and ## Preprocessor Operators

The # and ## preprocessor operators are used when using a macro #define. The # operator turns the argument it precedes into a quoted string. For example, given:

#define mkstr(s) # s

the preprocessor turns the line

printf(mkstr(I like C);

into

printf("I like C");

The ## operator concatenates two tokens. For example, given**:**

#define concat(a, b) a ## b

int xy=10;
printf("%d",concat(x, y);

the preprocessor turns the last line into:

printf("%d", xy);

Macros

A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. *Object-like* macros resemble data objects when used, *function-like* macros resemble function calls.

You may define any valid identifier as a macro, even if it is a C keyword. The preprocessor does not know anything about keywords. This can be useful if you wish to hide a keyword such as const from an older compiler that does not understand it. However, the preprocessor operator defined can never be defined as a macro, and C++'s named operators cannot be macros when you are compiling C++.

To define a macro that takes arguments, you use the #define command with a list of parameters in parentheses after the name of the macro. The parameters may be any valid C identifiers separated by commas at the top level (that is, commas that aren't within parentheses) and, optionally, by white-space characters. The left parenthesis must follow the macro name immediately, with no space in between.

For example, here's a macro that computes the maximum of two numeric values:

#define min(X, Y) ((X)>(Y) ? (X):(Y))

Standard Predefined Macros

The standard predefined macros are specified by the C and/or C++ language standards, so they are available with all compilers that implement those standards. Older compilers may not provide all of them. Their names all start with double underscores.

- **__FILE__**

This macro expands to the name of the current input file, in the form of a C string constant. This is the path by which the preprocessor opened the file, not the short names specified in #include or as the input file name argument. For example, "/usr/local/include/myheader.h" is a possible expansion of this macro.

- **__LINE__**

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

__FILE__ and __LINE__ are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

An #include directive changes the expansions of __FILE__ and __LINE__ to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the #include directive, the expansions of __FILE__ and __LINE__ revert to the values they had before the #include (but __LINE__ is then incremented by one as processing moves to the line after the #include).

- **__DATE__**

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Feb 12 1996". If the day of the month is less than 10, it is padded with a space on the left.

- **__TIME__**

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like "23:59:01".

- **__STDC__**

In normal operation, this macro expands to the constant 1, to signify that this compiler conforms to ISO Standard C.

- **__STDC_VERSION__**

This macro expands to the C Standard's version number, a long integer constant of the form yyyymmL where yyyy and mm are the year and month of the Standard version. This signifies which version of the C Standard the compiler conforms to.

This macro is not defined if the -traditional option is used, nor when compiling C++ or Objective-C.

- **__STDC_HOSTED__**

This macro is defined, with value 1, if the compiler's target is a *hosted environment*. A hosted environment has the complete facilities of the standard C library available.

- **__cplusplus**

This macro is defined when the C++ compiler is in use. You can use __cplusplus to test whether a header is compiled by a C compiler or a C++ compiler. This macro is similar to __STDC_VERSION__, in that it expands to a version number.

Tips

- Do use the preprocessor directives as much as possible in your program as it makes the program more robust.
- Using the #defined and #ifndef directives help in prevent multiple definitions in header files.
- Conditional compilation with the use of preprocessor directives provides a very easy way for turning the debug code on and off.
- A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.
- Macros can be used for writing clear and easily comprehensible code.
- Do remember that whenever the macro name is used, it is replaced by the contents of the macro.

Summary

The preprocessor is a program that runs prior to compilation and potentially modifies a source code file. It may add code in response to the #include directive, conditionally include code in response to #if, #ifdef, #ifndef directives or define constants using the #define directive.

A simple macro is a kind of abbreviation. It is a name which stands for a fragment of code. Some standard pre-defined Macros include __FILE__, __LINE__, __DATE__, __TIME__ etc.

Chapter 6
Bitwise Operators and Macros

Bitwise Operators

An operator that manipulates individual bits is called a bitwise operator. The most familiar operators are the addition operator (+) etc and these operators work with bytes or

groups of bytes. Occasionally, however, programmers need to manipulate the bits within a byte.

C++ provides operators to work with the individual bits in integers. For this to be useful, we must have some idea of how integers are represented in binary. For example the decimal number 3 is represented as 11 in binary and the decimal number 5 is represented as 101 in binary.

List of bitwise operators

Purpose	Operator example
complement	~i
and	i&j
exclusive or	i^j
inclusive or	i|j
shift left	i<<n
shift right	i>>n

- can be used on any integer type (char, short, int, etc.)
- right shift might not do sign extension
- used for unpacking compressed data

Bitwise AND operator

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

Bitwise OR operator

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

Bitwise OR operator

0 XOR 0 = 0
0 XOR 1 = 1 x XOR 0 = x

1 XOR 0 = 1 x XOR 1 = ~x
1 XOR 1 = 0

Bitwise Left-Shift is useful when to want to MULTIPLY an integer (not floating point numbers) by a power of 2. The operator, like many others, takes 2 operands like this:

a << b

This expression returns the value of a multiplied by 2 to the power of b.

Bitwise Right-Shift does the opposite, and takes away bits on the right. Suppose we had:

a >> b

This expression returns the value of a divided by 2 to the power of b.

Applications of Bitwise operators

Bitwise operators have two main applications. The first is using them to combine several values into a single variable. Suppose you have a series of flag variables which will always have only one of two values: 0 or 1 (this could also be true or false). The smallest unit of memory you can allocate to a variable is a byte, which is eight bits. But why assign each of your flags eight bits, when each one only needs one bit? Using bitwise operators allows you to combine data in this way.

Example -- Convert to binary with bit operators

This program reads integers and prints them in binary, using the shift and "and" operators to extract the relevant bits.

// Print binary representation of integers

```
#include <iostream>
//using namespace std;
void main() {
  int n;
  while (cin >> n) {
     cout << "decimal: " << n << endl;
     // print binary with leading zeros
    cout << "binary : ";
     for (int i=31; i>=0; i--) {
        int bit = ((n >> i) & 1)
        cout << bit;
     }
     cout << endl;
  }//end loop }
```

Problems

Here are some modifications that could be made to this code.

1. It's difficult to read long sequences of digits. It's common to put a space after every 4 digits.
2. Suppress leading zeros. This is done most easily by defining a bool flag, setting it to false at the beginning of each conversion, setting it to true when a non-zero bit is encountered, and printing zeros only when this flag is set to true. Then there's the case of all zeros that requires another test.

Typedef

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. The names you define using **typedef** are NOT new data types. They are synonyms for the data types or combinations of data types they represent.

A **typedef** declaration does not reserve storage. When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

The following statements declare LENGTH as a synonym for **int**, and then use this **typedef** to declare length, width, and height as integral variables.

```
typedef int LENGTH;

LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, you can use **typedef** to define a **struct** type. For example:

```
typedef struct {

        int scruples;

        int drams;

        int grains;

       } WEIGHT;
```

The structure WEIGHT can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

The proposed feature is intended to be a natural application of existing template syntax to the existing **typedef** keyword. Interactions with the rest of the language are limited because **typedef** templates do not create a new type or extend the type system in any way; they only create synonyms for other types.

Macros

A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. *Object-like* macros resemble data objects when used, *function-like* macros resemble function calls.

You may define any valid identifier as a macro, even if it is a C keyword. The preprocessor does not know anything about keywords. This can be useful if you wish to hide a keyword such as const from an older compiler that does not understand it. However, the preprocessor operator defined can never be defined as a macro, and C++'s named operators cannot be macros when you are compiling C++.

Macro Arguments

Function-like macros can take *arguments*, just like true functions. To define a macro that uses arguments, you insert *parameters* between the pair of parentheses in the macro definition that make the macro function-like. The parameters must be valid C identifiers, separated by commas and optionally whitespace.

To invoke a macro that takes arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses, separated by commas. The invocation of the macro need not be restricted to a single logical line--it can cross as many lines in the source file as you wish. The number of arguments you give must match the number of parameters in the macro definition. When the macro is expanded, each use of a parameter in its body is replaced by the tokens of the corresponding argument. (You need not use all of the parameters in the macro body.)

As an example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs, and some uses.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
  x = min(a, b);          ==>  x = ((a) < (b) ? (a) : (b));
  y = min(1, 2);          ==>  y = ((1) < (2) ? (1) : (2));
  z = min(a + 28, *p);    ==>  z = ((a + 28) < (*p) ? (a + 28) : (*p));
```

Typecasting

Typecasting is making a variable of one type, act like another type for one single application. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. For example (char)a will make 'a' function as a char.

Types of Typecasting

There are two types of typecasting:

- Implicit typecasting
- Explicit typecasting (*coercion*)

Implicit typecasting is done by the compiler itself while the explicit typecasting is done by us, the developers.

Implicit type casting (coercion) is further divided in two types

- Promotion
- Demotion

Example:

```
#include <iostream.h>
int main()
{
  cout<<(char)65;

//The (char) is a typecast, telling the computer to
//interpret the 65 as alphabet’s first letter “A”
//character, not as a number.  It is going to give the
//ASCII output of the equivalent of the number 65(It
should //be the letter A).
  return 0;
}
```

One use for typecasting for is when you want to use the ASCII characters. For example, assume that we want to create our own chart of all 256 ASCII characters. To do this, we will need to use to typecast to allow us to print out the integer as its character equivalent.

```
#include <iostream.h>
int main()
{
 for(int x=0; x<256; x++)
 {                    //The ASCII character set is from 0 to 255
  cout<<x<<". "<<(char)x<<" ";
            //Note the use of the int version of x to
            //output a number and the use of (char) to
```

```
            // typecast the x into a character
      //which outputs the ASCII character that
            //corresponds to the current number
 }
 return 0;
}
```

Assertions

An assertion statement specifies a condition at some particular point in your program. An assertion specifies that a program satisfies certain conditions at particular points in its execution. There are three types of assertion:

Preconditions
- Specify conditions at the start of a function.

Post conditions
- Specify conditions at the end of a function.

Invariants
- Specify conditions over a defined region of a program.

An assertion violation indicates a bug in the program. Thus, assertions are an effective means of improving the reliability of programs. In other words, they are a systematic debugging tool.

Assertions and error-checking

It is important to distinguish between program errors and run- time errors:

1. A program error is a bug, and should never occur.
2. A run-time error can validly occur at any time during program execution.

Assertions are not a mechanism for handling run-time errors. For example, an assertion violation caused by the user inadvertently entering a negative number when a positive number is expected is poor program design. Cases like this must be handled by appropriate error-checking and recovery code (such as requesting another input), not by assertions.

Realistically, of course, programs of any reasonable size do have bugs, which appear at run-time. Exactly what conditions are to be checked by assertions and what by run-time error- checking code is a design issue. Assertions are very effective in reusable libraries, for example, since i) the library is small enough for it to be possible to guarantee bug-free operation, and ii) the library routines cannot perform error- handling because they do not know in what environment they will be used. At higher levels of a program, where operation is more complex, run-time error-checking must be designed into the code.

Turning assertions off

By default, ANSI C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the NDEBUG flag to your compiler, either by inserting

```
#define NDEBUG
```

in a header file such as stdhdr.h, or by calling your compiler with the -dNDEBUG option:

```
cc -dNDEBUG ...
```

This should be done only you are confident that your program is operating correctly, and only if program run-time is a pressing concern.

The switch and case keywords

The switch-case statement is a multi-way decision statement. Unlike the multiple decisions statement that can be created using if-else, the switch statement evaluates the conditional expression and tests it against numerous constant values. The branch corresponding to the value that the expression matches is taken during execution.

The value of the expressions in a switch-case statement must be an (integer, char, short, long), etc. Float and double are not **allowed.**

The syntax is :

```
switch( expression )
{
   case constant-expression1: statements1;
   [case constant-expression2:     statements2;]
   [case constant-expression3:     statements3;]
   [default : statements4;]
}
```

The **case** statements and the **default** statement can occur in any order in the **switch** statement. The **default** clause is an optional clause that is matched if none of the constants in the **case** statements can be matched.

Consider the example shown below:

```
switch( Grade )
{
  case 'A' : printf( "Excellent" );
  case 'B' : printf( "Good" );
  case 'C' : printf( "OK" );
  case 'D' : printf( "Mmmmm...." );
  case 'F' : printf( "You must do better than this" );
  default  : printf( "What is your grade anyway?" );
```

```
}
```

Here, if the Grade is 'A' then the output will be

```
Excellent
Good
OK
Mmmmm....
You must do better than this
What is your grade anyway?
```

This is because, in the 'C' **switch** statement, execution continues on into the next case clause if it is not explicitly specified that the execution should exit the **switch** statement. The correct statement would be:

```
switch( Grade )
{

case 'A' : printf( "Excellent" );
           break;

case 'B' : printf( "Good" );
             break;

      case 'C' : printf( "OK" );
             break;

case 'D' : printf( "Mmmmm...." );
             break;

      case 'F' : printf( "You must do better than this" );
             break;

      default  : printf( "What is your grade anyway?" );
           break;
      }
```

Although the **break** in the **default** clause (or in general, after the last clause) is not necessary, it is good programming practice to put it in anyway.

An example where it is better to allow the execution to continue into the next **case** statement:

```
     char Ch;
```

```
    .
    .
    switch( Ch )
    {
                          /* Handle lower-case characters */
       case 'a' :
       case 'b' :
                    .
            .
            .
       case 'z' :
         printf( "%c is a lower-case character.\n", Ch );
        printf( "Its upper-case is %c.\n" toupper(Ch) );
        break;

                          /* Handle upper-case characters */
     case 'A' :
     case 'B' :
            .
            .
            .
     case 'Z' :
         printf( "%c is a upper-case character.\n", Ch );
        printf( "Its lower-case is %c.\n" tolower(Ch) );
        break;

                          /*   Handle   digits   and   special
characters */

       default  :
          printf( "%c is not in the alphabet.\n", Ch );
        break;
    }
    ..
```

Tips

- Take extreme care while using the Bitwise operators as these operates on individual bits.
- Bitwise Left Shift << operator is useful when we to want to multiply an integer by a power of two.
- Bitwise Right Shift >> operator is useful when we to want to divide an integer by a power of two.

- Do remember that when an object is defined using a typedef identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

- To invoke a macro that takes arguments, you write the name of the macro followed by a list of actual arguments in parentheses, separated by commas.

- Any type-casting done by us is considered to be the explicit type casting. Implicit typecasting is always done by the compiler

- Do remember that when we use typecasting, then the data type of one variable is temporarily changed, while the original data type remains the same

- Assertions are an effective means of improving the reliability of programs. They are a systematic debugging tool.

- The value of the expressions in a switch-case statement must be an (integer, char, short, long), etc. Float and double are not allowed.

Summary

In this lecture, we have learned about the three major bitwise operators AND, OR, XOR. Bitwise operators operate on individual bits.

Using “typedefs” provide an easy way to avoid the long names during the declarations and thus make our code simpler. We have also discussed about the typecasting. It is making a variable of one type, act like another type for one single application. The two types of type casting include the implicit type casting and the explicit type casting.

In C, the assertions are implemented with standard assert macro, the argument to assert must be true when the macro is executed, otherwise the program aborts and printouts an error message.

The switch-case statement is a multi-way decision statement. Unlike the multiple decisions statement that can be created using if-else, the switch statement evaluates the conditional expression and tests it against numerous constant values.

Chapter 7

Calling Conventions, storage classes, and Variable Scope

Calling Convention

To call a function, you must often pass parameters to it. There are plenty of ways how this can be done. You either pass the parameters on the calling stack (place where the processor also places the temporary pointer to the code following the call, so it knows where to continue after the call was done), or you pass some of them in registers. Floating point values can also be passed on the stack of the coprocessor.

Calling conventions rule how parameters will be passed (stack only or registers), in which order they will be passed (from left to right, i.e. in the same order as they appear in source code, or the other way around), and which code will clean the stack after use, if necessary. There are a lot of possible combinations:

- **pascal**, the original calling convention for old Pascal programs;
- ***register,*** the current default calling convention in Delphi;
- ***cdecl,*** the standard calling convention for C and C++ code;
- ***stdcall,*** the default cross-language calling convention on 32-bit Windows;
- ***safecall,*** a special case of stdcall, which can be ignored for now.

Difference between __stdcall and __cdecl calling convention

cdecl and __stdcall just tells the compiler whether the called function or the calling function cleans up the stack. In __stdcall calling convention, the called function cleans up the stack when it is about to return. So if it is called in a bunch of different places, all of those calls do not need to extra code to clean up the stack after the function call.

In __cdecl calling convention, it is the caller function that is responsible for cleaning the stack, so every function call must also need to include extra code to clean up the stack after the function call.

Default Calling Convention for C programmes

The __cdecl is the default calling convention for C programs. In this calling convention, the stack is cleaned up by the caller. The __cdecl calling convention creates larger executables than __stdcall, because it requires each function call to include stack cleanup code.

The following list shows the implementation of _cdecl calling convention.

Element	Implementation
Argument-passing order	Right to left
Stack-maintenance responsibility	Calling function pops the arguments from the

	stack
Name-decoration convention	Underscore character (_) is prefixed to names
Case-translation convention	No case translation performed

Default Calling Convention for Windows Programmes

The __stdcall calling convention is used to call Win32 API functions. The callee cleans the stack

Functions that use this calling convention require a function prototype.

return-type **__stdcall** *function-name*[(*argument-list*)]

The following list shows the implementation of this calling convention.

Element	**Implementation**
Argument-passing order	Right to left.
Argument-passing convention	By value, unless a pointer or reference type is passed.
Stack-maintenance responsibility	Called function pops its own arguments from the stack.
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at sign (@) followed by the number of bytes (in decimal) in the argument list. Therefore, the function declared as int func(int a, double b) is decorated as follows: _func@12
Case-translation convention	None

Storage Class Modifiers

C has a concept of '*Storage classes*' which are used to define the scope (visibility) and life time of variables and/or functions.

1. Auto storage class

The default storage class for local variables is “auto storage class”. The auto storage class specifier lets you define a variable with automatic storage; its use and storage is restricted to the current block. The storage class keyword ***auto*** is optional in a data declaration. It is not permitted in a parameter declaration. A variable having the auto storage class specifier must be declared within a block. It cannot be used for file scope declarations.

Because automatic variables require storage only while they are actually being used, defining variables with the auto storage class can decrease the amount of memory required to run a program. However, having many large automatic objects may cause you to run out of stack space.

Example:

```
{
  int Count;
  auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

Declaring variables with the **auto** storage class can also make code easier to maintain, because a change to an **auto** variable in one function never affects another function (unless it is passed as an argument).

Initialization

You can initialize any **auto** variable except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Note: If you use the **goto** statement to jump into the middle of a block, automatic variables within that block are not initialized.

Storage Allocation

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered; storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

2. Register - Storage Class

Register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

- The register storage class specifier indicates to the compiler that a heavily used variable (such as a loop control variable) within a block scope data definition or a parameter declaration should be allocated a register to minimize access time.

- It is equivalent to the auto storage class except that the compiler places the object, if possible, into a machine register for faster access.

- An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.

The following example lines define automatic storage duration objects using the **register** storage class specifier:

```
register int score1 = 0, score2 = 0;
register unsigned char code = 'A';
register int *element = &order[0];
```

Initialization

You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression

representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

- Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block are made available. When the block is exited, the objects are no longer available for use.

- If a **register** object is defined within a function that is recursively invoked, the memory is allocated for the variable at each invocation of the block.

3. Static Storage Class

Static is the default storage class for global variables. An object having the **static** storage class specifier can be defined within a block or at file scope. If the definition occurs within a block, the object has no linkage. If the definition occurs at file scope, the object has internal linkage.

Example:

Two variables below (count and road) both have a static storage class. Static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

```
static int Count;
int Road;
main()
{
        printf("%d\n", Count);
        printf("%d\n", Road);
}
```

'static' can also be defined within a function! If this is done the variable is initialized at run time but is not re initialized when the function is called.

Example:

There is one very important use for 'static'. Consider this bit of code.

```
char * func(void);

main()
{
   char *Text1;
```

```
   Text1 = func();
}

char * func(void)
{
   char Text2[10]="martin";
   return(Text2);
}
```

Now, 'func' returns a pointer to the memory location where 'text2' starts BUT text2 has a storage class of 'auto' and will disappear when we exit the function and could be overwritten but something else. The answer is to specify:

```
static char Text[10]="martin";
```

The storage assigned to 'text2' will remain reserved for the duration of the program.

Initialization

We can initialize any **static** object with a constant expression or an expression that reduces to the address of a previously declared **extern** or static object, possibly modified by a constant expression. If you do not provide an initial value, the object receives the value of zero of the appropriate type.

Storage Allocation

Storage is allocated at compile time for static variables that are initialized. Un initialized static variables are mapped at compile time and initialized to 0 (zero) at load time. This storage is freed when the program finishes running. Beyond this, the language does not define the order of initialization of objects from different files.

Block Scope Usage

Use **static** variables to declare objects that retain their value from one execution of a block to the next execution of that block. The **static** storage class specifier keeps the variable from being reinitialized each time the block, where the variable is defined, runs. For example:

```
static float rate = 10.5;
```

Initialization of a **static** array is performed only once at compile time. The following examples show the initialization of an array of characters and an array of integers:

```
static char message[] = "startup completed";
static int integers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

File Scope Usage

The **static** storage class specifier causes the variable to be visible only in the file where it is declared. Files, therefore, cannot access file scope **static** variables declared in other files.

Restrictions

We cannot declare a **static** function at block scope.

4. Extern Storage Class

Extern defines a global variable that is visible to all object modules. When you use 'extern' the variable cannot be initialized as all it does is to point the variable name at a storage location that has been previously defined.

With extern keyword, we are actually pointing to such a variable that is already been defined in some other file.

```
Source 1                                    Source 2
--------                                    --------


extern int count;                           int count=5;

write()                                     main()
{                                           {
        printf("count is %d\n", count);             write();
}
}
```

Count in 'source 1' will have a value of 5. If source 1 changes the value of count - source 2 will see the new value

The **extern storage class** specifier lets you declare objects and functions that several source files can use. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

An **extern** variable, function definition, or declaration also makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any **extern** declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword **extern** is optional.

If we do not specify a storage class specifier, the function has external linkage.

Initialization

We can initialize any object with the extern storage class specifier at file scope. Similarly, we can also initialize an extern object with an initializer that must either:

- Appear as part of the definition and the initial value must be described by a constant expression. OR
- Reduce to the address of a previously declared object with static storage duration. This object may be modified by adding or subtracting an integral constant expression.

If we do not explicitly initialize an **extern** variable, its initial value is zero of the appropriate type. Initialization of an **extern** object is completed by the time the program starts running.

Storage Allocation

Storage is allocated at compile time for **extern** variables that are initialized. Un-initialized variables are mapped at compile time and initialized to 0 (zero) at load time. This storage is freed when the program finishes running.

Scope, Initialization and Lifetime of Variable

In the following section, we will discuss the scope and lifetime of variables.

Example:

Consider the example below:

```
int main ()
{
  float temp = 1.1;
  int a;
  int b;
  printf ("Value for a and b [int]: ");
  scanf ("%d%d", &a, &b);
```

```
  if ( a < b )
  {
    int temp = a;  /* this "temp" hides the other one */
    printf ("Smallest local ""temp"" = a*2 = %d\n", 2*temp);
  }  /* end of block; local "temp" deleted */

  else
  {
    int temp = b;  /* another "temp" hides the other one */
    printf ("Smallest local ""temp"" = b*3 = %d\n", 3*temp);
  }  /* end of block; other local "temp" deleted */

  printf ("Global ""temp"" used: %f\n", a * b + temp);

  return 0;
}
```

Points to be considered:

- each { } block creates a new scope
- variables declared and initialized in a scope are deleted when execution leaves scope
- note the f-format to print result with global variable

Stack

A **stack** is an abstract data type that permits insertion and deletion at only one end called the **top**. A stack is a collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop.

Note

Description of elements: A stack is defined to hold one type of data element. Only the element indicated by the *top* can be accessed. The elements are related to each other by the order in which they are put on the stack.

Description of operations: Among the standard operations for a stack are:

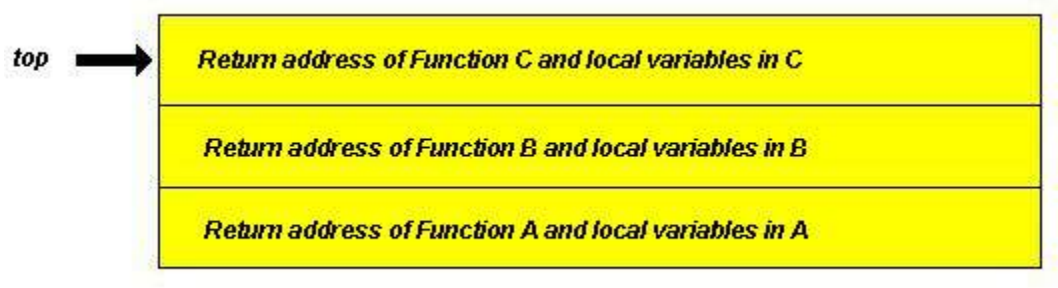
- insert an element on top of the stack (push)
- remove the top element from the stack (pop)
- Determine if the stack is empty.

An example of a stack is the pop-up mechanism that holds trays or plates in a cafeteria. The last plate placed on the stack (insertion) is the first plate off the stack (deletion). A stack is sometimes called a **Last-In, First-Out** or **LIFO** data structure. Stacks have

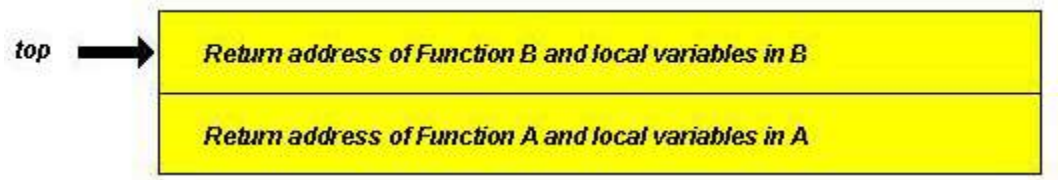
many uses in computing. They are used in solving such diverse problems as "evaluating an expression" to "traversing a maze."

Application of Stacks

A stack data structure is used when subprograms are called. The system must remember where to return after the called subprogram has executed. It must remember the contents of all local variables before control was transferred to the called subprogram. The return from a subprogram is to the instruction following the call that originally transferred control to the subprogram. Therefore, the return address and the local variables of the calling subprogram must be stored in a designated area in memory. For example, suppose function A has control and calls B which calls C which calls D. While D is executing, the return stack might look like this:



The first "return" would return (from D) to the return address in Function C and the return stack would then look like:



The last function called is the first one completed. Function C cannot finish execution until Function D has finished execution. The sequence in which these functions are executed is last-in, first-out. Therefore, a stack is the logical data structure to use for storing return addresses and local variables during subprogram invocations. You can see that the "stack" keeps the return addresses in the exact order necessary to reverse the steps of the forward chain of control as A calls B, B calls C, C calls D.

Const Access Modifier

The const keyword is used to create a read only variable. Once initialized, the value of the variable cannot be changed but can be used just like any other variable.

const int i = 10; // "i " cannot be changed in the program.

Constant Variables

Consider the following examples:

- **Constant pointer to variable data:**

```
 char * const ptr = buff.          // constant pointer to variable data
*ptr = 'a';
ptr = buff2;                       // it will be an error
```

Since we have declared ptr as a "constant pointer to variable data", so we can change the contents of the place where ptr is pointing at, i.e. data but being a constant variable, the ptr value i.e. the address it contains cannot be modified.

- **Variable pointer to Constant data:**

```
const char * ptr = buff.          //variable pointer to constant data
*ptr = 'a';                       // it will be an error
ptr = buf2;
```

Here, ptr has been declared as "variable pointer to constant data". In this case, the data to which the ptr is pointing to remains constant and cannot be modified after initialization. The contents of ptr (address) are variable and we can change the contents of ptr.

Command Line Arguments

C provides a fairly simple mechanism for retrieving command line parameters entered by the user. It passes an argv parameter to the main function in the program.

```
int main(int argc, char *argv[])
    {
    … … …
    }
```

In this code, the main program accepts two parameters, argv and argc. The argv parameter is an array of pointers to string that contains the parameters entered when the program was invoked at the command line. The argc integer contains a count of the number of parameters.

Tips

- Remember the calling conventions used by the functions you are using. It will give you a clearer image of what happens when some parameters are passed to a function.

- Automatic variables require storage only while they are actually being used, so defining variables with the auto storage class can decrease the amount of memory required to run a program.

- Avoid defining many large automatic objects as it may cause you to run out of stack space.

- Register access modifier should only be used for variables that require quick access - such as counters.

- Use static variables to declare objects that retain their value from one execution of a block to the next execution of that block since the static storage class specifier keeps the variable from being reinitialized each time the block runs.

- An extern declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword extern is optional.

- Please note that variables declared and initialized in a scope are deleted when execution leaves scope.

- A stack data structure is used when subprograms are called. It is the logical data structure to use for storing return addresses and local variables during subprogram invocations.

- Do not try to change the value of a constant variable declared with “const” keyword after it has been initialized.

Summary

The calling conventions tells us in which order the parameters will be passed in a function and whether the calling function or the called function is responsible for the cleaning of the stack.

The default calling convention for C programs is __cdecl and in this convention, the caller is responsible for cleaning the stack after the function call.

Similarly, the default calling convention for the windows programs is __stdcall. Here the called function itself has to do the stack clean up and so no extra code is required for stack clean up with each function call. It is very obvious that the __cdecl calling convention creates larger executables because it requires each function call to include the clean up code.

Storage classes are used to describe the scope and visibility of the variables and functions. The common storage classes discussed above are auto, register, static, and extern etc.

In the lifetime of variables, we have discussed that each { } block creates a new scope. Variables declared and initialized in a scope are deleted when execution leaves that scope

The const keyword is used to create a read only variable. Thus constant variables are not allowed to be modified after initialization.

Command line arguments provide an easy way to pass some parameters to the programme in the main function when the programme execution starts. When using an executable that requires startup arguments to debug, you can type these arguments at the command line, or from within the development environment.

Chapter 8
Windows Basics

8.1 Brief History of Win32

Before starting windows programming lets take a short look at the history of Windows.

- **In 1983** Windows announced for the first time in history.
- **In November 1985** Windows 1.0 is launched.
- **In April 1987** Windows 2.0 shipped.
- **In 1988** Windows/386 emerged out. This version of Windows supported Multiple **DOS boxes**. DOS boxes are the Consol windows which are enabled to get input or show output only in text form or character form and no GUI is supported in this mode.
- **November 1989** Win word 1.0 finally shipped.
- **In May 1990** Windows 3.0 shipped. It is designed to operate in 3 Modes. These are

    a. Real Mode or 8086 mode
    b. Protected Mode or 286 mode
    c. Enhanced or 386 mode with multiple DOS boxes and with support of **Virtual Memory**. In virtual Memory some of the area on Hard disk is used as system Memory.

- **Late 1991** Windows version 3.1 released. This version of Windows came with the Multimedia extensions that later became the part of Windows standard builds.
- **Late 1992** Windows NT beta version released. And with this version Win32 API also published. Windows NT offers preemptive **Multitasking.**
  Multitasking is of two types.

    a. Preemptive Multitasking: In Preemptive multi tasking, the operating system parcels out CPU time slices to each program.
    b. Cooperative or Non Preemptive Multitasking: In this type of multitasking each program can control the CPU for as long as it needs it. If a program is not using the CPU it can allow another program to use it temporarily.

- **Summer 1993**, Windows NT version 3.1 is launched. This version of windows is enabled to run as well, on MIPS and Alpha CPU's as Intel x86 CPU.
- **Summer 1994** Windows version 3.5 is launched.
- **In August 1995** Windows95 shipped. Windows95 was designed for home computing.
- **September 1995** Windows version 3.51 released. And it is considered as the most solid version of NT for servers.
- **Summer 1996,** Support for the MIPS and PowerPC machines are dropped.
- **June 1998**, Windows98 released with built in Internet Explorer version 4.

- **September 1998**, Visual Studio 6.0 released. Visual Studio6.0 consists of three languages i.e. Visual C++, Visual Basic and Visual J++. Visual studio comes in three categories, i.e. learner or student edition, professional and Enterprise edition.
  Visual C++ is a compiler that will be using in the Windows programming course. This Visual C++ compiler would be the part of Professional or Enterprise Visual Studio package.

- **Feb 2000** Windows 2000 released with major improvements. It is proved to be a much stable version than the earlier versions of Microsoft Windows series of Operating systems. It is also called Windows NT5.

8.2 Windows Components

Microsoft Windows consists of three important components. These are:

1. Kernel
2. GDI (Graphics Device Interface)
3. User

8.2.1 Kernel

Kernel is a main module of the operating system. This provides system services for managing threads, memory, and resources.
Kernel has to perform very important responsibilities e.g.

1. Process Management
2. File Management
3. Memory Management (System and Virtual Memory)

In Windows Operating System Kernel is implemented in the form of Kernel32.dll file. The Kernel is responsible for scheduling and synchronizing threads, processing, exception and interrupts. Loading applications and managing memory. Kernel is responsible for the system stability and efficiency.

8.2.2 GDI (Graphics Device Interface)

GDI is a subsystem responsible for displaying text and images on display devices and printers. The GDI processes Graphical function calls from a Windows-based application. It then passes those calls to the appropriate device driver, which generates the output on the display hardware. By acting as a buffer between applications and output devices, the GDI presents a device-independent view of the world for the application while interacting in a device-dependent format with the device.

GDI is responsible to display application’s graphics objects on Screen and Printer. The Applications that use GDI need not worry about Graphics Hardware because GDI

provides the suitable device independent interface. In GDI subsystem, anything we want to display or print, there's Device context or DC. Device context or DC is a logical term in windows. Whenever we have to display something, we get DC of display device, and whenever we have to print something we get DC of printer device.
GDI is implemented in the form of library GDI32.dll. This library contains all the APIs that need to draw graphics or text objects. We can write text, and draw rectangles, polygons, lines, points, etc by using Pens and Brushes:

Pens

A pen is a graphics tool that an application for Microsoft Windows uses to draw lines and curves. Drawing applications use pens to draw freehand lines, straight lines, and curves. Computer-aided design (CAD) applications use pens to draw visible lines, hidden lines, section lines, center lines, and so on. Word processing and desktop publishing applications use pens to draw borders and rules. Spreadsheet applications use pens to designate trends in graphs and to outline bar graphs and pie charts.

Each pen consists of three attributes: style, width, and color. While no limits are imposed on the width and color of a pen, the pen's style must be supported by the operating system. These styles are illustrated in the following figure.

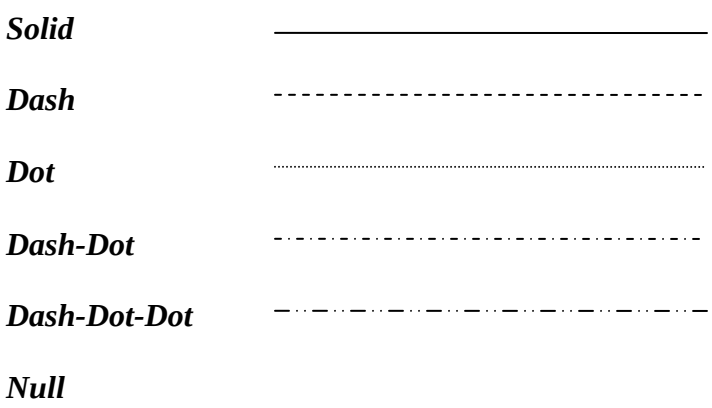


Figure 1 Pens types

Brushes

A brush is a graphics tool that a Windows based application uses to paint the interior of polygons, ellipses, and paths. Drawing applications use brushes to paint shapes; word processing applications use brushes to paint rules; computer-aided design (CAD) applications use brushes to paint the interiors of cross-section views; and spreadsheet applications use brushes to paint the sections of pie charts and the bars in bar graphs.

There are two types of brushes: logical and physical. A logical brush is one that you define in code as the ideal combination of colors and/or pattern that an application should use to paint shapes. A physical brush is one that a device driver creates, which is based on your logical-brush definition.

Brush Origin

When an application calls a drawing function to paint a shape, Windows positions a brush at the start of the paint operation and maps a pixel in the brush bitmap to the window origin of the client area. (The window origin is the upper-left corner of the window's client area.) The coordinates of the pixel, that Windows maps, are called the brush origin.

The default brush origin is located in the upper-left corner of the brush bitmap at the coordinates (0,0). Windows then copies the brush across the client area, forming a pattern that is as tall as the bitmap. The copy operation continues, row by row, until the entire client area is filled. However, the brush pattern is visible only within the boundaries of the specified shape. (Here, the term bitmap is used in its most literal sense—as an arrangement of bits—and does not refer exclusively to bits stored in an image file).

There are instances when the default brush origin should not be used. For example, it may be necessary for an application to use the same brush to paint the backgrounds of its parent and child windows and blend a child window's background with that of the parent window.

The following illustration shows a five-pointed star filled by using an application-defined brush. The illustration shows a zoomed image of the brush, as well as the location to which it was mapped at the beginning of the paint operation.

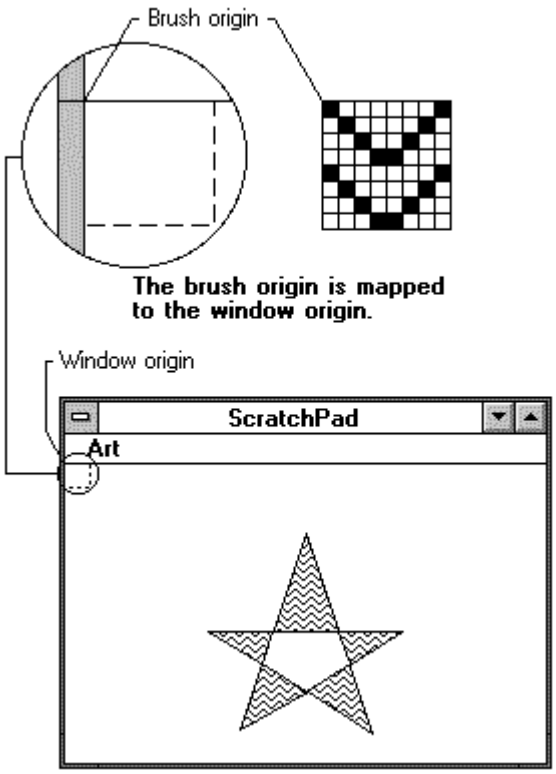


Figure 2 Brush Origin (Description from MSDN)

Logical Brush Types

Logical brushes come in three varieties: solid, pattern, and hatched.

- ✓ A solid brush consists of a color or pattern defined by some element of the Windows user interface (for example, you can paint a shape with the color and pattern conventionally used by Windows to display disabled buttons).
- ✓ A hatched brush consists of a combination of a color and of one of the six patterns defined by Win32. The following table illustrates the appearance of these predefined patterns.

Brush Style	**Illustration**
Backward diagonal	
Cross-hatched	
Diagonally cross-hatched	
Forward diagonal	

Horizontal	═══════════════
Vertical	‖‖‖‖‖‖‖‖‖‖‖‖‖‖‖‖‖‖‖

Figure 3 Brush Styles

8.2.3 User

User component manages all the user interface elements.
User interface elements include Dialogs, Menus, Text, Cursor, Controls, Clipboard, etc.
User component is implemented in User32.dll file. You would be familiar with all the user interface elements but the new thing for you might be Clipboard.

Clipboard

In Windows, data is shareable among applications. For example you are typing in Notepad and after you have typed, you want to copy all the text written in Notepad to another Editor, say, MS Word. How could this be possible? The answer is: through clipboard. Yes, in clipboard, firstly we copy all the data to clipboard and then paste that data to MS Word because clipboard is shareable object. All the text or image data you have previously copied can now be pasted in other application.

Following are some of the features of clipboard.

- User32.dll manages clipboard.
- Clipboard is used to cut copy and paste operations.
- Clipboard is temporary storage area. When you shut down windows, data saved in clipboard will be lost.

8.3 Handles in Windows

A term handle is a 32 bit number in Win32 environment. It is normally a type defined as void *. Handle has an extensive use in Windows. Using handles you can destroy, release and take other actions on the object. The basic types of handles in windows are:

- HANDLE
- HWND
- HINSTANCE

HWND is of type Handle to Window.
HINSTANCE is of type Handle to instance of the application.

Every application has unique identifier in Memory that is called an instance of the application.

8.4 Our first Win32 Program

For writing win32 program you should be familiar of programming concepts in C++.

First we will include header file windows.h in our source file because this header contains prototype of useful APIs that will be used in our windows programs. This header also contains some other headers required for commonly used APIs.

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

Every Windows GUI base program starts its execution from the WinMain API. And Every Windows console base program starts its execution from simple main function.
Here we will be discussing about Windows Graphical User Interface programs.
So we will start our program by writing WinMain.

```
int CALLBACK WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
        MessageBox(NULL, “This is the First Windows Program \n Author: Shahid”, “Virtual Uinversity”, MB_OK|MB_ICONINFORAMTION);
return 0;
}
```

WinMain:
WinMain is the starting point in Every Win32 GUI programs. WinMain has four parameters these are,

1. First is instance of the current application.
2. Second parameter is also an instance of this application which is used for the previous application of the same type that is already running. It is used only in Window 16bit editions or Windows 3.1. Windows 32bit editions do not support this parameter. It is here just for compatibility.
3. Third parameter is a command line argument of string type which is a type defined as char *.
4. Fourth parameter is windows style.

Calling Convention:
CALLBACK is a calling convention which is a type defined as __stdcall.
Return Value:
In our application WinMain returns 0. WinMain always returns a value of type integer.

MessageBox:

MessageBox API is used to display a message on screen. It comes in windows dialog form and has caption and client area, both of these contains strings. It also has buttons, like Ok, Cancel, Yes, No, Retry, Abort. It has four arguments.

1. First is the HANDLE to window, this handle is a parent handle of the messagebox. In our case it has no parent handle so we write NULL here that shows that it has parent desktop.
2. Second parameter is a string type which print string in the body of the message box which is also called client area of the message box
3. Third parameter is also a string type it prints string in a caption of the message box.
4. Fourth parameter is a buttons that will be displayed in the message box. We presented here some of the constants like MB_OK|MB_ICONINFORMATION which display ok button and icon information on the left side of the message box. These constants can be presented here by using bitwise OR operator.

8.5 Summary

In this Lecture, we studied about the windows history about when and how different editions were released. Then we learnt about the windows components: Kernel, GDI and User. Kernel is the heart of Operating system and GDI is a Graphics device interface in windows which is used to display and print graphics and text objects. Then we studied user component which is responsible to control all the dialogs, menu, windows and Windows controls, etc. ‘Handles in windows’ is the introductory part of handles used in windows. Handles are 32 bit number that may be void * type which is defined in ‘WinUser.h’ header file. Finally we wrote our first Win32 program. We always write WinMain in every windows program because WinMain is the starting point of Windows Programs. And we learnt how to display message box in WinMain function and then we returned from WinMain with the returned value Zero. And we discussed about the Message box API and explained its parameters as well.

Tips: Never use second parameter (HINSTANCE) of WinMain because it is not used in recent version of windows. In future lectures, we will discuss how to determine the existence of previous application.

Chapter 9
Windows Creation and Message Handling

9.1 Multiple Instances

Every running application is an Application Instance. So if you open more than one application, more than one instance will be running simultaneously. If you write a program and run it, this running program will be known as a process running in memory. Whenever you press ALT-CONTROL-DELETE, you can open Task Manager to watch all the processes present in task list, running under Windows. Each process can have one or more than one windows. Every process has at least one thread running, which is UI thread.

9.2 Window Class

Every window in *Windows* has its own registered Window class. This window class has set of attributes which are later used by windows. These attributes could be windows background brush, windows style, cursors, Icons, etc. So Windows class tells the Operating system about the characteristics and physical layout of its windows. Window Class is simply a structure named WNDCLASS or WNDCLASSEX that only contains set of attributes for window.

Each window class has an associated window procedure shared by all windows of the same class. The window procedure processes messages for all windows of that class and therefore, controls their behavior and appearance. For more information, see Window Procedures.

A process must register a window class before it creates a window. Registering a window class associates a window procedure, class styles and other class attributes particularly a class name. When a process specifies a class name in the *CreateWindow* or *CreateWindowEx* function, the system creates a window using a registered class name.

A window class defines the attributes of a window such as style, icon, cursor, menu, and window procedure. The first step in registering a window class is to fill a WNDCLASS structure. For more information, see Elements of a Window Class. Next step is to pass the structure to the **RegisterClass** function.

To register an application global class, specify the CS_GLOBALCLASS style in the **style** member of the **WNDCLASSEX** structure. When registering an application local class, do not specify the CS_GLOBALCLASS style.

If you register the window class using the ANSI version of **RegisterClassEx**, **RegisterClassExA**, the application requests that the system pass text parameters of messages to the windows of the created class using the ANSI character set; if you register the class using the Unicode version of **RegisterClassEx**, **RegisterClassExW**, the application requests that the system pass text parameters of messages to the windows of the created class using the Unicode character set. The *IsWindowUnicode* function enables

applications to query the nature of each window. For more information on ANSI and Unicode functions, see Conventions for Function Prototypes in Microsoft help documents.

The executable or DLL that registered the class is the owner of the class. The system determines class ownership from the **hInstance** member of the **WNDCLASSEX** structure passed to the **RegisterClassEx** function when the class is registered. For DLLs, the **hInstance** member must be the handle to the .dll instance.

Windows 2000 or Above: The class is not destroyed when the .dll that owns it is unloaded. Therefore, if the system calls the window procedure for a window of that class, it will cause an access violation, because the .dll containing the window procedure is no longer in memory. The process must destroy all windows using the class before the .dll is unloaded and call the **UnregisterClass** function.

```
ATOM RegisterClass(
      CONST WNDCLASS *lpWndClass
);
The complete description its parameters can be found from
Microsoft Developer Network
```

```
BOOL UnregisterClass(
    LPCTSTR lpClassName,
    HINSTANCE hInstance
);

The complete description its parameters can be found from
Microsoft Developer Network
```

This function inputs a pointer to CONST WNDCLASS structure and returns ATOM. ATOM is a unique identifier that will be returned from RegisterClass. ATOM is unsigned short value.

9.3 Elements of a Window Class

The elements of a window class define the default behavior of windows belonging to the class. The application that registers a window class assigns elements to the class by setting appropriate members in a **WNDCLASSEX** structure and passing the structure to the **RegisterClassEx** function. The *GetClassInfoEx* and *GetClassLong* functions retrieve information about a given window class. The *SetClassLong* function changes elements of a local or global class that the application has already registered.

The Structure of Window Class is as follows.

```
WNDCLASS Structure

typedef struct _WNDCLASS {
LPCTSTR     lpszClassName;
WNDPROC     lpfnWndProc;
UINT        style;
int         cbClsExtra;
int         cbWndExtra;
HINSTANCE   hInstance;
HICON       hIcon;
HCURSOR     hCursor;
HBRUSH      hbrBackground;
LPCTSTR     lpszMenuName;
LPCTSTR     lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Although a complete window class consists of many elements, the system requires the application which supplies a class name, the window-procedure address and an instance handle. Use the other elements to define default attributes for windows of the class, such as the shape of the cursor and the content of the menu for the window. You must initialize any unused members of the **WNDCLASSEX** structure to zero or NULL. The window class elements are as shown in the following table.

Element	Purpose
Class Name	Distinguishes the class from other registered classes.
Window Procedure Address	Pointer to the function that processes all messages sent to windows in the class and defines the behavior of the window.
Instance Handle	Identifies the application or .dll that registered the class.
Class Cursor	Defines the mouse cursor that the system displays for a window of the class.
Class Icons	Defines the large icon and the small icon (Windows NT 4.0 and later).
Class Background Brush.	Defines the color and pattern that fill the client area when the window is opened or painted.
Class Menu	Specifies the default menu for windows that do not explicitly define a menu.
Class Styles	Defines how to update the window after moving or resizing it, how to process double-clicks of the mouse, how to allocate space for the device context, and other aspects of the window.
Extra Class	Specifies the amount of extra memory, in bytes, that the system should

Memory	reserve for the class. All windows in the class share the extra memory and can use it for any application-defined purpose. The system initializes this memory to zero.
Extra Window Memory	Specifies the amount of extra memory, in bytes, that the system should reserve for each window belonging to the class. The extra memory can be used for any application-defined purpose. The system initializes this memory to zero.

9.3.1 Class Name

Every window class needs a *Class Name* to distinguish one class from another. Assign a class name by setting the **lpszClassName** member of the **WNDCLASSEX** structure to the address of a null-terminated string that specifies the name. Because window classes are process specific, window class names need to be unique only within the same process. Also, because class names occupy space in the system's private ATOM table, you should keep class name strings as short a possible.

The *GetClassName* function retrieves the name of the class to which a given window belongs.

9.3.2 Window Procedure Address

Every class needs a window-procedure address to define the entry point of the window procedure used to process all messages for windows in the class. The system passes messages to the procedure when it requires the window to carry out tasks, such as painting its client area or responding to input from the user. A process assigns a window procedure to a class by copying its address to the **lpfnWndProc** member of the **WNDCLASSEX** structure. For more information, see **Window Procedures**.

9.3.3 Instance Handle

Every window class requires an instance handle to identify the application or .dll that registers the class. The system requires instance handles to keep track of all modules. The system assigns a handle to each copy of a running executable or .dll.

The system passes an instance handle to the entry-point function of each executable. The executable or .dll assigns this instance handle to the class by copying it to the **hInstance** member of the **WNDCLASSEX** structure.

9.3.4 Class Cursor

The *class cursor* defines the shape of the cursor when it is in the client area of a window in the class. The system automatically sets the cursor to the given shape when the cursor enters the window's client area and ensures it keeps that shape while it remains in the client area. To assign a cursor shape to a window class, load a predefined cursor shape by using the *LoadCursor* function and then assign the returned cursor handle to the **hCursor**

member of the **WNDCLASSEX** structure. Alternatively, provide a custom cursor resource and use the *LoadCursor* function to load it from the application's resources.

The system does not require a class cursor. If an application sets the **hCursor** member of the **WNDCLASSEX** structure to NULL, no class cursor is defined. The system assumes the window sets the cursor shape each time the cursor moves into the window. A window can set the cursor shape by calling the *SetCursor* function whenever the window receives the WM_MOUSEMOVE message.

9.3.5 Class Icons

A *class icon* is a picture that the system uses to represent a window of a particular class. An application can have two class icons — one large and one small. The system displays a window's *large class icon* in the Task-switch window that appears when the user presses ALT+TAB, and in the large icon views of the task bar and explorer. The *small class icon* appears in a window's title bar and in the small icon views of the task bar and explorer.

To assign a large and small icon to a window class, specify the handles of the icons in the **hIcon** and **hIconSm** members of the **WNDCLASSEX** structure. The icon dimensions must conform to required dimensions for large and small class icons. For a large class icon, you can determine the required dimensions by specifying the SM_CXICON and SM_CYICON values in a call to the *GetSystemMetrics* function. For a small class icon, specify the SM_CXSMICON and SM_CYSMICON values.

If an application sets the **hIcon** and **hIconSm** members of the **WNDCLASSEX** structure to NULL, the system uses the default application icon as the large and small class icons for the window class. If you specify a large class icon but not a small one, the system creates a small class icon based on the large one. However, if you specify a small class icon but not a large one, the system uses the default application icon as the large class icon and the specified icon as the small class icon.

You can override the large or small class icon for a particular window by using the WM_SETICON message. You can retrieve the current large or small class icon by using the WM_GETICON message.

9.3.6 Class Background Brush

A *class background brush* prepares the client area of a window for subsequent drawing by the application. The system uses the brush to fill the client area with a solid color or pattern, thereby removing all previous images from that location whether they belong to the window or not. The system notifies a window that its background should be painted by sending the WM_ERASEBKGND message to the window.

To assign a background brush to a class, create a brush by using the appropriate GDI functions and assign the returned brush handle to the **hbrBackground** member of the **WNDCLASSEX** structure.

Instead of creating a brush, an application can set the **hbrBackground** member to one of the standard system color values. For a list of the standard system color values, see *System Colors* from *Microsoft Documentation.*

To use a standard system color, the application must increase the background-color value by one. For example, COLOR_BACKGROUND + 1 are the system background color. Alternatively, you can use the *GetSysColorBrush* function to retrieve a handle to a brush that corresponds to a standard system color, and then specify the handle in the **hbrBackground** member of the **WNDCLASSEX** structure.

The system does not require that a window class has a class background brush. If this parameter is set to NULL, the window must paint its own background whenever it receives the **WM_ERASEBKGND** message.

9.3.7 Class Menu

A *class menu* defines the default menu to be used by the windows in the class if no explicit menu is given when the windows are created. A menu is a list of commands from which a user can choose actions for the application to carry out.

You can assign a menu to a class by setting the **lpszMenuName** member of the **WNDCLASSEX** structure to the address of a null-terminated string that specifies the resource name of the menu. The menu is assumed to be a resource in the given application. The system automatically loads the menu when it is needed. If the menu resource is identified by an integer and not by a name, the application can set the **lpszMenuName** member to that integer by applying the MAKEINTRESOURCE macro before assigning the value.

The system does not require a class menu. If an application sets the **lpszMenuName** member of the **WNDCLASSEX** structure to NULL, window in the class has no menu bar. Even if no class menu is given, an application can still define a menu bar for a window when it creates the window.

If a menu is given for a class and a child window of that class is created, the menu is ignored.

9.3.8 Class Styles

The class styles define additional elements of the window class. Two or more styles can be combined by using the bitwise OR (|) operator. To assign a style to a window class, assign the style to the **style** member of the **WNDCLASSEX** structure. The class styles are as follows.

Style	Action
CS_BYTEALIGNCLIENT	Aligns the window's client area on a byte boundary (in the x direction). This style affects the width of the window and its horizontal placement on the display.
CS_BYTEALIGNWINDOW	Aligns the window on a byte boundary (in the x direction). This style affects the width of the window and its horizontal placement on the display.
CS_CLASSDC	Allocates one device context to be shared by all windows in the class. Because window classes are process specific, it is possible for multiple threads of an application to create a window of the same class. It is also possible for the threads to attempt to use the device context simultaneously. When this happens, the system allows only one thread to successfully finish its drawing operation.
CS_DBLCLKS	Sends a double-click message to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
CS_DROPSHADOW	**Windows XP:** Enables the drop shadow effect on a window. The effect is turned on and off through SPI_SETDROPSHADOW. Typically, this is enabled for small, short-lived windows such as menus to emphasize their Z order relationship to other windows.
CS_GLOBALCLASS	Specifies that the window class is an application global class. For more information.
CS_HREDRAW	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Disables **Close** on the window menu.
CS_OWNDC	Allocates a unique device context for each window in the class.
CS_PARENTDC	Sets the clipping rectangle of the child window to that of the parent window so that the child can draw on the parent. A window with the CS_PARENTDC style bit receives a regular device context from the system's cache of device contexts. It does not give the child the parent's device context or device context settings. Specifying CS_PARENTDC enhances an application's performance.
CS_SAVEBITS	Saves, as a bitmap, the portion of the screen image obscured by a window of this class. When the window is removed, the system uses the saved bitmap to restore the screen image, including other windows that were obscured. Therefore, the system does not send WM_PAINT messages to windows that were obscured if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image.

	This style is useful for small windows (for example, menus or dialog boxes) that are displayed briefly and then removed before other screen activity takes place. This style increases the time required to display the window, because the system must first allocate memory to store the bitmap.
CS_VREDRAW	Redraws the entire window if a movement or size adjustment changes the height of the client area.

9.4 Using Window Class (*Example*)

```
#include <windows.h>

// Declaration of Global variable
HINSTANCE hinst;
// Function prototypes.
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE
hInstPrev, LPSTR str, int cmd);
InitApplication(HINSTANCE);
InitInstance(HINSTANCE, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM,
LPARAM);

// Application entry point.
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE
hInstPrev, LPSTR str, int cmd)
{
    MSG msg;

    if (!InitApplication(hinstance))
        return FALSE;
return 0;
 }

//Initialize Application by registering class

BOOL InitApplication(HINSTANCE hinstance)
{
    WNDCLASSEX wcx;

    // Fill in the window class structure with
//parameters
    // that describe the main window.
```

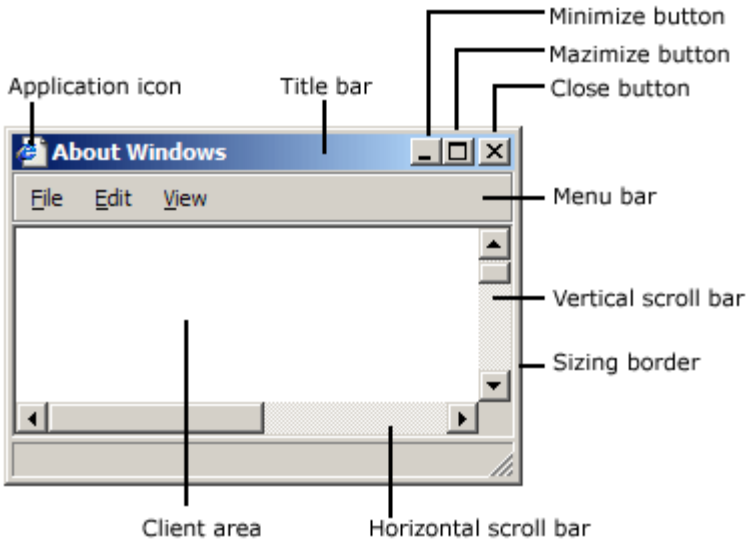
```
    wcx.cbSize = sizeof(wcx);          // size of
structure
    wcx.style = CS_HREDRAW |
        CS_VREDRAW;                    // redraw if
size changes
    wcx.lpfnWndProc = MainWndProc;     // points to
window procedure
    wcx.cbClsExtra = 0;                // no extra
class memory
    wcx.cbWndExtra = 0;                // no extra
window memory
    wcx.hInstance = hinstance;         // handle to
instance
    wcx.hIcon = LoadIcon(NULL,
        IDI_APPLICATION);              // predefined
app. icon
    wcx.hCursor = LoadCursor(NULL,
        IDC_ARROW);                    // predefined
arrow
    wcx.hbrBackground = GetStockObject(
        WHITE_BRUSH);                  // white
background brush
    wcx.lpszMenuName =  "MainMenu";    // name of menu
resource
    wcx.lpszClassName = "MainWClass";  // name of
window class
    wcx.hIconSm = LoadImage(hinstance, // small class
icon
        MAKEINTRESOURCE(5),
        IMAGE_ICON,
        GetSystemMetrics(SM_CXSMICON),
        GetSystemMetrics(SM_CYSMICON),
        LR_DEFAULTCOLOR);

    // Register the window class.

    return RegisterClassEx(&wcx);
}
```

9.5 About Windows

Every graphical Microsoft® Windows®-based application creates at least one window, called the *main window* that serves as the primary interface between the user and the application. Most applications also create other windows, either directly or indirectly, to perform tasks related to the main window. Each window plays a part in displaying output and receiving input from the user.

When you start an application, the system also associates a taskbar button with the application. The *taskbar button* contains the program icon and title. When the application is active, its taskbar button is displayed in the pushed state.

An application window includes elements such as a title bar, a menu bar, the window menu (formerly known as the system menu), the minimize button, the maximize button, the restore button, the close button, a sizing border, a client area, a horizontal scroll bar, and a vertical scroll bar. An application's main window typically includes all of these components. The following illustration shows these components in a typical main window.



9.5.1 Client Area

The *client area* is the part of a window where the application displays output, such as text or graphics. For example, a desktop publishing application displays the current page of a document in the client area. The application must provide a function, called a window procedure, to process input to the window and display output in the client area.

9.5.2 Nonclient Area

The title bars, menu bar, window menu, minimizes and maximize buttons, sizing border, and scroll bars are referred to collectively as the window's *nonclient area*. The system

manages most aspects of the *nonclient area, and* the application manages the appearance and behavior of its *client area*.

The *title bar* displays an application-defined icon and line of text; typically, the text specifies the name of the application or indicates the purpose of the window. An application specifies the icon and text when creating the window. The title bar also makes it possible for the user to move the window by using a mouse or other pointing device.

Most applications include a *menu bar* that lists the commands supported by the application. Items in the menu bar represent the main categories of commands. Clicking an item on the menu bar typically opens a pop-up menu whose items correspond to the tasks within a given category. By clicking a command, the user directs the application to carry out a task.

The *window menu* is created and managed by the system. It contains a standard set of menu items that, when chosen by the users, sets a window's size or position, closes the application, or performs tasks.

The buttons in the upper-right corner affect the size and position of the window. When you click the *maximize button*, the system enlarges the window to the size of the screen and positions the window, so it covers the entire desktop, minus the taskbar. At the same time, the system replaces the maximize button with the restore button. When you click the *restore button*, the system restores the window to its previous size and position. When you click the *minimize button*, the system reduces the window to the size of its taskbar button, positions the window over the taskbar button, and displays the taskbar button in its normal state. To restore the application to its previous size and position, click its taskbar button. When you click the *close button*, application exits.

The *sizing border* is an area around the perimeter of the window that enables the user to size the window by using a mouse or other pointing device.

The *horizontal scroll bar* and *vertical scroll bar* convert mouse or keyboard input into values that an application uses to shift the contents of the client area either horizontally or vertically. For example, a word-processing application that displays a lengthy document typically provides a vertical scroll bar to enable the user to page up and down through the document.

9.6 Prototype of CreateWindow

Here is a prototype of *CreateWindow* Function.

```
HWND CreateWindow(
        LPCTSTR lpClassName;      //class name (identification)
        LPCTSTR lpWindowName;  //Window caption bar Name
        DWORD dwStyle;            // style of the windows
        Int x;                    //starting X point of window on screen
```

```
        Int y;                  //starting Y point of window on screen
        Int width;              //Width of the window from starting point
        Int height;             //height of the window from starting Y point
        HWND hWndParent;        //handle the parent window if any
        HMENU hMenu;            // handle the Menu if any
        HINSTANCE hInstance;    //handle of the instance
        LPVOID lpParam;         //void parameter
);
//Documentation is described below
```

9.6.1 Class Name (*lpClassName*)

[in] Pointer to a null-terminated string or a class atom created by a previous call to the **RegisterClass** or **RegisterClassEx** function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

*If lpClassName is a string, it specifies the window class name. The class name can be any name registered with **RegisterClass** or **RegisterClassEx**, provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined system class names*

9.6.2 Window Name (*lpWindowName*).

[in] Pointer to a null-terminated string that specifies the window name.

*If the window style specifies a title bar, the window title pointed to by lpWindowName is displayed in the title bar. When using **CreateWindow** to create controls, such as buttons, check boxes, and static controls, use lpWindowName to specify the text of the control. When creating a static control with the SS_ICON style, use lpWindowName to specify the icon name or identifier. To specify an identifier, use the syntax "#num".*

9.6.3 Window Styles (*dwStyle*)

[in] Specifies the style of the window being created. This parameter can be a combination of ***Window Styles.***

The following styles can be specified wherever a window style is required.

Style	Meaning
WS_BORDER	Creates a window that has a thin-line border.
WS_CAPTION	Creates a window that has a title bar (includes the WS_BORDER style).
WS_CHILD	Creates a child window. A window with this style

	cannot have a menu bar. This style cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	Same as the WS_CHILD style.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.
WS_DISABLED	Creates a window that is initially disabled. A disabled window cannot receive input from the user. To change this after a window has been created, use **EnableWindow**.
WS_DLGFRAME	Creates a window that has a border of a style typically used with dialog boxes. A window with this style cannot have a title bar.
WS_GROUP	Specifies the first control of a group of controls. The group consists of this first control and all controls defined after it, up to the next control with the WS_GROUP style. The first control in each group usually has the WS_TABSTOP style so that the user can move from group to group. The user can subsequently change the keyboard focus from one control in the group to the next control in the group by using the direction keys. You can turn this style on and off to change dialog box navigation. To change this style after a window has been created, use **SetWindowLong**.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially minimized. Same as the WS_MINIMIZE style.
WS_MAXIMIZE	Creates a window that is initially maximized.
WS_MAXIMIZEBOX	Creates a window that has a maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_MINIMIZE	Creates a window that is initially minimized. Same as the WS_ICONIC style.

WS_MINIMIZEBOX	Creates a window that has a minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.
WS_OVERLAPPEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.
WS_POPUP	Creates a pop-up window. This style cannot be used with the WS_CHILD style.
WS_POPUPWINDOW	Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.
WS_SIZEBOX	Creates a window that has a sizing border. Same as the WS_THICKFRAME style.
WS_SYSMENU	Creates a window that has a window menu on its title bar. The WS_CAPTION style must also be specified.
WS_TABSTOP	Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style. You can turn this style on and off to change dialog box navigation. To change this style after a window has been created, use **SetWindowLong**.
WS_THICKFRAME	Creates a window that has a sizing border. Same as the WS_SIZEBOX style.
WS_TILED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.
WS_TILEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style.
WS_VISIBLE	Creates a window that is initially visible. This style can be turned on and off by using **ShowWindow** or **SetWindowPos**

WS_VSCROLL Creates a window that has a vertical scroll bar.

This is updated documents from Microsoft Help Desk.

Bitwise Inclusive-OR Operator '|'

The bitwise inclusive OR '|' operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1 (one). If both of the bits are 0 (zero), the result of the comparison is 0 (zero); otherwise, the result is 1 (one).

9.6.4 Horizontal Position of the Window (*x*)

This member specifies the initial horizontal position of the window. For an overlapped or pop-up window, the *x* parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *x* is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area.

If this parameter is set to CW_USEDEFAULT, the system selects the default position for the window's upper-left corner and ignores the y parameter. CW_USEDEFAULT is valid only for overlapped windows; if it is specified for a pop-up or child window, the x and y parameters are set to zero.

9.6.5 Vertical Position of the Window (*y*)

This member specifies the initial vertical position of the window. For an overlapped or pop-up window, the *y* parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *y* is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box, *y* is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the WS_VISIBLE style bit set and the x parameter is set to CW_USEDEFAULT, the system ignores the y parameter.

9.6.6 Width of the Window (*nWidth*)

This specifies the width, in device units, of the window. For overlapped windows, *nWidth* is either the window's width, in screen coordinates, or CW_USEDEFAULT. If *nWidth* is CW_USEDEFAULT, the system selects a default width and height for the window; the default width extends from the initial x-coordinate to the right edge of the screen, and the default height extends from the initial y-coordinate to the top of the icon area.

CW_USEDEFAULT is valid only for overlapped windows; if CW_USEDEFAULT is specified for a pop-up or child window, *nWidth* and *nHeight* are set to zero.

9.6.7 Height of the Window (*nHeight*)

This member specifies the height, in device units, of the window. For overlapped windows, *nHeight* is the window's height, in screen coordinates.

If nWidth is set to CW_USEDEFAULT, the system ignores nHeight.

9.6.8 Parent of the Window (*hWndParent*)

This member is a HANDLE to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

9.6.9 Menu of the Window (*hMenu*)

This member is a HANDLE to a menu, or specifies a child-window identifier depending on the window style. For an overlapped or pop-up window, *hMenu* identifies the menu to be used with the window; it can be NULL if the class menu is to be used. For a child window, *hMenu* specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

9.6.10 Instance Handle (*hInstance*)

This member is Application instance handle.

In Windows NT/2000 or later This value is ignored.

9.6.11 Long Param (*lpParam*)

This member is a pointer to a value to be passed to the window through the **CREATESTRUCT** structure passed in the *lParam* parameter the **WM_CREATE** message. If an application calls **CreateWindow** to create a multiple document interface (MDI) client window, *lpParam* must point to a **CLIENTCREATESTRUCT** structure.

9.6.12 Return Value

If the *CreateWindow* function is successful, then it returns a valid handle of the newly created window. Otherwise it returns NULL.

9.7 Using Windows (*Example*)

```
HINSTANCE hinst;
HWND hwndMain;

// Create the main window.

hwndMain = CreateWindowEx(
    0,                          // no extended styles
    "MainWClass",               // class name
    "Main Window",              // window name
    WS_OVERLAPPEDWINDOW |       // overlapped window
            WS_HSCROLL |        // horizontal scroll bar
            WS_VSCROLL,         // vertical scroll bar
    CW_USEDEFAULT,              // default horizontal
position
    CW_USEDEFAULT,              // default vertical
position
    CW_USEDEFAULT,              // default width
    CW_USEDEFAULT,              // default height
    (HWND) NULL,                // no parent or owner
window
    (HMENU) NULL,               // class menu used
    hinstance,                  // instance handle
    NULL);                      // no window creation data

if (!hwndMain)
    return FALSE;
```

Now Show the window using the flag specified by the program that started the application, and send the application WM_PAINT message.

```
ShowWindow(hwndMain, SW_SHOWDEFAULT);
UpdateWindow(hwndMain);
```

9.8 Messages in Windows

Unlike MS-DOS-based applications, Win32®-based applications are event-driven. They do not make explicit function calls (such as C run-time library calls) to obtain input. Instead, they wait for the system to pass input to them. The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system.

Note: We are presenting here a brief description of messages. Detailed discussion about messages, message routing, message types, message filtering etc will be given in next lectures.

9.8.1 Message Queuing

- Operating system keeps the generated messages in a queue.
- Every application has its own message queue.
- Messages generated in a system first reside in System Message Queue, then dispatch to application message queue and to the windows procedure.
- Windows programming is basically message driven programming.

9.8.2 Message Routing

The system uses two methods to route messages to a window procedure: posting messages to a first-in, first-out queue called a message queue, a system-defined memory object that temporarily stores messages, and sending messages directly to a window procedure.

Messages posted to a message queue are called queued messages. They are primarily the result of user input entered through the mouse or keyboard.

9.9 Window Procedure

Every window has its procedure that is called windows procedure. All messages that are sent be *DispatchMessage* API or *SendMessage* API will be received by windows procedure. So windows procedure is the particular address in memory that receives messages. Windows operating system gets this address through registered window class member *lpfnWndProc*. You have to provide address or name of window procedure in windows class.
Windows procedure receives four parameters,

```
LRESULT (CALLBACK *WNDPROC) (HWND hWnd,UINT message,WPARAM
wParam,LPARAM lParam);
```

9.9.1 Handle to Window(*hWnd*)

This member is a HANDLE to the window to which message was sent.

9.9.2 Message Type(*uMsg*)

This member specifies the message type; the message could be a Mouse message, character message, keyboard message, etc. Message is unsigned 32bit number.

9.9.3 Message's WPARAM(*wParam*)

This specifies additional message information. The contents of this parameter depend on the value of the *uMsg* parameter e.g. key down message keeps the key pressed value in this parameter.

9.9.4 Message's LPARAM(*lParam*)

This specifies additional message information. The contents of this parameter depend on the value of the *uMsg* parameter e.g. mouse down messages keep information of mouse pointer's x and y position in this parameter.

9.9.5 Return Value

The return value is the result of the message processing and depends on the message sent function.

9.10 Getting message from Message Queue

We can get message from message Queue by using *GetMessage* or *PeekMessage* APIs. The *GetMessage* function retrieves a message from the calling thread's message queue and also removes the message from the queue. And then function dispatches incoming sent messages until a posted message is available for retrieval.

GetMessage inputs four parameters,

```
BOOL GetMessage()
(
        LPMSG lpMsg,
        HWND hWnd,
        UINT wMsgFilterMin,
        UINT wMsgFilterMax
)
```

lpMsg

[out] Pointer to an MSG structure that receives message information from the thread's message queue.

hWnd

[in] Handle to the window whose messages are to be retrieved. The window must belong to the calling thread. The following value has a special meaning.

wMsgFilterMin

[in] Specifies the integer value of the lowest message value to be retrieved. Use WM_KEYFIRST to specify the first keyboard message or WM_MOUSEFIRST to specify the first mouse message.

wMsgFilterMax

[in] Specifies the integer value of the highest message value to be retrieved. Use WM_KEYLAST to specify the first keyboard message or WM_MOUSELAST to specify the last mouse message.

return Value

If the function retrieves a message other than **WM_QUIT**, the return value is nonzero. If the function retrieves the **WM_QUIT** message, the return value is zero.

If there is an error, the return value is -1. For example, the function fails if *hWnd* is an invalid window handle or *lpMsg* is an invalid pointer. To get extended error information, use GetLastError function.

9.11 Message Dispatching

After getting message from message queue, message is dispatched to the actual window procedure. For dispatching messages to window procedure, we use *DispatchMessage* API.

DispatchMessage inputs one argument that is pointer to MSG structure.

```
BOOL DispatchMessage
(
        MSG *lpMsg
)
```

Summary

In this lecture, we learnt how to Register Window class using RegisterClass API and how to set attributes of window class structure. After registration of Window class we learnt to use *CreateWindow* API. *CreateWindow* API uses window class name, caption name, style of window, starting points, width and height of windows and windows parent handle or handle to owner mainly. We have window handle in variable of type handle to window. Using window handle we can send different types of messages to the window.
Then we learnt how to get messages from application message queue. And after getting messages from application message queue, we dispatch messages to the windows message handling procedure. Windows message procedure inputs four parameters. These parameters are also the members of MSG structure. These parameters are important for every one who is developing applications for Windows.

Exercises

Write down a code which would create and destroy window successfully. Further more show message box when the user closes window.

Chapter 10
Architecture of Standard Win32 Application

10.1Creating Window Application

Now we are able to create a mature windows application. We are already familiar with windows creation, registration, message receiving and Message dispatching processes. Here we will make our knowledge more real through practical work. Let's make a full fledge window.

We will make a full fledge window application in four steps:

1. In first step, we will register a windows class.
2. In second step, we will create a window
3. In third step, we will make a message loop and message handling procedure
4. In our fourth step, we will write WinMain function and will make the application running.

Step 1 (*Registering a Window Class*)

We write our own function *RegisterAppWindow* which will register the window and return true or false in case of success or failure.

```
bool RegisterAppWindow(HINSTANCE hInstance)
{

/* Before registering class, we will have to fill the WNDCLASS structure.
*/

WNDCLASS wc;
wc.style = 0;
wc.lpfnWndProc=MyWindowProc;
wc.cbClsExtra = 0;
wc.cbWndExtra =0;
wc.hInstance = hInstance;
wc.hIcon = NULL;
wc.hCursor = LoadCursor(NULL, LoadCursor(IDC_ARROW));
wc.hbrBackground= (HBRUSH)GetStockObject(GRAY_BRUSH);
wc.lpszMenuName=NULL;
wc.lpszClassName="MyFirstWindowClass";

/*
We have discussed almost all the parameters of WNDCLASS structure in our previous lecture.
After filling the WNDCLASS structure, we will pass this as a reference to RegisterClass function. Because RegisterClass function take pointer to the structure WNDLCASS as a parameter.
*/

ATOM cAtom=RegisterClass(&wc);
```

```
if(!cAtom)
{
	MessageBox(NULL,"Error Register Window Class"," Virtual Uinversity",0);
	return 0;
}

return true;
}//End of Function RegisterAppWindow
```

Step 2 (Creating Window)

For Creation of window, we will use Win32 API *CreateWindow*. We have studied *CreateWindow* function in our previous lecture. Here we will use this function for creating window.

For creating window and checking return values, we will define our own function *InitApplication* which will internally call *CreateAppWindow* and returns true or false. It will return true in case of success and false in case failure.

```
bool InitApplication(HINSTANCE hInstance)
{
HWND hWnd=NULL;

/*
Call RegisterAppWindow Function and returns false if this function returns false because we need not to proceed forward unless our windows is not registered properly.
*/

if(!RegisterAppWindow(hInstance))
{
	return false;
}

/*
We already know that CreateWindow function returns handle to window, we will save this return handle in our hWnd variable.
*/
hWnd = CreateWindow("MyFirstWindowClass","Virtual Uinversity",
WS_OVERLAPPEDWINDOW|WS_VISIBLE,
	100,
	50,
	CW_USEDDEFAULT,
	CW_USEDDEFAULT,
	NULL,
	NULL,
	hInstance,	//Optional or not used in Window 2000 or above
	NULL);
/*
```

```
After receiving the return handling of the window, we are interested to know that what
information are in return handle. hWnd can have either handle to window or Null value.
Null value shows that CreateWindow function has not been successful and no window
could be created or window has successfully created otherwise.
*/
if (hWnd == NULL)
{
        MessageBox(NULL, “Cannot Create Window”,”Virtual Uinversity”,0);
        return false;
}

/*
After Successful creation of window, we have initially hidden window. For showing
window on the screen, we will use following API: ShowWindow
*/

ShowWindow(hWnd,SW_SHOWNORMAL);

/*
ShowWindow take second parameter which indicates windows show or hidden states.
*/
return true;
} //End of InitApplication
```

10.2 About Messages

Unlike MS-DOS-based applications, Windows-based applications are event-driven. They do not make explicit function calls (such as C run-time library calls) to obtain input. Instead, they wait for the system to pass input to them.

The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. For more information about window procedures, see Window Procedures.

Microsoft® Windows® XP: If a top-level window stops responding to messages for more than several seconds, the system considers the window to be hung. In this case, the system hides the window and replaces it with a ghost window that has the same Z order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually hung. When in the debugger mode, the system does not generate a ghost window

10.2.1 Windows Messages

The system passes input to a window procedure in the form of *messages*. Messages are generated by both the system and applications. The system generates a message at each input event — for example, when the user types, moves the mouse, or clicks a control such as a scroll bar. The system also generates messages in response to changes in the system brought about by an application, such as when an application changes the pool of system font resources or resizes one of its windows. An application can generate messages to direct its own windows to perform tasks or to communicate with windows in other applications.

The system sends a message to a window procedure with a set of four parameters:

- Window handle
- Message identifier
- Two values called message parameters.

The window handle identifies the window for which the message is intended. The system uses it to determine which window procedure should receive the message.

A message identifier is a named constant that identifies the purpose of a message. When a window procedure receives a message, it uses a message identifier to determine how to process the message. For example, the message identifier WM_PAINT tells the window procedure that the window's client area has changed and must be repainted.

Message parameters specify data or the location of data used by a window procedure when processing a message. The meaning and value of the message parameters depend on the message. A message parameter can contain an integer, packed bit flags, a pointer to a structure containing additional data, and so on. When a message does not use message parameters, they are typically set to NULL. A window procedure must check the message identifier to determine how to interpret the message parameters.

Message Types

This section describes the two types of messages:

- System-Defined Messages
- Application-Defined Messages

System-Defined Messages

The system sends or posts a *system-defined message* when it communicates with an application. It uses these messages to control the operations of applications and to provide input and other information for applications to process. An application can also send or post system-defined messages. Applications generally use these messages to control the operation of control windows created by using pre-registered window classes.

Each system-defined message has a unique message identifier and a corresponding symbolic constant (defined in the software development kit (SDK) header files) that states the purpose of the message. For example, the **WM_PAINT** constant requests that a window paint its contents.

Symbolic constants specify the category to which system-defined messages belong. The prefix of the constant identifies the type of window that can interpret and process the message. Following are the prefixes and their related message categories.

Prefix	Message category
ABM	Application desktop toolbar
BM	Button control
CB	Combo box control
CBEM	Extended combo box control
CDM	Common dialog box
DBT	Device
DL	Drag list box
DM	Default push button control
DTM	Date and time picker control
EM	Edit control
HDM	Header control
HKM	Hot key control
IPM	IP address control
LB	List box control
LVM	List view control
MCM	Month calendar control
PBM	Progress bar
PGM	Pager control
PSM	Property sheet
RB	Rebar control
SB	Status bar window
SBM	Scroll bar control
STM	Static control
TB	Toolbar
TBM	Trackbar
TCM	Tab control
TTM	Tooltip control
TVM	Tree-view control
UDM	Up-down control
WM	General window

General window messages cover a wide range of information and requests, including messages for mouse and keyboard input, menu and dialog box input, window creation and management.

Application-Defined Messages

An application can create messages to be used by its own windows or to communicate with windows in other processes. If an application creates its own messages, the window procedure that receives them must interpret the messages and provide appropriate processing.

Message-identifier values are used as follows:

- The system reserves message-identifier values in the range 0x0000 through 0x03FF (the value of WM_USER – 1) for system-defined messages. Applications cannot use these values for private messages.
- Values in the range 0x0400 (the value of **WM_USER**) through 0x7FFF are available for message identifiers for private window classes.
- If your application is marked version 4.0, you can use message-identifier values in the range 0x8000 (WM_APP) through 0xBFFF for private messages.
- The system returns a message identifier in the range 0xC000 through 0xFFFF when an application calls the RegisterWindowMessage function to register a message. The message identifier returned by this function is guaranteed to be unique throughout the system. Use of this function prevents conflicts that can arise if other applications use the same message identifier for different purposes.

10.2.2 Message Routing

The system uses two methods to route messages to a window procedure: posting messages to a first-in, first-out queue called a ***Message queue***, a system-defined memory object that temporarily stores messages, and sending messages directly to a window procedure.

Messages posted to a message queue are called *queued messages*. They are primarily the result of user input entered through the mouse or keyboard, such as WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_KEYDOWN, and WM_CHAR messages. Other queued messages include the timer, paint, and quit messages: WM_TIMER, **WM_PAINT**, and WM_QUIT. Most other messages, which are sent directly to a window procedure, are called *nonqueued messages*.

- Queued Messages
- Nonqueued Messages

Queued Messages

The system can display any number of windows at a time. To route mouse and keyboard input to the appropriate window, the system uses message queues.

The system maintains a single system message queue and one thread-specific message queue for each graphical user interface (GUI) thread. To avoid the overhead of creating a message queue for non–GUI threads, all threads are created initially without a message queue. The system creates a thread-specific message queue only when the thread makes its first call to one of the User or Windows Graphics Device Interface (GDI) functions.

Whenever the user moves the mouse, clicks the mouse buttons, or types on the keyboard, the device driver for the mouse or keyboard converts the input into messages and places them in the system message queue. The system removes the messages, one at a time, from the system message queue, examines them to determine the destination window, and then posts them to the message queue of the thread that created the destination window. A thread's message queue receives all mouse and keyboard messages for the windows created by the thread. The thread removes messages from its queue and directs the system to send them to the appropriate window procedure for processing.

With the exception of the **WM_PAINT** message, the system always posts messages at the end of a message queue. This ensures that a window receives its input messages in the proper first in, first out (FIFO) sequence. The **WM_PAINT** message, however, is kept in the queue and is forwarded to the window procedure only when the queue contains no other messages. Multiple **WM_PAINT** messages for the same window are combined into a single **WM_PAINT** message, consolidating all invalid parts of the client area into a single area. Combining **WM_PAINT** messages reduces the number of times a window must redraw the contents of its client area.

The system posts a message to a thread's message queue by filling an MSG structure and then copying it to the message queue.

Information in **MSG** includes:

```
typedef struct tagMSG {

        HWND hWnd;
        UINT message;
        WPARAM wParam,
        LPARAM lParam,
        DWORD time,
        POINT pt

}MSG;
```

hWnd: The handle of the window for which the message is intended (*hWnd*)
message: The message identifier (*message*)
wParam: The two message parameters (*wParam and lParam*)
lParam: The time the message was posted (*time*)
time: The mouse cursor position (*pt*)

A thread can post a message to its own message queue or to the queue of another thread by using the *PostMessage* or *PostThreadMessage* function.

An application can remove a message from its queue by using the *GetMessage* function.To examine a message without removing it from its queue, an application can use the *PeekMessage* function. This function fills **MSG** with information about the message.

After removing a message from its queue, an application can use the *DispatchMessage* function to direct the system to send the message to a window procedure for processing. **DispatchMessage** takes a pointer to **MSG** that was filled by a previous call to the **GetMessage** or **PeekMessage** function. **DispatchMessage** passes the window handle, the message identifier, and the two message parameters to the window procedure, but it does not pass the time the message was posted or mouse cursor position. An application can retrieve this information by calling the *GetMessageTime* and *GetMessagePos* functions while processing a message.

A thread can use the *WaitMessage* function to yield control to other threads when it has no messages in its message queue. The function suspends the thread and does not return until a new message is placed in the thread's message queue.

You can call the *SetMessageExtraInfo* function to associate a value with the current thread's message queue. Then call the *GetMessageExtraInfo* function to get the value associated with the last message retrieved by the **GetMessage** or **PeekMessage** function.

Nonqueued Messages

Nonqueued messages are sent immediately to the destination window procedure, bypassing the system message queue and thread message queue. The system typically sends nonqueued messages to notify a window of events that affect it. For example, when the user activates a new application window, the system sends the window a series of messages, including WM_ACTIVATE, WM_SETFOCUS, and WM_SETCURSOR. These messages notify the window that it has been activated, that keyboard input is being directed to the window, and that the mouse cursor has been moved within the borders of the window. Nonqueued messages can also result when an application calls certain system functions. For example, the system sends the WM_WINDOWPOSCHANGED message after an application uses the *SetWindowPos* function to move a window.

Some functions that send nonqueued messages are *BroadcastSystemMessage*, *BroadcastSystemMessageEx*, *SendMessage*, *SendMessageTimeout*, and *SendNotifyMessage*.

10.2.3 Message Handling

An application must remove and process messages posted to the message queues of its threads. A single-threaded application usually uses a message loop in its WinMain function to remove and send messages to the appropriate window procedures for processing. Applications with multiple threads can include a message loop in each thread that creates a window. The following sections describe how a message loop works and explain the role of a window procedure:

- Message Loop
- Window Procedure

Message Loop

A simple message loop consists of one function call to each of these three functions: **GetMessage**, TranslateMessage, and **DispatchMessage**. Note that if there is an error, **GetMessage** returns -1 -- thus the need for the special testing.

The **GetMessage** function retrieves a message from the queue and copies it to a structure of type **MSG**. It returns a nonzero value, unless it encounters the **WM_QUIT** message, in which case it returns FALSE and ends the loop. In a single-threaded application, ending the message loop is often the first step in closing the application. An application can end its own loop by using the *PostQuitMessage* function, typically in response to the WM_DESTROY message in the window procedure of the application's main window.

If you specify a window handle as the second parameter of **GetMessage**, only messages for the specified window are retrieved from the queue. **GetMessage** can also filter messages in the queue, retrieving only those messages that fall within a specified range.

A thread's message loop must include **TranslateMessage** if the thread is to receive character input from the keyboard. The system generates virtual-key messages (**WM_KEYDOWN** and WM_KEYUP) each time the user presses a key. A virtual-key message contains a virtual-key code that identifies which key was pressed, but not its character value. To retrieve this value, the message loop must contain **TranslateMessage**, which translates the virtual-key message into a character message (**WM_CHAR**) and places it back into the application message queue. The character message can then be removed upon a subsequent iteration of the message loop and dispatched to a window procedure.

The **DispatchMessage** function sends a message to the window procedure associated with the window handle specified in the **MSG** structure. If the window handle is HWND_TOPMOST, **DispatchMessage** sends the message to the window procedures of

all top-level windows in the system. If the window handle is NULL, **DispatchMessage** does nothing with the message.

An application's main thread starts its message loop after initializing the application and creating at least one window. Once started, the message loop continues to retrieve messages from the thread's message queue and to dispatch them to the appropriate windows. The message loop ends when the **GetMessage** function removes the **WM_QUIT** message from the message queue.

Only one message loop is needed for a message queue, even if an application contains many windows. **DispatchMessage** always dispatches the message to the proper window; this is because each message in the queue is an **MSG** structure that contains the handle of the window to which the message belongs.

You can modify a message loop in a variety of ways. For example, you can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages not specifying a window. You can also direct **GetMessage** to search for specific messages, leaving other messages in the queue. This is useful if you must temporarily bypass the usual FIFO order of the message queue.

An application that uses accelerator keys must be able to translate keyboard messages into command messages. To do this, the application's message loop must include a call to the *TranslateAccelerator* function. For more information about accelerator keys, see Keyboard Accelerators.

If a thread uses a modeless dialog box, the message loop must include the *IsDialogMessage* function so that the dialog box can receive keyboard input.

Window Procedure

A window procedure is a function that receives and processes all messages sent to the window. Every window class has a window procedure, and every window created with that class uses that same window procedure to respond to messages.

The system sends a message to a window procedure by passing the message data as arguments to the procedure. The window procedure then performs an appropriate action for the message; it checks the message identifier and, while processing the message, uses the information specified by the message parameters.

A window procedure does not usually ignore a message. If it does not process a message, it must send the message back to the system for default processing. The window procedure does this by calling the *DefWindowProc* function, which performs a default action and returns a message result. The window procedure must then return this value as its own message result. Most window procedures process just a few messages and pass the others on to the system by calling **DefWindowProc**.

Because a window procedure is shared by all windows belonging to the same class, it can process messages for several different windows. To identify the specific window affected by the message, a window procedure can examine the window handle passed with a message.

10.2.4 Message Filtering

An application can choose specific messages to retrieve from the message queue (while ignoring other messages) by using the **GetMessage** or **PeekMessage** function to specify a message filter. The filter is a range of message identifiers (specified by a first and last identifier), a window handle, or both. **GetMessage** and **PeekMessage** use a message filter to select which messages to retrieve from the queue. Message filtering is useful if an application must search the message queue for messages that have arrived later in the queue. It is also useful if an application must process input (hardware) messages before processing posted messages.

The WM_KEYFIRST and WM_KEYLAST constants can be used as filter values to retrieve all keyboard messages; the WM_MOUSEFIRST and WM_MOUSELAST constants can be used to retrieve all mouse messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, if an application filters for a **WM_CHAR** message in a window that does not receive keyboard input, the **GetMessage** function does not return. This effectively "hangs" the application.

10.2.5 Posting and Sending Messages

Any application can post and send messages. Like the system, an application posts a message by copying it to a message queue and sends a message by passing the message data as arguments to a window procedure. To post messages, an application uses the **PostMessage** function. An application can send a message by calling the **SendMessage**, **BroadcastSystemMessage**, SendMessageCallback, **SendMessageTimeout**, **SendNotifyMessage**, or SendDlgItemMessage function.

Posting Messages

An application typically posts a message to notify a specific window to perform a task. **PostMessage** creates an **MSG** structure for the message and copies the message to the message queue. The application's message loop eventually retrieves the message and dispatches it to the appropriate window procedure.

An application can post a message without specifying a window. If the application supplies a NULL window handle when calling **PostMessage**, the message is posted to the queue associated with the current thread. Because no window handle is specified, the application must process the message in the message loop. This is one way to create a message that applies to the entire application, instead of to a specific window.

Occasionally, you may want to post a message to all top-level windows in the system. An application can post a message to all top-level windows by calling **PostMessage** and specifying HWND_TOPMOST in the *hwnd* parameter.

A common programming error is to assume that the **PostMessage** function always posts a message. This is not true when the message queue is full. An application should check the return value of the **PostMessage** function to determine whether the message has been posted and, if it has not been, repost it.

Sending Messages

An application typically sends a message to notify a window procedure to perform a task immediately. The **SendMessage** function sends the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an edit control as its child window can set the text of the control by sending a message to it. The control can notify the parent window of changes to the text that are carried out by the user by sending messages back to the parent.

The **SendMessageCallback** function also sends a message to the window procedure corresponding to the given window. However, this function returns immediately. After the window procedure processes the message, the system calls the specified callback function.

Occasionally, you may want to send a message to all top-level windows in the system. For example, if the application changes the system time, it must notify all top-level windows about the change by sending a WM_TIMECHANGE message. An application can send a message to all top-level windows by calling **SendMessage** and specifying HWND_TOPMOST in the *hwnd* parameter. You can also broadcast a message to all applications by calling the **BroadcastSystemMessage** function and specifying BSM_APPLICATIONS in the *lpdwRecipients* parameter.

By using the *InSendMessage* or *InSendMessageEx* function, a window procedure can determine whether it is processing a message sent by another thread. This capability is useful when message processing depends on the origin of the message.

10.2.6 Message Deadlocks

A thread that calls the **SendMessage** function to send a message to another thread cannot continue executing until the window procedure that receives the message returns. If the receiving thread yields control while processing the message, the sending thread cannot continue executing, because it is waiting for **SendMessage** to return. If the receiving thread is attached to the same queue as the sender, it can cause an application deadlock to occur. (Note that journal hooks attach threads to the same queue.)

Note that the receiving thread needs not yield control explicitly; calling any of the following functions can cause a thread to yield control implicitly.

- DialogBox
- DialogBoxIndirect
- DialogBoxIndirectParam
- DialogBoxParam
- GetMessage
- MessageBox
- PeekMessage
- SendMessage

To avoid potential deadlocks in your application, consider using the **SendNotifyMessage** or **SendMessageTimeout** functions. Otherwise, a window procedure can determine whether a message it has received was sent by another thread by calling the **InSendMessage** or **InSendMessageEx** function. Before calling any of the functions in the preceding list while processing a message, the window procedure should first call **InSendMessage** or **InSendMessageEx**. If this function returns TRUE, the window procedure must call the ReplyMessage function before any function that causes the thread to yield control.

10.2.7 Broadcasting Messages

Each message consists of a message identifier and two parameters, *wParam* and *lParam*. The message identifier is a unique value that specifies the message purpose. The parameters provide additional information that is message-specific, but the *wParam* parameter is generally a type value that provides more information about the message.

A *message broadcast* is simply the sending of a message to multiple recipients in the system. To broadcast a message from an application, use the **BroadcastSystemMessage** function, specifying the recipients of the message. Rather than specify individual recipients, you must specify one or more types of recipients. These types are applications, installable drivers, network drivers, and system-level device drivers. The system sends broadcast messages to all members of each specified type.

The system typically broadcasts messages in response to changes that take place within system-level device drivers or related components. The driver or related component broadcasts the message to applications and other components to notify them of the change. For example, the component responsible for disk drives broadcasts a message whenever the device driver for the floppy disk drive detects a change of media such as when the user inserts a disk in the drive.

The system broadcasts messages to recipients in this order: system-level device drivers, network drivers, installable drivers, and applications. This means that system-level device drivers, if chosen as recipients, always get the first opportunity to respond to a message. Within a given recipient type, no driver is guaranteed to receive a given message before

any other driver. This means that a message intended for a specific driver must have a globally-unique message identifier so that no other driver unintentionally processes it.

You can also broadcast messages to all top-level windows by specifying HWND_BROADCAST in the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function.

Applications receive messages through the window procedure of their top-level windows. Messages are not sent to child windows. Services can receive messages through a window procedure or their service control handlers.

Note System-level device drivers use a related, system-level function to broadcast system messages.

Step 3 (Fetching messages and Message Procedure)

For Retrieving message from message queue we are going to write function which will have a message loop and *TranslateMessage* and *DispatchMessage* API's will be enclosed in the message loop.
We will call this function *RunApplication*.

```
int RunApplication()
{
MSG msg;

/*Main Application Message Loop*/

while(GetMessage (&msg, NULL, 0, 0)  >  0)
{
        /* translate virtual-key messages into character messages */

        TranslateMessage (&msg);

        /* dispatch message to windows procedure*/

        DispatchMessage (&msg);
}//End of Message Loop (while)

return (int)msg.wParam;
}//End of RunApplication
```

This type of message loop, we will be using in our windows applications. Now we will create Window Message Procedure. This message procedure will have the same name as given in *WNDLCASS* structure.

```
LRESULT CALLBACK MyWindowProc (HWND hWnd,UINT message,WPARAM
wParam,LPARAM lParam)
{
        switch(message)
        {
                case WM_LBUTTONDOWN:
                {
                  MessageBox(hWnd, “Left Mouse Button Pressed”,”Virtual
University”,0);
                   return 0;
                }
                case WM_DESTROY:
                {
/*
We have recieve WM_DESTROY message, this message removes window from screen
and release resource, it allocated. Now at this time, we should post WM_QUIT message
in a message queue so that message loop terminates. For posting WM_QUIT message we
use PostQuitMessage API.
*/
                        PostQuitMessage(0);
                        return 0;
                }
        }

/*
DefWindowProc(hWnd, message, wParam, lParam). This function calls the default
window procedure to provide default processing for any window messages that an
application does not process. This function ensures that every message is processed.
DefWindowProc () is called with the same parameters received by the window
procedure.
*/

DefWindowProc(hWnd,message,wParam,lParam);
} // End of MyWindowProc
```

Step 4 (*WinMain Function*)

In this step we will write Windows starting point WinMain function.

```
int CALLBACK WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{

/*
```

```
Call InitApplication Function and returns false if this function returns false because we
need not to proceed forward unless our windows is not registered and created properly.
*/

        if(!InitApplication(hInstance))
        {
           MessageBox(NULL,"Application could not be initialized properly","Virtual
University",MB_ICONHAND|MB_OK);
           return 0;
         }

/*
 Finally run the application until user press the close button and choose close from system
menu.
*/

return RunApplication();
}
```

Summary

In this lecture, we have created a full fledge window application which has all the characteristics of standard windows application. For creating window, we registered window class with the attributes required. After successful registration we created window. For creating window we used CreateWindow API. In CreateWindow API we mentioned some of the Window styles. Window styles are used to create different styles of windows. In this lecture we have created overlapped window with additional style like WS_VISIBLE or we can use ShowWindow function with WS_SHOWNORMAL style. We have also discussed many of the other windows style and their behaviors in our previous lecture. And after successful creation of window we made message handling procedure. Our message handling procedure checks left mouse button and in response of this it shows message box which is written some text, and other message are let them passed to default message procedure. We also studied Massages routing, message dispatching, message filtering, messages deadlock, message sending and message posting. For message receiving and dispatching, we created message loop which get the messages from message Queue and dispatch these messages to message procedure. Finally we coded a running application which displays windows and on pressing a left mouse button it shows a message box.

Chapter 11

User Interfaces

11.1 Hierarchy of Windows

The basic building block for displaying information in the Microsoft® Windows™ graphical environment is the window. Microsoft Windows manages how each window relates to all other windows in terms of visibility, ownership, and parent/child relationship. Windows uses this relationship information when creating, painting, destroying, sizing or displaying a window. A window can have many children's and may or may not have one parent. An example of windows is Notepad, calculator, word pad etc, are all windows.

A window shares the screen with other windows, including those from other applications. Only one window at a time can receive input from the user. The user can use the mouse, keyboard, or other input device to interact with this window and the application that owns it.

11.2 Threads

A thread is basically a path of execution through a program. It is also the smallest unit of execution that Win32 schedules. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all of the process's resources.

A process consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority may have to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to "balance" the CPU load.

Each thread in a process operates independently. Unless you make them visible to each other, the threads execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, must coordinate their work by using semaphores or another method of inter process communication.

So a thread is a process that is part of a larger process or application. A thread can execute any part of an application's code, including code that is currently being executed by another thread. All threads share the

- ✓ Virtual Address space
- ✓ Global variables
- ✓ Operating system resources of their respective processes.

Threads are two types of threads.

1. User-Interface Thread
2. Worker Thread

11.2.1 User-Interface Thread

In Windows, a thread that handles user input and responds to user events independently is User-Interface Thread. User-interface thread own one or more windows and have its own message queue. User-interface threads process messages, received from the system.

11.2.2 Worker Thread

A worker thread is commonly used to handle background tasks. Tasks such as calculation and background printing are good examples of worker threads.

11.3 Windows

In a graphical Microsoft® Windows®-based application, a window is a rectangular area of the screen where the application displays output and receives input from the user. Therefore, one of the first tasks of a graphical Windows-based application is to create a window.

A window shares the screen with other windows, including those from other applications. Only one window at a time can receive input from the user. The user can use the mouse, keyboard, or other input device to interact with this window and the application that owns it.

A Window may further contain more windows inside it. For example lets take a calculator, A calculator contains more windows in forms of buttons, radio buttons and check boxes.

- Every Window has its parent and zero or more siblings.
- Top level window has desktop as its parent.

11.3.1 Desktop Window

When you start the system, it automatically creates the desktop window. The *desktop window* is a system-defined window that paints the background of the screen and serves as the base for all windows displayed by all applications.

The desktop window uses a bitmap to paint the background of the screen. The pattern created by the bitmap is called the *desktop wallpaper*. By default, the desktop window uses the bitmap from a .bmp file specified in the registry as the desktop wallpaper.

The *GetDesktopWindow* function returns a handle to the desktop window.

A system configuration application, such as a Control Panel item, changes the desktop wallpaper by using the SystemParametersInfo function with the *wAction* parameter set to SPI_SETDESKWALLPAPER and the *lpvParam* parameter specifying a bitmap file name. **SystemParametersInfo** then loads the bitmap from the specified file, uses the bitmap to paint the background of the screen, and enters the new file name in the registry.

11.3.2 Application Windows

Every graphical Microsoft® Windows®-based application creates at least one window, called the *main window* that serves as the primary interface between the user and the application. Most applications also create other windows, either directly or indirectly, to perform tasks related to the main window. Each window plays a part in displaying output and receiving input from the user.

When you start an application, the system also associates a taskbar button with the application. The *taskbar button* contains the program icon and title. When the application is active, its taskbar button is displayed in the pushed state.

An application window includes elements such as a title bar, a menu bar, the window menu (formerly known as the system menu), the minimize button, the maximize button, the restore button, the close button, a sizing border, a client area, a horizontal scroll bar, and a vertical scroll bar. An application's main window typically includes all of these components. The following illustration shows these components in a typical main window.

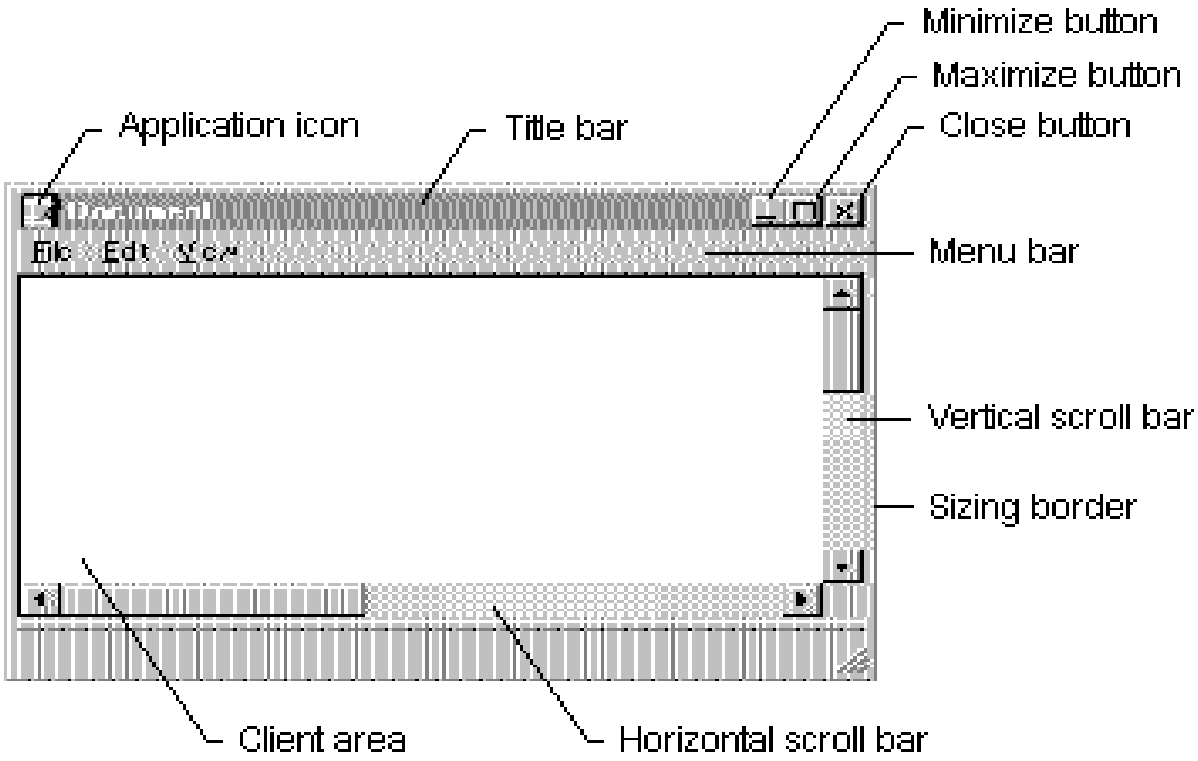


Figure 1 Windows Parts

11.3.2.1 Client Area

The *client area* is the part of a window where the application displays output, such as text or graphics. For example, a desktop publishing application displays the current page of a document in the client area. The application must provide a function, called a window procedure, to process input to the window and display output in the client area. For more information, see Window Procedures.

11.3.2.2 Nonclient Area

The title bar, menu bar, window menu, minimize and maximize buttons, sizing border, and scroll bars are referred to collectively as the window's *nonclient area*. The system manages most aspects of the nonclient area; the application manages the appearance and behavior of its client area.

The *title bar* displays an application-defined icon and line of text; typically, the text specifies the name of the application or indicates the purpose of the window. An application specifies the icon and text when creating the window. The title bar also makes it possible for the user to move the window by using a mouse or other pointing device.

Most applications include a *menu bar* that lists the commands supported by the application. Items in the menu bar represent the main categories of commands. Clicking an item on the menu bar typically opens a pop-up menu whose items correspond to the tasks within a given category. By clicking a command, the user directs the application to carry out a task.

The *window menu* is created and managed by the system. It contains a standard set of menu items that, when chosen by the user, set a window’s size or position, closes the application, or performs tasks.

The buttons in the upper-right corner affect the size and position of the window. When you click the *maximize button*, the system enlarges the window to the size of the screen and positions the window, so it covers the entire desktop, minus the taskbar. At the same time, the system replaces the maximize button with the restore button. When you click the *restore button*, the system restores the window to its previous size and position. When you click the *minimize button*, the system reduces the window to the size of its taskbar button, positions the window over the taskbar button, and displays the taskbar button in its normal state. To restore the application to its previous size and position, click its taskbar button. By clicking the *close button*, application exits.

The *sizing border* is an area around the perimeter of the window that enables the user to size the window by using a mouse or other pointing device.

The *horizontal scroll bar* and *vertical scroll bar* convert mouse or keyboard input into values that an application uses to shift the contents of the client area either horizontally or vertically. For example, a word-processing application that displays a lengthy document

typically provides a vertical scroll bar to enable the user to page up and down through the document.

11.3.3 Window Attributes

An application must provide the following information when creating a window. (With the exception of the Window Handle, which the creation function returns to uniquely identify the new window.)

- Class Name
- Window Name
- Window Style
- Extended Window Style
- Position
- Size
- Parent or Owner Window Handle
- Menu Handle or Child-Window Identifier
- Application Instance Handle
- Creation Data
- Window Handle

These window attributes are described in the following sections.

11.3.3.1 Class Name

Every window belongs to a window class. An application must register a window class before creating any windows of that class. The *window class* defines most aspects of a window's appearance and behavior. The chief component of a window class is the *window procedure*, a function that receives and processes all input and requests sent to the window. The system provides the input and requests in the form of *messages*. For more information, see Window Classes, **Window Procedures**, and Messages and Message Queues.

11.3.3.2 Window Name

A *window name* is a text string that identifies a window for the user. A main window, dialog box, or message box typically displays its window name in its title bar, if present. A control may display its window name, depending on the control's class. For example, buttons, edit controls, and static controls displays their window names within the rectangle occupied by the control. However, list boxes, combo boxes, and static controls do not display their window names.

To change the window name after creating a window, use the *SetWindowText* function. This function uses the *GetWindowTextLength* and *GetWindowText* functions to retrieve the current window-name string from the window.

11.3.3.3 Window Style

Every window has one or more window styles. A window style is a named constant that defines an aspect of the window's appearance and behavior that is not specified by the window's class. An application usually sets window styles when creating windows. It can also set the styles after creating a window by using the *SetWindowLong* function.

The system and, to some extent, the window procedure for the class, interpret the window styles.

Some window styles apply to all windows, but most apply to windows of specific window classes. The general window styles are represented by constants that begin with the WS_ prefix; they can be combined with the OR operator to form different types of windows, including main windows, dialog boxes, and child windows. The class-specific window styles define the appearance and behavior of windows belonging to the predefined control classes. For example, the SCROLLBAR class specifies a scroll bar control, but the SBS_HORZ and SBS_VERT styles determine whether a horizontal or vertical scroll bar control is created.

For lists of styles that can be used by windows, see the following topics:

- Window Styles
- Button Styles
- Combo Box Styles
- Edit Control Styles
- List Box Styles
- Rich Edit Control Styles
- Scroll Bar Control Styles
- Static Control Styles

11.3.3.4 Extended Window Style

Every window can optionally have one or more extended window styles. An *extended window style* is a named constant that defines an aspect of the window's appearance and behavior that is not specified by the window class or the other window styles. An application usually sets extended window styles when creating windows. It can also set the styles after creating a window by using the **SetWindowLong** function.

For more information, see CreateWindowEx.

11.3.3.5 Position

A window's position is defined as the coordinates of its upper left corner. These coordinates, sometimes called window coordinates, are always relative to the upper left corner of the screen or, for a child window, the upper left corner of the parent window's client area. For example, a top-level window having the coordinates (10,10) is placed 10

pixels to the right of the upper left corner of the screen and 10 pixels down from it. A child window having the coordinates (10,10) is placed 10 pixels to the right of the upper left corner of its parent window's client area and 10 pixels down from it.

The *WindowFromPoint* function retrieves a handle to the window occupying a particular point on the screen. Similarly, the *ChildWindowFromPoint* and *ChildWindowFromPointEx* functions retrieve a handle to the child window occupying a particular point in the parent window's client area. Although **ChildWindowFromPointEx** can ignore invisible, disabled, and transparent child windows, **ChildWindowFromPoint** cannot.

11.3.3.6 Size

A window's size (width and height) is given in pixels. A window can have zero width or height. If an application sets a window's width and height to zero, the system sets the size to the default minimum window size. To discover the default minimum window size, an application uses the *GetSystemMetrics* function with the SM_CXMIN and SM_CYMIN flags.

An application may need to create a window with a client area of a particular size. The *AdjustWindowRect* and *AdjustWindowRectEx* functions calculate the required size of a window based on the desired size of the client area. The application can pass the resulting size values to the **CreateWindowEx** function.

An application can size a window so that it is extremely large; however, it should not size a window so that it is larger than the screen. Before setting a window's size, the application should check the width and height of the screen by using **GetSystemMetrics** with the SM_CXSCREEN and SM_CYSCREEN flags.

11.3.3.7 Parent or Owner Window Handle

A window can have a parent window. A window that has a parent is called a *child window*. The *parent window* provides the coordinate system used for positioning a child window. Having a parent window affects aspects of a window's appearance; for example, a child window is clipped so that no part of the child window can appear outside the borders of its parent window. A window that has no parent, or whose parent is the desktop window, is called a *top-level window*. An application uses the EnumWindows function to obtain a handle to each of its top-level windows. **EnumWindows** passes the handle to each top-level window, in turn, to an application-defined callback function, EnumWindowsProc.

A window can own, or be owned by, another window. An owned window always appears in front of its owner window, is hidden when its owner window is minimized, and is destroyed when its owner window is destroyed. For more information, see Owned Windows.

11.3.3.8 Menu Handle or Child-Window Identifier

A child window can have a *child-window* identifier, a unique, application-defined value associated with the child window. Child-window identifiers are especially useful in applications that create multiple child windows. When creating a child window, an application specifies the identifier of the child window. After creating the window, the application can change the window's identifier by using the **SetWindowLong** function, or it can retrieve the identifier by using the GetWindowLong function.

Every window, except a child window, can have a menu. An application can include a menu by providing a menu handle either when registering the window's class or when creating the window.

11.3.3.9 Application Instance Handle

Every application has an instance handle associated with it. The system provides the instance handle to an application when the application starts. Because it can run multiple copies of the same application, the system uses instance handles internally to distinguish one instance of an application from another. The application must specify the instance handle in many different windows, including those that create windows.

11.3.3.10 Creation Data

Every window can have application-defined creation data associated with it. When the window is first created, the system passes a pointer to the data on to the window procedure of the window being created. The window procedure uses the data to initialize application-defined variables.

11.3.3.11 Window Handle

After creating a window, the creation function returns a *window handle* that uniquely identifies the window. A window handle has the **HWND** data type; an application must use this type when declaring a variable that holds a window handle. An application uses this handle in other functions to direct their actions to the window.

An application can use the *FindWindow* function to discover whether a window with the specified class name or window name exists in the system. If such a window exists, **FindWindow** returns a handle to the window. To limit the search to the child windows of a particular application, use the *FindWindowEx* function.

The *IsWindow* function determines whether a window handle identifies a valid, existing window. There are special constants that can replace a window handle in certain functions. For example, an application can use HWND_BROADCAST in the *SendMessage* and *SendMessageTimeout* functions, or HWND_DESKTOP in the *MapWindowPoint*s function.

11.3.4 Multithread Applications

A Windows-based application can have multiple threads of execution, and each thread can create windows. The thread that creates a window must contain the code for its window procedure.

An application can use the *EnumThreadWindows* function to enumerate the windows created by a particular thread. This function passes the handle to each thread window, in turn, to an application-defined callback function, *EnumThreadWndProc*.

The *GetWindowThreadProcessId* function returns the identifier of the thread that created a particular window.

To set the show state of a window created by another thread, use the *ShowWindowAsync* function.

11.4 Controls and Dialog Boxes

An application can create several types of windows in addition to its main window, including controls and dialog boxes.

A *control* is a window that an application uses to obtain a specific piece of information from the user, such as the name of a file to open or the desired point size of a text selection. Applications also use controls to obtain information needed to control a particular feature of an application. For example, a word-processing application typically provides a control to let the user turn word wrapping on and off. For more information, see Windows Controls.

Controls are always used in conjunction with another window—typically, a dialog box. A *dialog box* is a window that contains one or more controls. An application uses a dialog box to prompt the user for input needed to complete a command. For example, an application that includes a command to open a file would display a dialog box that includes controls in which the user specifies a path and file name. Dialog boxes do not typically use the same set of window components as does a main window. Most have a title bar, a window menu, a border (non-sizing), and a client area, but they typically do not have a menu bar, minimize and maximize buttons, or scroll bars. For more information, see Dialog Boxes.

A *message box* is a special dialog box that displays a note, caution, or warning to the user. For example, a message box can inform the user of a problem the application has encountered while performing a task. For more information, see Message Boxes.

11.4.1 Edit Control

An edit control is selected and receives the input focus when a user clicks the mouse

inside it or presses the TAB key. After it is selected, the edit control displays its text (if any) and a flashing caret that indicates the insertion point. The user can then enter text, move the insertion point, or select text to be edited by using the keyboard or the mouse. An edit control can send notification messages to its parent window in the form of WM_COMMAND messages.

11.4.2 Static controls

- A static control is a control that enables an application to provide the user with informational text and graphics that typically require no response.

- Applications often use static controls to label other controls or to separate a group of controls. Although static controls are child windows, they cannot be selected. Therefore, they cannot receive the keyboard focus.

Example of static control is a text in message box.

11.4.3 Scroll Bar

A window in a Win32®-based application can display a data object, such as a document or a bitmap that is larger than the window's client area. When provided with a scroll bar, the user can scroll a data object in the client area to bring into view the portions of the object that extend beyond the borders of the window.
Scroll bar is of two types. Horizontal Scroll bars and Vertical Scroll bar.

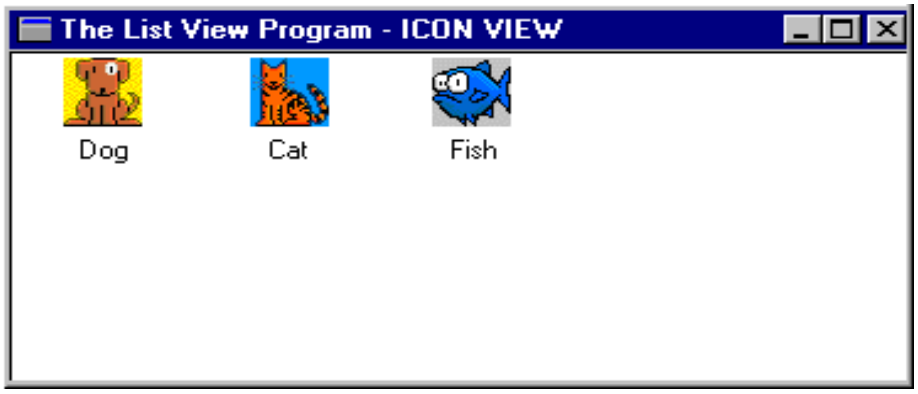
11.5 Common Controls

The common controls are a set of windows that are implemented by the common control library, which is a dynamic-link library (DLL) included with the Microsoft® Windows® operating system. Like other control windows, a common control is a child window that an application uses in conjunction with another window to perform I/O tasks.

Common controls are of these types.

- Date Time Picker Control.

- List View Control.



11.6 Other user Interface Elements

The following are the user interface elements used in Windows.

- Cursors (Mouse shape)
- Icons (Windows Desktop Icons)
- Bitmaps (Images with RGB color values.)
- Accelerators (CTRL + S) Short Key combinations.

11.7 Windows Messages (brief description)

The following are the some of the windows messages

- WM_CLOSE
- WM_COMMAND
- WM_CREATE
- WM_DESTROY
- WM_ENABLE
- WM_LBUTTONDOWN
- WM_PAINT
- WM_RBUTTONDOWN
- WM_SYSCOMMAND
- WM_QUIT
- WM_SETTEXT

11.7.1 WM_SYSCOMMAND

A window receives this message when the user chooses a command from the window menu (formerly known as the system or control menu) or when the user chooses the maximize button, minimize button, restore button, or close button.

wParam:

This parameter specifies the type of system command requested. This parameter can be one of the following values.

```
SC_MAXIMIZE
SC_MINIMIZE
SC_CLOSE
SC_RESTORE
SC_MAXIMIZE
```

lParam

This parameter is the low-order word specifies the horizontal position of the cursor, in screen coordinates, if a window menu command is chosen with the mouse. Otherwise, this parameter is not used. The high-order word specifies the vertical position of the cursor, in screen coordinates, if a window menu command is chosen with the

mouse. This parameter is -1 if the command is chosen using a system accelerator, or zero if using a mnemonic.

The Window Procedure (Switch Only)

```
case WM_SYSCOMMAND:
{
	wParam &= 0xFFF0;  // lower 4-bits used by system

	switch(wParam)
	{
		case SC_MAXIMIZE:
		wParam = SC_MINIMIZE;//we handle this message and change it to
		//SC_MINIMIZE

		return DefWindowProc(hWnd, message, wParam, lParam);

		case SC_MINIMIZE:
		wParam = SC_MAXIMIZE; //we handle this message and change it to
		//SC_MAXIMIZE

		return DefWindowProc(hWnd, message, wParam, lParam);

		case SC_CLOSE:
		if(MessageBox(hWnd, "Are you sure to quit?",
			"Please Confirm", MB_YESNO) == IDYES)
		DestroyWindow(hWnd);
		break;

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

	break;

	case WM_DESTROY: PostQuitMessage(0); break;

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

Swap Minimize/Maximize

Application swap buttons Maximize and Minimize with each other, as it is described in windows procedure.

Summary

In this lecture, we learnt more about windows, its hierarchy and types. We learnt about windows types, these are owned windows and child windows. We must keep this point in mind that owned windows and child windows have different concepts. Owned windows have the handle of their owner windows, these handle make a chain of owned windows. We read about the behavior of owned windows and owner windows. We knew that if we bring some change to owner window then the owned windows will response on some changes like minimize and destroying operations. We also knew about child windows that these are the part of its parent's client area. After we knew about threads and their types, threads are two types one is User interface thread and second is working thread. UI (User interface) thread is attached with user interfaces like windows, messages and dialog boxes. We gained a little knowledge about controls. And after that we learnt how to make a windows procedure with responses system menu including close, maximize and minimize button.

Exercise

1. Write down a code which able to
   i. Create window on screen with default coordinates
   ii. Show Edit control on top left corner in the client area.
   iii. Display button besides the edit control, which contains text 'Show text on client area'
   iv. After pressing button, text must display on the client area.
   v. And when the user closes the application, it must show message box which will be containing 'Thanks for using my application'

Chapter 12

Window Classes

12.1 System classes

Up till now, we have been registering window classes before creating a window. A number of window classes are pre-registered / pre-coded in Windows, and their window procedures are also pre-written.

A system class is a window class registered by the system. Many system classes are available for all processes to use, while others are used only internally by the system. Because the system registers these classes, a process cannot destroy them.

Microsoft Windows NT/Windows 2000/Windows XP: The system registers the system classes for a process, the first time one of its threads calls a User or a Windows Graphics Device Interface (GDI) function.

There are two types of System Window Classes.

1. Those which can be used by the user processes.
2. Those which can only be used by the system

The following table describes the system classes that are available for use by all processes.

Class	Description
Button	The class for a button.
ComboBox	The class for a combo box.
Edit	The class for an edit control.
ListBox	The class for a list box.
MDIClient	The class for an MDI client window.
ScrollBar	The class for a scroll bar.
Static	The class for a static control.

The following table describes the system classes that are available only for use by the system.

Class	Description
ComboLBox	The class for the list box contained in a combo box.
DDEMLEvent	Windows NT/Windows 2000/Windows XP: The class for Dynamic Data Exchange Management Library (DDEML) events.
Message	Windows 2000/Windows XP: The class for a message-only window.
#32768	The class for a menu.
#32769	The class for the desktop window.

#32770	The class for a dialog box.
#32771	The class for the task switch window.
#32772	Windows NT/Windows 2000/Windows XP: The class for icon titles.

Because these classes are pre-registered, that's why we do not call *RegisterClass* or do not need to register the window class before creating such a window.

12.2 Styles of System Classes

The Following are the styles of some of the system window classes

Button Styles

BS_3STATE
This style creates a button that is the same as a check box, except that the check box can be grayed, as well as, checked or cleared. Use the grayed state to show that the state of the check box is not determined.

BS_AUTO3STATE
This style creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and cleared.

BS_AUTOCHECKBOX
This style creates a button that is the same as a check box, except that the check state automatically toggles between checked and cleared, each time the user selects the check box.

BS_AUTORADIOBUTTON
This style creates a button that is the same as a radio button, except that when the user selects it, the system automatically sets the button's check state to checked and automatically sets the check state for all other buttons in the same group to cleared.

BS_CHECKBOX
This style creates a small, empty check box with text. By default, the text is displayed to the right of the check box. To display the text to the left of the check box, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style).

BS_DEFPUSHBUTTON
This style creates a push button that behaves like a BS_PUSHBUTTON style, but it has also a heavy black border. If the button is in a dialog box, the user can select the button by pressing the ENTER key, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely (default) option.

BS_GROUPBOX
This style creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper left corner.

BS_LEFTTEXT
This style places text on the left side of the radio button or check box when combined with a radio button or check box style. This style is same as the BS_RIGHTBUTTON style.

BS_OWNERDRAW
This style creates an owner-drawn button. The owner window receives a WM_DRAWITEM message when a visual aspect of the button has changed. Do not combine the BS_OWNERDRAW style with any other button styles.

BS_PUSHBUTTON
This style creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button.

BS_RADIOBUTTON
This style creates a small circle with text. By default, the text is displayed to the right of the circle. To display the text to the left of the circle, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style). Use radio buttons for groups of related, but mutually exclusive choices.

BS_USERBUTTON
This style has become obsolete, but provided for compatibility with 16-bit versions of Windows. Applications should use BS_OWNERDRAW instead.

BS_BITMAP
This style specifies that the button displays a bitmap.

BS_BOTTOM
This style places text at the bottom of the button rectangle.

BS_CENTER
This style centers text horizontally in the button rectangle.

BS_ICON
This style specifies that the button displays an icon.

BS_FLAT
This style specifies that the button is two-dimensional; it does not use the default shading to create a 3-D image.

BS_LEFT
This style Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button.

BS_MULTILINE
This style wraps the button text to multiple lines if the text string is too long to fit on a single line in the button rectangle.

BS_NOTIFY
This style enables a button to send BN_KILLFOCUS and BN_SETFOCUS to help notification messages to its parent window.

Note that buttons send the BN_CLICKED notification message regardless of whether it has this style. To get BN_DBLCLK notification messages, the button must have the BS_RADIOBUTTON or BS_OWNERDRAW style.

BS_PUSHLIKE
This style makes a button (such as a check box, three-state check box, or radio button) and look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.

BS_RIGHT
This style right-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is right justified on the right side of the check box or radio button.

BS_RIGHTBUTTON
This style positions a radio button's circle or a check box's square on the right side of the button rectangle. This is same as the BS_LEFTTEXT style.

BS_TEXT
This style specifies that the button displays text.

BS_TOP
This style places text at the top of the button rectangle.

BS_TYPEMASK
Microsoft Windows 2000: A composite style bit that results from using the OR operator on BS_* style bits. It can be used to mask out valid BS_* bits from a given bitmask. Note that this is out of date and does not correctly include all valid styles. Thus, you should not use this style.

BS_VCENTER
This style places text in the middle (vertically) of the button rectangle.

12.3 Creating *Button* Window Class (*Example*)

For button, we will use our well known API *CreateWindow* to create a button.

```
hWnd = CreateWindow( "BUTTON", "Virtual University", BS_RADIOBUTTON |
WS_VISIBLE | WS_OVERLAPPEDWINDOW | WS_CAPTION, 50, 50, 200, 100,
NULL, NULL, hInstance, NULL);
```

This button has no parent. If you want to place this button on any window, you should provide *hWndParent* member with parent Window handle and add WS_CHILD style in its dwStyle member.

12.4 Get and Set Window Long

The **SetWindowLong** function changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

```
LONG SetWindowLong(
        HWND hWnd,              // handle to window
        int nIndex,             // offset of value to set
        LONG dwNewLong          // new value
);
```

hWnd

Handle to the window and, indirectly, the class to which the Window belongs.

nIndex

This member specifies the zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of an integer. To set any other value, specify one of the following values.

GWL_EXSTYLE, Sets a new extended window style.
GWL_STYLE, Sets a new Window Style

GWL_WNDPROC :Sets a new address for the window procedure.

In Windows NT/2000/XP, You cannot change this attribute if the window does not belong to the same process as the calling thread.

GWL_HINSTANCE: Sets a new application instance handle.
GWL_ID, Sets a new identifier of the window.

GWL_USERDATA: Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.

The following values are also available when the *hWnd* parameter identifies a dialog box.

DWL_DLGPROC: Sets the new address of the dialog box procedure.
DWL_MSGRESULT: Sets the return value of a message processed in the dialog box procedure.

DWL_USER: Sets new extra information that is private to the application, such as handles or pointers.

dwNewLong

This style specifies the replacement value.

```
LONG GetWindowLong(
        HWND hWnd,          // handle to window
        int  nIndex         // offset of value to retrieve
        );
```

Certain window data is cached, so changes you make using **SetWindowLong** will not take effect until you call the *SetWindowPos* function. Specifically, if you change any of the frame styles, you must call **SetWindowPos** with the SWP_FRAMECHANGED flag for the cache to be updated properly.
If you use **SetWindowLong** with the GWL_WNDPROC index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the WindowProc callback function.
If you use **SetWindowLong** with the DWL_MSGRESULT index to set the return value for a message processed by a dialog procedure, you should return TRUE directly afterwards. Otherwise, if you call any function that results in your dialog procedure receiving a window message, the nested window message could overwrite the return value you set using DWL_MSGRESULT.
Calling **SetWindowLong** with the GWL_WNDPROC index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class, created by another process. The **SetWindowLong** function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling CallWindowProc. This allows the application to create a chain of window procedures.
Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the WNDCLASSEX structure used with the *RegisterClassEx* function.

12.5 Sub-Classing

Sub-classing allows you to change the behavior of an existing window, typically a control, by inserting a message map to intercept the window's messages. For example, suppose you have a dialog box with an edit control that you want to accept only non-numeric characters. You could do this by intercepting WM_CHAR messages destined for the edit control and discarding any messages indicating that a numeric character has been entered.

Subclassing is a technique that allows an application to intercept messages destined for another window. An application can augment, monitor, or modify the default behavior of a window by intercepting messages meant for another window. Sub-classing is an effective way to change or extend the behavior of a window without redeveloping the window. *Subclassing* the default control window classes (button controls, edit controls, list controls, combo box controls, static controls, and scroll bar controls) is a convenient way to obtain the functionality of the control and to modify its behavior. For example, if a multi-line edit control is included in a dialog box and the user presses the ENTER key, the dialog box closes. By *subclassing* the edit control, an application can have the edit control insert a carriage return and line feed into the text without exiting the dialog box. An edit control does not have to be developed specifically for the needs of the application

The Basics

The first step in creating a window is registering a window class by filling a **WNDCLASS** structure and calling **RegisterClass**. One element of the **WNDCLASS** structure is the address of the window procedure for this window class. When a window is created, the 32-bit versions of the Microsoft Windows operating system take the address of the window procedure in the **WNDCLASS** structure and copy it to the new window's information structure. When a message is sent to the window, Windows calls the window procedure through the address in the window's information structure. To subclass a window, you substitute a new window procedure that receives all the messages meant for the original

window by substituting the window procedure address with the new window procedure address.
When an application subclasses a window, it can take three actions with the message: (1) pass the message to the original window procedure; (2) modify the message and pass it to the original window procedure; (3) not pass the message.
The application subclassing a window can decide when to react to the messages it receives. The application can process the message before, after, or both before and after passing the message to the original window procedure.

Types of Subclassing

The two types of subclassing are *instance subclassing* and *global subclassing*.

- Instance subclassing is subclassing an individual window's information structure. With instance subclassing, only the messages of a particular window instance are sent to the new window procedure.
- Global subclassing is replacing the address of the window procedure in the **WNDCLASS** structure of a window class. All subsequent windows created with this class have the substituted window procedure's address. Global subclassing affects only windows created after the subclass has occurred. At the time of the subclass, if any windows of the window class that is being globally subclassed exist, the existing windows are not affected by the global subclass. If the application needs to affect the behavior of the existing windows, the application must subclass each existing instance of the window class.

Win32 Subclassing Rules

Two sub-classing rules apply to instance and global sub-classing in Win32.

Subclassing is allowed only within a process. An application cannot subclass a window or class that belongs to another process.

The reason for this rule is simple: Win32 processes have separate address spaces. A window procedure has an address in a particular process. In a different process, that address does not contain the same window procedure. As a result, substituting an address from one process with an address from another process does not provide the desired result, so the 32-bit versions of Windows do not allow this substitution (that is, subclassing from a different process) to take place. The SetWindowLong and SetClassLong functions prevent this type of subclassing. You can not subclass a window or class that is in another process. End of story.

One way to add subclassing code into another process is much more complicated: It involves using the OpenProcess, WriteProcessMemory, and CreateRemoteThread functions to inject code into the other process. I don't recommend this method and won't go into any details on how to do it. For developers who insist on using this method,

The subclassing process may not use the original window procedure address directly.

In Win16, an application could use the window procedure address returned from *SetWindowLong* or *SetClassLong* to call the procedure directly. After all, the return value is simply a pointer to a function, so why not just call it? In Win32, this is a definitive no-no.

The value returned from *SetWindowLong* and *GetClassLong* may not be a pointer to the previous window procedure at all. Win32 may return a pointer to a data structure that it can use to call the actual window procedure. This occurs in Windows NT when an application subclasses a Unicode window with a non-Unicode window procedure, or a non-Unicode window with a Unicode window procedure. In this case, the operating system must perform a translation between Unicode and ANSI for the messages the window receives. If an application uses the pointer to this structure to directly call the window procedure, the application will immediately generate an exception. The only way to use the window procedure address returned from SetWindowLong or SetClassLong is as a parameter to CallWindowProc.

Instance Subclassing

The *SetWindowLong* function is used to subclass an instance of a window. The application must have the address of the subclass function. The subclass function is the function that receives the messages from Windows and passes the messages to the original window procedure. The subclass function must be exported in the application's or the DLL's module definition file.
The application *subclassing* the window calls *SetWindowLong* with the handle to the window the application wants to subclass, the GWL_WNDPROC option (defined in WINDOWS.H), and the address of the new subclass function. *SetWindowLong* returns a DWORD, which is the address of the original window procedure for the window. The application must save this address to pass the intercepted messages to the original window procedure and to remove the subclass from the window. The application passes the messages to the original window procedure by calling CallWindowProc with the address of the original window procedure and the hWnd, Message, wParam, and lParam parameters used in Windows messaging. Usually, the application simply passes the arguments it receives from Windows to CallWindowProc.
The application also needs the original window procedure address for removing the subclass from the window. The application removes the subclass from the window by calling SetWindowLong again. The application passes the address of the original window procedure with the GWL_WNDPROC option and the handle to the window being subclassed.

The following code subclasses and removes a subclass to an edit control:

```
LONG FAR PASCAL SubClassFunc(HWND hWnd,UINT Message,WPARAM wParam,
    LONG lParam);

FARPROC lpfnOldWndProc;
HWND hEditWnd;

//
// Create an edit control and subclass it.
// The details of this particular edit control are not important.
//
hEditWnd = CreateWindow("EDIT", "EDIT Test",
                    WS_CHILD | WS_VISIBLE | WS_BORDER ,
                    0, 0, 50, 50,
                    hWndMain,
                    NULL,
```

```
                    hInst,
                    NULL);
//
// Now subclass the window that was just created.
//
lpfnOldWndProc = (FARPROC)SetWindowLong(hEditWnd,
              GWL_WNDPROC, (DWORD) SubClassFunc);
.
.
.
//
// Remove the subclass for the edit control.
//
SetWindowLong(hEditWnd, GWL_WNDPROC, (DWORD) lpfnOldWndProc);

//
// Here is a sample subclass function.
//
LONG FAR PASCAL SubClassFunc(   HWND hWnd,
            UINT Message,
            WPARAM wParam,
            LONG lParam)
{
  //
  // When the focus is in an edit control inside a dialog box, the
  //  default ENTER key action will not occur.
  //

  if ( Message == WM_GETDLGCODE )
     return DLGC_WANTALLKEYS;

  return CallWindowProc(lpfnOldWndProc, hWnd, Message, wParam,
                    lParam);
}
```

12.6 Get or Set ClassLong

The GetClassLong() function retrieves the specified 32-bit (LONG) value from the WNDCLASS structure associated with the specified window. This will can be back ground brush, handle to instance, handle to windows procedure and handle to Icon etc.

```
LONG SetClassLong(
HWND hWnd,          // handle to window
int nIndex,         // offset of value to set n  LONG dwNewLong // new value
);
```

Parameters

hWnd

Handle to the window and, indirectly, the class to which the window belongs.

nIndex

This member specifies the 32-bit value to replace. To set a 32-bit value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third 32-bit integer. To set any other value from the **WNDCLASSEX** structure, specify one of the following values.

GCL_CBCLSEXTRA: Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA: Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated.

GCL_HBRBACKGROUND: Replaces a handle to the background brush associated with the class.

GCL_HCURSOR: Replaces a handle to the cursor associated with the class.
GCL_HICON: Replaces a handle to the icon associated with the class.

GCL_HICONSM: Replace a handle to the small icon associated with the class.

GCL_HMODULE: Replaces a handle to the module that registered the class.

GCL_MENUNAME: Replaces the address of the menu name string. The string identifies the menu resource associated with the class.

GCL_STYLE: Replaces the window-class style bits.

GCL_WNDPROC: Replaces the address of the window procedure associated with the class.

dwNewLong

This member specifies the replacement value.

Return Value

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the value was not previously set, the return value is zero.
If the function fails, the return value is zero. To get extended error information, call *GetLastError*.

```
LONG GetClassLong(
HWND hWnd,          // handle to window
int nIndex          // offset of value to retrieve
);
```

If you use the **SetClassLong** function and the GCL_WNDPROC index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the *WindowProc* callback function.

Calling **SetClassLong** with the GCL_WNDPROC index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.
Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the **WNDCLASSEX** structure used with the *RegisterClass* function.
Use the **SetClassLong** function with care. For example, it is possible to change the background color for a class by using **SetClassLong**, but this change does not immediately repaint all windows belonging to the class.

Difference between SetWindowLong and SetClassLong

- In SetWindowLong(), behavior of a single window is modified.
- In SetClassLong(), behavior of the window class is modified.

12.7 Sub-Classing (*Elaboration*)

We elaborate sub-classing by using following examples.

```
DLGPROC oldWindowProc;
hWnd = CreateWindow("BUTTON", "Virtual University",    BS_AUTOCHECKBOX |
WS_VISIBLE |    WS_OVERLAPPEDWINDOW,
50, 50, 200, 100,
 NULL, NULL, hInstance, NULL);

oldWindowProc = (WNDPROC) SetWindowLong ( hWnd,
 GWL_WNDPROC, (LONG) myWindowProc);

while(GetMessage(&msg, NULL, 0, 0)  >  0)
{
        DispatchMessage(&msg);
}

return msg.wParam;
```

New Window Procedure

```
LRESULT CALLBACK myWindowProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
         switch (message)
        {
        case WM_LBUTTONDOWN:
                MessageBox(hWnd, "Left mouse button pressed.", "Message", MB_OK);
                 DestroyWindow(hWnd);
                break;

        case WM_DESTROY:
                PostQuitMessage(0);
```

```
            break;

            default:
            return CallWindowProc(oldWindowProc, hWnd, message,
            wParam, lParam);
        }

return 0;
}
```

12.8 Supper-Classing

Super-classing defines a class that adds new functionality to a predefined window class, such as the button or list box controls.

Superclassing involves creating a new class that uses the window procedure of an existing class for basic functionality.

Super-Classing (*Example*)

Super-classing defines a class that adds new functionality to a predefined window class, such as the button or list box controls.

The following example defines a new class with partly or wholly modified behavior of a pre-defined window class.

```
DLGPROC oldWindowProc; // Global variable
WNDCLASS wndClass;
GetClassInfo(hInstance, “BUTTON”, &wndClass);
```

GetClassInfo API gets the information about class. Information includes windows style, procedure, background brush, icon and cursors.

```
WndClass.hInstance = hInstance;
wndClass.lpszClassName = “BEEPBUTTON”;
OldWindowProc = wndClas.lpfnWndProc;
wndClas.lpfnWndProc = myWindowProc;
```

After getting class information we fill the new window class and register it again by using *RegisterClass* API

```
if(! RegisterClass( &wndClass ) )
{
        return 0;
}
```

After registering new window class with the new procedure, create a window with the new register class name. This registered class name will be different from the old registered class name.

```
hWnd = CreateWindow(“BEEPBUTTON", "Virtual University",       WS_VISIBLE |
WS_OVERLAPPEDWINDOW,
50, 50, 200, 100,
NULL, NULL, hInstance, NULL);

noldWindowProc = (WNDPROC)SetWindowLong(hWnd, GWL_WNDPROC,
(LONG)myWindowProc);
```

```
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
        if(msg.message == WM_LBUTTONUP)
        DispatchMessage(&msg);
}

return msg.wParam
```

New Window Procedure

This new windows procedure *myWindowProc* will call the old window procedure after its normal message processing.

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

         switch (message)
        {
        case WM_LBUTTONDOWN:
        MessageBeep(0xFFFFFFFF);
        Break;
        default:
        return CallWindowProc(oldWindowProc, hWnd, message, wParam, lParam);

        }

        return 0;
}
```

Tips: After implementation of Sub-classing or Super Classing don’t forget to call window procedure function.

Summary

In this lecture, we learnt about system windows classes. System window classes include buttons, combo boxes, list box, etc. We studied about Button System Window class. We discussed how to change windows attributes by using SetWindowLong and GetWindowLong APIs. Using SetWindowLong and GetWindowLong, we can also make a new procedure and change the message behavior of a window. Using SetClassLong and GetClassLong, we can change one of the attributes of a registered class. Changing class values will effect the change for every window that is using this class. This is called sub-classing. We also knew about Super-classing in which we register new window class by using the properties of previous window class.

Chapter 13

Graphics Device Interface

13.1 GDI (Graphics Device Interface)

In previous lectures we have got some understanding about GDI. In this lecture, we will take a detail look on Graphics Device Interface and its Device independency.

The graphical component of the Microsoft® Windows™ graphical environment is the graphics device interface (GDI). It communicates between the application and the device driver, which performs the hardware-specific functions that generate output. By acting as a buffer between applications and output devices, GDI presents a device-independent view of the world for the application while interacting in a device-dependent format with the device.

In the GDI environment there are two working spaces—the logical and the physical. Logical space is inhabited by applications; it is the "ideal" world in which all colors are available, all fonts scale, and output resolution is phenomenal. Physical space, on the other hand, is the real world of devices, with limited color, strange output formats, and differing drawing capabilities. In Windows, an application does not need to understand the quirkiness of a new device. GDI code works on the new device if the device has a device driver.

GDI concepts mapped between the logical and the physical are objects (pens, brushes, fonts, palettes, and bitmaps), output primitives, and coordinates.

Objects are converted from logical objects to physical objects using the realization process. For example, an application creates a logical pen by calling *CreatePen* with the appropriate parameters. When the logical pen object is selected into a device context (DC) using *SelectObject*, GDI realizes the pen into a physical pen object that is used to communicate with the device. GDI passes the logical object to the device, and the device creates a device-specific object containing device-specific information. During realization, requested (logical) color is mapped to available colors, fonts are matched to the best available fonts, and patterns are prepared for output. Font selection is more complex than other realizations, and GDI, not the driver, performs most of the realization work. Similarly, palette realization (done at RealizePalette time as opposed to SelectPalette time) is done entirely within GDI. Bitmaps are an exception to the object realization process; although they have the device-independent bitmap (DIB) logical form, bitmap objects are always device specific and are never actually realized.

Output primitives are similarly passed as "logical" requests (to stretch the definition) to the device driver, which draws the primitive to the best of its ability and resolution. If the driver cannot handle a certain primitive—for example, it cannot draw an ellipse—GDI simulates the operation. For an Ellipse call, GDI calculates a polygon that represents a digitized ellipse. The resulting polygon can then be simulated as a *polyline* and a series of *scanline* fills if the device cannot draw polygons itself. The application, though, does not care what system component does the actual work; the primitive gets drawn.

An application can set up for itself any logical coordinate system, using SetMapMode, *SetWindowExt*, *SetWindowOrg*, *SetViewportExt*, and *SetViewportOrg*. In GDI that coordinate system is mapped to the device coordinate system, in which one unit equals one pixel and (0,0) defines the topmost, leftmost pixel on the output surface. The device driver sees only coordinates in its own space, whereas the application operates only in a coordinate space of its own, disregarding the physical pixel layout of the destination.

By maintaining the two separate but linked spaces, logical for the applications and physical for the devices, GDI creates a device-independent interface. Applications that make full use of the logical space and avoid device-specific assumptions can expect to operate successfully on any output device.

13.2 GDI Objects and its API's

This topic will discuss Graphics Device Objects and the API's used to create, select, get, release, draw and delete GDI objects.

GDI objects Creation

Each type of object has a routine or a set of routines that is used to create that object.

Pens are created with the **CreatePen** and the **CreatePenIndirect** functions. An application can use either function to define three pen attributes: style, width, and color. The background mode during output determines the color (if any) of the gaps in any nonsolid pen. The PS_INSIDEFRAME style allows dithered wide pens and a different mechanism for aligning the pen on the outside of filled primitives.

Brushes are created with the CreateSolidBrush, CreatePatternBrush, CreateHatchBrush, CreateDIBPatternBrush, and CreateBrushIndirect functions. Unlike other objects, brushes have distinct types that are not simply attributes. Hatch brushes are special because they use the current background mode (set with the SetBkMode function) for output.

Fonts are created with the CreateFont and CreateFontIndirect functions. An application can use either function to specify the 14 attributes that define the desired size, shape, and style of the logical font.

Bitmaps are created with the CreateBitmap, CreateBitmapIndirect, CreateCompatibleBitmap, and CreateDIBitmap functions. An application can use all four functions to specify the dimensions of the bitmap. An application uses the CreateBitmap and CreateBitmapIndirect functions to create a bitmap of any color format. The CreateCompatibleBitmap and CreateDIBitmap functions use the color format of the device context. A device supports two bitmap formats: monochrome and device-specific color. The monochrome format is the same for all devices. Using an output device context (DC) creates a bitmap with the native color format; using a memory DC creates a bitmap that matches the color format of the bitmap currently selected into that DC. (The DCs color format changes based on the color format of the currently selected bitmap.)

Palette objects are created with the CreatePalette function. Unlike pens, brushes, fonts, and bitmaps, the logical palette created with this function can be altered later with the SetPaletteEntries function or, when appropriate, with the AnimatePalette function.

Regions can be created with the CreateRectRgn, CreateRectRgnIndirect, CreateRoundRectRgn, CreateEllipticRgn, CreateEllipticRgnIndirect, CreatePolygonRgn, and CreatePolyPolygonRgn functions. Internally, the region object that each function creates is composed of a union of rectangles with no vertical overlap. Regions created based on nonrectangular primitives simulate the complex shape with a series of rectangles, roughly corresponding to the scanlines that would be used to paint the primitive. As a result, an elliptical region is stored as many short rectangles (a bit fewer than the height of the ellipse), which leads to more cumbersome and slower region calculations and clipping. Coordinates used for creating regions are not specified in logical units as they are for other objects. The graphics device interface (GDI) uses them without transformation. GDI translates coordinates for clip regions to be relative to the upper-left corner of a window when applicable. Region objects can be altered with the CombineRgn and OffsetRgn functions.

What Happens During Selection

Selecting a logical object into a DC involves converting the logical object into a physical object that the device driver uses for output. This process is called realization. The principle is the same for all objects, but the actual operation is different for each object type. When an application changes the logical device mapping of a DC (by changing the mapping mode or the window or viewport definition), the system re-realizes the currently selected pen and font before they are used the next time. Changing the DCs coordinate mapping scheme alters the physical interpretation of the logical pens width and the logical fonts height and width by essentially reselecting the two objects.

Pens are the simplest of objects. An application can use three attributes to define a logical pen—width, style, and color. Of these, the width and the color are converted from logical values to physical values. The width is converted based on the current mapping mode (a width of 0 results in a pen with a one-pixel width regardless of mapping mode), and the color is mapped to the closest color the device can represent. The physical color is a solid color (that is, it has no dithering). If the pen style is set to PS_INSIDEFRAME and the physical width is not 1, however, the pen color can be dithered. The pen style is recorded in the physical object, but the information is not relevant until the pen is actually used for drawing.

Logical brushes have several components that must be realized to make a physical brush. If the brush is solid, a physical representation must be calculated by the device driver; it can be a dithered color (represented as a bitmap with multiple colors that when viewed by the human eye approximates a solid color that cannot be shown as a single pixel on the device), or it can be a solid color. Pattern brush realization involves copying the bitmap that defines the pattern and, for color patterns, ensuring that the color format is compatible with the device. Usually, the device driver also builds a monochrome version of a color pattern for use with monochrome bitmaps. With device-independent bitmap (DIB) patterns, GDI converts the DIB into a device-dependent bitmap using SetDIBits

before it passes a normal pattern brush to the device driver. The selection of a DIB pattern brush with a two-color DIB and DIB_RGB_COLORS into a monochrome DC is a special case; GDI forces the color table to have black as index 0 and white as index 1 to maintain foreground and background information. The device driver turns hatched brushes into pattern brushes using the specified hatch scheme; the foreground and background colors at the time of selection are used for the pattern. All brush types can be represented at the device-driver level as bitmaps (usually 8-by-8) that are repeatedly blted as appropriate. To allow proper alignment of these bitmaps, GDI realizes each physical brush with a brush origin. The default origin is (0,0) and can be changed with the SetBrushOrg function (discussed in more detail below).

The GDI component known as the font mapper examines every physical font in the system to find the one that most closely matches the requested logical font. The mapper penalizes any font property that does not match. The physical font chosen is the one with the smallest penalty. The possible physical fonts that are available are raster, vector, TrueType fonts installed in the system, and device fonts built into or downloaded to the output device. The logical values for height and width of the font are converted to physical units based on the current mapping mode before the font mapper examines them.

Selecting a bitmap into a memory DC involves nothing more than performing some error checking and setting a few pointers. If the bitmap is compatible with the DC and is not currently selected elsewhere, the bits are locked in memory and the appropriate fields are set in the DC. Most GDI functions treat a memory DC with a selected bitmap as a regular device DC; only the device driver acts differently, based on whether the output destination is memory or the actual device. The color format of the bitmap defines the color format of the memory DC. When a memory DC is created with CreateCompatibleDC, the default monochrome bitmap is selected into it, and the color format of the DC is monochrome. When an appropriate color bitmap (one whose color resolution matches that of the device) is selected into the DC, the color format of the DC changes to reflect this event. This behavior affects the result of the CreateCompatibleBitmap function, which creates a monochrome bitmap for a monochrome DC and a color bitmap for a color DC.

Palettes are not automatically realized during the selection process. The RealizePalette function must be explicitly called to realize a selected palette. If a palette is realized on a nonpalette device, nothing happens. On a palette device, the logical palette is color-matched to the hardware palette to get the best possible matching. Subsequent references to a color in the logical palette are mapped to the appropriate hardware palette color.

Nothing is actually realized when a clip region is selected into a DC. A copy of the region is made and placed in the DC. This new clip region is then intersected with the current visible region (computed by the system and defining how much of the window is visible on the screen), and the DC is ready for drawing. Calling SelectObject with a region is equivalent to using the SelectClipRgn function.

Memory Usage

The amount of memory each object type consumes in GDIs heap and in the global memory heap depends on the type of the object.

This topic is discussed in Microsoft Documentation 2003 Release.

The following table describes memory used for storing logical objects.

Object type	GDI heap use (in bytes)	Global memory use (in bytes)
Pen	10 + sizeof(LOGPEN)	0
Brush	10 + sizeof(LOGBRUSH) + 6	0
pattern brush	same as brush + copy of bitmap	
Font	10 + sizeof(LOGFONT)	0
Bitmap	10 + 18	32 + room for bits
Palette	10 + 10	4 + (10 * num entries)
rectangular region	10 + 26	0
solid complex region	rect region + (6 * (num scans –1))	0
region with hole	region + (2 * num scans with hole)	0

When an object is selected into a DC, it may have corresponding physical (realized) information that is stored globally and in GDIs heap. The table below details that use. The size of realized versions of objects that devices maintain is determined by the device.

Object type	GDI heap use (in bytes)	Global memory use
pen	10 + 8 + device info	0
brush	10 + 14 + device info	0
font	55 (per realization)	font data (per physical font)
bitmap	0	0
palette	0	0
region	intersection of region with visible region	0

As a result of the font caching scheme, several variables determine how much memory a realized font uses. If two logical fonts are mapped to the same physical font, only one copy of the actual font is maintained. For TrueType fonts, glyph data is loaded only upon request, so the size of the physical font grows (memory permitted) as more characters are needed. When the font can grow no larger, characters are cached to use the available

space. The font data stored for a single physical font ranges from 48 bytes for a hardware font to 120K for a large bitmapped font.

Physical pens and brushes are not deleted from the system until the corresponding object is deleted. The physical object that corresponds to a selected logical object is locked in GDIs heap. (It is unlocked upon deselection.) Similarly, a font "instance" is cached in the system to maintain a realization of a specific logical font on a specific device with a specific coordinate mapping. When the logical font is deleted, all of its instances are removed as well.

When the clip region intersects with the visible region, the resulting intersection is roughly the same size as the initial clip region. This is always the case when the DC belongs to the topmost window and the clip region is within the windows boundary.

Creating vs. Recreating

If an application uses an object repeatedly, should the object be created once and cached by the application, or should the application recreate the object every time it is needed and delete it when that part of the drawing is complete? Creating on demand is simpler and saves memory in GDIs heap (objects do not remain allocated for long). Caching the objects within an application involves more work, but it can greatly increase the speed of object selection and realization, especially for fonts and sometimes for palettes.

The speed gains are possible because GDI caches physical objects. Although realizing a new logical pen or brush simply involves calling the device driver, realizing a logical font involves a cumbersome comparison of the logical font with each physical font available in the system. An application that wants to minimize font-mapping time should cache logical font handles that are expected to be used again. All previous font-mapping information is lost when a logical font handle is deleted; a recreated logical font must be realized from scratch.

Applications should cache palette objects for two reasons (both of which apply only on palette devices). Most importantly, because bitmaps on palette devices are stored based on a specific logical bitmap, using a different palette alters the bitmaps coloration and meaning. The second reason is a speed issue; the foreground realization of a palette is cached by GDI and is not calculated after the first realization. A new foreground realization must be computed from scratch for a newly created palette (or a palette altered by the SetPaletteEntries function or unrealized with the UnrealizeObject function).

Stock Objects

During initialization, GDI creates a number of predefined objects that any application can use. These objects are called stock objects. With the exception of regions and bitmaps, every object type has at least one defined stock object. An application calls the GetStockObject function to get a handle to a stock object, and the returned handle is then used as a standard object handle. The only difference is that no new memory is used because no new object is created. Also, because the system owns the stock objects, an

application is not responsible for deleting the object after use. Calling the DeleteObject function with a stock object does nothing.

Several stock fonts are defined in the system, the most useful being SYSTEM_FONT. This font is the default selected into a DC and is used for drawing the text in menus and title bars. Because this object defines only a logical font, the physical font that is actually used depends on the mapping mode and on the resolution of the device. A screen DC with a mapping mode of MM_TEXT has the system font as the physical font, but if the mapping mode is changed or if a different device is used, the physical font is no longer guaranteed to be the same. A change of behavior for Windows version 3.1 is that a stock font is never affected by the current mapping mode; it is always realized as if MM_TEXT were being used. Note that a font created by an application as a copy of a stock font does not have this immunity to scaling.

No stock bitmap in the system is accessible by means of the GetStockObject function, but GDI uses a default one-by-one monochrome bitmap as a stock object. This default bitmap is selected into a memory DC during creation of that DC. The bitmaps handle can be obtained by selecting a bitmap into a freshly minted memory DC; the return value from the SelectObject function is the stock bitmap.

Error Handling

The two common types of errors associated with objects are failure to create and failure to select. Both are most commonly associated with low-memory conditions.

During the creation process, GDI allocates a block of memory to store the logical object information. When the heap is full, applications cannot create any more objects until some space is freed. Bitmap creation tends to fail not because GDIs heap is full but because available global memory is insufficient for storing the bits themselves. Palettes also have a block of global memory that must be allocated by GDI to hold the palette information. The standard procedure for handling a failed object creation is to use a corresponding stock object in its place, although a failed bitmap creation is usually more limiting. An application usually warns the user that memory is low when an object creation or selection fails.

Out-of-memory conditions can also occur when a physical object is being realized. Realization also involves GDI allocating heap memory, and realizing fonts usually involves global memory as well. If the object was realized in the past for the same DC, new allocation is unnecessary (see the "Creating vs. Recreating" section). If a call to SelectObject returns an error (0), no new object is selected into the DC, and the previously selected object is not deselected.

Another possible error applies only to bitmaps. Attempting to select a bitmap with a color format that does not match the color format of the DC results in an error. Monochrome bitmaps can be selected into any memory DC, but color bitmaps can be selected only into a memory DC of a device that has the same color format. Additionally, bitmaps can be selected only into memory DCs; they cannot be selected into a DC connected to an actual output device or into metafile DCs.

Some object selections do not fail. Selecting a default object (WHITE_BRUSH, BLACK_PEN, SYSTEM_FONT, or DEFAULT_PALETTE stock objects) into a screen DC or into a screen-compatible memory DC does not fail when the mapping mode is set to MM_TEXT. Also, a bitmap with a color format matching a memory DC always successfully selects into that DC. Palette selection has no memory requirements and always succeeds.

Deletion of GDI Objects

All applications should delete objects when they are no longer needed. To delete an object properly, first deselect it from any DC into which it was previously selected. To deselect an object, an application must select a different object of the same type into the DC. Common practice is to track the original object that was selected into the DC and select it back when all work is accomplished with the new object. When a region is selected into a DC with the *SelectObject* or *SelectClipRgn* function, GDI makes a copy of the object for the DC, and the original region can be deleted at will.

```
hNewPen = CreatePen(1, 1, RGB(255, 0, 0));

if (hNewPen)   //if the new pen is selected then ok else do

{      hOldPen = SelectObject(hDC, hNewPen);

}

else

      hOldPen = NULL;              // no selection

      Rectangle(hDC,x,y,ex,ey)       // drawing operations

if (hOldPen)

   SelectObject(hDC, hOldPen);   // deselect hNewPen (if selected)

if (hNewPen)

   DeleteObject(hDC, hNewPen);   // delete pen if created
```

An alternative method is to select in a stock object returned from the *GetStockObject* function. This approach is useful when it is not convenient to track the original object. A DC is considered "clean" of application-owned objects when all currently selected objects are stock objects. The three exceptions to the stock object rule are fonts (only the SYSTEM_FONT object should be used for this purpose); bitmaps, which do not have a stock object defined (the one-by-one monochrome stock bitmap is a constant object that is the default bitmap of a memory DC); and regions, which have no stock object and have no need for one.

```
hNewPen = CreatePen(1, 1, RGB(255, 0, 0));

if (hNewPen)

{

    if (SelectObject(hDC, hNewPen))

    {

        SelectObject(hDC, GetStockObject(BLACK_PEN));

    }

    DeleteObject(hDC, hNewPen);

}
```

Note: The rumor that an application should never delete a stock object is far from the truth. Calling the DeleteObject function with a stock object does nothing. Consequently, an application need not ensure that an object being deleted is not a stock object.

UNREALIZEOBJECT

The *UnrealizeObject* function affects only brushes and palettes. As its name implies, the *UnrealizeObject* function lets an application force GDI to re-realize an object from scratch when the object is next realized in a DC.

The *UnrealizeObject* function lets an application reset the origin of the brush. When a patterned, hatched, or dithered brush is used, the device driver handles it as an eight-by-eight bitmap. During use, the driver aligns a point in the bitmap, known as the brush origin, to the upper-left corner of the DC. The default brush origin is (0,0). If an application wants to change the brush origin, it uses the *SetBrushOrg* function. This function does not change the origin of the current brush; it sets the origin of the brush for the next time that the brush is realized. The origin of a brush that has never been selected into a DC can be set as follows:

```
// Create the brush.

hBrush = CreatePatternBrush(.....);

// Set the origin as needed.

SetBrushOrg(hDC, X, Y);

// Select (and realize) the brush with the chosen origin.

SelectObject(hDC, hBrush);
```

If, on the other hand, the brush is currently selected into a DC, calling the *SetBrushOrg* function alone accomplishes nothing. Because the new origin does not take effect until the brush is realized anew, the application must force this re-realization by using the *UnrealizeObject* function before the brush is reselected into a DC. The following sample code changes the origin of a brush that is initially selected into a DC:

```
// Deselect the brush from the DC.

hBrush = SelectObject(hDC, GetStockObject(BLACK_BRUSH));

// Set a new origin.

SetBrushOrg(hDC, X, Y);

// Unrealize the brush to force re-realization.

UnrealizeObject(hBrush);

// Select (and hence re-realize) the brush.

SelectObject(hDC, hBrush);
```

The UnrealizeObject function can also be called for a palette object, although the effect is a bit more subtle. (As is common with the palette functions, nothing happens on a nonpalette device.) The function forces the palette to be realized from scratch the next time the palette is realized, thereby ignoring any previous mapping. This is useful in situations in which an application expects that the palette will realize differently the next time around, perhaps matching more effectively with a new system palette and not forcing a system palette change. Any bitmaps created with the original realization of the palette are no longer guaranteed to be valid.

Special Cases

Palette objects are selected into DCs using the *SelectPalette* function. The reason for this additional, seemingly identical, function is that palette selection has an additional parameter that defines whether the palette is being selected as a foreground or as a background palette, which affects palette realization on palette devices. Calling the *SelectObject* function with a palette returns an error. Palettes are deleted using the *DeleteObject* function.

A clip region can be selected into a DC by calling either the *SelectClipRgn* or the *SelectObject* function. Both functions perform identically with the exception of selecting a NULL handle in place of a region. *SelectClipRgn* can be used to clear the current clipping state by calling the function as follows:

Note: Parameter description of the API's used above, can be best found from Microsoft site, or contact Virtual University resource.

13.3 GDI from the Driver's Perspective (*for advanced users*)

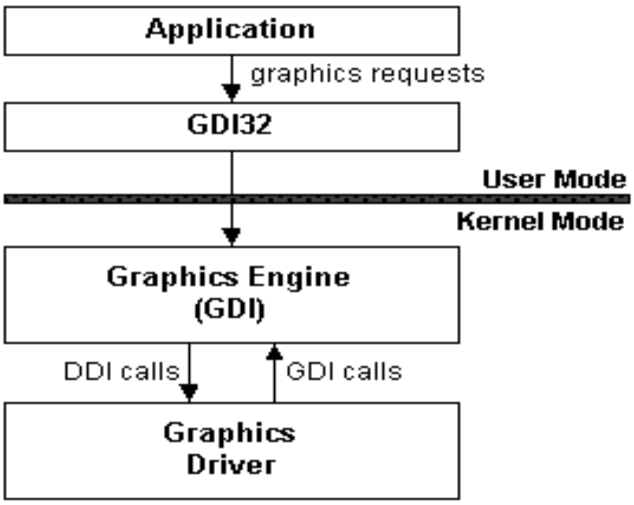
Note: The documentation depicted below is for the advanced readers or those who are interested to know more about GDI driver model. Novice can skip this topic.

GDI is the intermediary support between a Windows NT-based graphics driver and an application. Applications call Win32 GDI functions to make graphics output requests. These requests are routed to kernel-mode GDI. Kernel-mode GDI then sends these requests to the appropriate graphics driver, such as a display driver or printer driver. Kernel-mode GDI is a system-supplied module that cannot be replaced.

GDI communicates with the graphics driver through a set of graphics device driver interface (graphics DDI) functions. These functions are identified by their Drv prefix. Information is passed between GDI and the driver through the input/output parameters of these entry points. The driver must support certain DrvXxx functions for GDI to call. The driver supports GDI's requests by performing the appropriate operations on its associated hardware before returning to GDI.

GDI includes many graphics output capabilities in itself, eliminating the need for the driver to support these capabilities and thereby making it possible to reduce the size of the driver. GDI also exports service functions that the driver can call, further reducing the amount of support the driver must provide. GDI service functions are identified by their Eng prefix, and functions that provide access to GDI-maintained structures have names in the form XxxOBJ_Xxx.

The following figure shows this flow of communication.



Graphics Driver and GDI Interaction

More on GDI and its usage in Win32 environment contact Virtual University Resource.

13.4 Device Context (DC)

We have studied a lot about GDI and its objects and now, we will know how to display GDI objects using Device context.
A device context is a structure that defines a set of graphic objects and their associated attributes, as well as the graphic modes that affect output. The graphic objects include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. The remainder of this section is divided into the following three areas.

Display Device Context Cache

The system maintains a cache of display device contexts that it uses for common, parent, and window device contexts. The system retrieves a device context from the cache whenever an application calls the *GetDC* or *BeginPaint* function; the system returns the DC to the cache when the application subsequently calls the *ReleaseDC* or *EndPaint* function.

There is no predetermined limit on the amount of device contexts that a cache can hold; the system creates a new display device context for the cache if none is available. Given this, an application can have more than five active device contexts from the cache at a time. However, the application must continue to release these device contexts after use. Because new display device contexts for the cache are allocated in the application's heap space, failing to release the device contexts eventually consumes all available heap space. The system indicates this failure by returning an error when it cannot allocate space for the new device context. Other functions unrelated to the cache may also return errors.

Display Device Context Defaults

Upon first creating a display device context, the system assigns default values for the attributes (that is, drawing objects, colors, and modes) that comprise the device context. The following table shows the default values for the attributes of a display device context.

Attribute	Default value
Background color	Background color setting from Control Panel (typically, white).
Background mode	OPAQUE
Bitmap	None
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire window or client area with the update region clipped, as appropriate. Child and pop-up windows in the client area may also be clipped.
Palette	DEFAULT_PALETTE
Current pen position	(0,0)
Device origin	Upper left corner of the window or the client area.
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT
Inter character spacing	0
Mapping mode	MM_TEXT
Pen	BLACK_PEN
Polygon-fill mode	ALTERNATE
Stretch mode	BLACKONWHITE
Text color	Text color setting from Control Panel (typically, black).
Viewport extent	(1,1)
Viewport origin	(0,0)
Window extent	(1,1)
Window origin	(0,0)

An application can modify the values of the display device context attributes by using selection and attribute functions, such as *SelectObject*, *SetMapMode*, and *SetTextColor*.

For example, an application can modify the default units of measure in the coordinate system by using *SetMapMode* to change the mapping mode.

Changes to the attribute values of a common, parent, or window device context are not permanent. When an application releases these device contexts, the current selections, such as mapping mode and clipping region, are lost as the context is returned to the cache. Changes to a class or private device context persist indefinitely. To restore them to their original defaults, an application must explicitly set each attribute.

Common Display Device Context

A common device context is used for drawing in the client area of the window. The system provides a common device context by default for any window whose window class does not explicitly specify a display device context style. Common device contexts are typically used with windows that can be drawn without extensive changes to the device context attributes. Common device contexts are convenient because they do not require additional memory or system resources, but they can be inconvenient if the application must set up many attributes before using them.

The system retrieves all common device contexts from the display device context cache.

An application can retrieve a common device context immediately after the window is created. Because the common device context is from the cache, the application must always release the device context as soon as possible after drawing. After the common device context is released, it is no longer valid and the application must not attempt to draw with it. To draw again, the application must retrieve a new common device context, and continue to retrieve and release a common device context each time it draws in the window. If the application retrieves the device context handle by using the *GetDC* function, it must use the *ReleaseDC* function to release the handle. Similarly, for each *BeginPaint* function, the application must use a corresponding *EndPaint* function.

When the application retrieves the device context, the system adjusts the origin so that it aligns with the upper left corner of the client area. It also sets the clipping region so that output to the device context is clipped to the client area. Any output that would otherwise appear outside the client area is clipped. If the application retrieves the common device context by using *BeginPaint*, the system also includes the update region in the clipping region to further restrict the output.

When an application releases a common device context, the system restores the default values for the attributes of the device context. An application that modifies attribute values must do so each time it retrieves a common device context. Releasing the device context releases any drawing objects the application may have selected into it, so the application need not release these objects before releasing the device context. In all cases, an application must never assume that the common device context retains non default selections after being released.

Private Display Device Context

A *private device context* enables an application to avoid retrieving and initializing a display device context each time the application must draw in a window. Private device contexts are useful for windows that require many changes to the values of the attributes of the device context to prepare it for drawing. Private device contexts reduce the time required to prepare the device context and therefore the time needed to carry out drawing in the window.

An application directs the system to create a private device context for a window by specifying the CS_OWNDC style in the window class. The system creates a unique private device context each time it creates a new window belonging to the class. Initially, the private device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves changes to the device context for the life of the window or until the application makes additional changes.

An application can retrieve a handle to the private device context by using the *GetDC* function any time after the window is created. The application must retrieve the handle only once. Thereafter, it can keep and use the handle any number of times. Because a private device context is not part of the display device context cache, an application need never release the device context by using the *ReleaseDC* function.

The system automatically adjusts the device context to reflect changes to the window, such as moving or sizing. This ensures that any overlapping windows are always properly clipped; that is, no action is required by the application to ensure clipping. However, the system does not revise the device context to include the update region. Therefore, when processing a WM_PAINT message, the application must incorporate the update region either by calling BeginPaint or by retrieving the update region and intersecting it with the current clipping region. If the application does not call BeginPaint, it must explicitly validate the update region by using the ValidateRect or ValidateRgn function. If the application does not validate the update region, the window receives an endless series of WM_PAINT messages.

Because BeginPaint hides the caret if a window is showing it, an application that calls BeginPaint should also call the EndPaint function to restore the caret. EndPaint has no other effect on a private device context.

Although a private device context is convenient to use, it is expensive in terms of system resources, requiring 800 or more bytes to store. Private device contexts are recommended when performance considerations outweigh storage costs.

The system includes the private device context when sending the WM_ERASEBKGND message to the application. The current selections of the private device context, including mapping mode, are in effect when the application or the system processes these messages. To avoid undesirable effects, the system uses logical coordinates when erasing the background; for example, it uses the *GetClipBox* function to retrieve the logical

coordinates of the area to erase and passes these coordinates to the *FillRect* function. Applications that process these messages can use similar techniques. The system supplies a window device context with the WM_ICONERASEBKGND message regardless of whether the corresponding window has a private device context.

An application can use the *GetDCEx* function to force the system to return a common device context for the window that has a private device context. This is useful for carrying out quick touch-ups to a window without changing the current values of the attributes of the private device context.

Class Display Device Context

By using a class device context, an application can use a single display device context for every window belonging to a specified class. Class device contexts are often used with control windows that are drawn using the same attribute values. Like private device contexts, class device contexts minimize the time required to prepare a device context for drawing.

The system supplies a class device context for a window if it belongs to a window class having the CS_CLASSDC style. The system creates the device context when creating the first window belonging to the class and then uses the same device context for all subsequently created windows in the class. Initially, the class device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves all changes, except for the clipping region and device origin, until the last window in the class has been destroyed. A change made for one window applies to all windows in that class.

An application can retrieve the handle for the class device context by using the *GetDC* function any time after the first window has been created. The application can keep and use the handle without releasing it because the class device context is not part of the display device context cache. If the application creates another window in the same window class, the application must retrieve the class device context again. Retrieving the device context sets the correct device origin and clipping region for the new window. After the application retrieves the class device context for a new window in the class, the device context can no longer be used to draw in the original window without again retrieving it for that window. In general, each time it must draw in a window, an application must explicitly retrieve the class device context for the window.

Applications that use class device contexts should always call *BeginPaint* when processing a WM_PAINT message. The function sets the correct device origin and clipping region for the window, and incorporates the update region. The application should also call EndPaint to restore the caret if *BeginPaint* hide it. *EndPaint* has no other effect on a class device context.

The system passes the class device context when sending the WM_ERASEBKGND message to the application, permitting the current attribute values to affect any drawing carried out by the application or the system when processing this message. The system

supplies a window device context with the WM_ICONERASEBKGND message regardless of whether the corresponding window has a class device context. As it could with a window having a private device context, an application can use *GetDCEx* to force the system to return a common device context for the window that has a class device context.

Note: Use of class device contexts is not recommended.

Window Display Device Context

A window device context enables an application to draw anywhere in a window, including the nonclient area. Window device contexts are typically used by applications that process the WM_NCPAINT and WM_NCACTIVATE messages for windows with custom nonclient areas. Using a window device context is not recommended for any other purpose.

An application can retrieve a window device context by using the *GetWindowDC* or *GetDCEx* function with the DCX_WINDOW option specified. The function retrieves a window device context from the display device context cache. A window that uses a window device context must release it after drawing by using the *ReleaseDC* function as soon as possible. Window device contexts are always from the cache; the CS_OWNDC and CS_CLASSDC class styles do not affect the device context.

When an application retrieves a window device context, the system sets the device origin to the upper left corner of the window instead of the upper left corner of the client area. It also sets the clipping region to include the entire window, not just the client area. The system sets the current attribute values of a window device context to the same default values as a common device context. An application can change the attribute values, but the system does not preserve any changes when the device context is released.

Parent Display Device Context

A parent device context enables an application to minimize the time necessary to set up the clipping region for a window. An application typically uses parent device contexts to speed up drawing for control windows without requiring a private or class device context. For example, the system uses parent device contexts for push button and edit controls. Parent device contexts are intended for use with child windows only, never with top-level or pop-up windows.

An application can specify the CS_PARENTDC style to set the clipping region of the child window to that of the parent window so that the child can draw in the parent. Specifying CS_PARENTDC enhances an application's performance because the system doesn't need to keep recalculating the visible region for each child window.

Attribute values set by the parent window are not preserved for the child window; for example, the parent window cannot set the brush for its child windows. The only property preserved is the clipping region. The window must clip its own output to the limits of the window. Because the clipping region for the parent device context is identical to the

parent window, the child window can potentially draw over the entire parent window, but the parent device context must not be used in this way.

The system ignores the CS_PARENTDC style if the parent window uses a private or class device context, if the parent window clips its child windows, or if the child window clips its child windows or sibling windows.

Window Update Lock

A window update lock is a temporary suspension of drawing in a window. The system uses the lock to prevent other windows from drawing over the tracking rectangle whenever the user moves or sizes a window. Applications can use the lock to prevent drawing if they carry out similar moving or sizing operations with their own windows.

An application uses the *LockWindowUpdate* function to set or clear a window update lock, specifying the window to lock. The lock applies to the specified window and all of its child windows. When the lock is set, the GetDC and BeginPaint functions return a display device context with a visible region that is empty. Given this, the application can continue to draw in the window, but all output is clipped. The lock persists until the application clears it by calling LockWindowUpdate, specifying NULL for the window. Although LockWindowUpdate forces a window's visible region to be empty, the function does not make the specified window invisible and does not clear the WS_VISIBLE style bit.

After the lock is set, the application can use the *GetDCEx* function, with the DCX_LOCKWINDOWUPDATE value, to retrieve a display device context to draw over the locked window. This allows the application to draw a tracking rectangle when processing keyboard or mouse messages. The system uses this method when the user moves and sizes windows. GetDCEx retrieves the display device context from the display device context cache, so the application must release the device context as soon as possible after drawing.

While a window update lock is set, the system creates an accumulated bounding rectangle for each locked window. When the lock is cleared, the system uses this bounding rectangle to set the update region for the window and its child windows, forcing an eventual WM_PAINT message. If the accumulated bounding rectangle is empty (that is, if no drawing has occurred while the lock was set), the update region is not set.

Accumulated Bounding Rectangle

The accumulated bounding rectangle is the smallest rectangle enclosing the portion of a window or client area affected by recent drawing operations. An application can use this rectangle to conveniently determine the extent of changes caused by drawing operations. It is sometimes used in conjunction with LockWindowUpdate to determine which portion of the client area must be redrawn after the update lock is cleared.

An application uses the *SetBoundsRect* function (specifying DCB_ENABLE) to begin accumulating the bounding rectangle. The system subsequently accumulates points for

the bounding rectangle as the application uses the specified display device context. The application can retrieve the current bounding rectangle at any time by using the *GetBoundsRect* function. The application stops the accumulation by calling *SetBoundsRect* again, specifying the DCB_DISABLE value.

13.5 Steps involved in output of a text string in the client area of the application

The following points are adopted to output a text string.

1. Get the handle to the Device Context for the window's client area from the GDI.
2. Use the Device Context for writing / painting in the client area of the window.
3. Release the Device context.

Printing Text String *(Example)*

```
HDC hdc;

hdc = GetDC(hWnd);            //Get the DC

char *str="This is Gdi program";

TextOut(hdc,10,10,str , strlen(str));  //output a text

ReleaseDC(hWnd,hdc);          //release a DC
```

13.6 GetDC

The GetDC function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

hDC = GetDC(hWnd);

hWnd

Handle to the window whose DC is to be retrieved. If this value is NULL, GetDC retrieves the DC for the entire screen.

The GetDC function retrieves a common, class, or private DC depending on the class style of the specified window. For class and private DCs, GetDC leaves the previously assigned attributes unchanged. However, for common DCs, GetDC assigns default attributes to the DC each time it is retrieved. For example, the default font is System, which is a bitmap font. Because of this, the handle for a common DC returned by GetDC

does not tell you what font, color, or brush was used when the window was drawn. To determine the font, call GetTextFace.

Note: that the handle to the DC can only be used by a single thread at any one time.

After painting with a common DC, the ReleaseDC function must be called to release the DC. Class and private DCs do not have to be released. ReleaseDC must be called from the same thread that called GetDC. The number of DCs is limited only by available memory.

13.7 TextOut

The TextOut() function writes a character string at the specified location, using the currently selected font, background color, and text color.

```
BOOL TextOut(
  HDC hdc,                // handle to DC
  int nXStart,            // x-coordinate of starting position
  int nYStart,            // y-coordinate of starting position
  LPCTSTR lpString,       // character string
  int cbString            // number of characters
);
```

hdc is a HANDLE to the device context.
nXStart: Specifies the x-coordinate, in logical coordinates, of the reference point that the system uses to align the string.
nYStart: Specifies the y-coordinate, in logical coordinates, of the reference point that the system uses to align the string.
lpString: Pointer to the string to be drawn. The string does not need to be zero-terminated, since cbString specifies the length of the string.
cbString: Specifies the length of the string. For the ANSI function it is a BYTE count and for the Unicode function it is a WORD count. Note that for the ANSI function, characters in SBCS code pages take one byte each while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one WORD while Unicode surrogates are two WORDs.

The interpretation of the reference point depends on the current text-alignment mode. An application can retrieve this mode by calling the GetTextAlign function; an application can alter this mode by calling the SetTextAlign function.

By default, the current position is not used or updated by this function. However, an application can call the SetTextAlign function with the fMode parameter set to TA_UPDATECP to permit the system to use and update the current position each time the application calls TextOut for a specified device context. When this flag is set, the system ignores the nXStart and nYStart parameters on subsequent TextOut calls.

```
// Obtain the window's client rectangle

GetClientRect(hwnd, &r);

/* THE FIX: by setting the background mode
 to transparent, the region is the text itself */

// SetBkMode(hdc, TRANSPARENT);

// Send some text out into the world

TCHAR text[ ] = "You can bring horse to water, but you can not make it drink";

TextOut(hdc,r.left,r.top,text, ARRAYSIZE(text)); //ARRAYSIZE is a string length
```

13.8 ReleaseDC

The ReleaseDC function releases a device context (DC), freeing it for use by other applications. The effect of the ReleaseDC function depends on the type of DC. It frees only common and window DCs. It has no effect on class or private DCs.

```
int ReleaseDC(
HWND hWnd, // handle to window
HDC hDC // handle to DC
);
```

hWnd: Handle to the window whose DC is to be released.
hDC: Handle to the DC to be released.

The application must call the ReleaseDC function for each call to the GetWindowDC function and for each call to the GetDC function that retrieves a common DC.

An application cannot use the ReleaseDC function to release a DC that was created by calling the CreateDC function; instead, it must use the DeleteDC function. ReleaseDC must be called from the same thread that called GetDC.

13.9 WM_PAINT

When a minimized window is maximized, Windows requests the application to repaint the client area.

Windows sends a WM_PAINT message for repainting a window.

13.10 BeginPaint

Begin Paint function performs following tasks.

- The BeginPaint() function prepares the specified window for painting and fills a PAINTSTRUCT structure with information about the painting.
- BeginPaint() first erases the background of window's client area by sending WM_ERASEBKGND message.
- If the function succeeds, the return value is the handle to a display device context for the specified window.

```
HDC BeginPaint(
HWND hwnd,                  // handle to window
LPPAINTSTRUCT lpPaint    // paint information
);
```

hwnd: Handle to the window to be repainted.
lpPaint: Pointer to the PAINTSTRUCT structure that will receive painting information.

The BeginPaint function automatically sets the clipping region of the device context to exclude any area outside the update region. The update region is set by the InvalidateRect or InvalidateRgn function and by the system after sizing, moving, creating, scrolling, or any other operation that affects the client area. If the update region is marked for erasing, BeginPaint sends a WM_ERASEBKGND message to the window.

An application should not call BeginPaint except in response to a WM_PAINT message. Each call to BeginPaint must have a corresponding call to the EndPaint function.

If the caret is in the area to be painted, BeginPaint automatically hides the caret to prevent it from being erased.

If the window's class has a background brush, BeginPaint uses that brush to erase the background of the update region before returning.

13.11 EndPaint

EndPaint is used to free the system resources reserved by the BeginPaint().
This function is required for each call to the BeginPaint() function, but only after painting is complete.

```
BOOL EndPaint(
  HWND hWnd,                // handle to window
  CONST PAINTSTRUCT *lpPaint  // paint data
);
```

hWnd: Handle to the window that has been repainted.
lpPaint: Pointer to a PAINTSTRUCT structure that contains the painting information retrieved by BeginPaint.

Return Value: The return value is always nonzero.

13.12 WM_SIZING

Whenever a window is resized, system sends WM_SIZING message to the application that owns the window.

In this message we can print a string each time when window is being sizing. The following example shows our statement.

```
case WM_SIZING:
        hDC = GetDC(hWnd);

        char *str=”First GDI Call in WM_SIZING Message”;

        TextOut(hDC, 0, 0, str, strlen(str));

        ReleaseDC(hWnd, hDC);
break;
```

13.13 CS_HREDRAW and CS_VREDRAW

After specifying CS_HREDRAW and CS_VREDRAW, window will send WM_PAINT message each time when window redraw either horizontally or vertically.
To send WM_PAINT message whenever a window is resized, we specify CS_HREDRAW, CS_VREDRAW class styles in WNDCLASS structure while registering the class.

Summary

In this lecture, we discussed window’s most important component—GDI (Graphics device context) in detailed. GDI is very much useful for every programmer because it gives platform independent interface. So whenever we want to write something on screen or on printer we take a device context of that particular device either display or printer. We used *GetDC* functions for getting device context of a display device or printer device to output a graphics or text data. Printing or drawing can always be done through Device context provided by Windows.

Whenever window needs to draw or paint in its client area it receives WM_PAINT message.

Tips

1) GetDC provides you handle to the device context from the cache sometimes. So be careful when using this handle and you must release device context after using it or when it is useless. Do not try to delete device context handle because it is shared to many applications so release it not delete it.
2) Try to perform painting in client area always in WM_PAINT message. (recommended)

Exercises

1. Write an application that uses Private Device Context. Using that device context, display center aligned text.

2. Before starting of above application, show a dialog box which gives option to the user to change background brush.

Chapter 14
Painting and Drawing

14.1 Painting in a Window

The *WM_PAINT* message is sent when the system or another application makes a request to paint a portion of an application's window. The message is sent by the DispatchMessage function to a window procedure when the application obtains a *WM_PAINT* message from message Queue by using the *GetMessage* or *PeekMessage* functions.
A window receives this message through its *WindowProc* function.
Windows always specifies invalid area of any window in terms of a least bounding rectangle; hence, the entire window is not repainted.

WM_PAINT message is generated by the system only when any part of application window becomes invalid.

The WM_PAINT message is generated by the system and should not be sent by an application.

The *DefWindowProc* function validates the update region. The function may also send the WM_NCPAINT message to the window procedure if the window frame must be painted and send the WM_ERASEBKGND message if the window background must be erased.

The system sends this message when there are no other messages in the application's message queue. *DispatchMessage* determines where to send the message; *GetMessage* determines which message to dispatch. GetMessage returns the WM_PAINT message when there are no other messages in the application's message queue, and *DispatchMessage* sends the message to the appropriate window procedure.

A window may receive internal paint messages as a result of calling *RedrawWindow* with the RDW_INTERNALPAINT flag set. In this case, the window may not have an update region. An application should call the *GetUpdateRect* function to determine whether the window has an update region. If *GetUpdateRect* returns zero, the application should not call the *BeginPaint* and *EndPaint* functions.

An application must check for any necessary internal painting by looking at its internal data structures for each WM_PAINT message, because a WM_PAINT message may have been caused by both a non-NULL update region and a call to RedrawWindow with the RDW_INTERNALPAINT flag set.

The system sends an internal WM_PAINT message only once. After an internal WM_PAINT message is returned from GetMessage or PeekMessage or is sent to a window by UpdateWindow, the system does not post or send further WM_PAINT messages until the window is invalidated or until RedrawWindow is called again with the RDW_INTERNALPAINT flag set.

For some common controls, the default WM_PAINT message processing checks the wParam parameter. If wParam is non-NULL, the control assumes that the value is an HDC and paints using that device context.

14.1.1 When to Draw in a Window

An application draws in a window at a variety of times: when first creating a window, when changing the size of the window, when moving the window from behind another window, when minimizing or maximizing the window, when displaying data from an opened file, and when scrolling, changing, or selecting a portion of the displayed data.

The system manages actions such as moving and sizing a window. If an action affects the content of the window, the system marks the affected portion of the window as ready for updating and, at the next opportunity, sends a **WM_PAINT** message to the window procedure of the window. The message is a signal to the application to determine what must be updated and to carry out the necessary drawing.

Some actions are managed by the application, such as displaying open files and selecting displayed data. For these actions, an application can mark for updating the portion of the window affected by the action, causing a **WM_PAINT** message to be sent at the next opportunity. If an action requires immediate feedback, the application can draw while the action takes place, without waiting for **WM_PAINT**. For example, a typical application highlights the area the user selects rather than waiting for the next **WM_PAINT** message to update the area.

In all cases, an application can draw in a window as soon as it is created. To draw in the window, the application must first retrieve a handle to a display device context for the window. Ideally, an application carries out most of its drawing operations during the processing of **WM_PAINT** messages. In this case, the application retrieves a display device context by calling the **BeginPaint** function. If an application draws at any other time, such as from within **WinMain** or during the processing of keyboard or mouse messages, it calls the **GetDC** or **GetDCEx** function to retrieve the display DC.

14.1.2 The WM_PAINT Message

Typically, an application draws in a window in response to a **WM_PAINT** message. The system sends this message to a window procedure when changes to the window have altered the content of the client area. The system sends the message only if there are no other messages in the application message queue.

Upon receiving a **WM_PAINT** message, an application can call **BeginPaint** to retrieve the display device context for the client area and use it in calls to GDI functions to carry out whatever drawing operations are necessary to update the client area. After completing the drawing operations, the application calls the **EndPaint** function to release the display device context.

Before **BeginPaint** returns the display device context, the system prepares the device context for the specified window. It first sets the clipping region for the device context to be equal to the intersection of the portion of the window that needs updating and the portion that is visible to the user. Only those portions of the window that have changed are redrawn. Attempts to draw outside this region are clipped and do not appear on the screen.

The system can also send **WM_NCPAINT** and **WM_ERASEBKGND** messages to the window procedure before **BeginPaint** returns. These messages direct the application to draw the nonclient area and window background. The *nonclient area* is the part of a window that is outside of the client area. The area includes features such as the title bar, window menu (also known as the **System** menu), and scroll bars. Most applications rely on the default window function, **DefWindowProc**, to draw this area and therefore pass the **WM_NCPAINT** message to this function. The *window background* is the color or pattern that a window is filled with before other drawing operations begin. The background covers any images previously in the window or on the screen under the window. If a window belongs to a window class having a class background brush, the **DefWindowProc** function draws the window background automatically.

BeginPaint fills a **PAINTSTRUCT** structure with information such as the dimensions of the portion of the window to be updated and a flag indicating whether the window background has been drawn. The application can use this information to optimize drawing. For example, it can use the dimensions of the update region, specified by the **rcPaint** member, to limit drawing to only those portions of the window that need updating. If an application has very simple output, it can ignore the update region and draw in the entire window, relying on the system to discard (clip) any unneeded output. Because the system clips drawing that extends outside the clipping region, only drawing that is in the update region is visible.

BeginPaint sets the update region of a window to NULL. This clears the region, preventing it from generating subsequent **WM_PAINT** messages. If an application processes a **WM_PAINT** message but does not call **BeginPaint** or otherwise clear the update region, the application continues to receive **WM_PAINT** messages as long as the region is not empty. In all cases, an application must clear the update region before returning from the **WM_PAINT** message.

After the application finishes drawing, it should call **EndPaint**. For most windows, **EndPaint** releases the display device context, making it available to other windows. **EndPaint** also shows the caret, if it was previously hidden by **BeginPaint**. **BeginPaint** hides the caret to prevent drawing operations from corrupting it.

14.1.3 Drawing Without the WM_PAINT Message

Although applications carry out most drawing operations while the **WM_PAINT** message is processing, it is sometimes more efficient for an application to draw directly in a window without relying on the **WM_PAINT** message. This can be useful when the

user needs immediate feedback, such as when selecting text and dragging or sizing an object. In such cases, the application usually draws while processing keyboard or mouse messages.

To draw in a window without using a **WM_PAINT** message, the application uses the **GetDC** or **GetDCEx** function to retrieve a display device context for the window. With the display device context, the application can draw in the window and avoid intruding into other windows. When the application has finished drawing, it calls the **ReleaseDC** function to release the display device context for use by other applications.

When drawing without using a **WM_PAINT** message, the application usually does not invalidate the window. Instead, it draws in such a fashion that it can easily restore the window and remove the drawing. For example, when the user selects text or an object, the application typically draws the selection by inverting whatever is already in the window. The application can remove the selection and restore the original contents of the window by simply inverting again.

The application is responsible for carefully managing any changes it makes to the window. In particular, if an application draws a selection and an intervening **WM_PAINT** message occurs, the application must ensure that any drawing done during the message does not corrupt the selection. To avoid this, many applications remove the selection, carry out usual drawing operations, and then restore the selection when drawing is complete.

14.1.4 Window Background

The window background is the color or pattern used to fill the client area before a window begins drawing. The window background covers whatever was on the screen before the window was moved there, erasing existing images and preventing the application's new output from being mixed with unrelated information.

The system paints the background for a window or gives the window the opportunity to do so by sending it a **WM_ERASEBKGND** message when the application calls **BeginPaint**. If an application does not process the message but passes it to **DefWindowProc**, the system erases the background by filling it with the pattern in the background brush specified by the window's class. If the brush is not valid or the class has no background brush, the system sets the **fErase** member in the **PAINTSTRUCT** structure that **BeginPaint** returns, but carries out no other action. The application then has a second chance to draw the window background, if necessary.

If it processes **WM_ERASEBKGND**, the application should use the message's *wParam* parameter to draw the background. This parameter contains a handle to the display device context for the window. After drawing the background, the application should return a nonzero value. This ensures that **BeginPaint** does not erroneously set the **fErase** member of the **PAINTSTRUCT** structure to a nonzero value (indicating the background should be erased) when the application processes the subsequent **WM_PAINT** message.

An application can define a class background brush by assigning a brush handle or a system color value to the **hbrBackground** member of the **WNDCLASS** structure when registering the class with the **RegisterClass** function. The **GetStockObject** or **CreateSolidBrush** function can be used to create a brush handle. A system color value can be one of those defined for the **SetSysColors** function. (The value must be increased by one before it is assigned to the member.)

An application can process the **WM_ERASEBKGND** message even though a class background brush is defined. This is typical in applications that enable the user to change the window background color or pattern for a specified window without affecting other windows in the class. In such cases, the application must not pass the message to **DefWindowProc**.

It is not necessary for an application to align brushes, because the system draws the brush using the window origin as the point of reference. Given this, the user can move the window without affecting the alignment of pattern brushes.

14.2 Window Coordinate System

The coordinate system for a window is based on the coordinate system of the display device. The basic unit of measure is the device unit (typically, the pixel). Points on the screen are described by x- and y-coordinate pairs. The x-coordinates increase to the right; y-coordinates increase from top to bottom. The origin (0,0) for the system depends on the type of coordinates being used.

The system and applications specify the position of a window on the screen in *screen coordinates*. For screen coordinates, the origin is the upper-left corner of the screen. The full position of a window is often described by a **RECT** structure containing the screen coordinates of two points that define the upper-left and lower-right corners of the window.

The system and applications specify the position of points in a window by using *client coordinates*. The origin in this case is the upper-left corner of the window or client area. Client coordinates ensure that an application can use consistent coordinate values while drawing in the window, regardless of the position of the window on the screen.

The dimensions of the client area are also described by a **RECT** structure that contains client coordinates for the area. In all cases, the upper-left coordinate of the rectangle is included in the window or client area, while the lower-right coordinate is excluded. Graphics operations in a window or client area are excluded from the right and lower edges of the enclosing rectangle.

Occasionally, applications may be required to map coordinates in one window to those of another window. An application can map coordinates by using the **MapWindowPoints** function. If one of the windows is the desktop window, the function effectively converts

screen coordinates to client coordinates and vice versa; the desktop window is always specified in screen coordinates.

14.3 Window Regions

In addition to the update region, every window has a *visible region* that defines the window portion visible to the user. The system changes the visible region for the window whenever the window changes size or whenever another window is moved such that it obscures or exposes a portion of the window. Applications cannot change the visible region directly, but the system automatically uses the visible region to create the clipping region for any display device context retrieved for the window.

The *clipping region* determines where the system permits drawing. When the application retrieves a display device context using the **BeginPaint**, **GetDC**, or **GetDCEx** function, the system sets the clipping region for the device context to the intersection of the visible region and the update region. Applications can change the clipping region by using functions such as **SetWindowRgn**, **SelectClipPath** and **SelectClipRgn**, to further limit drawing to a particular portion of the update area.

The WS_CLIPCHILDREN and WS_CLIPSIBLINGS styles further specify how the system calculates the visible region for a window. If a window has one or both of these styles, the visible region excludes any child window or sibling windows (windows having the same parent window). Therefore, drawing that would otherwise intrude in these windows will always be clipped.

14.4 Condition in which PAINT message is sent *(briefly)*

- Any hidden part of window becomes visible Window is resized (and CS_VREDRAW, CS_HREDRAW style bits were set while registering the window class).
- Program scrolls its window.
- *InvalidateRect* or *InvalidateRgn* is called by the application.

14.5 Condition in which PAINT message may be sent

- A dialog is dismissed.
- A drop-down menu disappears.
- A tool tip is displayed and then it hides.

14.6 Condition in which PAINT message never sent

- An icon is dragged over the window.
- The mouse cursor is moved

14.7 PAINT Reference

14.7.1 InvalidateRect Function

InvalidateRect function is used to make window or part of it, invalidate.

```
BOOL InvalidateRect(
HWND hWnd,              // handle to window
CONST RECT *lpRect,     // rectangle coordinates
BOOL bErase             // erase state
);
```

hWnd: Handle to the window whose update region has changed. If this parameter is NULL, the system invalidates and redraws all windows, and sends the **WM_ERASEBKGND** and **WM_NCPAINT** messages to the window procedure before the function returns.

lpRect: Pointer to a **RECT** structure that contains the client coordinates of the rectangle to be added to the update region. If this parameter is NULL, the entire client area is added to the update region.

bErase: Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased when the **BeginPaint** function is called. If this parameter is FALSE, the background remains unchanged.

Return Values: If the function succeeds, the return value is nonzero.If the function fails, the return value is zero.

14.7.2 PAINTSTRUCT Structure

The **PAINTSTRUCT** structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc;                   //Handle to the Device context
  BOOL fErase; /*erase back ground of this parameter is true*/
  RECT rcPaint;               /*rectangle to the invalidate region*/
  BOOL fRestore;
  BOOL fIncUpdate;            //updation true/false
  BYTE rgbReserved[32];       //rgb values
} PAINTSTRUCT, *PPAINTSTRUCT;
```

hdc

Handle to the display DC to be used for painting.

fErase

Specifies whether the background must be erased. This value is nonzero if the application should erase the background. The application is responsible for

erasing the background if a window class is created without a background brush. For more information, see the description of the **hbrBackground** member of the **WNDCLASS** structure.

rcPaint

Specifies a **RECT** structure that specifies the upper left and lower right corners of the rectangle in which the painting is requested, in device units relative to the upper-left corner of the client area.

fRestore

Reserved; used internally by the system.

fIncUpdate

Reserved; used internally by the system.

rgbReserved

Reserved; used internally by the system.

14.8 Other GDI Text Output Functions

14.8.1 DrawText

The *DrawText* function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

```
int DrawText(
  HDC hDC,           // handle to DC
  LPCTSTR lpString, // text to draw
  int nCount,        // text length
  LPRECT lpRect,     // formatting dimensions
  UINT uFormat       // text-drawing options
);
```

hDC: Handle to the device context.

lpString: Pointer to the string that specifies the text to be drawn. If the *nCount* parameter is –1, the string must be null-terminated.

If *uFormat* includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

nCount: Specifies the length of the string. For the ANSI function it is a BYTE count and for the Unicode function it is a **WORD** count. Note that for the ANSI function, characters in SBCS code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one **WORD** while Unicode surrogates are two **WORD**s. If *nCount* is –1, then the *lpString* parameter is assumed to be a pointer to a null-terminated string and **DrawText** computes the character count automatically.

lpRect: Pointer to a **RECT** structure that contains the rectangle (in logical coordinates) in which the text is to be formatted.

uFormat: Specifies the method of formatting the text. This parameter can be one or more of the following values.

Value	Description
DT_BOTTOM	Justifies the text to the bottom of the rectangle. This value is used only with the DT_SINGLELINE value.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, **DrawText** uses the width of the rectangle pointed to by the *lpRect* parameter and extends the base of the rectangle to bound the last line of text. If the largest word is wider than the rectangle, the width is expanded. If the text is less than the width of the rectangle, the width is reduced. If there is only one line of text, **DrawText** modifies the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawText** returns the height of the formatted text but does not draw the text.
DT_CENTER	Centers text horizontally in the rectangle.
DT_EDITCONTROL	Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.
DT_END_ELLIPSIS	For displayed text, if the end of a string does not fit in the rectangle, it is truncated and ellipses are added. If a word that is not at the end of the string goes beyond the limits of the rectangle, it is truncated without ellipses. The string is not modified unless the DT_MODIFYSTRING flag is specified. Compare with DT_PATH_ELLIPSIS and DT_WORD_ELLIPSIS.
DT_EXPANDTABS	Expands tab characters. The default

	number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX	**Windows 2000/XP:** Ignores the ampersand (&) prefix character in the text. The letter that follows will not be underlined, but other mnemonic-prefix characters are still processed. For example:
	`input string: "A&bc&&d"`
	`normal: "A̲bc&d"`
	`DT_HIDEPREFIX: "Abc&d"`
	
	Compare with DT_NOPREFIX and DT_PREFIXONLY.

DT_INTERNAL	Uses the system font to calculate text metrics.
DT_LEFT	Aligns text to the left.
DT_MODIFYSTRING	Modifies the specified string to match the displayed text. This value has no effect unless DT_END_ELLIPSIS or DT_PATH_ELLIPSIS is specified.
DT_NOCLIP	Draws without clipping. **DrawText** is somewhat faster when DT_NOCLIP is used.
DT_NOFULLWIDTHCHARBREAK	**Windows 98/Me, Windows 2000/XP:** Prevents a line break at a DBCS (double-wide character string), so that the line breaking rule is equivalent to SBCS strings. For example, this can be used in Korean windows, for more readability of icon labels. This value has no effect unless DT_WORDBREAK is specified.
DT_NOPREFIX	Turns off processing of prefix characters. Normally, **DrawText** interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. For example,

```
input string:   "A&bc&&d"
normal:         "Abc&d"
DT_NOPREFIX:    "A&bc&&d"
```

Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_PATH_ELLIPSIS — For displayed text, replaces characters in the middle of the string with ellipses so that the result fits in the specified rectangle. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much as possible of the text after the last backslash.

The string is not modified unless the DT_MODIFYSTRING flag is specified.

Compare with DT_END_ELLIPSIS and DT_WORD_ELLIPSIS.

DT_PREFIXONLY — **Windows 2000/XP:** Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any other characters in the string. For example,

```
input string:    "A&bc&&d"
normal:          "Abc&d"
DT_PREFIXONLY:   " _   "
```

Compare with DT_HIDEPREFIX and DT_NOPREFIX.

DT_RIGHT — Aligns text to the right.

DT_RTLREADING — Layout in right-to-left reading order for bi-directional text when the font selected into the *hdc* is a Hebrew or Arabic font. The default reading order for all text is left-to-right.

DT_SINGLELINE — Displays text on a single line only. Carriage returns and line feeds do not break the line.

DT_TABSTOP — Sets tab stops. Bits 15–8 (high-order byte of the low-order word) of the *uFormat* parameter specify the number of characters for each tab. The default number of characters per tab is eight. The DT_CALCRECT, DT_EXTERNALLEADING,

	DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.
DT_TOP	Justifies the text to the top of the rectangle.
DT_VCENTER	Centers text vertically. This value is used only with the DT_SINGLELINE value.
DT_WORDBREAK	Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *lpRect* parameter. A carriage return-line feed sequence also breaks the line. If this is not specified, output is on one line.
DT_WORD_ELLIPSIS	Truncates any word that does not fit in the rectangle and adds ellipses. Compare with DT_END_ELLIPSIS and DT_PATH_ELLIPSIS.

Return Values: If the function succeeds, the return value is the height of the text in logical units. If DT_VCENTER or DT_BOTTOM is specified, the return value is the offset from

lpRect-> top to the bottom of the drawn text

If the function fails, the return value is zero.

The *DrawText* function uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawText** clips the text so that it does not appear outside the specified rectangle. Note that text with significant overhang may be clipped, for example, an initial "W" in the text string or text that is in italics. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the specified rectangle, the **DrawText** function does not attempt to substitute a smaller font.

The **DrawText** function supports only fonts whose escapement and orientation are both zero.

The text alignment mode for the device context must include the TA_LEFT, TA_TOP, and TA_NOUPDATECP flags.

14.8.2 TabbedTextOut

The *TabbedTextOut* function writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions. Text is written in the currently selected font, background color, and text color.

```
LONG TabbedTextOut(
  HDC hDC,                    // handle to DC
  int X,                      // x-coord of start
  int Y,                      // y-coord of start
  LPCTSTR lpString,           // character string
  int nCount,                 // number of characters
  int nTabPositions,          // number of tabs in array
  CONST LPINT lpnTabStopPositions, // array of tab positions
  int nTabOrigin              // start of tab expansion
);
```

hDC: Handle to the device context.

X: Specifies the x-coordinate of the starting point of the string, in logical units.

Y: Specifies the y-coordinate of the starting point of the string, in logical units.

lpString: Pointer to the character string to draw. The string does not need to be zero-terminated, since *nCount* specifies the length of the string.

nCount: Specifies the length of the string pointed to by *lpString*. For the ANSI function it is a BYTE count and for the Unicode function it is a **WORD** count. Note that for the ANSI function, characters in SBCS code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one **WORD** while Unicode surrogates are two **WORD**s.

nTabPositions: Specifies the number of values in the array of tab-stop positions.

lpnTabStopPositions: Pointer to an array containing the tab-stop positions, in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array.

nTabOrigin: Specifies the x-coordinate of the starting position from which tabs are expanded, in logical units.

Return Values: If the function succeeds, the return value is the dimensions, in logical units, of the string. The height is in the high-order word and the width is in the low-order word.

If the function fails, the return value is zero.

If the *nTabPositions* parameter is zero and the *lpnTabStopPositions* parameter is NULL, tabs are expanded to eight times the average character width.

If *nTabPositions* is 1, the tab stops are separated by the distance specified by the first value in the *lpnTabStopPositions* array.

If the *lpnTabStopPositions* array contains more than one value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

The *nTabOrigin* parameter allows an application to call the **TabbedTextOut** function several times for a single line. If the application calls **TabbedTextOut** more than once with the *nTabOrigin* set to the same value each time, the function expands all tabs relative to the position specified by *nTabOrigin*.

By default, the current position is not used or updated by the **TabbedTextOut** function. If an application needs to update the current position when it calls **TabbedTextOut**, the application can call the **SetTextAlign** function with the *wFlags* parameter set to TA_UPDATECP. When this flag is set, the system ignores the *X* and *Y* parameters on subsequent calls to the **TabbedTextOut** function, using the current position instead.

14.9 Primitive Shapes

Primitive shapes include: Lines and Curves, filled shapes like:

Ellipse, Chord, Pie, Polygon, Rectangles

14.9.1 Lines

Line can be drawn using *MoveToEx* and *LineTo* Function.
MoveToEx function moves the points at specified location.

Note: MoveToEx effects all drawing functions.

The *LineTo* function draws a line from the current position up to, but not including, the specified point.

```
BOOL LineTo(
  HDC hdc,    // device context handle
  int nXEnd,  // x-coordinate of ending point
  int nYEnd   // y-coordinate of ending point
);
```

hdc: Handle to a device context.
nXEnd: Specifies the x-coordinate, in logical units, of the line's ending point.
nYEnd: Specifies the y-coordinate, in logical units, of the line's ending point.

Return Values: If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

14.9.2 Rectangle

- The *Rectangle()* function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush.

- The rectangle is outlined using currently selected pen and filled using the currently selected brush of n the window's device context.

```
BOOL Rectangle(
HDC hdc,        // handle to DC
int nLeftRect,   // x-coord of upper-left corner of rectangle
int nTopRect,    // y-coord of upper-left corner of rectangle
int nRightRect,  // x-coord of lower-right corner of rectangle
int nBottomRect  // y-coord of lower-right corner of rectangle
);
```

14.9.3 Polygon

The *Polygon()* function draws a polygon consisting of two or more vertices connected by straight lines. The polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode.

```
BOOL Polygon(
HDC hdc,                    // handle to DC
CONST POINT *lpPoints,      // polygon vertices
int Count                   // count of polygon vertices
);
```

14.10 Stock Objects

Pre-defined GDI objects in Windows are:

- Pens
- Brushes
- Fonts
- Palettes

14.10.1 GetStockObject Function

The *GetStockObject* function retrieves a handle to one of the stock pens, brushes, fonts, or palettes.

```
HGDIOBJ GetStockObject(
  int fnObject   // stock object type
);
```

fnObject: Specifies the type of stock object. This parameter can be one of the following values.

Value	Meaning
BLACK_BRUSH	Black brush.
DKGRAY_BRUSH	Dark gray brush.
DC_BRUSH	**Windows 2000/XP:** Solid color brush. The default color is white. The color can be changed by using the **SetDCBrushColor** function. For more information, see the Remarks section.
GRAY_BRUSH	Gray brush.
HOLLOW_BRUSH	Hollow brush (equivalent to NULL_BRUSH).
LTGRAY_BRUSH	Light gray brush.
NULL_BRUSH	Null brush (equivalent to HOLLOW_BRUSH).
WHITE_BRUSH	White brush.
BLACK_PEN	Black pen.
DC_PEN	**Windows 2000/XP:** Solid pen color. The default color is white. The color can be changed by using the **SetDCPenColor** function. For more information, see the Remarks section.
WHITE_PEN	White pen.
ANSI_FIXED_FONT	Windows fixed-pitch (monospace) system font.
ANSI_VAR_FONT	Windows variable-pitch (proportional space) system font.
DEVICE_DEFAULT_FONT	**Windows NT/2000/XP:** Device-dependent font.
DEFAULT_GUI_FONT	Default font for user interface objects such as menus and dialog boxes. This is MS Sans Serif. Compare this with SYSTEM_FONT.
OEM_FIXED_FONT	Original equipment manufacturer (OEM) dependent fixed-pitch (monospace) font.
SYSTEM_FONT	System font. By default, the system uses the system font to draw menus, dialog box controls, and text. **Windows 95/98 and Windows NT:** The system font is MS Sans Serif. **Windows 2000/XP:** The system font is Tahoma
SYSTEM_FIXED_FONT	Fixed-pitch (monospace) system font. This stock object is provided only for compatibility with 16-bit Windows versions earlier than 3.0.
DEFAULT_PALETTE	Default palette. This palette consists of the static colors in the system palette.

Return Values: If the function succeeds, the return value is a handle to the requested logical object. If the function fails, the return value is NULL.

Use the DKGRAY_BRUSH, GRAY_BRUSH, and LTGRAY_BRUSH stock objects only in windows with the CS_HREDRAW and CS_VREDRAW styles. Using a gray stock brush in any other style of window can lead to misalignment of brush patterns after a window is moved or sized. The origins of stock brushes cannot be adjusted.

The HOLLOW_BRUSH and NULL_BRUSH stock objects are equivalent.

The font used by the DEFAULT_GUI_FONT stock object could change. Use this stock object when you want to use the font that menus, dialog boxes, and other user interface objects use.

14.11 SelectObject

The *SelectObject* function selects an object into the specified device context (DC). The new object replaces the previous object of the same type.

```
HGDIOBJ SelectObject(
  HDC hdc,          // handle to DC
  HGDIOBJ hgdiobj   // handle to object
);
```

hdc: Handle to the DC.
Hgdiobj: Handle to the object to be selected. The specified object must have been created by using one of the following functions.

Object	Functions
Bitmap	Created by any bitmap function like **CreateBitmap**, **CreateCompatibleBitmap** and **CreateDIBSection** etc. (Bitmaps can be selected for memory DCs only, and for only one DC at a time.)
Brush	Created by **CreateBrushIndirect or CreateSolidBrush**
Font	Created by **CreateFont** function
Pen	Created by **CreatePen**
Region	Created by any region function e.g. **CreatePolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**

Return Values: If the selected object is not a region and the function succeeds, the return value is a handle to the object being replaced.

If an error occurs and the selected object is not a region, the return value is NULL. Otherwise, it is HGDI_ERROR.

This function returns the previously selected object of the specified type. An application should always replace a new object with the original, default object after it has finished drawing with the new object.

An application cannot select a bitmap into more than one DC at a time.

14.12 Example

```
SelectObject(hdc,GetStockObject(DC_PEN));
SetDCPenColor(hdc,RGB(00,0xff,00);
Rectangle(0,0,20,20);
SetDCPenColor(hdc,RGB(00,00,0xff));
Rectangle(0,0,20,20)

/* The brush color can be changed in a similar manner. SetDCPenColor
 and SetDCBrushColor can be used interchangeably with GetStockObject
 to change the current color.
*/

SelectObject(hDC,GetStockObject(DC_BRUSH));
SetDCBrushColor(hDC,0x0)

// Provides the same flexibility as:

SelectObject(hDC,GetStockOBject(BLACK_BRUSH));

// It is not necessary to call DeleteObject to delete stock objects.
```

Summary

In this lecture, we studied about painting in windows, for painting we used an important message i.e. WM_PAINT. This message is always sent when windows need to paint its portion or region that was previously invalidated. Windows paint only its region when it become invalidates otherwise it is not sent WM_PAINT message. In some cases, WM_PAINT messages are not sent—when menu drops down or small dialog boxes appear. We also studied about invalidation and validation of a region. If region is invalidated then it will be receiving WM_PAINT messages until it become validate.
At the end of the lecture we studied about sending and posting of messages. Messages can be sent to windows procedure directly or it can be posted to message queue. All the sent messages are not returned until the message is processed but the posted messages are returned after posting the message in queue.

Tips

All GDI Objects create handles, these handles can be of bitmaps, regions, or fonts handle. These handles must be deleted after using these objects. Keep it in mind and make a practice to delete all the objects when they are left unused.

Exercises

1. Create a round rectangular region and paint it using your own created hatched brush. Also write the text which should be clipped to the region.
2. On pressing the mouse left button in region, region must show message box, bearing text you have pressed mouse in region.

Chapter 15
Windows Management

15.1 Z-Order

- The Z order of a window indicates the window's position in a stack of overlapping windows. This window stack is oriented along an imaginary axis, the z-axis, extending outward from the screen.
- The window at the top of the Z order overlaps all other windows.
- The window at the bottom of the Z order is overlapped by all other windows.

15.2 Windows Review

15.2.1 CreateWindow

CreateWindow function have been discussing in our previous lectures. Much of its details including styles, class name, parent handles, instance handle and coordinates, etc have been discussed in chapter 11.
CreateWindow Function is used to create window. CreateWindow function can create parent, child, popup and overlapped windows with dimensions x, y, width and height.

```
HWND CreateWindow
(
LPCTSTR lpClassName,   // registered class name
LPCTSTR lpWindowName,    // window name
DWORD dwStyle,   // window style
int x,   // horizontal position of window
int y,   // vertical position of window
int nWidth,    // window width
int nHeight,   // window height
HWND hWndParent,   // handle to parent or owner window
HMENU hMenu,   // menu handle or child identifier
HINSTANCE hInstance,   // handle to application instance
LPVOID lpParam   // window-creation data
);
```

15.2.2 Child Windows

Following are the characteristics of child windows.

- A child window always appears within the client area of its parent window.
- Child windows are most often as controls.
- A child window sends WM_COMMAND notification messages to its parent window.

- When a child window is created a unique identifier for that window is specified in hMenu parameter of CreateWindow()

15.2.3 Window Procedure

```
LRESULT CALLBACK WindowProc
(
        HWND hwnd, // handle to window
        UINT uMsg, // WM_COMMAND
        WPARAM wParam, // notification code and identifier
        LPARAM lParam // handle to control (HWND)
);
```

15.2.4 Notification code

Common controls are normally taken as child windows that send notification messages to the parent window when events, such as input from the user, occur in the control. The application relies on these notification messages to determine what action the user wants it to take. Except for trackbars, which use the WM_HSCROLL and WM_VSCROLL messages to notify their parent of changes, common controls send notification messages as WM_NOTIFY messages.

15.2.5 WM_COMMAND Notification code

- The wParam parameter of Window Procedure contains the notification code and control identifier.
- low word: ID of the control n high word: notification code
- BUTTON *BN_CLICKED*
- EDIT *EN_CHANGE* etc

15.3 Example Application

For demonstration purpose we are going to create an example application.

15.3.1 Description

Our application will be parent-child window application. This application will consist of three push buttons of names:

- RECTANGLE
- CIRCLE
- MESSAGE”

And Edit child window or edit control in its client area.
Floating popup window with caption bar and one push button bearing a name"QUIT APPLICATION".

15.3.2 Objectives

- Parent-child communication
- Message Routing
- Use of GDI function calls

15.3.3 Windows Management Functions

Building our application, we will use following windows management functions in our application.

Windows management function - I

```
HWND GetParent
(
        HWND hWnd   // handle to child window
);
```

GetParent function returns the parent handle of the specified child. This function will be useful when the parent of the child window to use.

Windows management function - II

```
HWND GetDlgItem
(
        HWND hDlg, // handle to dialog box
        int nIDDlgItem // control identifier
);
```

GetDlgItem function returns the handle of a dialog item. Using this function we can easily get the handle of the edit control, displayed on dialog box.

```
HWND FindWindow
(
        LPCTSTR lpClassName,  // class name
        LPCTSTR lpWindowName // window name
);
```

FindWindow function finds the window with the given class name or window name.

15.3.4 Window classes

The Window classes used in this application are:

- mainWindowClass
- popupWindowClass
- System Window Classes

15.3.4.1 Main Window class

```
wc.lpfnWndProc  = mainWindowProc;
wc.hInstance    = hAppInstance = hInstance;
wc.hCursor    = LoadCursor(NULL,      IDC_UPARROW);
wc.hbrBackground= (HBRUSH)GetStockObject (GRAY_BRUSH);
wc.lpszClassName= "MainWindowClass";

if(!RegisterClass(&wc))
{
        return 0;
}
```

15.3.4.2 Popup Window class

```
wc .lpfnWndProc   = popupWindowProc;
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wc.hCursor        = LoadCursor(NULL, IDC_HELP);
wc.lpszClassName = "PopupWindowClass";

if(!RegisterClass(&wc))
{
        return 0;
}
```

15.3.4.3 System Window classes

System window classes are pre-registered. They do not need to register in our application. In this application, we will only used to create them not to register them.

15.3.5 Creating Main Windows

Create a Main Window of the Application.

```
hWndMain = CreateWindow("MainWindowClass",
"Virtual University",
```

```
WS_OVERLAPPEDWINDOW | WS_VISIBLE,
100, 100, 400, 300,
NULL, NULL, hInstance, NULL
);

// check the returned handle, don't need to proceed if the returned handle is NULL
if(!hWnd)
{
	MessageBox(NULL,"Cannot Create Main Window","Error",
MB_ICONHAND|MB_OK);
	return 0;
}
```

Now create a popup window with caption and visible style bit on.

```
hWndPopup = CreateWindow("PopupWindowClass", //window name (optional)
"Popup Window",  //class name
WS_POPUP | WS_CAPTION | WS_VISIBLE,
250, 250, 300, 250,
hWndMain, NULL, hInstance, NULL
);
```

15.3.6 Creating Child Windows

Create a button window bearing a text "Rectangle".

```
hWndButton=CreateWindow("BUTTON",
"Rectangle",
WS_CHILD | WS_VISIBLE,
10, 10, 100, 50,
hWndMain, 5,
hInstance, NULL
);
```

Create an Edit Window bearing a text "Message"

```
CreateWindow("EDIT",
"Message",
WS_CHILD | WS_VISIBLE n | ES_LOWERCASE,
10, 190, 200, 25,
hWndMain, 8,
hInstance, NULL
);
```

15.3.7 User defined Messages

System defined messages are already defined in WINUSER.H

```
#define WM_LBUTTONDOWN 0x0201 (513)
#define WM_DESTROY 0x0002 (2)
#define WM_QUIT 0x0012 (18)
#define WM_USER 0x0400 (1024)
```
These message are already defined, user don't need to define them again.

Here, we will define our own messages.

```
#define WM_DRAW_FIGURE WM_USER+786
//user defined message are in valid range, in our case it is WM_USER + 786
```

15.3.8 Application's Main Window Procedure

Message send to main window will be received and processed in mainWndProc function. In windows procedure we will process WM_COMMAND message. In WM_COMMAND message we check the LOWORD and HIWORD parameters of the messages. In LOWORD we have the control ID, always and in HIWORD we have notification code. Using control id we identify a window.

```
case WM_COMMAND:
        wControlID = LOWORD(wParam);
        wNotificationCode = HIWORD(wParam);
        if(wNotificationCode == BN_CLICKED)
        {
            switch(wControlID)
            {
            case 5:
            SendMessage(hWndPopup, WM_DRAW_FIGURE,   RECTANGLE, 0);
            break;
            case 6:
            SendMessage(hWndPopup, WM_DRAW_FIGURE,     CIRCLE, 0); break;
            case 7: SendMessage(hWndPopup, WM_DRAW_FIGURE,
            TEXT_MESSAGE, 0); break;
             }
```

15.3.8.1 Drawing in Popup Window- I

here we check the button if button rectangle is pressed then draw a rectangle with Light Gray stock brush.

```
case WM_DRAW_FIGURE:

hDC = GetDC(hWndPopup);
switch(wParam)
{
        case RECTANGLE:
        SelectObject(hDC,GetStockObject(
        LTGRAY_BRUSH));
        Rectangle(hDC, 50, 10, 230, 150);
        break;
}
```

15.3.8.2 Drawing in Popup Window- II

In case of Circle, we create hatch brush, we created a hatch brush which has a style of diagonal cross lines. After creating a hatch brush we select it on device context to draw a figure. After drawing, brush must be deleted.

```
case CIRCLE:

hBrush = CreateHatchBrush(HS_DIAGCROSS, RGB(170, 150, 180));
SelectObject(hDC, hBrush);
Ellipse(hDC, 70, 10, 210, 150);
DeleteObject(hBrush);
break;
```

15.3.8.3 Drawing in Popup Window- III

By pressing the button a text must be display with the stock object Ansi variable fonts and background brush. Text are displayed using TextOut GDI function.

```
case TEXT_MESSAGE:
{
        TextOut(hDC, 50, 100, "Virtual University", 18);
        SelectObject(hDC, GetStockObject(
        ANSI_VAR_FONT));
        SetBkColor(hDC, RGB(10, 255, 20));
        TextOut(hDC, 50, 115, "knowledge Beyond Boundaries", 27);
        break;
}
```

```
ReleaseDC(hWndPopup, hDC);
```

15.3.9 Informing back to Main Window

```
case WM_DRAW_FIGURE:

hDC = GetDC(hWndPopup);
hwndEdit = GetDlgItem(GetParent(hWnd), 8);

switch(wParam)
{
case RECTANGLE:
SelectObject(hDC,GetStockObject(LTGRAY_BRUSH));
Rectangle(hDC, 50, 10, 230, 150);
SendMessage(hwndEdit, WM_SETTEXT, 0, "Rectangle DrAwN!");
break;
}
```

15.3.10 Quit Application via control in Popup window

Main window is destroyed through button's notification message BN_CLICKED. Main Window can be destroyed using DestroyWindow Function.

```
WM_CREATE:
CreateWindow("BUTTON",
"Quit Application",
WS_CHILD | WS_VISIBLE, n 75, 155, 150, 40, hWnd, 1234,
hAppInstance, NULL);
break;

case WM_COMMAND:
wControlID = LOWORD(wParam);
wNotificationCode = HIWORD(wParam);
if(wNotificationCode == BN_CLICKED)
{
switch(wControlID)
{
case 1234:
DestroyWindow(GetParent(hWnd));
break;
}

}
```

Summary

This chapter uses Windows management functions whose details have been discussed in our previous lectures. These functions are very helpful to interact with windows and hierarchy of windows and also with windows handling, windows manipulation and windows management. Our main objective in this application was to create a full fledge application. Before continue, we overviewed all the functions that we had to use. Function includes GetParent, GetDlgItem, CreateWindow and notification codes that are sent to window by controls or other child windows. Controls are normally considered are child windows, because these can be placed in any windows and become the part of the window but controls can be main window. Notification messages are considered to transfer informations to parent window by child windows. Child windows can send notification message to parent windows which aim only to inform about some events to parent window. The notification events could be e.g. in case of edit control is selection change i.e. EN_SELCHANGE or EN_CLICKED in case of button. Finally we wrote a code for our application. This code displays a window and three child windows including button that contains text like rectangle, messages etc. we also make a popup window, popup window is not a child window. Popup windows are very useful when to show message on screen or working in full screen modes in Microsoft Windows Operating systems.

Exercises

1. Create a full screen popup window and then create another popup window with a caption box, system menu and with no close style.
2. Create your own status bar and show it in a window at its proper location. This status bar should display current time.

Chapter 16
Input Devices

16.1 Keyboard

Keyboard is an external device in computer. Keyboard is used to input data in computer system. An application receives keyboard input in the form of messages posted to its windows.

16.1.1 Keyboard Input Model

The system provides device-independent keyboard support for applications by installing a keyboard device driver appropriate for the current keyboard. The system provides language-independent keyboard support by using the language-specific keyboard layout currently selected by the user or the application. The keyboard device driver receives scan codes from the keyboard, which are sent to the keyboard layout where they are translated into messages and posted to the appropriate windows in your application.

Assigned to each key on a keyboard is a unique value called a *scan code*; it is a device-dependent identifier for the key on the keyboard. A keyboard generates two scan codes when the user types a key—one when the user presses the key and another when the user releases the key.

The keyboard device driver interprets a scan code and translates (maps) it to a *virtual-key code; virtual-key code* is a device-independent value defined by the system that identifies the purpose of a key. After translating a scan code, the keyboard layout creates a message that includes the scan code, the virtual-key code, and other information about the keystroke, and then places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Eventually, the thread's message loop removes the message and passes it to the appropriate window procedure for processing. The following figure illustrates the keyboard input model.

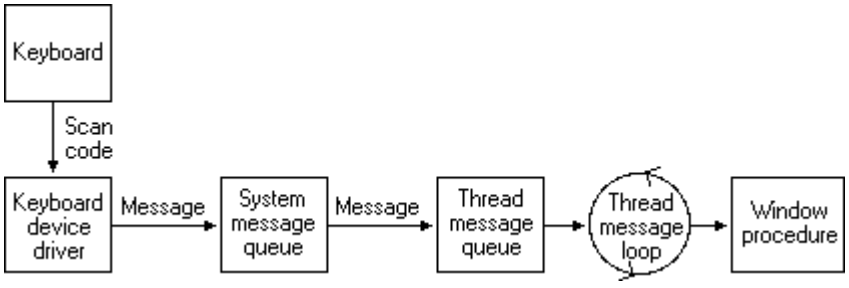


Figure 1

16.1.2 Keyboard Focus and Activation

The system posts keyboard messages to the message queue of the foreground thread that created the window with the keyboard focus. The *keyboard focus* is a temporary property of a window. The system shares the keyboard with all windows on the display by shifting the keyboard focus, at the user's direction, from one window to another. The window that has the keyboard focus receives (from the message queue of the thread that created it) all keyboard messages until the focus changes to a different window.

A thread can call the GetFocus function to determine which of its windows (if any) currently has the keyboard focus. A thread can give the keyboard focus to one of its windows by calling the SetFocus function. When the keyboard focus changes from one window to another, the system sends a WM_KILLFOCUS message to the window that has lost the focus, and then sends a WM_SETFOCUS message to the window that has gained the focus.

The concept of keyboard focus is related to that of the active window. The *active window* is the top-level window the user is currently working with. The window with the keyboard focus is either the active window, or a child window of the active window. To help the user identify the active window, the system places it at the top of the Z order and highlights its title bar (if it has one) and border.

The user can activate a top-level window by clicking it, selecting it using the ALT+TAB or ALT+ESC key combination, or selecting it from the Task List. A thread can activate a top-level window by using the SetActiveWindow function. It can determine whether a top-level window it created is active by using the GetActiveWindow function.

When one window is deactivated and another activated, the system sends the WM_ACTIVATE message. The low-order word of the *wParam* parameter is zero if the window is being deactivated and nonzero if it is being activated. When the default window procedure receives the **WM_ACTIVATE** message, it sets the keyboard focus to the active window.

To block keyboard and mouse input events from reaching applications, use BlockInput. Note. the **BlockInput** function will not interfere with the asynchronous keyboard input-state table. This means that calling the SendInput function while input is blocked will change the asynchronous keyboard input-state table.

16.1.3 Keystroke Messages

Pressing a key causes a WM_KEYDOWN or WM_SYSKEYDOWN message to be placed in the thread message queue attached to the window that has the keyboard focus. Releasing a key causes a WM_KEYUP or WM_SYSKEYUP message to be placed in the queue.

Key-up and key-down messages typically occur in pairs, but if the user holds down a key long enough to start the keyboard's automatic repeat feature, the system generates a

number of **WM_KEYDOWN** or **WM_SYSKEYDOWN** messages in a row. It then generates a single **WM_KEYUP** or **WM_SYSKEYUP** message when the user releases the key.

16.1.3.1 System and non system keystrokes

The system makes a distinction between system keystrokes and nonsystem keystrokes. System keystrokes produce system keystroke messages, **WM_SYSKEYDOWN** and **WM_SYSKEYUP**. Nonsystem keystrokes produce nonsystem keystroke messages, **WM_KEYDOWN** and **WM_KEYUP**.

If your window procedure must process a system keystroke message, make sure that after processing the message, the procedure passes it to the DefWindowProc function. Otherwise, all system operations involving the ALT key will be disabled whenever the window has the keyboard focus, that is, the user won't be able to access the window's menus or System menu, or use the ALT+ESC or ALT+TAB key combination to activate a different window.

System keystroke messages are primarily used by the system rather than by an application. The system uses them to provide its built-in keyboard interface to menus and to allow the user to control which window is active. System keystroke messages are generated when the user types a key in combination with the ALT key, or when the user types and no window has the keyboard focus (for example, when the active application is minimized). In this case, the messages are posted to the message queue attached to the active window.

Nonsystem keystroke messages are used by application windows; the **DefWindowProc** function does nothing with them. A window procedure can discard any nonsystem keystroke messages that it does not need.

16.1.3.2 Virtual key codes Described

The *wParam* parameter of a keystroke message contains the virtual-key code of the key that was pressed or released. A window procedure processes or ignores a keystroke message, depending on the value of the virtual-key code.

A typical window procedure processes only a small subset of the keystroke messages that it receives and ignores the rest. For example, a window procedure might process only **WM_KEYDOWN** keystroke messages, and only those that contain virtual-key codes for the cursor movement keys, shift keys (also called control keys), and function keys. A typical window procedure does not process keystroke messages from character keys. Instead, it uses the TranslateMessage function to convert the message into character messages.

16.1.3.3 Keystroke Message Flags

The *lParam* parameter of a keystroke message contains additional information about the keystroke that generated the message. This information includes the repeat count, the scan code, the extended-key flag, the context code, the previous key-state flag, and the transition-state flag. The following illustration shows the locations of these flags and values in the *lParam* parameter.

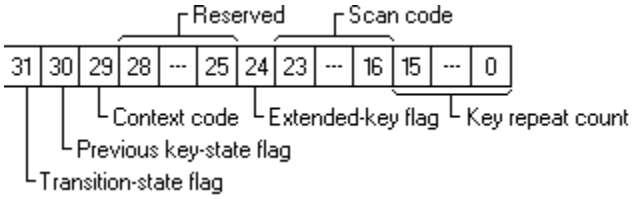


Figure 2

An application can use the following values to manipulate the keystroke flags.

KF_ALTDOWN	Manipulates the ALT key flag, which indicated if the ALT key is pressed.
KF_DLGMODE	Manipulates the dialog mode flag, which indicates whether a dialog box is active.
KF_EXTENDED	Manipulates the extended key flag.
KF_MENUMODE	Manipulates the menu mode flag, which indicates whether a menu is active.
KF_REPEAT	Manipulates the repeat count.
KF_UP	Manipulates the transition state flag.

Repeat Count

You can check the repeat count to determine whether a keystroke message represents more than one keystroke. The system increments the count when the keyboard generates **WM_KEYDOWN** or **WM_SYSKEYDOWN** messages faster than an application can process them. This often occurs when the user holds down a key long enough to start the keyboard's automatic repeat feature. Instead of filling the system message queue with the resulting key-down messages, the system combines the messages into a single key down message and increments the repeat count. Releasing a key cannot start the automatic repeat feature, so the repeat count for **WM_KEYUP** and **WM_SYSKEYUP** messages is always set to 1.

Scan Code

The scan code is the value that the keyboard hardware generates when the user presses a key. It is a device-dependent value that identifies the key pressed, as opposed to the character represented by the key. An application typically ignores scan codes. Instead, it uses the device-independent virtual-key codes to interpret keystroke messages.

Extended-Key Flag

The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the enhanced keyboard. The extended keys consist of the ALT and CTRL keys on the right-hand side of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; the NUM LOCK key, the BREAK (CTRL+PAUSE) key, the PRINT SCRN key, and the divide (/) and ENTER keys in the numeric keypad. The extended-key flag is set if the key is an extended key.

Context Code

The context code indicates whether the ALT key was down when the keystroke message was generated. The code is 1 if the ALT key was down and 0 if it was up.

Previous Key-State Flag

The previous key-state flag indicates whether the key that generated the keystroke message was previously up or down. It is 1 if the key was previously down and 0 if the key was previously up. You can use this flag to identify keystroke messages generated by the keyboard's automatic repeat feature. This flag is set to 1 for **WM_KEYDOWN** and **WM_SYSKEYDOWN** keystroke messages generated by the automatic repeat feature. It is always set to 0 for **WM_KEYUP** and **WM_SYSKEYUP** messages.

Transition-State Flag

The transition-state flag indicates whether pressing a key or releasing a key generated the keystroke message. This flag is always set to 0 for **WM_KEYDOWN** and **WM_SYSKEYDOWN** messages; it is always set to 1 for **WM_KEYUP** and **WM_SYSKEYUP** messages.

16.1.4 Character Messages

Keystroke messages provide a lot of information about keystrokes, but they don't provide character codes for character keystrokes. To retrieve character codes, an application must include the **TranslateMessage** function in its thread message loop. **TranslateMessage** passes a **WM_KEYDOWN** or **WM_SYSKEYDOWN** message to the keyboard layout. The layout examines the message's virtual-key code and, if it corresponds to a character key, it provides the character code equivalent (taking into account the state of the SHIFT

and CAPS LOCK keys). It then generates a character message that includes the character code and places the message at the top of the message queue. The next iteration of the message loop removes the character message from the queue and dispatches the message to the appropriate window procedure.

16.1.4.1 Non-system Character Messages

A window procedure can receive the following character messages: WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, WM_SYSDEADCHAR, and WM_UNICHAR. The **TranslateMessage** function generates a **WM_CHAR** or **WM_DEADCHAR** message when it processes a **WM_KEYDOWN** message. Similarly, it generates a **WM_SYSCHAR** or **WM_SYSDEADCHAR** message when it processes a **WM_SYSKEYDOWN** message.

An application that processes keyboard input typically ignores all but the **WM_CHAR** and **WM_UNICHAR** messages, passing any other messages to the **DefWindowProc** function. Note that **WM_CHAR** uses 16-bit Unicode transformation format (UTF) while **WM_UNICHAR** uses UTF-32. The system uses the **WM_SYSCHAR** and **WM_SYSDEADCHAR** messages to implement menu mnemonics.

The *wParam* parameter of all character messages contains the character code of the character key that was pressed. The value of the character code depends on the window class of the window receiving the message. If the Unicode version of the RegisterClass function was used to register the window class, the system provides Unicode characters to all windows of that class. Otherwise, the system provides ASCII character codes. For more information, see Unicode and Character Sets.

The contents of the *lParam* parameter of a character message are identical to the contents of the *lParam* parameter of the key-down message that was translated to produce the character message.

16.1.4.2 Dead-Character Messages

Some non-English keyboards contain character keys that are not expected to produce characters by them. Instead, they are used to add a diacritic to the character produced by the subsequent keystroke. These keys are called *dead keys*. The circumflex key on a German keyboard is an example of a dead key. To enter the character consisting of an "o" with a circumflex, a German user would type the circumflex key followed by the "o" key. The window with the keyboard focus would receive the following sequence of messages:

- ✓ **WM_KEYDOWN**
- ✓ **WM_DEADCHAR**
- ✓ **WM_KEYUP**
- ✓ **WM_KEYDOWN**
- ✓ **WM_CHAR**
- ✓ **WM_KEYUP**

TranslateMessage generates the **WM_DEADCHAR** message when it processes the **WM_KEYDOWN** message from a dead key. Although the *wParam* parameter of the **WM_DEADCHAR** message contains the character code of the diacritic for the dead key, an application typically ignores the message. Instead, it processes the **WM_CHAR** message generated by the subsequent keystroke. The **WM_CHAR** parameter of the **WM_CHAR** message contains the character code of the letter with the diacritic. If the subsequent keystroke generates a character that cannot be combined with a diacritic, the system generates two **WM_CHAR** messages. The *wParam* parameter of the first contains the character code of the diacritic; the *wParam* parameter of the second contains the character code of the subsequent character key.

The **TranslateMessage** function generates the **WM_SYSDEADCHAR** message when it processes the **WM_SYSKEYDOWN** message from a system dead key (a dead key that is pressed in combination with the ALT key). An application typically ignores the **WM_SYSDEADCHAR** message.

16.1.5 Key Status

While processing a keyboard message, an application may need to determine the status of another key besides the one that generated the current message. For example, a word-processing application that allows the user to press SHIFT+END to select a block of text must check the status of the SHIFT key whenever it receives a keystroke message from the END key. The application can use the *GetKeyState* function to determine the status of a virtual key at the time the current message was generated; it can use the *GetAsyncKeyState* function to retrieve the current status of a virtual key.

The keyboard layout maintains a list of names. The name of a key that produces a single character is the same as the character produced by the key. The name of a noncharacter key such as TAB and ENTER is stored as a character string. An application can retrieve the name of any key from the device driver by calling the *GetKeyNameText* function.

16.1.6 Key Stroke and Character Translations

The system includes several special purpose functions that translate scan codes, character codes, and virtual-key codes provided by various keystroke messages. These functions include *MapVirtualKey*, *ToAscii*, *ToUnicode*, and *VkKeyScan.*

16.1.7 Hot-key Support

A *hot key* is a key combination that generates a WM_HOTKEY message, a message the system places at the top of a thread's message queue, bypassing any existing messages in the queue. Applications use hot keys to obtain high-priority keyboard input from the user. For example, by defining a hot key consisting of the CTRL+C key combination, an application can allow the user to cancel a lengthy operation.

To define a hot key, an application calls the RegisterHotKey function, specifying the combination of keys that generates the **WM_HOTKEY** message, the handle to the

window to receive the message, and the identifier of the hot key. When the user presses the hot key, a **WM_HOTKEY** message is placed in the message queue of the thread that created the window. The *wParam* parameter of the message contains the identifier of the hot key. The application can define multiple hot keys for a thread, but each hot key in the thread must have a unique identifier. Before the application terminates, it should use the UnregisterHotKey function to destroy the hot key.

Applications can use a hot key control to make it easy for the user to choose a hot key. Hot key controls are typically used to define a hot key that activates a window; they do not use the **RegisterHotKey** and **UnregisterHotKey** functions. Instead, an application that uses a hot key control typically sends the WM_SETHOTKEY message to set the hot key. Whenever the user presses the hot key, the system sends a WM_SYSCOMMAND message specifying SC_HOTKEY. For more information about hot key controls, see "Using Hot Key Controls" in Hot Key Controls.

16.1.8 Languages, Locals, and Keyboard Layouts

A *language* is a natural language, such as English, French, and Japanese. A *sublanguage* is a variant of a natural language that is spoken in a specific geographical region, such as the English sublanguages spoken in England and the United States. Applications use values, called language identifiers, to uniquely identify languages and sublanguages.

Applications typically use *locales* to set the language in which input and output is processed. Setting the locale for the keyboard, for example, affects the character values generated by the keyboard. Setting the locale for the display or printer affects the glyphs displayed or printed. Applications set the locale for a keyboard by loading and using keyboard layouts. They set the locale for a display or printer by selecting a font that supports the specified locale.

A keyboard layout not only specifies the physical position of the keys on the keyboard but also determines the character values generated by pressing those keys. Each layout identifies the current input language and determines which character values are generated by which keys and key combinations.

Every keyboard layout has a corresponding handle that identifies the layout and language. The low word of the handle is a language identifier. The high word is a device handle specifying the physical layout, or is zero indicating a default physical layout. The user can associate any input language with a physical layout. For example, an English-speaking user who very occasionally works in French can set the input language of the keyboard to French without changing the physical layout of the keyboard. This means the user can enter text in French using the familiar English layout.

Applications are generally not expected to manipulate input languages directly. Instead, the user sets up language and layout combinations, and then switches among them. When the user clicks into text marked with a different language, the application calls the ActivateKeyboardLayout function to activate the user's default layout for that language. If the user edits text in a language which is not in the active list, the application can call

the LoadKeyboardLayout function with the language to get a layout based on that language.

The **ActivateKeyboardLayout** function sets the input language for the current task. The *hkl* parameter can be either the handle to the keyboard layout or a zero-extended language identifier. Keyboard layout handles can be obtained from the **LoadKeyboardLayout** or GetKeyboardLayoutList function. The HKL_NEXT and HKL_PREV values can also be used to select the next or previous keyboard.

The GetKeyboardLayoutName function retrieves the name of the active keyboard layout for the calling thread. If an application creates the active layout using the **LoadKeyboardLayout** function, **GetKeyboardLayoutName** retrieves the same string used to create the layout. Otherwise, the string is the primary language identifier corresponding to the locale of the active layout. This means the function may not necessarily differentiate among different layouts with the same primary language so, it cannot return specific information about the input language. The GetKeyboardLayout function, however, can be used to determine the input language.

The **LoadKeyboardLayout** function loads a keyboard layout and makes the layout available to the user. Applications can make the layout immediately active for the current thread by using the KLF_ACTIVATE value. An application can use the KLF_REORDER value to reorder the layouts without also specifying the KLF_ACTIVATE value. Applications should always use the KLF_SUBSTITUTE_OK value when loading keyboard layouts to ensure that the user's preference, if any, is selected.

For multilingual support, the **LoadKeyboardLayout** function provides the KLF_REPLACELANG and KLF_NOTELLSHELL flags. The KLF_REPLACELANG flag directs the function to replace an existing keyboard layout without changing the language. Attempting to replace an existing layout using the same language identifier but without specifying KLF_REPLACELANG is an error. The KLF_NOTELLSHELL flag prevents the function from notifying the shell when a keyboard layout is added or replaced. This is useful for applications that add multiple layouts in a consecutive series of calls. This flag should be used in all but the last call.

The UnloadKeyboardLayout function is restricted in that it cannot unload the system default input language. This ensures that the user always has one layout available for enter text using the same character set as used by the shell and file system.

16.1.9 Keyboard Messages (*brief*)

The following are some of the keyboard messages.

WM_KEYDOWN: when the key down
WM_KEYUP: when the key up
WM_SYSKEYDOWN: when the system key down, e.g. ALT key
WM_SYSKEYUP: when the system key up

When user press Ctrl + key then two messages of WM_KEYDOWN and two messages WM_KEYUP are sent to the Application Message Queue.

16.1.10 Key down message format

The **WM_KEYDOWN** message is posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is *not* pressed.

```
WM_KEYDOWN

    WPARAM wParam       //wParam of the key down
    LPARAM lParam;      /*lParam   of   the   key   down
messages*/
```

wParam: Specifies the virtual-key code of the nonsystem key.

lParam: Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

0-15: Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23: Specifies the scan code. The value depends on the OEM.
24: Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28: Reserved; do not use.
29: Specifies the context code. The value is always 0 for a **WM_KEYDOWN** message.
30: Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is zero if the key is up.
31: Specifies the transition state. The value is always zero for a **WM_KEYDOWN** message.

16.1.11 Character message format

The **WM_CHAR** message is posted to the window with the keyboard focus when a WM_KEYDOWN message is translated by the *TranslateMessage* function. The **WM_CHAR** message contains the character code of the key that was pressed.

```
WM_CHAR

    WPARAM wParam /*character code in this message
```

```
    LPARAM lParam; /*data or scan code of character
parameter*/
```

wParam: Specifies the character code of the key.
lParam: Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag.

16.1.12 Getting Key State

GetKeyState function gets the key state either its pressed or un-pressed.

```
SHORT GetKeyState(
        Int vVirtKey)  //virtual key code
);
```

There is another function available which is:

```
SHORT GetAsyncKeyState(
        Int vVirtKey            //virtual key code
);
```

Their complete description can be found from Microsoft Help System.

16.1.13 Character Message Processing

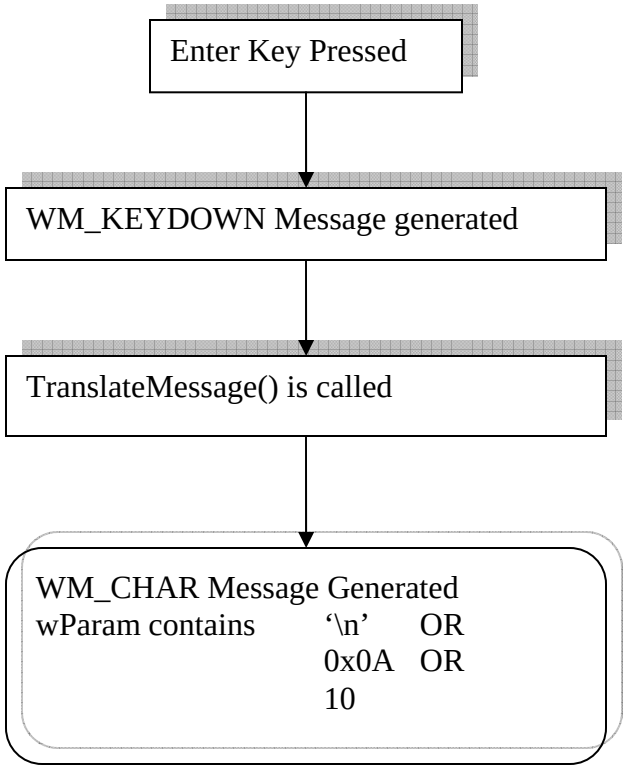


Figure 3

16.2 Caret

A caret is a blinking line, block, or bitmap in the client area of a window. The caret typically indicates the place at which text or graphics will be inserted.

The system provides one caret per message queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

Use the *CreateCaret* function to specify the parameters for a caret. The system forms a caret by inverting the pixel color within the rectangle specified by the caret's position, width, and height. The width and height are specified in logical units; therefore, the appearance of a caret is subject to the window's mapping mode.

16.2.1 Caret Visibility

After the caret is defined, use the *ShowCaret* function to make the caret visible. When the caret appears, it automatically begins flashing. To display a solid caret, the system inverts every pixel in the rectangle; to display a gray caret, the system inverts every other pixel; to display a bitmap caret, and the system inverts only the white bits of the bitmap.

16.2.2 Caret Blink Time

The elapsed time, in milliseconds, required to invert the caret is called the *blink time*. The caret will blink as long as the thread that owns the message queue has a message pump processing the messages.

The user can set the blink time of the caret using the Control Panel and applications should respect the settings that the user has chosen. An application can determine the caret's blink time by using the GetCaretBlinkTime function. If you are writing an application that allows the user to adjust the blink time, such as a Control Panel applet, use the SetCaretBlinkTime function to set the rate of the blink time to a specified number of milliseconds.

The *flash time* is the elapsed time, in milliseconds, required to display, invert, and restore the caret's display. The flash time of a caret is twice as much as the blink time.

16.2.3 Caret Position

You can determine the position of the caret using the GetCaretPos function. The position, in client coordinates, is copied to a structure specified by a parameter in **GetCaretPos**. An application can move a caret in a window by using the SetCaretPos function. A window can move a caret only if it already owns the caret. **SetCaretPos** can move the caret whether it is visible or not.

16.2.4 Removing a Caret

You can temporarily remove a caret by hiding it, or you can permanently remove the caret by destroying it. To hide the caret, use the *HideCaret* function. This is useful when your application must redraw the screen while processing a message, but must keep the caret out of the way. When the application finishes drawing, it can display the caret again by using the *ShowCaret* function. Hiding the caret does not destroy its shape or invalidate the insertion point. Hiding the caret is cumulative, that is, if the application calls *HideCaret* five times, it must also call *ShowCaret* five times before the caret will reappear.

To remove the caret from the screen and destroy its shape, use using the *DestroyCaret* function. *DestroyCaret* destroys the caret only if the window involved in the current task owns the caret.

16.2.5 Caret Functions

The following functions are used to handle a caret

- CreateCaret()
- DestroyCaret()
- SetCaretPos()

16.3 Mouse

In old ages or in Dos age, mouse is initialized with the interrupt INT 33h. Now-a-days we don’t need to use interrupt and its services because we have windows which provides APIs instead of interrupts. So using these APIs we can handle mouse and its other properties.

The mouse is an important, but optional, user-input device for applications. A well-written application should include a mouse interface, but it should not depend solely on the mouse for acquiring user input. The application should provide full keyboard support as well.

An application receives mouse input in the form of messages that are sent or posted to its windows.

16.3.1 Mouse Cursor

When the user moves the mouse, the system moves a bitmap on the screen called the *mouse cursor*. The mouse cursor contains a single-pixel point called the *hot spot*, a point that the system tracks and recognizes as the position of the cursor. When a mouse event occurs, the window that contains the hot spot typically receives the mouse message

resulting from the event. The window need not be active or have the keyboard focus to receive a mouse message.

The system maintains a variable that controls mouse speed, that is, the distance the cursor moves when the user moves the mouse. You can use the SystemParametersInfo function with the SPI_GETMOUSE or SPI_SETMOUSE flag to retrieve or set mouse speed. For more information about mouse cursors, see Cursors.

16.3.2 Mouse Capture

The system typically posts a mouse message to the window that contains the cursor hot spot when a mouse event occurs. An application can change this behavior by using the *SetCapture* function to route mouse messages to a specific window. The window receives all mouse messages until the application calls the *ReleaseCapture* function or specifies another capture window, or until the user clicks a window created by another thread.

When the mouse captures changes, the system sends a WM_CAPTURECHANGED message to the window that is losing the mouse capture. The *lParam* parameter of the message specifies a handle to the window that is gaining the mouse capture.

Only the foreground window can capture mouse input. When a background window attempts to capture mouse input, it receives messages only for mouse events that occur when the cursor hot spot is within the visible portion of the window.

Capturing mouse input is useful if a window must receive all mouse input, even when the cursor moves outside the window. For example, an application typically tracks the cursor position after a mouse button down event, following the cursor until a mouse button up event occurs. If an application has not captured mouse input and the user releases the mouse button outside the window, the window does not receive the button-up message.

A thread can use the *GetCapture* function to determine whether one of its windows has captured the mouse. If one of the thread's windows has captured the mouse, *GetCapture* retrieves a handle to the window.

16.3.3 Mouse Configuration

Although the mouse is an important input device for applications, not every user necessarily has a mouse.

An application can determine whether the system includes a mouse by passing the SM_MOUSEPRESENT value to the GetSystemMetrics function.

Windows supports a mouse having up to three buttons. On a three-button mouse, the buttons are designated as the left, middle, and right buttons. Messages and named constants related to the mouse buttons use the letters L, M, and R to identify the buttons. The button on a single-button mouse is considered to be the left button. Although

Windows supports a mouse with multiple buttons, most applications use the left button primarily and the others minimally, if at all.

*An application can determine the number of buttons on the mouse by passing the SM_CMOUSEBUTTONS value to the **GetSystemMetrics** function.*

To configure the mouse for a left-handed user, the application can use the *SwapMouseButton* function to reverse the meaning of the left and right mouse buttons. Passing the SPI_SETMOUSEBUTTONSWAP value to the ***SystemParametersInfo*** function is another way to reverse the meaning of the buttons. Note, however, that the mouse is a shared resource, so reversing the meaning of the buttons affects all applications

16.3.4 Mouse Messages

The mouse generates an input event when the user moves the mouse, or presses or releases a mouse button. The system converts mouse input events into messages and posts them to the appropriate thread's message queue. When mouse messages are posted faster than a thread can process them, the system discards all but the most recent mouse message.

A window receives a mouse message when a mouse event occurs while the cursor is within the borders of the window, or when the window has captured the mouse. Mouse messages are divided into two groups: client area messages and nonclient area messages. Typically, an application processes client area messages and ignores nonclient area messages.

16.3.4.1 Client Area Mouse Messages

A window receives a client area mouse message when a mouse event occurs within the window's client area. The system posts the WM_MOUSEMOVE message to the window when the user moves the cursor within the client area. It posts one of the following messages when the user presses or releases a mouse button while the cursor is within the client area.

Message	Meaning
WM_LBUTTONDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_LBUTTONUP	The left mouse button was released.
WM_MBUTTONDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.
WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.
WM_RBUTTONUP	The right mouse button was released.

In addition, an application can call the *TrackMouseEvent* function to have the system send two other messages. It posts the WM_MOUSEHOVER message when the cursor hovers over the client area for a certain time period. It posts the WM_MOUSELEAVE message when the cursor leaves the client area.

Message Parameters

The *lParam* parameter of a client area mouse message indicates the position of the cursor hot spot. The low-order word indicates the x-coordinate of the hot spot, and the high-order word indicates the y-coordinate. The coordinates are specified in client coordinates. In the client coordinate system, all points on the screen are specified relative to the coordinates (0, 0) of the upper-left corner of the client area.

The *wParam* parameter contains flags that indicate the status of the other mouse buttons and the CTRL and SHIFT keys at the time of the mouse event. You can check for these flags when mouse-message processing depends on the state of another mouse button or of the CTRL or SHIFT key. The *wParam* parameter can be a combination of the following values.

Value	Meaning
MK_CONTROL	The CTRL key is down.
MK_LBUTTON	The left mouse button is down.
MK_MBUTTON	The middle mouse button is down.
MK_RBUTTON	The right mouse button is down.
MK_SHIFT	The SHIFT key is down.

Double Click Messages

The system generates a double-click message when the user clicks a mouse button twice in quick succession. When the user clicks a button, the system establishes a rectangle centered around the cursor hot spot. It also marks the time at which the click occurred. When the user clicks the same button a second time, the system determines whether the hot spot is still within the rectangle and calculates the time elapsed since the first click. If the hot spot is still within the rectangle and the elapsed time does not exceed the double-click time-out value, the system generates a double-click message.

An application can get and set double-click time-out values by using the *GetDoubleClickTime* and *SetDoubleClickTime* functions, respectively. Alternatively, the application can set the double-click–time-out value by using the SPI_SETDOUBLECLICKTIME flag with the **SystemParametersInfo** function. It can also set the size of the rectangle that the system uses to detect double-clicks by passing the SPI_SETDOUBLECLKWIDTH and SPI_SETDOUBLECLKHEIGHT flags to

SystemParametersInfo. Note, however, that setting the double-click–time-out value and rectangle affects all applications.

An application-defined window does not, by default, receive double-click messages. Because of the system overhead involved in generating double-click messages, these messages are generated only for windows belonging to classes that have the **CS_DBLCLKS** class style. Your application must set this style when registering the window class. For more information, see Window Classes.

A double-click message is always the third message in a four-message series. The first two messages are the button-down and button-up messages generated by the first click. The second click generates the double-click message followed by another button-up message. For example, double-clicking the left mouse button generates the following message sequence:

- **WM_LBUTTONDOWN**
- **WM_LBUTTONUP**
- **WM_LBUTTONDBLCLK**
- **WM_LBUTTONUP**

Because a window always receives a button-down message before receiving a double-click message, an application typically uses a double-click message to extend a task it began during a button-down message. For example, when the user clicks a color in the color palette of Microsoft Paint, Paint displays the selected color next to the palette. When the user double-clicks a color, Paint displays the color and opens the **Edit Colors** dialog box.

16.3.4.2 Non Client Area Mouse Messages

A window receives a nonclient area mouse message when a mouse event occurs in any part of a window except the client area. A window's nonclient area consists of its border, menu bar, title bar, scroll bar, window menu, minimize button, and maximize button.

The system generates nonclient area messages primarily for its own use. For example, the system uses nonclient area messages to change the cursor to a two-headed arrow when the cursor hot spot moves into a window's border. A window must pass nonclient area mouse messages to the **DefWindowProc** function to take advantage of the built-in mouse interface.

There is a corresponding nonclient area mouse message for each client area mouse message. The names of these messages are similar except that the named constants for the nonclient area messages include the letters NC. For example, moving the cursor in the nonclient area generates a *WM_NCMOUSEMOVE* message, and pressing the left mouse button while the cursor is in the nonclient area generates a *WM_NCLBUTTONDOWN* message.

The *lParam* parameter of a nonclient area mouse message is a structure that contains the x- and y-coordinates of the cursor hot spot. Unlike coordinates of client area mouse messages, the coordinates are specified in screen coordinates rather than client coordinates. In the screen coordinate system, all points on the screen are relative to the coordinates (0,0) of the upper-left corner of the screen.

The *wParam* parameter contains a hit-test value, a value that indicates where in the nonclient area the mouse event occurred.

16.3.4.3 The WM_NCHITTEST Message

Whenever a mouse event occurs, the system sends a WM_NCHITTEST message to either the window that contains the cursor hot spot or the window that has captured the mouse. The system uses this message to determine whether to send a client area or nonclient area mouse message. An application that must receive mouse movement and mouse button messages must pass the **WM_NCHITTEST** message to the **DefWindowProc** function.

The *lParam* parameter of the **WM_NCHITTEST** message contains the screen coordinates of the cursor hot spot. The **DefWindowProc** function examines the coordinates and returns a hit-test value that indicates the location of the hot spot. The hit-test value can be one of the following values.

Value	**Location of hot spot**
HTBORDER	In the border of a window that does not have a sizing border.
HTBOTTOM	In the lower-horizontal border of a window.
HTBOTTOMLEFT	In the lower-left corner of a window border.
HTBOTTOMRIGHT	In the lower-right corner of a window border.
HTCAPTION	In a title bar.
HTCLIENT	In a client area.
HTCLOSE	In a **Close** button.
HTERROR	On the screen background or on a dividing line between windows (same as HTNOWHERE, except that the **DefWindowProc** function produces a system beep to indicate an error).
HTGROWBOX	In a size box (same as HTSIZE).
HTHELP	In a **Help** button.
HTHSCROLL	In a horizontal scroll bar.
HTLEFT	In the left border of a window.
HTMENU	In a menu.
HTMAXBUTTON	In a **Maximize** button.
HTMINBUTTON	In a **Minimize** button.
HTNOWHERE	On the screen background or on a dividing line between windows.
HTREDUCE	In a **Minimize** button.

HTRIGHT	In the right border of a window.
HTSIZE	In a size box (same as HTGROWBOX).
HTSYSMENU	In a **System** menu or in a **Close** button in a child window.
HTTOP	In the upper-horizontal border of a window.
HTTOPLEFT	In the upper-left corner of a window border.
HTTOPRIGHT	In the upper-right corner of a window border.
HTTRANSPARENT	In a window currently covered by another window in the same thread.
HTVSCROLL	In the vertical scroll bar.
HTZOOM	In a **Maximize** button.

If the cursor is in the client area of a window, **DefWindowProc** returns the HTCLIENT hit-test value to the window procedure. When the window procedure returns this code to the system, the system converts the screen coordinates of the cursor hot spot to client coordinates, and then posts the appropriate client area mouse message.

The **DefWindowProc** function returns one of the other hit-test values when the cursor hot spot is in a window's nonclient area. When the window procedure returns one of these hit-test values, the system posts a nonclient area mouse message, placing the hit-test value in the message's *wParam* parameter and the cursor coordinates in the *lParam* parameter.

16.3.5 Screen and Client Area Coordinates

Screen coordinates start from the top left corner of the screen and end to right bottom coordinates of the screen.
But Client Area coordinates start from the top left coordinate of the client area of the window and ends with right-bottom coordinate of the client area of the window.

These coordinates can be converted to each other. Conversion from screen area coordinates to client area coordinates can be done by using function

```
BOOL ScreenToClient(
HWND hWnd,              //handle to the window
LPPOINT lpPoint         //point structure
);
```

And conversion from client area coordinates of window to screen area coordinates can be done by using function.

```
BOOL ClientToScreen(
HWND hWnd,              //handle to the window
LPPOINT lpPoint         //point structure
);
```

Summary

In this lecture, we studied about the input devices. Input devices include keyboard and mouse. Keyboard is used to input the system. Whenever we press a key on keyboard, we generate a message. This message directly goes to the Operating system and then rout to our application. Keyboard messages include Key down and key up messages. Another type of messages is character messages these messages also come from keyboard. Keyboard messages are translated to their Character values and then send to the application in form of character message. Mouse is another input device. Almost all user interfaces uses mouse as input device as well as keyboard. Mouse device is optional in the system but useful in complex applications. Mouse can be used to point anywhere on screen. Mouse sends different messages e.g. mouse can send left button down message when the mouse left button is down and in the same way left button up message when the left button is up. During the movement of mouse pointer on screen mouse move message is always sent. During input session, caret is used to position the keyboard. Caret shows character can be placed where the caret is blinking.

Exercises

1. Create your own status bar and show it in a window at its proper location. This status bar should display current time and NUM Lock, CAPS LOCK, SCROLL lock states. If these keys are pressed show them otherwise don’t show.
2. Using the mouse messages draw a line which starts when a mouse left button is down and end when the mouse left button is up. During the mouse pressed state if ESC key is pressed the process should be cancelled and line should not be drawn.

Chapter 17

Resources

Resource is binary data that you can add to the executable file of a Windows-based application. A resource can be either standard or defined. The data in a standard resource describes an icon, cursor, menu, dialog box, bitmap, enhanced metafile, font, accelerator table, message-table entry, string-table entry, or version information. An application-defined resource, also called a custom resource, contains any data required by a specific application.

17.1 Types of windows resources

Following are the Windows Resources are used in windows.

- Accelerator
- String Table
- Icon
- Bitmap
- Dialog
- Menu
- Cursor
- Version

17.2 Resource Definition Statements

The resource-definition statements define the resources that the resource compiler puts in the resource (.Res) file. After the .Res file is linked to the executable file, the application can load its resources at run time as needed. All resource statements associate an identifying name or number with a given resource.

The resource-definition statements can be divided into the following categories:

- Resources
- Controls
- Statements

The following tables describe the resource-definition statements.

Resources

Resource	Description
ACCELERATORS	Defines menu accelerator keys.
BITMAP	Defines a bitmap by naming it and specifying the name of the file that contains it. (To use a particular bitmap, the application requests it by name.)
CURSOR	Defines a cursor or animated cursor by naming it and specifying the name of the file that contains it. (To use a particular cursor, the application requests it by name.)

DIALOG	Defines a template that an application can use to create dialog boxes.
DIALOGEX	Defines a template that an application can use to create dialog boxes.
FONT	Specifies the name of a file that contains a font.
ICON	Defines an icon or animated icon by naming it and specifying the name of the file that contains it. (To use a particular icon, the application requests it by name.)
MENU	Defines the appearance and function of a menu.
MENUEX	Defines the appearance and function of a menu.
MESSAGETABLE	Defines a message table by naming it and specifying the name of the file that contains it. The file is a binary resource file generated by the message compiler.
POPUP	Defines a menu item that can contain menu items and submenus.
RCDATA	Defines data resources. Data resources let you include binary data in the executable file.
STRINGTABLE	Defines string resources. String resources are Unicode or ASCII strings that can be loaded from the executable file.
User-Defined	Defines a resource that contains application-specific data.
VERSIONINFO	Defines a version-information resource. Contains information such as the version number, intended operating system, and so on.

Controls

Control	Description
AUTO3STATE	Creates an automatic three-state check box control.
AUTOCHECKBOX	Creates an automatic check box control.
AUTORADIOBUTTON	Creates an automatic radio button control.
CHECKBOX	Creates a check box control.
COMBOBOX	Creates a combo box control.
CONTROL	Creates an application-defined control.
CTEXT	Creates a centered-text control.
DEFPUSHBUTTON	Creates a default pushbutton control.
EDITTEXT	Creates an edit control.
GROUPBOX	Creates a group box control.
ICON	Creates an icon control. This control is an icon displayed in a dialog box.
LISTBOX	Creates a list box control.
LTEXT	Creates a left-aligned text control.
PUSHBOX	Creates a push box control.
PUSHBUTTON	Creates a push button control.

RADIOBUTTON	Creates a radio button control.
RTEXT	Creates a right-aligned control.
SCROLLBAR	Creates a scroll bar control.
STATE3	Creates a three-state check box control.

Statements

Statement	Description
CAPTION	Sets the title for a dialog box.
CHARACTERISTICS	Specifies information about a resource that can be used by tool that can read or write resource-definition files.
CLASS	Sets the class of the dialog box.
EXSTYLE	Sets the extended window style of the dialog box.
FONT	Sets the font with which the system will draw text for the dialog box.
LANGUAGE	Sets the language for all resources up to the next **LANGUAGE** statement or to the end of the file. When the **LANGUAGE** statement appears before the beginning of the body of an **ACCELERATORS**, **DIALOG**, **MENU**, **RCDATA**, or **STRINGTABLE** resource definition, the specified language applies only to that resource.
MENU	Sets the menu for the dialog box.
MENUITEM	Defines a menu item.
STYLE	Sets the window style for the dialog box.
VERSION	Specifies version information for a resource that can be used by tool that can read or write resource-definition files.

17.3 .rc files (*resource files*)

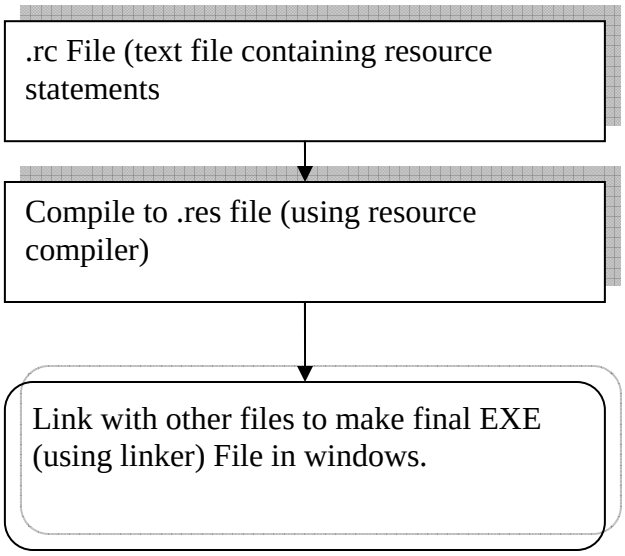


Figure 1

17.4 Resource Statements in Resource File

ICON resource statement in a resource file (.rc)

```
#define IDI_ICON 101

IDI_ICON     ICON        DISCARDABLE     “vu.ico”
Integer id   reserved                     icon
             word                         filename
101          ICON        DISCARDABLE     “vu.ico”
```

17.5 Using Resource Compiler (RC)

To start RC, use the **RC** command.

```
RC [[options]] script-file
```

The *script-file* parameter specifies the name of the resource-definition file that contains the names, types, filenames, and descriptions of the resources to be compiled. The *options* parameter can be one or more of the following command-line options.

Options

/?
Displays a list of **RC** command-line options.

/d
Defines a symbol for the preprocessor that you can test with the **#ifdef** directive.

/fo *resname*
Uses *resname* for the name of the .RES file.

/h
Displays a list of **RC** command-line options.

/i
Searches the specified directory before searching the directories specified by the INCLUDE environment variable.

/l *codepage*
Specifies default language for compilation. For example, -l409 is equivalent to including the following statement at the top of the resource script file:
`LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US`

For more information, see Language Identifiers.

Alternatively, you use #pragma code_page(409) in the .RC file.

/n Null terminates all strings in the string table.

/r Ignored. Provided for compatibility with existing makefiles.

/u Undefines a symbol for the preprocessor.

/v Display messages that report on the progress of the compiler.

/x Prevents RC from checking the INCLUDE environment variable when searching for header files or resource files.

Options are not case sensitive and a hyphen (-) can be used in place of a slash mark (/). You can combine single-letter options if they do not require any additional parameters. For example, the following two commands are equivalent:

```
rc /V /X SAMPLE.RC
rc -vx sample.rc
```

17.6 Loading an Icon from the resource table

The **LoadIcon** function loads the specified icon resource from the executable (.exe) file associated with an application instance.

```
HICON LoadIcon(

    HINSTANCE hInstance,  /*handle to the instance*/
    LPCTSTR lpIconName    /*string to the icon data*/
);
```

hInstance: Handle to an instance of the module whose executable file contains the icon to be loaded. This parameter must be NULL when a standard icon is being loaded.

lpIconName:
Pointer to a null-terminated string that contains the name of the icon resource to be loaded. Alternatively, this parameter can contain the resource identifier in the low-order word and zero in the high-order word. Use the MAKEINTRESOURCE macro to create this value.

> To use one of the predefined icons, set the *hInstance* parameter to NULL and the *lpIconName* parameter to one of the following values.
>
> IDI_APPLICATION: Default application icon.
> IDI_ASTERISK: Same as IDI_INFORMATION.
> IDI_ERROR: Hand-shaped icon.
> IDI_EXCLAMATION: Same as IDI_WARNING.
> IDI_HAND: Same as IDI_ERROR.
> IDI_INFORMATION: Asterisk icon.
> IDI_QUESTION: Question mark icon.
> IDI_WARNING: Exclamation point icon.
> IDI_WINLOGO: Windows logo icon.

Return Value:

If the function succeeds, the return value is a handle to the newly loaded icon.
If the function fails, the return value is NULL. To get extended error information, use *GetLastError.*

LoadIcon loads the icon resource only if it has not been loaded; otherwise, it retrieves a handle to the existing resource. The function searches the icon resource for the icon most appropriate for the current display. The icon resource can be a color or monochrome bitmap.

LoadIcon can only load an icon whose size conforms to the SM_CXICON and SM_CYICON system metric values. Use the **LoadImage** function to load icons of other sizes.

17.7 String table in a resource file

```
#include “resource.h”

STRINGTABLE DISCARDABLE
BEGIN
        IDS_STRING1         “This is Virtual University"
        IDS_STRING2         "MyWindowClass"
        IDS_STRING3         “My Novel Programme"
END
```

17.8 Loading String

The **LoadString** function loads a string resource from the executable file associated with a specified module, copies the string into a buffer, and appends a terminating null character.

```
int LoadString(

    HINSTANCE hInstance,//handle to application instance*/
    UINT uID,             /*//id of the string*/
    LPTSTR lpBuffer,      /*buffer to receive string data*/
    int nBufferMax        /*maximum buffer size is available
for the string data to store*/
);
```

hInstance: Handle to an instance of the module whose executable file contains the string resource. To get the handle for the application itself, use GetModuleHandle(NULL).

uID: Specifies the integer identifier of the string to be loaded.

lpBuffer: Pointer to the buffer to receive the string.

nBufferMax: Specifies the size of the buffer, in **TCHAR**s. This refers to bytes for versions of the function or **WCHAR**s for Unicode versions. The string is truncated and null terminated if it is longer than the number of characters specified.

*Return Value:*If the function succeeds, the return value is the number of **TCHAR**s copied into the buffer, not including the null-terminating character, or zero if the string resource does not exist. To get extended error information, call GetLastError.

17.9 Keyboard Accelerator

A *keyboard accelerator*, also known as a shortcut key, is a keystroke or combination of keystrokes that generates a WM_COMMAND message. Keyboard accelerators are often used as shortcuts for commonly used menu commands, but you can also use them to generate commands that have no equivalent menu items. Include keyboard accelerators for any common or frequent actions, and provide support for the common shortcut keys where they apply.

You can use an ASCII character code or a *virtual-key* code to define the accelerator. A virtual key is a device-independent value that identifies the purpose of a keystroke as interpreted by the Windows keyboard device driver. An ASCII character code makes the accelerator case-sensitive. The ASCII "C" character can define the accelerator as ALT+c rather than ALT+C. Because accelerators do not need to be case-sensitive, most applications use virtual-key codes for accelerators rather than ASCII character codes.

To create an accelerator table

1. Use a resource compiler to define an *accelerator table* resource and add it to your executable file.

   An accelerator table consists of an array of **ACCEL** data structures, each of which defines an individual accelerator.

2. Call the **LoadAccelerators** function at run time to load the accelerator table and to retrieve the handle of the accelerator table.
3. Pass a handle to the accelerator table to the **TranslateAccelerator** function to activate the accelerator table.

17.10 Defining an Accelerator

```
#define ID_DO_BACK 1001
#define ID_ACC2 1002
#define ID_DRAWSTRING 1003
```

```
ACCELERATOR ACCELERATORS DISCARDABLE
BEGIN
        VK_BACK,        ID_DO_BACK,    VIRTKEY,  ALT,  NOINVERT
        VK_DELETE,      ID_ACC2,    VIRTKEY,  ALT,  NOINVERT     … …… .. .. .
        . ..  . . ..     "^S",          ID_DRAWSTRING,          ASCII, NOINVERT
END

Labelling: Virtual Key or ASCII    ID
Options(VIRTKEY, ASCII, ALT, CONTROL)
```

17.11 Loading Accelerator Resource

The **LoadAccelerators** function loads the specified accelerator table.

```
HACCEL LoadAccelerators(

    HINSTANCE hInstance,  /*//handle to the application
instance*/
    LPCTSTR lpTableName   /*string to the table name*/
);
```

hInstance: Handle to the module whose executable file contains the accelerator table to load.
lpTableName: Pointer to a null-terminated string that contains the name of the accelerator table to load. Alternatively, this parameter can specify the resource identifier of an accelerator-table resource in the low-order word and zero in the high-order word. To create this value, use the MAKEINTRESOURCE macro.

Return Value: If the function succeeds, the return value is a handle to the loaded accelerator table.
If the function fails, the return value is NULL. To get extended error information, call GetLastError.

17.12 Translate Accelerator

The ***TranslateAccelerator*** function processes accelerator keys for menu commands. The function translates a WM_KEYDOWN or WM_SYSKEYDOWN message to a WM_COMMAND or WM_SYSCOMMAND message (if there is an entry for the key in the specified accelerator table) and then sends the **WM_COMMAND** or **WM_SYSCOMMAND** message directly to the appropriate window procedure. **TranslateAccelerator** does not return until the window procedure has processed the message.

```
int TranslateAccelerator(

    HWND hWnd,             /*handle to the window to whom
accelerator attached*/
    HACCEL hAccTable,      /*accelerate table*/
    LPMSG lpMsg            /*MSG structure*/
);
```

hWnd: Handle to the window whose messages are to be translated.

hAccTable: Handle to the accelerator table. The accelerator table must have been loaded by a call to the LoadAccelerators function or created by a call to the *CreateAcceleratorTable* function.

lpMsg: Pointer to an MSG structure that contains message information retrieved from the calling thread's message queue using the *GetMessage* or *PeekMessage* function.

Return Value: If the function succeeds, the return value is nonzero.If the function fails, the return value is zero.

To differentiate the message that this function sends from messages sent by menus or controls, the high-order word of the *wParam* parameter of the **WM_COMMAND** or **WM_SYSCOMMAND** message contains the value 1.

Accelerator key combinations used to select items from the **window** menu are translated into **WM_SYSCOMMAND** messages; all other accelerator key combinations are translated into **WM_COMMAND** messages.

An accelerator need not correspond to a menu command.

If the accelerator command corresponds to a menu item, the application is sent WM_INITMENU and WM_INITMENUPOPUP messages, as if the user were trying to display the menu. However, these messages are not sent if any of the following conditions exist:

- The window is disabled.
- The accelerator key combination does not correspond to an item on the **window** menu and the window is minimized.
- A mouse capture is in effect. For information about mouse capture, see the SetCapture function.

If the specified window is the active window and no window has the keyboard focus (which is generally the case if the window is minimized), **TranslateAccelerator** translates WM_SYSKEYUP and **WM_SYSKEYDOWN** messages instead of WM_KEYUP and **WM_KEYDOWN** messages.

If an accelerator keystroke occurs that corresponds to a menu item when the window that owns the menu is minimized, **TranslateAccelerator** does not send a **WM_COMMAND** message. However, if an accelerator keystroke occurs that does not match any of the items in the window's menu or in the window menu, the function send a **WM_COMMAND** message, even if the window is minimized.

17.13 Translate Accelerator at Work

VK_BACK, ID_DO_BACK, VIRTKEY, ALT, NOINVERT

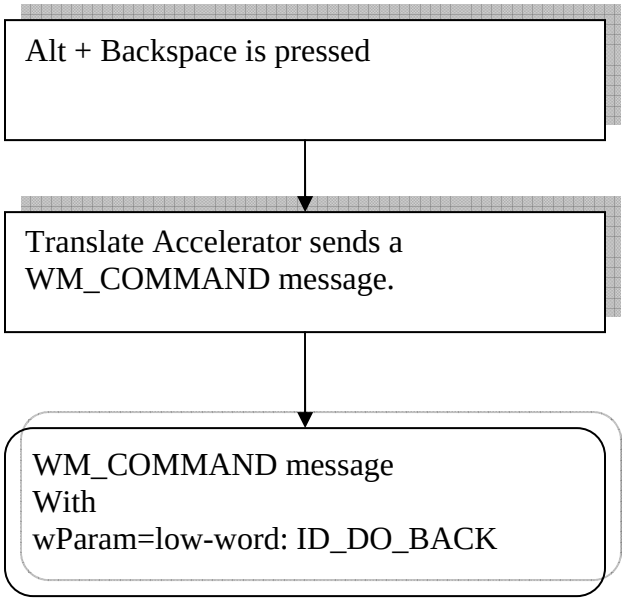


Figure 2

17.14 Handling Accelerator Keys

```
HACCEL hAccel;

//Load the accelerator table

hAccel = LoadAccelerator(hInstance, MAKEINTRESOURCE(ACCELERATOR))

While(GetMessage(&msg, .. .,. . . , .. . ))
{

//Call translateAccelerator to test if accelerator is pressed

If( !TranslateAccelerator(msg.hwnd, hAccel, &msg))
```

```
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
}
```

17.14.1 Windows Procedure

```
case WM_COMMAND:

if(LOWORD(wParam) == ID_DO_BACK)
{
 // accelerator is pressed
}
```

Summary

In this lecture, we have been studying about resources. Resources are also very much important subject in Windows executable files. Resources are separately compiled using resource compiler. Resource compiler (RC) compiles them to binary resource and these binary resource are then become the part of final executable file. Resource files are simply text script files. Resource can be loaded from any DLL and EXE module. For loading the resource, we have useful resource functions like *LoadString* that loads a string from resource table and Load Icon etc. that loads an Icon data from resource data.

Exercises

Practice to design your own resource including menus, bitmaps, dialogs, etc in Visual Studio Resource Developer.

Chapter 18

String and Menu Resource

18.1 Menus

A *menu* is a list of items that specify options or groups of options (a submenu) for an application. Clicking a menu item opens a submenu or causes the application to carry out a command.

18.1.1 Menu bar and Menus

A menu is arranged in a hierarchy. At the top level of the hierarchy is the *menu bar*; which contains a list of *menus*, which in turn can contain *submenus.* A menu bar is sometimes called a *top-level menu*, and the menus and submenus are also known as *pop-up menus*.

A menu item can either carry out a command or open a submenu. An item that carries out a command is called a *command item* or a *command*.

An item on the menu bar almost always opens a menu. Menu bars rarely contain command items. A menu opened from the menu bar drops down from the menu bar and is sometimes called a *drop-down menu*. When a drop-down menu is displayed, it is attached to the menu bar. A menu item on the menu bar that opens a drop-down menu is also called a *menu name*.

The menu names on a menu bar represent the main categories of commands that an application provides. Selecting a menu name from the menu bar typically opens a menu whose menu items correspond to the commands in a category. For example, a menu bar might contain a **File** menu name that, when clicked by the user, activates a menu with menu items such as **New**, **Open**, and **Save**. To get information about a menu bar, call GetMenuBarInfo.

Only an overlapped or pop-up window can contain a menu bar; a child window cannot contain one. If the window has a title bar, the system positions the menu bar just below it. A menu bar is always visible. A submenu is not visible, however, until the user selects a menu item that activates it. For more information about overlapped and pop-up windows, see Window Types.

Each menu must have an owner window. The system sends messages to a menu's owner window when the user selects the menu or chooses an item from the menu.

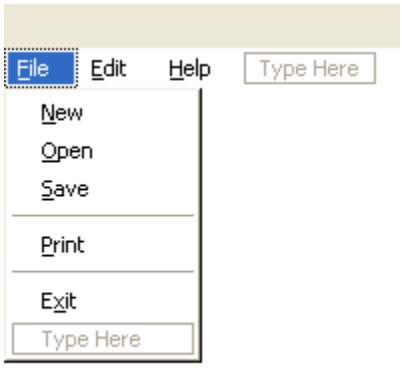


Figure 1 Menu in Visual Studio Editor

18.1.1.1 Short cut Menus

The system also provides *shortcut menus*. A shortcut menu is not attached to the menu bar; it can appear anywhere on the screen. An application typically associates a shortcut menu with a portion of a window, such as the client area, or with a specific object, such as an icon. For this reason, these menus are also called *context menus*.

A shortcut menu remains hidden until the user activates it, typically by right-clicking a selection, a toolbar, or a taskbar button. The menu is usually displayed at the position of the caret or mouse cursor.

18.1.1.2 The Window Menu

The **Window** menu (also known as the **System** menu or **Control** menu) is a pop-up menu defined and managed almost exclusively by the operating system. The user can open the window menu by clicking the application icon on the title bar or by right-clicking anywhere on the title bar.

The **Window** menu provides a standard set of menu items that the user can choose to change a window's size or position, or close the application. Items on the window menu can be added, deleted, and modified, but most applications just use the standard set of menu items. An overlapped, pop-up, or child window can have a window menu. It is uncommon for an overlapped or pop-up window not to include a window menu.

When the user chooses a command from the **Window** menu, the system sends a WM_SYSCOMMAND message to the menu's owner window. In most applications, the window procedure does not process messages from the window menu. Instead, it simply passes the messages to the DefWindowProc function for system-default processing of the message. If an application adds a command to the window menu, the window procedure must process the command.

An application can use the GetSystemMenu function to create a copy of the default window menu to modify. Any window that does not use the **GetSystemMenu** function to make its own copy of the window menu receives the standard window menu.

18.1.2 Menu Handles

The system generates a unique handle for each menu. A *menu handle* is a value of the **HMENU** type. An application must specify a menu handle in many of the menu functions. You receive a handle to a menu bar when you create the menu or load a menu resource.

To retrieve a handle to the menu bar for a menu that has been created or loaded, use the GetMenu function. To retrieve a handle to the submenu associated with a menu item, use the GetSubMenu or GetMenuItemInfo function. To retrieve a handle to a window menu, use the **GetSystemMenu** function.

18.1.3 State of Menu Items

Following are the states of Menu items:

- Checked (MF_CHECKED)
- Unchecked (MF_UNCHECKED)
- Enabled (MF_ENABLED)
- Disabled (MF_DISABLED)
- Grayed (MF_GRAYED)
- Separator (MF_SEPARATOR)
- Highlight (MF_HILIGHT)

18.2 Menu Resource Definition Statement

```
IDR_MY_MENU MENU DISCARDABLE
BEGIN
        POPUP "&Tools“
        BEGIN
                MENUITEM "Write &Text", ID_TOOLS_WRITE_TEXT, GRAYED
                MENUITEM SEPARATOR
                POPUP "&Draw"
                BEGIN
                MENUITEM "&Rectangle",  ID_TOOLS_DRAW_RECTANGLE
                MENUITEM "&Circle",     ID_TOOLS_DRAW_CIRCLE, CHECKED
                MENUITEM "&Ellipse",    ID_TOOLS_DRAW_ELLIPSE
                END
                MENUITEM SEPARATOR
                MENUITEM "&Erase All",  ID_TOOLS_ERASE_ALL, INACTIVE
                END
```

```
    MENUITEM "&About...",     ID_ABOUT
    END
```

Clicking on menu item sends a WM_COMMAND message to its parent.
WM_COMMAND message contains Menu item ID in the low word of WPARAM and handle in LPARAM.

18.3 Loading Menu

The **LoadMenu** function loads the specified menu resource from the executable (.exe) file associated with an application instance.

```
HMENU LoadMenu(

    HINSTANCE hInstance,  //handle to the instance of the */
    LPCTSTR lpMenuName    /* Menu Name */
);
```

hInstance: Handle to the module containing the menu resource to be loaded.

lpMenuName: Pointer to a null-terminated string that contains the name of the menu resource. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. To create this value, use the MAKEINTRESOURCE macro.

Return Value: If the function succeeds, the return value is a handle to the menu resource. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

18.4 Specify default class Menu

You can specify default class menu for windows by assigning Menu name to the lpszMenuName parameter in window class.

```
wc. lpszMenuName= (LPCTSTR)IDR_MENU1;
……..
……..
if(!RegisterClass(&wc))
{
        return 0;
}
```

18.5 Specify Menu in CreateWindow

Menu can be specifying in *hMenu* parameter of CreateWindow function. *hMenu* is the handle of the menu so Menu handle must be specify here rather its name. if the handle of the menu is specified then this will override class window menu.

18.6 Example Application

Now we will practically discuss the menus and Timers by making an application. In this application we will display menu which will be enabled and disabled.

18.6.1 Resource Definition strings

```
#include “resource.h”

STRINGTABLE DISCARDABLE
BEGIN
IDS_APP_NAME            "Virtual University"
IDS_CLASS_NAME          "MyWindowClass"
END
```

18.6.2 Resource Definition Icon

```
IDI_MAIN_ICON          ICON    DISCARDABLE     "VU.ICO"

Icon file name is VU.ICO
```

18.6.3 Application Menus

```
IDR_FIRST_MENU MENU DISCARDABLE
BEGIN
        POPUP “&File"
BEGIN
        MENUITEM "E&xit",     ID_FILE_EXIT
END

POPUP "&Timer"
BEGIN
        MENUITEM "&Start",
        ID_TIMER_START
        MENUITEM "Sto&p",
        ID_TIMER_STOP, GRAYED
END
END
```

18.6.4 Application Window Class

```
#define BUFFER_SIZE 128

TCHAR windowClassName[BUFFER_SIZE];
LoadString(hInstance, IDS_CLASS_NAME, windowClassName, BUFFER_SIZE);
wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MAIN_ICON));
wc.lpszMenuName = MAKEINTRESOURCE(IDR_FIRST_MENU);
wc.lpszClassName = windowClassName
```

18.6.5 CreateWindow

```
#define BUFFER_SIZE 128
TCHAR windowName[BUFFER_SIZE];
… … …
LoadString(hInstance, IDS_APP_NAME, windowName, BUFFER_SIZE);
hWnd = CreateWindow(windowClassName, windowName, ...
```

18.6.6 Window Procedure

```
static int count;
static BOOL bTimerStarted;
 . . . . . .
case WM_CREATE:
count=0;
bTimerStarted=FALSE

case WM_COMMAND:
switch( LOWORD(wParam) )
{
        case ID_TIMER_START:
        SetTimer(hWnd, ID_TIMER, 1000, NULL);
        bTimerStarted=TRUE;
        hOurMenu = GetMenu(hWnd);
        EnableMenuItem(hOurMenu, ID_TIMER_START, MF_BYCOMMAND |
MF_GRAYED);
        EnableMenuItem(hOurMenu, ID_TIMER_STOP, MF_BYCOMMAND |
MF_ENABLED);
        DrawMenuBar(hWnd);

        Case ID_TIMER_STOP:
        KillTimer(hWnd, ID_TIMER);
        bTimerStarted=FALSE;
        hOurMenu = GetMenu(hWnd);
```

```
        EnableMenuItem(hOurMenu, ID_TIMER_STOP, MF_BYCOMMAND |
MF_GRAYED);
        EnableMenuItem(hOurMenu, ID_TIMER_START, MF_BYCOMMAND |
MF_ENABLED);
        DrawMenuBar(hWnd);
        break;

        case ID_FILE_EXIT:
                DestroyWindow(hWnd);

        case WM_TIMER:
        switch(wParam)
        {
                case ID_TIMER:
                ++count;   count %= 10;
                GetClientRect(hWnd, &rect);
                InvalidateRect(hWnd, &rect, TRUE); break; }
                break;
        }

        TCHAR msg[10];

        case WM_PAINT:

        hDC = BeginPaint(hWnd, &ps);
        wsprintf(msg, "Count: %2d", count);
        TextOut(hDC, 10, 10, msg, lstrlen(msg));
        EndPaint(hWnd, &ps);

        break;

        case WM_DESTROY:
        if(bTimerStarted)
        KillTimer(hWnd, ID_TIMER);
        PostQuitMessage(0);
        break;
```

18.6.7 Keyboard Accelerator

```
IDR_ACCELERATOR ACCELERATORS DISCARDABLE
BEGIN    "P",  ID_TIMER_STOP,     VIRTKEY, CONTROL, NOINVERT
         "S",  ID_TIMER_START,    VIRTKEY, CONTROL, NOINVERT
         "X", ID_FILE_EXIT,      VIRTKEY, ALT, NOINVERT
END

IDR_FIRST_MENU MENU DISCARDABLE
```

```
BEGIN
   POPUP "&File"
   BEGIN
         MENUITEM "E&xit\tAlt+X",              ID_FILE_EXIT
   END
   POPUP "&Timer"
   BEGIN
         MENUITEM "&Start\tCtrl+S",            ID_TIMER_START
         MENUITEM "Sto&p\tCtrl+P",             ID_TIMER_STOP, GRAYED
   END
END
```

18.6.8 Message Loop

```
HACCEL hAccelerators;
hAccelerators = LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDR_ACCELERATOR));
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
         if(!TranslateAccelerator(msg.hwnd, hAccelerators, &msg))
         {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
         }
}
```

Summary

In this lecture, we studied about the menus resources and their entry in resource definition file. Using menu accelerators you can use short cut keys to operate menus. At the end we discussed an example application which enables or disable the menus. Menus are used by almost every application except some games or other system tools. Using menus we can watch different facilities or action provided by application.

Exercises

1. Show a popup menu whenever the mouse right button is up inside the client area. The pop-up menu should contain at least three items.
2. Using the mouse messages draw a line which starts when a mouse left button is down and end when the mouse left button is up. During the mouse pressed state if ESC key is pressed the popup menu should be displayed which include menu item Exit. If user press on the exit button process must be cancelled.
   Use the PeekMessage function and filter the mouse messages only, for this you will have to create another mouse loop that will be created when the mouse left button is down and ends when mouse left button is up.

Chapter 19

Menu And Dialogs

19.1 Menus

We have discussed Menus in our previous lecture, here we will know more about menus and their use in Windows Applications.

19.2 Menu Items

Command Items and Items that Open Submenus

When the user chooses a command item, the system sends a command message to the window that owns the menu. If the command item is on the window menu, the system sends the **WM_SYSCOMMAND** message. Otherwise, it sends the WM_COMMAND message.

Handle is associated with each menu item that opens a submenu. When the user points to such an item, the system opens the submenu. No command message is sent to the owner window. However, the system sends a WM_INITMENUPOPUP message to the owner window before displaying the submenu. You can get a handle to the submenu associated with an item by using the **GetSubMenu** or **GetMenuItemInfo** function.

A menu bar typically contains menu names, but it can also contain command items. A submenu typically contains command items, but it can also contain items that open nested submenus. By adding such items to submenus, you can nest menus to any depth. To provide a visual cue for the user, the system automatically displays a small arrow to the right of the text of a menu item that opens a submenu.

Menu-Item Identifier

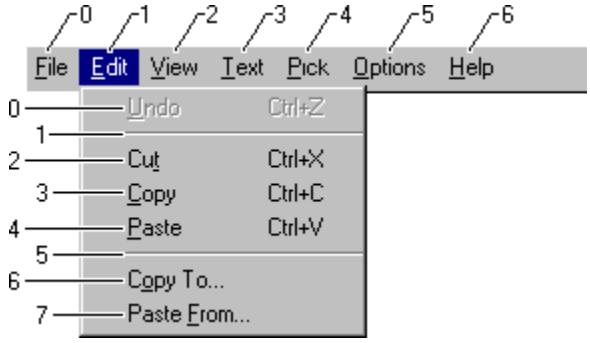
Associated with each menu item is a unique, application-defined integer, called a *menu-item identifier*. When the user chooses a command item from a menu, the system sends the item's identifier to the owner window as part of a **WM_COMMAND** message. The window procedure examines the identifier to determine the source of the message, and processes the message accordingly. In addition, you can specify a menu item using its identifier when you call menu functions; for example, to enable or disable a menu item.

Menu items that open submenus have identifiers just as command items do. However, the system does not send a command message when such an item is selected from a menu. Instead, the system opens the submenu associated with the menu item.

To retrieve the identifier of the menu item at a specified position, use the GetMenuItemID or **GetMenuItemInfo** function.

Menu-Item Position

In addition to having a unique identifier, each menu item in a menu bar or menu has a unique position value. The leftmost item in a menu bar, or the top item in a menu, has position zero. The position value is incremented for subsequent menu items. The system assigns a position value to all items in a menu, including separators. The following illustration shows the position values of items in a menu bar and in a menu.



When calling a menu function that modifies or retrieves information about a specific menu item, you can specify the item using either its identifier or its position. For more information, see Menu Modifications.

Default Menu Items

A submenu can contain one default menu item. When the user opens a submenu by double-clicking, the system sends a command message to the menu's owner window and closes the menu as if the default command item had been chosen. If there is no default command item, the submenu remains open. To retrieve and set the default item for a submenu, use the GetMenuDefaultItem and SetMenuDefaultItem functions.

Selected and Clear Menu Items

A menu item can be either selected or clear. The system displays a bitmap next to selected menu items to indicate their selected state. The system does not display a bitmap next to clear items, unless an application-defined "clear" bitmap is specified. Only menu items in a menu can be selected; items in a menu bar cannot be selected.

Applications typically check or clear a menu item to indicate whether an option is in effect. For example, suppose an application has a toolbar that the user can show or hide by using a **Toolbar** command on a menu. When the toolbar is hidden, the **Toolbar** menu item is clear. When the user chooses the command, the application checks the menu item and shows the toolbar.

A check mark attribute controls whether a menu item is selected. You can set a menu item's check mark attribute by using the CheckMenuItem function. You can use the

GetMenuState function to determine whether a menu item is currently selected or cleared.

Instead of **CheckMenuItem** and **GetMenuState**, you can use the **GetMenuItemInfo** and SetMenuItemInfo functions to retrieve and set the check state of a menu item.

Sometimes, a group of menu items corresponds to a set of mutually exclusive options. In this case, you can indicate the selected option by using a selected radio menu item (analogous to a radio button control). Selected radio items are displayed with a bullet bitmap instead of a check mark bitmap. To check a menu item and make it a radio item, use the CheckMenuRadioItem function.

By default, the system displays a check mark or bullet bitmap next to selected menu items and no bitmap next to cleared menu items. However, you can use the SetMenuItemBitmaps function to associate application-defined selected and cleared bitmaps with a menu item. The system then uses the specified bitmaps to indicate the menu item's selected or cleared state.

Application-defined bitmaps associated with a menu item must be the same size as the default check mark bitmap, the dimensions of which may vary depending on screen resolution. To retrieve the correct dimensions, use the GetSystemMetrics function. You can create multiple bitmap resources for different screen resolutions; create one bitmap resource and scale it, if necessary; or create a bitmap at run time and draw an image in it. The bitmaps may be either monochrome or color. However, because menu items are inverted when highlighted, the appearance of certain inverted color bitmaps may be undesirable. For more information, see Bitmaps.

Enabled, Grayed, and Disabled Menu Items

A menu item can be enabled, grayed, or disabled. By default, a menu item is enabled. When the user chooses an enabled menu item, the system sends a command message to the owner window or displays the corresponding submenu, depending on what kind of menu item it is.

When menu items are not available to the user, they should be grayed or disabled. Grayed and disabled menu items cannot be chosen. A disabled item looks just like an enabled item. When the user clicks on a disabled item, the item is not selected, and nothing happens. Disabled items can be useful in, for example, a tutorial that presents a menu that looks active but isn't.

An application grays an unavailable menu item to provide a visual cue to the user that a command is not available. You can use a grayed item when an action is not appropriate (for example, you can gray the Print command in the File menu when the system does not have a printer installed).

The EnableMenuItem function enables, grays, or disables a menu item. To determine whether a menu item is enabled, grayed, or disabled, use the **GetMenuItemInfo** function.

Instead of **GetMenuItemInfo**, you can also use the **GetMenuState** function to determine whether a menu item is enabled, grayed, or disabled.

Highlighted Menu Items

The system automatically highlights menu items on menus as the user selects them. However, highlighting can be explicitly added or removed from a menu name on the menu bar by using the HiliteMenuItem function. This function has no effect on menu items on menus. When **HiliteMenuItem** is used to highlight a menu name, though, the name only appears to be selected. If the user presses the ENTER key, the highlighted item is not chosen. This feature might be useful in, for example, a training application that demonstrates the use of menus.

Owner-Drawn Menu Items

An application can completely control the appearance of a menu item by using an *owner-drawn item*. Owner-drawn items require an application to take total responsibility for drawing selected (highlighted), selected, and cleared states. For example, if an application provided a font menu, it could draw each menu item by using the corresponding font; the item for Roman would be drawn with roman, the item for Italic would be drawn in italic, and so on. For more information, see Creating Owner-Drawn Menu Items.

Menu Item Separators and Line Breaks

The system provides a special type of menu item, called a *separator*, which appears as a horizontal line. You can use a separator to divide a menu into groups of related items. A separator cannot be used in a menu bar, and the user cannot select a separator.

When a menu bar contains more menu names than will fit on one line, the system wraps the menu bar by automatically breaking it into two or more lines. You can cause a line break to occur at a specific item on a menu bar by assigning the MFT_MENUBREAK type flag to the item. The system places that item and all subsequent items on a new line.

When a menu contains more items than will fit in one column, the menu will be truncated. You can cause a column break to occur at a specific item in a menu by assigning the MFT_MENUBREAK type flag to the item or using the MENUBREAK option in the MENUITEM statement. The system places that item and all subsequent items in a new column. The MFT_MENUBARBREAK type flag has the same effect, except that a vertical line appears between the new column and the old.

If you use the AppendMenu, InsertMenu, or ModifyMenu functions to assign line breaks, you should assign the type flags MF_MENUBREAK or MF_MENUBARBREAK.

19.3 Drop Down Menus

Drop down menus are submenu. For example you are working with notepad and you are going to make a new file, for this you press on a file menu and menu drops itself down and you select new from that menu, so this menu is called drop down menu. This drop down menu is called submenu.

19.4 Get Sub Menu

The **GetSubMenu** function retrieves a handle to the drop-down menu or submenu activated by the specified menu item.

```
HMENU GetSubMenu(

    HMENU hMenu,
    int nPos
);
```

hMenu: Handle to the menu.
nPos: Specifies the zero-based relative position in the specified menu of an item that activates a drop-down menu or submenu.

Return Value: If the function succeeds, the return value is a handle to the drop-down menu or submenu activated by the menu item. If the menu item does not activate a drop-down menu or submenu, the return value is NULL.

19.5 Example Application

Here we create an application which will demonstrate menus.

19.5.1 Popup Menu (*Resource File View*)

Popup menu is a main menu which may have sub menu.

```
IDR_MENU_POPUP MENU DISCARDABLE
BEGIN     POPUP "Popup Menu"
BEGIN    MENUITEM "&Line",           ID_POPUPMENU_LINE
         MENUITEM "&Circle",              ID_POPUPMENU_CIRCLE
         MENUITEM "&Rectangle",        ID_POPUPMENU_RECTANGLE
POPUP "&Other"
BEGIN       MENUITEM "&Polygon", ID_OTHER_POLYGON
         MENUITEM "&Text Message", ID_OTHER_TEXTMESSAGE
END

END
```

```
END
```

19.5.2 The WM_RBUTTONDOWN message

```
WM_RBUTTONDOWN

    WPARAM wParam
    LPARAM lParam;
```

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.
MK_CONTROL: The CTRL key is down.
MK_LBUTTON: The left mouse button is down.
MK_MBUTTON: The middle mouse button is down.
MK_RBUTTON: The right mouse button is down.
MK_SHIFT: The SHIFT key is down.
MK_XBUTTON1

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.
The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Value:

If an application processes this message, it should return zero

19.5.3 Structure to represent Points

POINT structure contains LONG (long)x and LONG y.

```
typedef struct tagPOINT
{
        LONG x;       //horizontal number
        LONG y;       //vertical number
} POINT;
```

POINTS structure contains SHORT (short) x, and SHORT y

```
typedef struct tagPOINTS
{
        SHORT x;   //horizontal short integer
        SHORT y;      //vertical short integer
} POINTS;
```

19.5.4 Main Window Procedure

```
POINTS pts; POINT pt;
… … … …
 case WM_RBUTTONDOWN:

 pts = MAKEPOINTS(lParam);
 pt.x = pts.x;
 pt.y = pts.y;

ClientToScreen(hWnd, &pt);  //convert the window coordinates to the screen coordinates

result = TrackPopupMenu(hPopupMenu, TPM_LEFTALIGN | TPM_TOPALIGN |
TPM_RETURNCMD | TPM_LEFTBUTTON,
pt.x, pt.y, 0, hWnd, 0
);
```

19.5.5 Set Menu Item Information

The **SetMenuInfo** function sets information for a specified menu.

```
BOOL SetMenuInfo(

    HMENU hmenu,   //handle to the menu
    LPCMENUINFO lpcmi    //menu informations
);
```

hmenu: Handle to a menu.

lpcmi: Pointer to a MENUINFO structure for the menu.

Return Value:

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

19.5.6 System Menu

The ***GetSystemMenu*** function allows the application to access the window menu (also known as the system menu or the control menu) for copying and modifying.

```
HMENU GetSystemMenu(

    HWND hWnd,            //handle to the window
    BOOL bRevert          //action specification
);
```

hWnd: Handle to the window that will own a copy of the window menu.

bRevert: Specifies the action to be taken. If this parameter is FALSE, **GetSystemMenu** returns a handle to the copy of the window menu currently in use. The copy is initially identical to the window menu, but it can be modified. If this parameter is TRUE, **GetSystemMenu** resets the window menu back to the default state. The previous window menu, if any, is destroyed.

Return Value: If the *bRevert* parameter is FALSE, the return value is a handle to a copy of the window menu. If the *bRevert* parameter is TRUE, the return value is NULL.

Any window that does not use the **GetSystemMenu** function to make its own copy of the window menu receives the standard window menu.

The window menu initially contains items with various identifier values, such as SC_CLOSE, SC_MOVE, and SC_SIZE.

Menu items on the window menu send WM_SYSCOMMAND messages.

All predefined window menu items have identifier numbers greater than 0xF000. If an application adds commands to the window menu, it should use identifier numbers less than 0xF000.

The system automatically grays items on the standard window menu, depending on the situation. The application can perform its own checking or graying by responding to the WM_INITMENU message that is sent before any menu is displayed.

19.5.7 System Menu Identifiers

The window menu initially contains items with various identifier values, such as Figure labelled as

SC_MOVE Move

SC_SIZE Size
SC_CLOSE Close

19.6 Time Differences

There are two time differences are available in windows one is

Local Time -
And
UTC (Universal Coordinated Time) historically GMT (Greenwich Mean Time)

19.7 Time Information in Windows

```
VOID GetSystemTime(
 LPSYSTEMTIME lpSystemTime   // system time
 );
```

This function retrieves the system time in UTC format.

```
VOID GetLocalTime(
LPSYSTEMTIME lpSystemTime   // system time
 );
```

This function retrieves the current local date and time.

```
IDR_FIRST_MENU MENU DISCARDABLE
BEGIN
          POPUP "&File"
BEGIN
         MENUITEM "E&xit",   ID_FILE_EXIT n END
POPUP "F&ormat"
BEGIN
       MENUITEM "&UTC",   ID_FORMAT_UTC
       MENUITEM "&Local Time", ID_FORMAT_LOCALTIME
END
END
```

19.8 Clock Example (*Window Procedure*)

```
static SYSTEMTIME st;
          enum Format { UTC, LOCAL };
          static enum Format format;

case WM_CREATE:
                    SetTimer(hWnd, ID_TIMER, 1000, NULL);
```

```
                format=LOCAL;
                GetLocalTime(&st);
        hOurMenu = GetMenu(hWnd);
                CheckMenuItem(hOurMenu, ID_FORMAT_LOCALTIME,
MF_BYCOMMAND | MF_CHECKED);
Break;

case WM_COMMAND:
                switch(LOWORD(wParam))
                {
                case ID_FORMAT_UTC:
                        if(format == UTC)
                                break;
                        format = UTC;
                        hOurMenu = GetMenu(hWnd);
                result = CheckMenuItem(hOurMenu, ID_FORMAT_UTC,
MF_BYCOMMAND | MF_CHECKED);
                        result = CheckMenuItem(hOurMenu,
ID_FORMAT_LOCALTIME, MF_BYCOMMAND | MF_UNCHECKED);
                DrawMenuBar(hWnd);
                (format == UTC) ? GetSystemTime(&st) : GetLocalTime(&st);
                GetClientRect(hWnd, &rect);
                InvalidateRect(hWnd, &rect, TRUE);
                        break;


case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        wsprintf(msg, "Hour: %2d:%02d:%02d", st.wHour, st.wMinute, st.wSecond);
        TextOut(hDC, 10, 10, msg, lstrlen(msg));
        EndPaint(hWnd, &ps);
        break;

case WM_TIMER:
        if(wParam == ID_TIMER)
        {
        (format == UTC) ? GetSystemTime(&st) : GetLocalTime(&st);
        GetClientRect(hWnd, &rect);
        InvalidateRect(hWnd, &rect, TRUE);
                break;
        }
        break;
```

19.9 Dialogs

Dialogs are important resource in windows. Most of the information in window are displayed in dialog boxes. Simple example of dialog boxes is about dialog box or properties are shown in normally in dialog boxes.

A dialog box is a temporary window an application creates to retrieve user input. An application typically uses dialog boxes to prompt the user for additional information for menu items. A dialog box usually contains one or more controls (child windows) with which the user enters text, chooses options, or directs the action.

Windows also provides predefined dialog boxes that support common menu items such as **Open** and **Print**. Applications that use these menu items should use the common dialog boxes to prompt for this user input, regardless of the type of application.

Dialogs are of two types.

- Modal Dialog Boxes
- Modeless Dialog Boxes

19.9.1 Modal Dialog Boxes

A modal dialog box should be a pop-up window having a **window** menu, a title bar, and a thick border; that is, the dialog box template should specify the WS_POPUP, WS_SYSMENU, WS_CAPTION, and DS_MODALFRAME styles. Although an application can designate the WS_VISIBLE style, the system always displays a modal dialog box regardless of whether the dialog box template specifies the WS_VISIBLE style. An application must not create a modal dialog box having the WS_CHILD style. A modal dialog box with this style disables itself, preventing any subsequent input from reaching the application.

An application creates a modal dialog box by using either the **DialogBox** or DialogBoxIndirect function. **DialogBox** requires the name or identifier of a resource containing a dialog box template; **DialogBoxIndirect** requires a handle to a memory object containing a dialog box template. The DialogBoxParam and DialogBoxIndirectParam functions also create modal dialog boxes; they are identical to the previously mentioned functions but pass a specified parameter to the dialog box procedure when the dialog box is created.

When creating the modal dialog box, the system makes it the active window. The dialog box remains active until the dialog box procedure calls the EndDialog function or the system activates a window in another application. Neither the user nor the application can make the owner window active until the modal dialog box is destroyed.

When the owner window is not already disabled, the system automatically disables the window and any child windows belonging to it when it creates the modal dialog box. The owner window remains disabled until the dialog box is destroyed. Although a dialog box procedure could potentially enable the owner window at any time, enabling the owner defeats the purpose of the modal dialog box and is not recommended. When the dialog box procedure is destroyed, the system enables the owner window again, but only if the modal dialog box caused the owner to be disabled.

As the system creates the modal dialog box, it sends the **WM_CANCELMODE** message to the window (if any) currently capturing mouse input. An application that receives this message should release the mouse capture so that the user can move the mouse in the modal dialog box. Because the system disables the owner window, all mouse input is lost if the owner fails to release the mouse upon receiving this message.

To process messages for the modal dialog box, the system starts its own message loop, taking temporary control of the message queue for the entire application. When the system retrieves a message that is not explicitly for the dialog box, it dispatches the message to the appropriate window. If it retrieves a WM_QUIT message, it posts the message back to the application message queue so that the application's main message loop can eventually retrieve the message.

The system sends the WM_ENTERIDLE message to the owner window whenever the application message queue is empty. The application can use this message to carry out a background task while the dialog box remains on the screen. When an application uses the message in this way, the application must frequently yield control (for example, by using the PeekMessage function) so that the modal dialog box can receive any user input. To prevent the modal dialog box from sending the **WM_ENTERIDLE** messages, the application can specify the DS_NOIDLEMSG style when creating the dialog box.

An application destroys a modal dialog box by using the **EndDialog** function. In most cases, the dialog box procedure calls **EndDialog** when the user clicks **Close** from the dialog box's **window** menu or clicks the **OK** or **Cancel** button in the dialog box. The dialog box can return a value through the **DialogBox** function (or other creation functions) by specifying a value when calling the **EndDialog** function. The system returns this value after destroying the dialog box. Most applications use this return value to determine whether the dialog box completed its task successfully or was canceled by the user. The system does not return control from the function that creates the dialog box until the dialog box procedure has called the **EndDialog** function.

19.9.2 Modeless Dialog Boxes

A modeless dialog box should be a pop-up window having a **window** menu, a title bar, and a thin border; that is, the dialog box template should specify the WS_POPUP, WS_CAPTION, WS_BORDER, and WS_SYSMENU styles. The system does **not** automatically display the dialog box unless the template specifies the WS_VISIBLE style.

An application creates a modeless dialog box by using the **CreateDialog** or CreateDialogIndirect function. **CreateDialog** requires the name or identifier of a resource containing a dialog box template; **CreateDialogIndirect** requires a handle to a memory object containing a dialog box template. Two other functions, CreateDialogParam and CreateDialogIndirectParam, also create modeless dialog boxes; they pass a specified parameter to the dialog box procedure when the dialog box is created.

CreateDialog and other creation functions return a window handle for the dialog box. The application and the dialog box procedure can use this handle to manage the dialog box. For example, if WS_VISIBLE is not specified in the dialog box template, the application can display the dialog box by passing the window handle to the ShowWindow function.

A modeless dialog box neither disables the owner window nor sends messages to it. When creating the dialog box, the system makes it the active window, but the user or the application can change the active window at any time. If the dialog box does become inactive, it remains above the owner window in the Z order, even if the owner window is active.

The application is responsible for retrieving and dispatching input messages to the dialog box. Most applications use the main message loop for this. To permit the user to move to and select controls by using the keyboard, however, the application must call the IsDialogMessage function. For more information about this function, see Dialog Box Keyboard Interface.

A modeless dialog box cannot return a value to the application as a modal dialog box does, but the dialog box procedure can send information to the owner window by using the SendMessage function.

An application must destroy all modeless dialog boxes before terminating. It can destroy a modeless dialog box by using the DestroyWindow function. In most cases, the dialog box procedure calls **DestroyWindow** in response to user input, such as clicking the **Cancel** button. If the user never closes the dialog box in this way, the application must call **DestroyWindow**.

DestroyWindow invalidates the window handle for the dialog box, so any subsequent calls to functions that use the handle return error values. To prevent errors, the dialog box procedure should notify the owner that the dialog box has been destroyed. Many applications maintain a global variable containing the handle for the dialog box. When the dialog box procedure destroys the dialog box, it also sets the global variable to NULL, indicating that the dialog box is no longer valid.

The dialog box procedure must not call the **EndDialog** function to destroy a modeless dialog box.

19.9.3 Message Box Function

A message box is a special dialog box that an application can use to display messages and prompt for simple input. A message box typically contains a text message and one or more buttons. An application creates the message box by using the MessageBox or MessageBoxEx function, specifying the text and the number and types of buttons to display. Note that currently there is no difference between how **MessageBox** and **MessageBoxEx** work.

Although the message box is a dialog box, the system takes complete control of the creation and management of the message box. This means the application does not provide a dialog box template and dialog box procedure. The system creates its own template based on the text and buttons specified for the message box and supplies its own dialog box procedure.

A message box is a modal dialog box and the system creates it by using the same internal functions that **DialogBox** uses. If the application specifies an owner window when calling **MessageBox** or **MessageBoxEx**, the system disables the owner. An application can also direct the system to disable all top-level windows belonging to the current thread by specifying the MB_TASKMODAL value when creating the dialog box.

The system can send messages to the owner, such as WM_CANCELMODE and WM_ENABLE, just as it does when creating a modal dialog box. The owner window should carry out any actions requested by these messages.

19.9.4 Modal Loop

Modal loop is run by Modal dialogs and process message as does application message loop. That’s why program execution is transfer to modal loop so the modal loop itself gets messages and dispatch message.

19.9.5 Dialog Resource Template

```
IDD_DIALOG_ABOUT DIALOG DISCARDABLE  0, 0, 265, 124
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
        DEFPUSHBUTTON   "OK",IDOK,208,7,50,14
        PUSHBUTTON      "Cancel",IDCANCEL,208,24,50,14
        LTEXT           "Some copyright text", IDC_STATIC,
                        67, 27,107,47
        ICON           IDI_ICON_VU,IDC_STATIC,17,14,20,20
END
```

19.9.6 Creating a Modal Dialog

Modal Dialog runs the dialog modal loop. And handle all the messages in message queue.

```
INT_PTR DialogBox(
  HINSTANCE hInstance,  // handle to module
  LPCTSTR lpTemplate,   // dialog box template
  HWND hWndParent,      // handle to owner window
  DLGPROC lpDialogFunc  // dialog box procedure
);
```

Summary

In this lecture, we studied about menus and dialogs. Dialogs are another useful and multipurpose resource in windows. Dialogs are used to display temporary information and other data. Dialogs are of two types: One is modal dialogs and second is modeless dialogs. Modal dialogs do not return control to the application until they are not ended or destroyed. Modeless dialogs act like a normal windows they return control after they have created. Message boxes are normally modal dialog boxes.

Exercises

1. Create a Modeless dialog box. On pressing the mouse left button on the client area of the Modeless dialog, another modal dialog should appear. And after pressing the right mouse button on the dialog a text name ‘Exercise’ should be displayed.

Chapter 20

Dialogs

20.1 Dialog Box Templates

A dialog box template is binary data that describes the dialog box, defining its height, width, style, and the controls it contains. To create a dialog box, the system either loads a dialog box template from the resources in the application's executable file or uses the template passed to it in global memory by the application. In either case, the application must supply a template when creating a dialog box.

A developer creates template resources by using a resource compiler or a dialog box editor. A resource compiler converts a text description into a binary resource, and a dialog box editor saves an interactively constructed dialog box as a binary resource.

To create a dialog box without using template resources, you must create a template in memory and pass it to the **CreateDialogIndirectParam** or **DialogBoxIndirectParam** function, or to the **CreateDialogIndirect** or **DialogBoxIndirect** macro.

A dialog box template in memory consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format. In a standard template, the header is a DLGTEMPLATE structure followed by additional variable-length arrays; and the data for each control consists of a DLGITEMTEMPLATE structure followed by additional variable-length arrays. In an extended dialog box template, the header uses the DLGTEMPLATEEX format and the control definitions use the DLGITEMTEMPLATEEX format.

You can create a memory template by allocating a global memory object and filling it with the standard or extended header and control definitions. A memory template is identical in form and content to a template resource. Many applications that use memory templates first use the *LoadResource* function to load a template resource into memory, and then modify the loaded resource to create a new memory template.

20.1.1 Dialog Box Templates Styles

Every dialog box template specifies a combination of style values that define the appearance and features of the dialog box. The style values can be window styles, such as WS_POPUP and WS_SYSMENU, and dialog box styles, such as DS_MODALFRAME. The number and type of styles for a template depends on the type and purpose of the dialog box.

The system passes all window styles specified in the template to the CreateWindowEx function when creating the dialog box. The system may pass one or more extended styles depending on the specified dialog box styles. For example, when the template specifies DS_MODALFRAME, the system uses WS_EX_DLGMODALFRAME when creating the dialog box.

Most dialog boxes are pop-up windows that have a window menu and a title bar. Therefore, the typical template specifies the WS_POPUP, WS_SYSMENU, and WS_CAPTION styles. The template also specifies a border style: WS_BORDER for modeless dialog boxes and DS_MODALFRAME for modal dialog boxes. A template may specify a window type other than pop-up (such as WS_OVERLAPPED) if it creates a customized window instead of a dialog box.

The system always displays a modal dialog box regardless of whether the WS_VISIBLE style is specified. When the template for a modeless dialog box specifies the WS_VISIBLE style, the system automatically displays the dialog box when it is created. Otherwise, the application is responsible for displaying the dialog box by using the **ShowWindow** function.

The following table lists the dialog box styles that you can specify when you create a dialog box. You can use these styles in calls to the CreateWindow and **CreateWindowEx** functions, in the **style** member of the **DLGTEMPLATE** and **DLGTEMPLATEEX** structures, and in the statement of a dialog box definition in a resource file.

Value	Meaning
	Gives the dialog box a non-bold font, and draws three-dimensional borders around control windows in the dialog box.
DS_3DLOOK	The DS_3DLOOK style is required only by applications compiled for Windows NT 3.51. The system automatically applies the three-dimensional look to dialog boxes created by applications compiled for Windows 95/98/Me and later versions of Windows NT.
DS_ABSALIGN	Indicates that the coordinates of the dialog box are screen coordinates. If this style is not specified, the coordinates are client coordinates.
DS_CENTER	Centers the dialog box in the working area of the monitor that contains the owner window. If no owner window is specified, the dialog box is centered in the working area of a monitor determined by the system. The working area is the area not obscured by the taskbar or any application bars.
DS_CENTERMOUSE	Centers the dialog box on the mouse cursor.
DS_CONTEXTHELP	Includes a question mark in the title bar of the dialog box. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a control in the dialog box, the control receives a WM_HELP message. The control should pass the message to the dialog box procedure, which should call the function using the HELP_WM_HELP command. The help application displays a pop-up window that typically contains help for the control.

	Note that DS_CONTEXTHELP is only a placeholder. When the dialog box is created, the system checks for DS_CONTEXTHELP and, if it is there, adds WS_EX_CONTEXTHELP to the extended style of the dialog box. WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.
DS_CONTROL	Creates a dialog box that works well as a child window of another dialog box, much like a page in a property sheet. This style allows the user to tab among the control windows of a child dialog box, use its accelerator keys, and so on.
DS_FIXEDSYS	Causes the dialog box to use the SYSTEM_FIXED_FONT instead of the default SYSTEM_FONT. This is a mono-space font compatible with the System font in 16-bit versions of Windows earlier than 3.0.
DS_LOCALEDIT	Applies to 16-bit applications only. This style directs edit controls in the dialog box to allocate memory from the application's data segment. Otherwise, edit controls allocate storage from a global memory object.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and **window** menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOFAILCREATE	**Windows 95/98/Me:** Creates the dialog box even if errors occur — for example, if a child window cannot be created or if the system cannot create a special data segment for an edit control.
DS_NOIDLEMSG	Suppresses **WM_ENTERIDLE** messages that the system would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SETFONT	Indicates that the header of the dialog box template (either standard or extended) contains additional data specifying the font to use for text in the client area and controls of the dialog box. If possible, the system selects a font according to the specified font data. The system passes a handle to the font to the dialog box and to each control by sending them the WM_SETFONT message. For descriptions of the format of this font data, see **DLGTEMPLATE** and **DLGTEMPLATEEX**. If neither DS_SETFONT nor DS_SHELLFONT is specified, the dialog box template does not include the font data.
DS_SETFOREGROUND	Causes the system to use the *SetForegroundWindow* function to bring the dialog box to the foreground. This style is useful for modal dialog boxes that require immediate attention from

	the user regardless of whether the owner window is the foreground window.
	Indicates that the dialog box should use the system font. The **typeface** member of the extended dialog box template must be set to MS Shell Dialog. Otherwise, this style has no effect. It is also recommended that you use the DIALOGEX Resource, rather than the DIALOG Resource.
DS_SHELLFONT	The system selects a font using the font data specified in the **pointsize**, **weight**, and **italic** members. The system passes a handle to the font to the dialog box and to each control by sending them the **WM_SETFONT** message. For descriptions of the format of this font data, see **DLGTEMPLATEEX**.
	If neither DS_SHELLFONT nor DS_SETFONT is specified, the extended dialog box template does not include the font data.
DS_SYSMODAL	This style is obsolete and is included for compatibility with 16-bit versions of Windows. If you specify this style, the system creates the dialog box with the WS_EX_TOPMOST style. This style does not prevent the user from accessing other windows on the desktop. Do not combine this style with the DS_CONTROL style.

20.1.2 Dialog Box Measurements

Every dialog box template contains measurements that specify the position, width, and height of the dialog box and the controls it contains. These measurements are device independent, so an application can use a single template to create the same dialog box for all types of display devices. This ensures that a dialog box will have the same proportions and appearance on all screens despite differing resolutions and aspect ratios between screens.

The measurements in a dialog box template are specified in dialog template units. To convert measurements from dialog template units to screen units (pixels), use the *MapDialogRect* function, which takes into account the font used by the dialog box and correctly converts a rectangle from dialog template units into pixels. For dialog boxes that use the system font, you can use the *GetDialogBaseUnits* function to perform the conversion calculations yourself, although using ***MapDialogRect*** is simpler.

The template must specify the initial coordinates of the upper left corner of the dialog box. Usually the coordinates are relative to the upper left corner of the owner window's client area. When the template specifies the DS_ABSALIGN style or the dialog box has no owner, the position is relative to the upper left corner of the screen. The system sets

this initial position when creating the dialog box, but permits an application to adjust the position before displaying the dialog box. For example, an application can retrieve the dimensions of the owner window, calculate a new position that centers the dialog box in the owner window, and then set the position by using the *SetWindowPos* function.

The template should specify a dialog box width and height that does not exceed the width and height of the screen and ensures that all controls are within the client area of the dialog box. Although the system permits a dialog box to be any size, creating one that is too small or too large can prevent the user from providing input, defeating the purpose of the dialog box. Many applications use more than one dialog box when there are a large number of controls. In such cases, the initial dialog box usually contains one or more buttons that the user can choose to display the next dialog box.

20.1.3 Dialog Box Controls

The template specifies the position, width, height, style, identifier, and window class for each control in the dialog box. The system creates each control by passing this data to the **CreateWindowEx** function. Controls are created in the order they are specified in the template. The template should specify the appropriate number, type, and order of controls to ensure that the user can enter the input needed to complete the task associated with the dialog box.

For each control, the template specifies style values that define the appearance and operation of the control. Every control is a child window and therefore must have the WS_CHILD style. To ensure that the control is visible when the dialog box is displayed, each control must also have the WS_VISIBLE style. Other commonly used window styles are WS_BORDER for controls that have optional borders, WS_DISABLED for controls that should be disabled when the dialog box is initially created, and WS_TABSTOP and WS_GROUP for controls that can be accessed using the keyboard. The WS_TABSTOP and WS_GROUP styles are used in conjunction with the dialog keyboard interface described later in this topic.

The template may also specify control styles specific to the control's window class. For example, a template that specifies a button control must give a button control style such as BS_PUSHBUTTON or BS_CHECKBOX. The system passes the control styles to the control window procedure through the WM_CREATE message, allowing the procedure to adapt the appearance and operation of the control.

The system converts the position coordinates and the width and height measurements from dialog base units to pixels, before passing these to ***CreateWindowEx***. When the system creates a control, it specifies the dialog box as the parent window. This means the system always interprets the position coordinates of the control as client coordinates, relative to the upper left corner of the dialog box's client area.

The template specifies the window class for each control. Typical dialog box contains controls belonging to the predefined control window classes such as the button and edit

control window classes. In this case, the template specifies window classes by supplying the corresponding predefined atom values for the classes. When a dialog box contains a control belonging to a custom control window class, the template gives the name of that registered window class or the atom value currently associated with the name.

Each control in a dialog box must have a unique identifier to distinguish it from other controls. Controls send information to the dialog box procedure through WM_COMMAND messages, so the control identifiers are essential for the procedure to determine which control sent a specified message. The only exception to this rule is control identifiers for static controls. Static controls do not require unique identifiers because they send no **WM_COMMAND** messages.

To permit the user to close the dialog box, the template should specify at least one push button and give it the control identifier IDCANCEL. To permit the user to choose between completing or canceling the task associated with the dialog box, the template should specify two push buttons, labeled **OK** and **Cancel**, with control identifiers of IDOK and IDCANCEL, respectively.

A template also specifies optional text and creation data for a control. The text typically provides labels for button controls or specifies the initial content of a static text control. The creation data is one or more bytes of data that the system passes to the control window procedure when creating the control. Creation data is useful for controls that require more information about their initial content or style than is specified by other data. For example, an application can use creation data to set the initial setting and range for a scroll bar control.

20.1.4 Dialog Box Window Menu

The system gives a dialog box a **window** menu when the template specifies the WS_SYSMENU style. To prevent inappropriate input, the system automatically disables all items in the menu except **Move** and **Close**. The user can click **Move** to move the dialog box. When the user clicks **Close**, the system sends a **WM_COMMAND** message to the dialog box procedure with the *wParam* parameter set to IDCANCEL. This is identical to the message sent by the **Cancel** button when the user clicks it. The recommended action for this message is to close the dialog box and cancel the requested task.

Although other menus in dialog boxes are not recommended, a dialog box template can specify a menu by supplying the identifier or the name of a menu resource. In this case, the system loads the resource and creates the menu for the dialog box. Applications typically use menu identifiers or names in templates when using the templates to create custom windows rather than dialog boxes.

20.1.5 Dialog Box Fonts

The system uses the average character width of the dialog box font to calculate the position and dimensions of the dialog box. By default, the system draws all text in a dialog box using the SYSTEM_FONT font.

To specify a font for a dialog box other than the default, you must create the dialog box using a dialog box template. In a template resource, use the FONT Statement. In a dialog box template, set the DS_SETFONT or DS_SHELLFONT style and specify a point size and a typeface name. Even if a dialog box template specifies a font in this manner, the system always uses the system font for the dialog box title and dialog box menus.

When the dialog box has the DS_SETFONT or DS_SHELLFONT style, the system sends a **WM_SETFONT** message to the dialog box procedure and to each control as it creates the control. The dialog box procedure is responsible for saving the font handle passed with the **WM_SETFONT** message and selecting the handle into the display device context whenever it writes text to the window. Predefined controls do this by default.

The system font can vary between different versions of Windows. To have your application use the system font no matter which system it is running on, use DS_SHELLFONT with the typeface MS Shell Dlg, and use the **DIALOGEX Resource** instead of the **DIALOG Resource**. The system maps this typeface such that your dialog box will use the Tahoma font on Windows 2000/Windows XP, and the MS Sans Serif font on earlier systems.

Note that DS_SHELLFONT has no effect if the typeface is not MS Shell Dlg.

20.1.6 Templates in Memory

A dialog box template in memory consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format. In a standard template, the header is a **DLGTEMPLATE** structure followed by additional variable-length arrays. The data for each control consists of a **DLGITEMTEMPLATE** structure followed by additional variable-length arrays. In an extended dialog box template, the header uses the **DLGTEMPLATEEX** format and the control definitions use the **DLGITEMTEMPLATEEX** format.

To distinguish between a standard template and an extended template, check the first 16-bits of a dialog box template. In an extended template, the first **WORD** is 0xFFFF; any other value indicates a standard template.

If you create a dialog template in memory, you must ensure that the each of the **DLGITEMTEMPLATE** or **DLGITEMTEMPLATEEX** control definitions is aligned on **DWORD** boundaries. In addition, any creation data that follows a control definition

must be aligned on a **DWORD** boundary. All of the other variable-length arrays in a dialog box template must be aligned on **WORD** boundaries.

Template Header

In both the standard and extended templates for dialog boxes, the header includes the following general information:

- The location and dimensions of the dialog box
- The window and dialog box styles for the dialog box
- The number of controls in the dialog box. This value determines the number of **DLGITEMTEMPLATE** or **DLGITEMTEMPLATEEX** control definitions in the template.
- An optional menu resource for the dialog box. The template can indicate that the dialog box does not have a menu, or it can specify an ordinal value or null-terminated Unicode string that identifies a menu resource in an executable file.
- The window class of the dialog box. This can be either the predefined dialog box class, or an ordinal value or null-terminated Unicode string that identifies a registered window class.
- A null-terminated Unicode string that specifies the title for the dialog box window. If the string is empty, the title bar of the dialog box is blank. If the dialog box does not have the WS_CAPTION style, the system sets the title to the specified string but does not display it.
- If the dialog box has the DS_SETFONT style, the header specifies the point size and typeface name of the font to use for the text in the client area and controls of the dialog box.

In an extended template, the **DLGTEMPLATEEX** header also specifies the following additional information:

- The help context identifier of the dialog box window when the system sends a **WM_HELP** message.
- If the dialog box has the DS_SETFONT or DS_SHELLFONT style, the header specifies the font weight and indicates whether the font is italic.

Control Definitions

Following the template header is one or more control definitions that describe the controls of the dialog box. In both the standard and extended templates, the dialog box header has a member that indicates the number of control definitions in the template. In a standard template, each control definition consists of a **DLGITEMTEMPLATE** structure followed by additional variable-length arrays. In an extended template, the control definitions use the **DLGITEMTEMPLATEEX** format.

In both the standard and extended templates, the control definition includes the following information:

- The location and dimensions of the control.
- The window and control styles for the control.
- The control identifier.
- The window class of the control. This can be either the ordinal value of a predefined system class or a null-terminated Unicode string that specifies the name of a registered window class.
- A null-terminated Unicode string that specifies the initial text of the control, or an ordinal value that identifies a resource, such as an icon, in an executable file.
- An optional variable-length block of creation data. When the system creates the control, it passes a pointer to this data in the *lParam* parameter of the **WM_CREATE** message that it sends to the control.

In an extended template, the control definition also specifies a help context identifier for the control when the system sends a **WM_HELP** message.

20.2 When to Use a DialogBox

Most applications use dialog boxes to prompt for additional information for menu items that require user input. Using a dialog box is the only recommended way for an application to retrieve the input. For example, a typical **Open** menu item requires the name of a file to open, so an application should use a dialog box to prompt the user for the name. In such cases, the application creates the dialog box when the user clicks the menu item and destroys the dialog box immediately after the user supplies the information.

Many applications also use dialog boxes to display information or options while the user works in another window. For example, word processing applications often use a dialog box with a text-search option. While the application searches for the text, the dialog box remains on the screen. The user can then return to the dialog box and search for the same word again; or the user can change the entry in the dialog box and search for a new word. Applications that use dialog boxes in this way typically create one when the user clicks the menu item and continue to display it for as long as the application runs or until the user explicitly closes the dialog box.

To support the different ways applications use dialog boxes, there are two types of dialog box: modal and modeless. A *modal dialog box* requires the user to supply information or cancel the dialog box before allowing the application to continue. Applications use modal dialog boxes in conjunction with menu items that require additional information before they can proceed. A *modeless dialog box* allows the user to supply information and return to the previous task without closing the dialog box. Modal dialog boxes are simpler to manage than modeless dialog boxes because they are created, perform their task, and are destroyed by calling a single function.

To create either a modal or modeless dialog box, an application must supply a dialog box template to describe the dialog box style and content; the application must also supply a dialog box procedure to carry out tasks. The *dialog box template* is a binary description

of the dialog box and the controls it contains. The developer can create this template as a resource to be loaded from the application's executable file, or created in memory while the application runs. The *dialog box procedure* is an application-defined callback function that the system calls when it has input for the dialog box or tasks for the dialog box to carry out. Although a dialog box procedure is similar to a window procedure, it does not have the same responsibilities.

An application typically creates a dialog box by using either the DialogBox or CreateDialog function. **DialogBox** creates a modal dialog box; **CreateDialog** creates a modeless dialog box. These two functions load a dialog box template from the application's executable file and create a pop-up window that matches the template's specifications. There are other functions that create a dialog box by using templates in memory; they pass additional information to the dialog box procedure as the dialog box is created.

Dialog boxes usually belong to a predefined, exclusive window class. The system uses this window class and its corresponding window procedure for both modal and modeless dialog boxes. When the function is called, it creates the window for the dialog box as well as the windows for the controls in the dialog box, and then sends selected messages to the dialog box procedure. While the dialog box is visible, the predefined window procedure manages all messages, processing some messages and passing others to the dialog box procedure so that the procedure can carry out tasks. Applications do not have direct access to the predefined window class or window procedure, but they can use the dialog box template and dialog box procedure to modify the style and behavior of a dialog box.

20.3 Dialog Box Owner window

Most dialog boxes have an owner window (or more simply, an owner). When creating the dialog box, the application sets the owner by specifying the owner's window handle. The system uses the owner to determine the position of the dialog box in the Z order so that the dialog box is always positioned above its owner. Also, the system can send messages to the window procedure of the owner, notifying it of events in the dialog box.

The system automatically hides or destroys the dialog box whenever its owner is hidden or destroyed. This means the dialog box procedure requires no special processing to detect changes to the state of the owner window.

Because the typical dialog box is used in conjunction with a menu item, the owner window is usually the window containing the menu. Although it is possible to create a dialog box that has no owner, it is not recommended. For example, when a modal dialog box has no owner, the system does not disable any of the application's other windows and allows the user to continue to carry out work in the other windows, defeating the purpose of the modal dialog box.

When a modeless dialog box has no owner, the system neither hides nor destroys the dialog box when other windows in the application are hidden or destroyed. Although this does not defeat the purpose of the modeless dialog box, it requires that the application carry out special processing to ensure the dialog box is hidden and destroyed at appropriate times.

20.4 Creating Modal Dialog

The **DialogBoxParam** function creates a modal dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the *lParam* parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
INT_PTR DialogBoxParam(

    HINSTANCE hInstance,        //handle to the istance
    LPCTSTR lpTemplateName,     //name of the template*/
    HWND hWndParent,            //parent handle if any*/
    DLGPROC lpDialogFunc,       //dialog function
procedure*/
    LPARAM dwInitParam          /*initialize parameters*/
);
```

hInstance: Handle to the module whose executable file contains the dialog box template.

lpTemplateName: Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent: Handle to the window that owns the dialog box.

lpDialogFunc: Pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam: Specifies the value to pass to the dialog box in the *lParam* parameter of the **WM_INITDIALOG** message.

Return Value: If the function succeeds, the return value is the value of the *nResult* parameter specified in the call to the EndDialog function used to terminate the dialog box.

If the function fails because the *hWndParent* parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of

Microsoft® Windows®. If the function fails for any other reason, the return value is –1. To get extended error information, call GetLastError.

The **DialogBoxParam** function uses the CreateWindowEx function to create the dialog box. **DialogBoxParam** then sends a **WM_INITDIALOG** message (and a WM_SETFONT message if the template specifies the DS_SETFONT or DS_SHELLFONT style) to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the WS_VISIBLE style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the **EndDialog** function, **DialogBoxParam** destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the *nResult* parameter specified by the dialog box procedure when it called **EndDialog**.

20.5 Dialog Procedure

The **DialogProc** function is an application-defined callback function used with the CreateDialog and DialogBox families of functions. It processes messages sent to a modal or modeless dialog box. The **DLGPROC** type defines a pointer to this callback function. **DialogProc** is a placeholder for the application-defined function name.

```
INT_PTR CALLBACK DialogProc(

    HWND hwndDlg,               //handle to the dialog
    UINT uMsg,                  //message structure
    WPARAM wParam,              //wParam
    LPARAM lParam               //lParam
);
```

hwndDlg: Handle to the dialog box.

uMsg: Specifies the message.

wParam: Specifies additional message-specific information.

lParam: Specifies additional message-specific information.

Return Value: Typically, the dialog box procedure should return TRUE if it processed the message, and FALSE if it did not. If the dialog box procedure returns FALSE, the dialog manager performs the default dialog operation in response to the message.

If the dialog box procedure processes a message that requires a specific return value, the dialog box procedure should set the desired return value by calling SetWindowLong(*hwndDlg*, DWL_MSGRESULT, *lResult*) immediately before returning

TRUE. Note that you must call **SetWindowLong** immediately before returning TRUE; doing so earlier may result in the DWL_MSGRESULT value being overwritten by a nested dialog box message.

You should use the dialog box procedure only if you use the dialog box class for the dialog box. This is the default class and is used when no explicit class is specified in the dialog box template. Although the dialog box procedure is similar to a window procedure, it must not call the *DefWindowProc* function to process unwanted messages. Unwanted messages are processed internally by the dialog box window procedure.

20.6 The WM_INITDIALOG Message

The **WM_INITDIALOG** message is sent to the dialog box procedure immediately before a dialog box is displayed. Dialog box procedures typically use this message to initialize controls and carry out any other initialization tasks that affect the appearance of the dialog box.

```
WM_INITDIALOG

    WPARAM wParam
    LPARAM lParam;
```

wParam

Handle to the control to receive the default keyboard focus. The system assigns the default keyboard focus only if the dialog box procedure returns TRUE.

lParam

Specifies additional initialization data. This data is passed to the system as the *lParam* parameter in a call to the CreateDialogIndirectParam, CreateDialogParam, DialogBoxIndirectParam, or DialogBoxParam function used to create the dialog box. For property sheets, this parameter is a pointer to the PROPSHEETPAGE structure used to create the page. This parameter is zero if any other dialog box creation function is used.

Return Value:

The dialog box procedure should return TRUE to direct the system to set the keyboard focus to the control specified by *wParam*. Otherwise, it should return FALSE to prevent the system from setting the default keyboard focus.

The dialog box procedure should return the value directly. The DWL_MSGRESULT value set by the *SetWindowLong* function is ignored.

The control to receive the default keyboard focus is always the first control in the dialog box that is visible, not disabled, and that has the WS_TABSTOP style. When the dialog box procedure returns TRUE, the system checks the control to ensure that the procedure

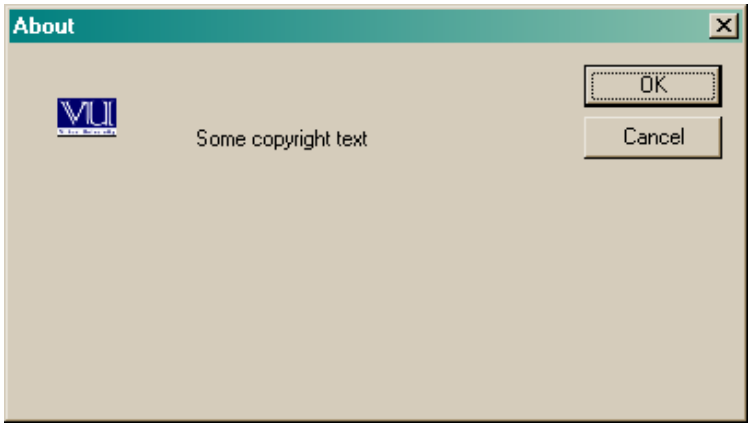
has not disabled it. If it has been disabled, the system sets the keyboard focus to the next control that is visible, not disabled, and has the WS_TABSTOP.

An application can return FALSE only if it has set the keyboard focus to one of the controls of the dialog box.

20.7 Using Dialog Procedure

```
BOOL CALLBACK AboutAuthorDialog(HWND hDlg, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
        switch(message)
        {
        case WM_INITDIALOG:
                return TRUE;
        case WM_COMMAND:
                switch(LOWORD(wParam))
                {
                case IDOK:
                case IDCANCEL:
                        EndDialog(hDlg, 0);
                        return TRUE;
                }
                break;
        }
        return FALSE;
}
```

20.8 Screen Shot of About Modal Dialog



20.9 Dialog Box Messages and functions

Following are the description of dialog box functions.

20.9.1 Retrieve handle of the control

The **GetDlgItem** function retrieves a handle to a control in the specified dialog box.

```
HWND GetDlgItem(

    HWND hDlg,
    int nIDDlgItem
);
```

hDlg
[in] Handle to the dialog box that contains the control.
nIDDlgItem
[in] Specifies the identifier of the control to be retrieved.

Return Value:

If the function succeeds, the return value is the window handle of the specified control.
If the function fails, the return value is NULL, indicating an invalid dialog box handle or a nonexistent control. To get extended error information, call GetLastError.

You can use the **GetDlgItem** function with any parent-child window pair, not just with dialog boxes. As long as the *hDlg* parameter specifies a parent window and the child window has a unique identifier (as specified by the *hMenu* parameter in the CreateWindow or CreateWindowEx function that created the child window), **GetDlgItem** returns a valid handle to the child window.

20.9.2 Set Window Text

The **SetWindowText** function changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, **SetWindowText** cannot change the text of a control in another application.

```
BOOL SetWindowText(

    HWND hWnd,
    LPCTSTR lpString
);
```

hWnd: Handle to the window or control whose text is to be changed.
lpString: Pointer to a null-terminated string to be used as the new title or control text.

Return Value:

If the function succeeds, the return value is nonzero.

If the target window is owned by the current process, **SetWindowText** causes a WM_SETTEXT message to be sent to the specified window or control. If the control is a list box control created with the WS_CAPTION style, however, **SetWindowText** sets the text for the control, not for the list box entries.

To set the text of a control in another process, send the **WM_SETTEXT** message directly instead of calling **SetWindowText**.

The **SetWindowText** function does not expand tab characters (ASCII code 0x09). Tab characters are displayed as vertical bar (|) characters.

20.9.3 Retrieve the identifier of the specified control

The **GetDlgCtrlID** function retrieves the identifier of the specified control.

```
int GetDlgCtrlID(

    HWND hwndCtl   /*handle to the control whose id is
required*/
);
```

hwndCtl: Handle to the control.

Return Value

If the function succeeds, the return value is the identifier of the control.

GetDlgCtrlID accepts child window handles as well as handles of controls in dialog boxes. An application sets the identifier for a child window when it creates the window by assigning the identifier value to the *hmenu* parameter when calling the CreateWindow or CreateWindowEx function.

Although **GetDlgCtrlID** may return a value if *hwndCtl* is a handle to a top-level window, top-level windows cannot have identifiers and such a return value is never valid.

20.9.4 Retrieve the text associated with the specified control in Dialog

The **GetDlgItemText** function retrieves the title or text associated with a control in a dialog box.

```
UINT GetDlgItemText(

    HWND hDlg,              /*handle to the dialog*/
    int nIDDlgItem,         /*id of the control */
    LPTSTR lpString,        /*text data*/
    int nMaxCount           /*maximum limit of the text*/
);
```

hDlg: Handle to the dialog box that contains the control.

nIDDlgItem: Specifies the identifier of the control whose title or text is to be retrieved.

lpString: Pointer to the buffer to receive the title or text.

nMaxCount: Specifies the maximum length, in **TCHARs**, of the string to be copied to the buffer pointed to by *lpString*. If the length of the string, including the NULL character, exceeds the limit, the string is truncated.

Return Value:

> If the function succeeds, the return value specifies the number of **TCHARs** copied to the buffer, not including the terminating NULL character.
>
> If the function fails, the return value is zero. To get extended error information, call GetLastError.

If the string is as long as or longer than the buffer, the buffer will contain the truncated string with a terminating NULL character.

The **GetDlgItemText** function sends a WM_GETTEXT message to the control.

For the ANSI version of the function, the number of **TCHARs** is the number of bytes; for the Unicode version, it is the number of characters.

20.9.5 Sends a message to the specified control in a dialog box

The **SendDlgItemMessage** function sends a message to the specified control in a dialog box.

```
LRESULT SendDlgItemMessage(

    HWND hDlg,              /*handle to the dialog*/
    int nIDDlgItem,         /*id of the dialog item*/
    UINT Msg,               /*message type*/
    WPARAM wParam,          /*message wParam*/
    LPARAM lParam           /*message lParam*/
);
```

hDlg: Handle to the dialog box that contains the control.

nIDDlgItem: Specifies the identifier of the control that receives the message.

Msg: Specifies the message to be sent.

wParam: Specifies additional message-specific information.

lParam: Specifies additional message-specific information.

Return Value:

> The return value specifies the result of the message processing and depends on the message sent.

The **SendDlgItemMessage** function does not return until the message has been processed.

Using **SendDlgItemMessage** is identical to retrieving a handle to the specified control and calling the SendMessage function.

Example

In this example we send a message to edit control of EM_LIMITTEXT. This message will limit the text to the given number say 25 in our case. Edit control will not receive more than this limit.

```
EM_LIMITTEXT
wParam,      // text length
lParam       // not used; must be zero
//Sets the text limit of an edit control

//This message is sent by sendDlgItemMessage function.
SendDlgItemMessage(hEdit, EM_LIMITTEXT, (WPARAM)25, (LPARAM)0);
```

20.9.6 Setting or getting text associated with a window or control

```
WM_GETTEXT
wParam,      // number of characters to copy
lParam       // text buffer
```

Get Text Message retrieve the text associated with the window. This text could be a caption text on any window or the text displayed in edit controls.

```
WM_SETTEXT
wParam,      // not used; must be zero
lParam       // window-text string (LPCTSTR)
```

Set Text set the text in window.

GetWindowText() function internally sends a WM_GETTEXT message to get the text.
SetWindowText() function internally sends a WM_SETTEXT message to set the text.

20.9.7 Set or retrieve current selection in an edit control

Setting or getting the current selection in an edit control we use two message EM_SETSEL and EM_GETSEL.

```
EM_SETSEL  or EM_GETSEL
wParam,      // starting position
lParam       // ending position
```

20.10 Creating Modeless Dialog

In our previous lecture, we have studied Modeless dialogs. Here we will create the modeless dialogs.
Modeless dialogs are created with *CreateDialog* function.

```
HWND CreateDialog(

    HINSTANCE hInstance,  /*handle to the instance*/
    LPCTSTR lpTemplate,   /*template name*/
    HWND hWndParent,      /*handle to the parent*/
    DLGPROC lpDialogFunc  /*dialog function*/
);
```

hInstance: Handle to the module whose executable file contains the dialog box template.

lpTemplate: Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.
hWndParent: Handle to the window that owns the dialog box.

lpDialogFunc: Pointer to the dialog box procedure.

Return Value:

> If the function succeeds, the return value is the handle to the dialog box.
> If the function fails, the return value is NULL. To get extended error information, call GetLastError.

The **CreateDialog** function uses the CreateWindowEx function to create the dialog box. **CreateDialog,** then sends a WM_INITDIALOG message (and a WM_SETFONT

message if the template specifies the DS_SETFONT or DS_SHELLFONT style) to the dialog box procedure. The function displays the dialog box if the template specifies the WS_VISIBLE style. Finally, **CreateDialog** returns the window handle to the dialog box.

After **CreateDialog** returns, the application displays the dialog box (if it is not already displayed) by using the ShowWindow function. The application destroys the dialog box by using the DestroyWindow function. To support keyboard navigation and other dialog box functionality, the message loop for the dialog box must call the IsDialogMessage function.

20.10.1 Showing Modeless Dialog

Modeless dialogs are initially hidden unless their property of visibility is not set.

For showing Dialog we can use *ShowWindow* function, which can show any window created in Windows.

```
BOOL ShowWindow(

    HWND hWnd,     /*handle to the window*/
    int nCmdShow   /*show style*/
);
```

hWnd: Handle to the window.
nCmdShow: Specifies how the window is to be shown. This parameter is ignored the first time an application calls **ShowWindow**, if the program that launched the application provides a STARTUPINFO structure. Otherwise, the first time **ShowWindow** is called, the value should be the value obtained by the WinMain function in its *nCmdShow* parameter. In subsequent calls, this parameter can be one of the following values.

SW_HIDE: Hides the window and activates another window.

SW_MAXIMIZE: Maximizes the specified window.

SW_MINIMIZE: Minimizes the specified window and activates the next top-level window in the Z order.
SW_RESTORE: Activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when restoring a minimized window.

SW_SHOW: Activates the window and displays it in its current size and position.

SW_SHOWDEFAULT: Sets the show state based on the SW_ value specified in the **STARTUPINFO** structure passed to the CreateProcess function by the program that started the application.

SW_SHOWMAXIMIZED: Activates the window and displays it as a maximized window.

SW_SHOWMINIMIZED: Activates the window and displays it as a minimized window.

SW_SHOWMINNOACTIVE: Displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED, except the window is not activated.

SW_SHOWNA: Displays the window in its current size and position. This value is similar to SW_SHOW, except the window is not activated.

SW_SHOWNOACTIVATE: Displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL, except the window is not active.

SW_SHOWNORMAL: Activates and displays a window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when displaying the window for the first time.

Return Value

If the window was previously visible, the return value is nonzero.
If the window was previously hidden, the return value is zero.

To perform certain special effects when showing or hiding a window, use AnimateWindow.

The first time an application calls **ShowWindow**, it should use the **WinMain** function's *nCmdShow* parameter as its *nCmdShow* parameter. Subsequent calls to **ShowWindow** must use one of the values in the given list, instead of the one specified by the **WinMain** function's *nCmdShow* parameter.

As noted in the discussion of the *nCmdShow* parameter, the *nCmdShow* value is ignored in the first call to **ShowWindow** if the program that launched the application specifies startup information in the structure. In this case, **ShowWindow** uses the information specified in the **STARTUPINFO** structure to show the window. On subsequent calls, the application must call **ShowWindow** with *nCmdShow* set to SW_SHOWDEFAULT to use the startup information provided by the program that launched the application. This behavior is designed for the following situations:

- Applications create their main window by calling CreateWindow with the WS_VISIBLE flag set.
- Applications create their main window by calling **CreateWindow** with the WS_VISIBLE flag cleared, and later call **ShowWindow** with the SW_SHOW flag set to make it visible.

20.10.2 Processing Dialog Messages

The **IsDialogMessage** function determines whether a message is intended for the specified dialog box and, if it is, processes the message.

```
BOOL IsDialogMessage(

    HWND hDlg,      /*handle to the dialog*/
    LPMSG lpMsg     /*message structure */
);
```

hDlg: Handle to the dialog box.

lpMsg: Pointer to an MSG structure that contains the message to be checked.

Return Value:

If the message has been processed, the return value is nonzero.
If the message has not been processed, the return value is zero.

Although the **IsDialogMessage** function is intended for modeless dialog boxes, you can use it with any window that contains controls, enabling the windows to provide the same keyboard selection as is used in a dialog box.

When **IsDialogMessage** processes a message, it checks for keyboard messages and converts them into selections for the corresponding dialog box. For example, the TAB key, when pressed, selects the next control or group of controls, and the DOWN ARROW key, when pressed, selects the next control in a group. Because the **IsDialogMessage** function performs all necessary translating and dispatching of messages, a message processed by **IsDialogMessage** must not be passed to the TranslateMessage or DispatchMessage function.

IsDialogMessage sends WM_GETDLGCODE messages to the dialog box procedure to determine which keys should be processed.

IsDialogMessage can send DM_GETDEFID and DM_SETDEFID messages to the window. These messages are defined in the Winuser.h header file as WM_USER and **WM_USER** + 1, so conflicts are possible with application-defined messages having the same values.

20.10.3 Message Loop to dispatch messages to a modeless dialog

```
While(GetMessage(&msg, NULL, 0, 0)  > 0)

/*get message from the queue and check its validity, it should be greater than zero*/
{
```

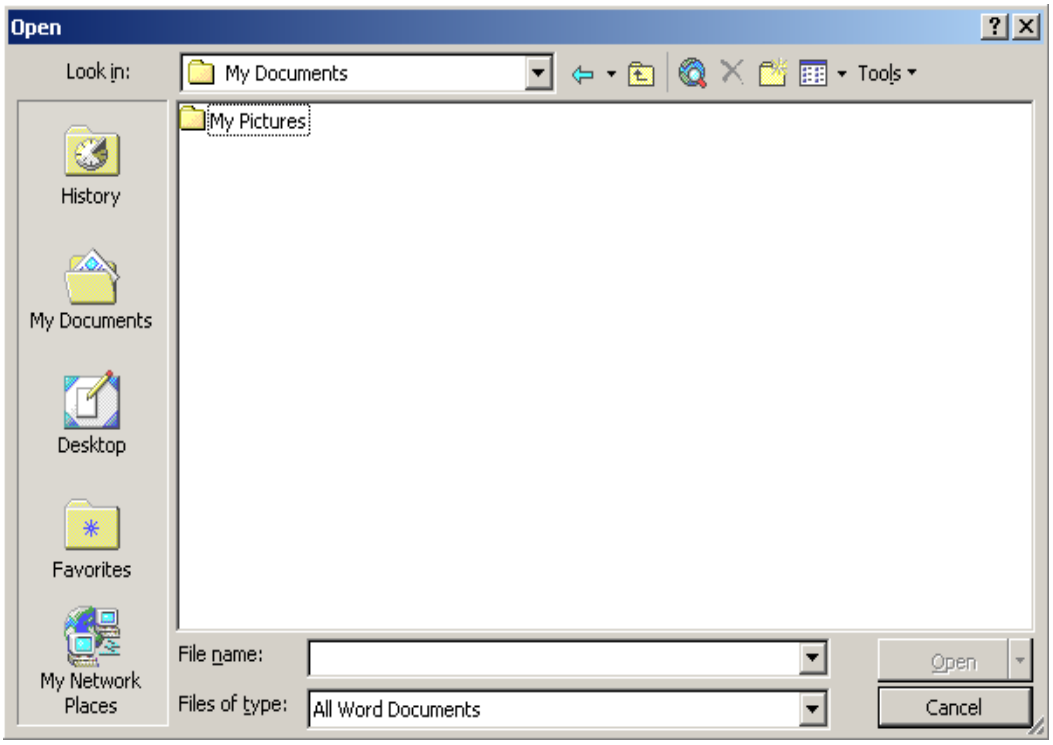
```
/*check if this is the dialog message otherwise send it to the window procedure*/

        if(!IsDialogMessage(hDlg, &msg))
        {
                /*translate message before dispatching it*/
                TranslateMessage(&msg);
                /*now dispatch to the window procedure*/
                DispatchMessage(&msg);
        }
}
```

Modeless dialogs can be destroyed by calling *DestroyWindow* function.

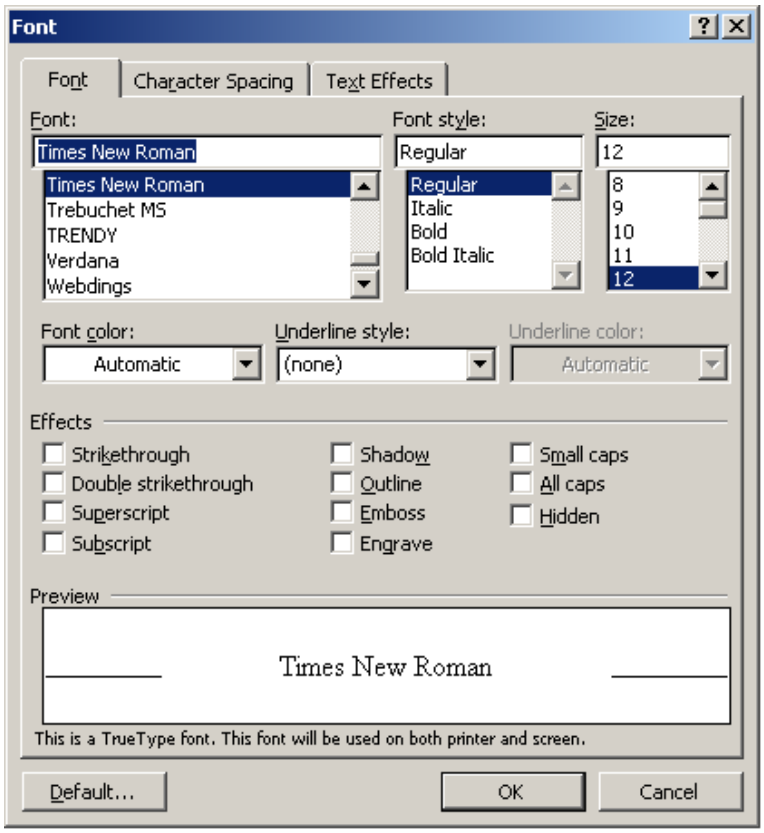
20.11 Windows Common Dialogs

Windows common dialogs are of the following types.

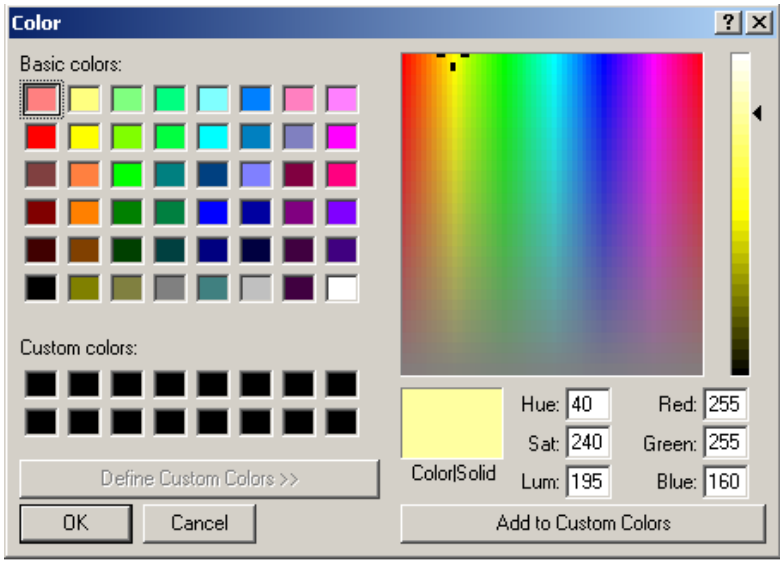
20.11.1 Open File Dialog



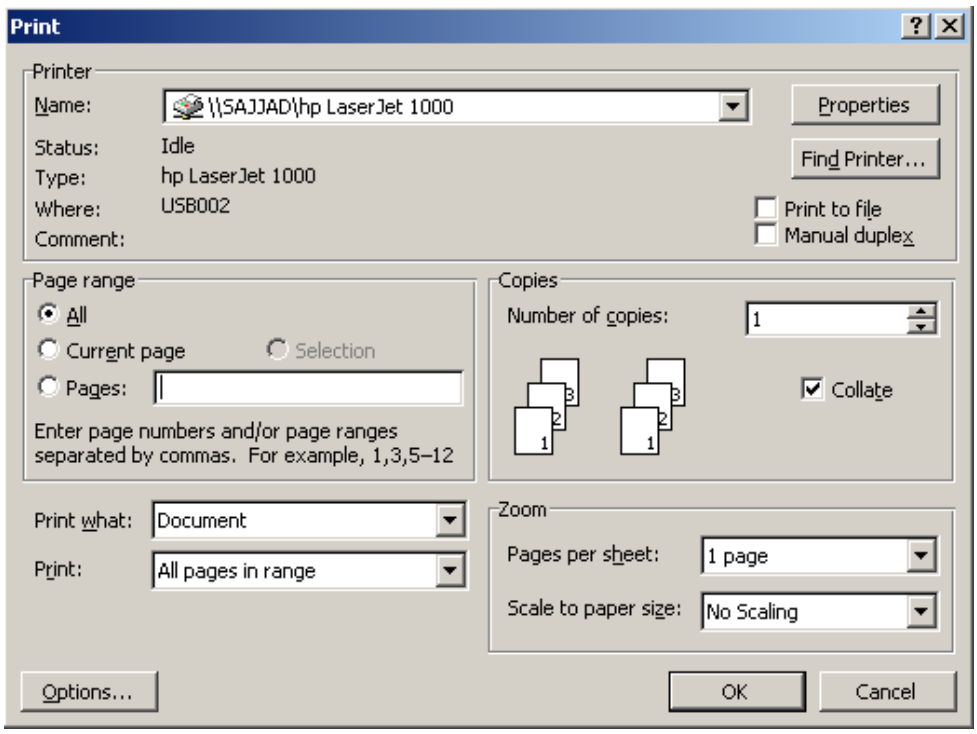
20.11.2 Choose Font Dialog



20.11.3 Choose Color Dialog



20.11.4 Print Dialog



Summary

A dialog box template is binary data that describes the dialog box, defining its height, width, style, and the controls it contains. Dialog box template becomes later part of the executable file. Dialogs are of two types: Modal dialogs and Modeless dialogs. For dispatching message for Modeless dialogs, we can use *IsDialogMessage* Function in our main Message Loop. In windows lot of common dialogs are available. Print dialog is used to print the document, this dialog show the setting for printer and its path name. Choose color dialog help the user to choose the color of its own choice. File Open dialog are useful to open and save the files on disk.

Exercises

1. Create a notepad like window and facilitate the user to save and open the text in a file. For saving the file use open file and save file dialog.

Chapter 21

Using Dialogs and Windows Controls

21.1 Windows Common Dialogs

In our previous lecture, we have viewed common dialogs and in this lecture we will learn to use them. Following are the Windows common dialog names and the functions that create these common dialogs.

- Choose color:
  - For creating the color dialog we use function ChooseColor(&CHOOSCOLOR). This function inputs CHOOSCOLOR structure and create the color dialog.

- Find:
  - FindText(&FINDREPLACE) function create the find text dialog. This dialog helps to find and replace text in text document. This function inputs FINDREPLACE structure.

- Choose font:
  - ChooseFont(&CHOOSEFONT) function create a Choose Font dialog. This dialog helps choose the font from installed system fonts. This function inputs CHOOSEFONT structure.

- Open File:
  - GetOpenFilename(&OPENFILENAM) function creates a dialog that lets the user specify the drive, directory, and the name of a file or set of files to open.

- Print
  - PrintDlg() function open dialog which help to print the document.

- Save As:
  - GetSaveFilename(&OPENFILENAM) function open a file dialog which help to save a file on the drive

21.2 Dialog Units

Dialog Unit (DLU): A unit of horizontal or vertical distance within a dialog box. A horizontal DLU is the average width of the current dialog box font divided by 4. A vertical DLU is the average height of the current dialog-box font divided by 8.
Dialogs Units have also explained in our previous lectures.

21.3 Groups and Focus

- WS_GROUP style specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the WS_GROUP style FALSE after the first control belongs to the same group. The next control with the WS_GROUP style starts the next group (that is, one group ends where the next begins)

- Focus: for setting focus on any control in dialog the SetFocus and GetFocus functions are used.

21.4 Edit Control

Dialog boxes and controls support communication between applications and their users. An *edit control* is a rectangular control window typically used in a dialog box to permit the user to enter and edit text by typing on the keyboard.

21.4.1 Edit Control Features

An edit control is selected and receives the input focus when a user clicks the mouse inside it or presses the TAB key. After it is selected, the edit control displays its text (if any) and a flashing caret that indicates the insertion point. The user can then enter text, move the insertion point, or select text to be edited by using the keyboard or the mouse. An edit control can send notification messages to its parent window in the form of WM_COMMAND messages. A parent window can send messages to an edit control in a dialog box by calling the *SendDlgItemMessage* function.

The system provides both single-line edit controls (sometimes called SLEs) and multiline edit controls (sometimes called MLEs). Edit controls belong to the EDIT window class.

A combo box is a control that combines much of the functionality of an edit control and a list box. In a combo box, the edit control displays the current selection and the list box presents options a user can select. Many developers use the dialog boxes provided in the common dialog box library (Comdlg32.dll) to perform tasks that otherwise might require customized edit controls.

21.4.2 Edit Control Notification Messages

The user makes editing requests by using the keyboard and mouse. The system sends each request to the edit control's parent window in the form of a **WM_COMMAND** message. The message includes the edit control identifier in the low-order word of the *wParam* parameter, the handle of the edit control in the *lParam* parameter, and an edit

control notification message corresponding to the user's action in the high-order word of the *wParam* parameter.

An application should examine each notification message and respond appropriately. The following table lists each edit control notification message and the action that generates it.

Notification message	User action
EN_CHANGE	The user has modified text in an edit control. The system updates the display before sending this message (unlike EN_UPDATE).
EN_ERRSPACE	The edit control cannot allocate enough memory to meet a specific request.
EN_HSCROLL	The user has clicked the edit control's horizontal scroll bar. The system sends this message before updating the screen.
EN_KILLFOCUS	The user has selected another control.
EN_MAXTEXT	While inserting text, the user has exceeded the specified number of characters for the edit control. Insertion has been truncated. This message is also sent either when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted exceeds the width of the edit control or when an edit control does not have the ES_AUTOVSCROLL style and the total number of lines to be inserted exceeds the height of the edit control.
EN_SETFOCUS	The user has selected this edit control.
EN_UPDATE	The user has altered the text in the edit control and the system is about to display the new text. The system sends this message after formatting the text, but before displaying it, so that the application can resize the edit control window.
EN_VSCROLL	The user has clicked the edit control's vertical scroll bar or has scrolled the mouse wheel over the edit control. The system sends this message before updating the screen.

In addition, the system sends a WM_CTLCOLOREDIT message to an edit control's parent window before the edit control is drawn. This message contains a handle of the edit control's display context (DC) and a handle of the child window. The parent window can use these handles to change the edit control's text and background colors.

21.4.3 Edit Control Default Message Processing

The window procedure for the predefined edit control window class carries out default processing for all messages that the edit control procedure does not process. When the edit control procedure returns FALSE for any message, the predefined window procedure checks the messages and carries out the following default actions.

Message	Default action
EM_CANUNDO	Returns TRUE if the edit control operation can be undone.
EM_CHARFROMPOS	Returns the character index and line index of the character nearest the specified point.
EM_EMPTYUNDOBUFFER	Empties the undo buffer and sets the undo flag retrieved by the **EM_CANUNDO** message to FALSE. The system automatically clears the undo flag whenever the edit control receives a WM_SETTEXT or **EM_SETHANDLE** message.
EM_FMTLINES	Adds or removes soft line-break characters (two carriage returns and a line feed) to the ends of wrapped lines in a multiline edit control. It is not processed by single-line edit controls.
EM_GETFIRSTVISIBLELINE	Returns the zero-based index of the first visible character in a single-line edit control or the zero-based index of the uppermost visible line in a multiline edit control.
EM_GETHANDLE	Returns a handle identifying the buffer containing the multiline edit control's text. It is not processed by single-line edit controls.
EM_GETLIMITTEXT	Returns the current text limit, in characters.
EM_GETLINE	Copies characters in a single-line edit control to a buffer and returns the number of characters copied. In a multiline edit control, retrieves a line of text from the control and returns the number of characters copied.
EM_GETLINECOUNT	Returns the number of lines in the edit control.
EM_GETMARGINS	Returns the widths of the left and right margins.
EM_GETMODIFY	Returns a flag indicating whether the content of an edit control has been modified.
EM_GETPASSWORDCHAR	Returns the character that edit controls use in conjunction with the ES_PASSWORD style.
EM_GETRECT	Returns the coordinates of the formatting rectangle in an edit control.
EM_GETSEL	Returns the starting and ending character positions of the current selection in the edit control.
EM_GETTHUMB	Returns the position of the scroll box in the vertical scroll bar in a multiline edit control.
EM_GETWORDBREAKPROC	Returns the address of the current Wordwrap function in an edit control.
EM_LINEFROMCHAR	Returns the zero-based number of the line in a multiline edit control that contains a specified character index.

	This message is the reverse of the EM_LINEINDEX message. It is not processed by single-line edit controls.
EM_LINEINDEX	Returns the character of a line in a multiline edit control. This message is the reverse of the **EM_LINEFROMCHAR** message. It is not processed by single-line edit controls.
EM_LINELENGTH	Returns the length, in characters, of a single-line edit control. In a multiline edit control, returns the length, in characters, of a specified line.
EM_LINESCROLL	Scrolls the text vertically in a single-line edit control or horizontally in a multiline edit control (when the control has the ES_LEFT style). The *lParam* parameter specifies the number of lines to scroll vertically, starting from the current line. The *wParam* parameter specifies the number of characters to scroll horizontally, starting from the current character.
EM_POSFROMCHAR	Returns the client coordinates of the specified character.
EM_REPLACESEL	Replaces the current selection with the text in an application-supplied buffer, sends the parent window **EN_UPDATE** and **EN_CHANGE** messages, and updates the undo buffer.
EM_SCROLL	Scrolls the text vertically in a multiline edit control. This message is equivalent to sending a WM_VSCROLL message to the edit control. It is not processed by single-line edit controls.
EM_SCROLLCARET	Scrolls the caret into view in an edit control.
EM_SETFONT	Unsupported.
EM_SETHANDLE	Sets a handle to the memory used as a text buffer, empties the undo buffer, resets the scroll positions to zero, and redraws the window.
EM_SETLIMITTEXT	Sets the maximum number of characters the user may enter in the edit control. **Windows NT/2000/XP:** For single-line edit controls, this value is either 0x7FFFFFFE or the value of the *wParam* parameter, whichever is smaller. For multiline edit controls, this value is either –1 or the value of the *wParam* parameter, whichever is smaller. **Windows 95/98/Me:** For single-line edit controls, this value is either 0x7FFE or the value of the *wParam* parameter, whichever is smaller. For multiline edit controls, this value is either 0xFFFF or the value of the *wParam* parameter, whichever is smaller.

EM_SETMARGINS	Sets the widths of the left and right margins, and redraws the edit control to reflect the new margins.
EM_SETMODIFY	Sets or clears the modification flag to indicate whether the edit control has been modified.
EM_SETPASSWORDCHAR	Defines the character that edit controls use in conjunction with the ES_PASSWORD style.
EM_SETREADONLY	Sets or removes the read-only style (ES_READONLY) in an edit control.
EM_SETRECT	Sets the formatting rectangle for the multiline edit control and redraws the window. It is not processed by single-line edit controls.
EM_SETRECTNP	Sets the formatting rectangle for the multiline edit control but does not redraw the window. It is not processed by single-line edit controls.
EM_SETSEL	Selects a range of characters in the edit control by setting the starting and ending positions to be selected.
EM_SETTABSTOPS	Sets tab-stop positions in the multiline edit control. It is not processed by single-line edit controls.
EM_SETWORDBREAKPROC	Replaces the default Wordwrap function with an application-defined Wordwrap function.
EM_UNDO	Removes any text that was just inserted or inserts any deleted characters and sets the selection to the inserted text. If necessary, sends the **EN_UPDATE** and **EN_CHANGE** notification messages to the parent window.
WM_CHAR	Writes a character to the single-line edit control and sends the **EN_UPDATE** and **EN_CHANGE** notification messages to the parent window. Writes a character to the multiline edit control. Handles the accelerator keys for standard functions, such as CTRL+C for copying and CTRL+V for pasting. In multiline edit controls, also processes TAB, and CTRL+TAB keystrokes to move among the controls in a dialog box and to insert tabs into multiline edit controls. Uses the MessageBeep function for illegal characters.
WM_CLEAR	Clears the current selection, if any, in an edit control. If there is no current selection, deletes the character to the right of the caret. If the user presses the SHIFT key, this cuts the selection to the clipboard, or deletes the character to the left of the caret when there is no selection. If the user presses the CTRL key, this deletes the selection, or deletes to the end of the line when there is no selection.

WM_COPY	Copies text to the clipboard unless the style is ES_PASSWORD, in which case the message returns zero.
WM_CREATE	Creates the edit control and notifies the parent window with TRUE for success or –1 for failure.
WM_CUT	Cuts the selection to the clipboard, or deletes the character to the left of the cursor if there is no selection.
WM_ENABLE	Causes the rectangle to be redrawn in gray for single-line edit controls. Returns the enabled state for single-line and multiline edit controls.
WM_ERASEBKGND	Fills the multiline edit control window with the current color of the edit control.
WM_GETDLGCODE	Returns the following values: DLGC_WANTCHARS, DLGC_HASSETSEL, and DLGC_WANTARROWS. In multiline edit controls, it also returns DLGC_WANTALLKEYS. If the user presses ALT+BACKSPACE, it also returns DLGC_WANTMESSAGE.
WM_GETFONT	Returns the handle of the font being used by the control, or NULL if the control uses the system font.
WM_GETTEXT	Copies the specified number of characters to a buffer and returns the number of characters copied.
WM_GETTEXTLENGTH	Returns the length, in characters, of the text in an edit control. The length does not include the null-terminating character.
WM_HSCROLL	Scrolls the text in a multiline edit control horizontally and handles scroll box movement.
WM_KEYDOWN	Performs standard processing of the virtual-key codes.
WM_KILLFOCUS	Removes the keyboard focus of an edit control window, destroys the caret, hides the current selection, and notifies the parent window that the edit control has lost the focus.
WM_LBUTTONDBLCLK	Clears the current selection and selects the word under the cursor. If the SHIFT key is depressed, extends the selection to the word under the cursor.
WM_LBUTTONDOWN	Changes the current insertion point. If the SHIFT key is depressed, extends the selection to the position of the cursor. In multiline edit controls, also sets the timer to automatically scroll when the user holds down the mouse button outside the multiline edit control window.
WM_LBUTTONUP	Releases the mouse capture and sets the text insertion point in the single-line edit control. In a multiline edit control, also kills the timer set in the

	WM_LBUTTONDOWN message.
WM_MOUSEMOVE	Changes the current selection in the single-line edit control, if the mouse button is down. In a multiline edit controls, also sets the timer to automatically scroll if the user holds down the mouse button outside the multiline edit control window.
WM_NCCREATE	Pointer to the CREATESTRUCT structure for the window. This message is sent to the **WM_CREATE** message when a window is first created.
WM_NCDESTROY	Frees all memory associated with the edit control window, including the text buffer, undo buffer, tab-stop buffer, and highlight brush.
WM_PAINT	Erases the background, fills the window with the current color of the edit control window, draws the border (if any), sets the font and draws any text, and shows the text-insertion caret.
WM_PASTE	Pastes text from the clipboard into the edit control window at the caret position.
WM_SETFOCUS	Sets the keyboard focus of an edit control window (shows the current selection, if it was hidden, and creates the caret).
WM_SETFONT	Sets the font and optionally redraws the edit control.
WM_SETTEXT	Copies text to the single-line edit control, notifies the parent window when there is insufficient memory, empties the undo buffer, and sends the **EN_UPDATE** and **EN_CHANGE** notification messages to the parent window. In multiline edit controls, also rewraps the lines (if necessary) and sets the scroll positions.
WM_SIZE	Changes the size of the edit control window and ensures that the minimum size accommodates the height and width of a character.
WM_SYSCHAR	Returns TRUE if the user presses ALT+BACKSPACE; otherwise takes no action.
WM_SYSKEYDOWN	Undoes the last action if the user presses ALT+BACKSPACE; otherwise takes no action.
WM_TIMER	Scrolls the text in the edit control window if the user holds down the mouse button outside the multiline edit control window.
WM_UNDO	Removes any text that was just inserted or inserts any deleted characters and sets the selection to the inserted text. If necessary, sends the **EN_UPDATE** and **EN_CHANGE** notification messages to the parent window.

WM_VSCROLL	Scrolls a multiline edit control vertically and handles scroll box movement. It is not processed by single-line edit controls.

The predefined edit control window procedure passes all other messages to the *DefWindowProc* function for default processing.

21.5 Button

A *button* is a control the user can click to provide input to an application.

21.5.1 Button Types and Styles

There are five styles of a button:

- Check Boxes
- Group Boxes
- Owner Drawn Buttons
- Push Buttons
- Radio Buttons

Check Boxes

A *check box* consists of a square box and application-defined text (label), an icon, or a bitmap, that indicates a choice the user can make by selecting the button. Applications typically display check boxes in a group box to permit the user to choose from a set of related, but independent options. For example, an application might present a group of check boxes from which the user can select error conditions that produce warning beeps.

A check box can be one of four styles: standard, automatic, three-state, and automatic three-state, as defined by the constants BS_CHECKBOX, BS_AUTOCHECKBOX, BS_3STATE, and BS_AUTO3STATE, respectively. Each style can assume two check states: checked (a check mark inside the box) or cleared (no check mark). In addition, a three-state check box can assume an indeterminate state (a grayed box inside the check box). Repeatedly clicking a standard or automatic check box toggles it from checked to cleared and back again. Repeatedly clicking a three-state check box toggles it from checked to cleared to indeterminate and back again.

When the user clicks a check box (of any style), the check box receives the keyboard focus. The system sends the check box's parent window a WM_COMMAND message containing the BN_CLICKED notification code. The parent window doesn't acknowledge this message if it comes from an automatic check box or automatic three-state check box, because the system automatically sets the check state for those styles. But the parent window must acknowledge the message if it comes from a check box or three-state check box because the parent window is responsible for setting the check state

for those styles. Regardless of the check box style, the system automatically repaints the check box once its state is changed.

Group Boxes

A *group box* is a rectangle that surrounds a set of controls, such as check boxes or radio buttons, with application-defined text (label) in its upper left corner. The sole purpose of a group box is to organize controls related by a common purpose (usually indicated by the label). The group box has only one style, defined by the constant BS_GROUPBOX. Because a group box cannot be selected, it has no check state, focus state, or push state. An application cannot send messages to a group box.

Owner Drawn Buttons

Unlike radio buttons, an *owner-drawn button* is painted by the application, not by the system, and has no predefined appearance or usage. Its purpose is to provide a button whose appearance and behavior are defined by the application alone. There is only one owner-drawn button style: BS_OWNERDRAW.

When the user selects an owner-drawn button, the system sends the button's parent window a **WM_COMMAND** message containing the **BN_CLICKED** notification code, just as it does for a button that is not owner-drawn. The application must respond appropriately.

Push Buttons

A *push button* is a rectangle containing application-defined text (label), an icon, or a bitmap that indicates what the button does when the user selects it. A push button can be one of two styles: standard or default, as defined by the constants BS_PUSHBUTTON and BS_DEFPUSHBUTTON. A standard push button is typically used to start an operation. It receives the keyboard focus when the user clicks it. A default push button, on the other hand, is typically used to indicate the most common or default choice. It is a button that the user can select by simply pressing ENTER when a dialog box has the input focus.

When the user clicks a push button (of either style), it receives the keyboard focus. The system sends the button's parent window a **WM_COMMAND** message that contains the **BN_CLICKED** notification code. In response, the dialog box typically closes and carries out the operation indicated by the button.

The default push button cannot be a check box, a radio button, or an ownerdraw button at the same time.

Radio Buttons

A *radio button* consists of a round button and application-defined text (a label), an icon, or a bitmap that indicates a choice the user can make by selecting the button. An application typically uses radio buttons in a group box to permit the user to choose from a set of related, but mutually exclusive options. For example, the application might present a group of radio buttons from which the user can select a format preference for text selected in the client area. The user could select a left-aligned, right-aligned, or centered format by selecting the corresponding radio button. Typically, the user can select only one option at a time from a set of radio buttons.

A radio button can be one of two styles: standard or automatic, as defined by the constants BS_RADIOBUTTON and BS_AUTORADIOBUTTON. Each style can assume two check states: checked (a dot in the button) or cleared (no dot in the button). Repeatedly selecting a radio button (standard or automatic) toggles it from checked to cleared and back again.

When the user selects either state, the radio button receives the keyboard focus. The system sends the button's parent window a **WM_COMMAND** message containing the **BN_CLICKED** notification code. The parent window doesn't acknowledge this message if it comes from an automatic radio button because the system automatically sets the check state for that style. But the parent window should acknowledge the message if it comes from a radio button because the parent window is responsible for setting the check state for that style. Regardless of the radio button style, the system automatically repaints the button as its state changes.

When the user selects an automatic radio button, the system automatically sets the check state of all other radio buttons within the same group to clear. The same behavior is available for standard radio buttons by using the WS_GROUP style, as discussed in Dialog Boxes.

21.5.2 Notification Messages from Button

When the user clicks a button, its state changes, and the button sends notification messages to its parent window. For example, a push button control sends the BN_CLICKED notification message whenever the user chooses the button. In all cases (except for BCN_HOTITEMCHANGE), the low-order word of the *wParam* parameter contains the control identifier, the high-order word of *wParam* contains the notification code, and the *lParam* parameter contains the control window handle.

Both the message and the parent window's response depend on the type, style, and current state of the button. Following are the button notification messages an application should monitor and process.

Message	Description
BCN_HOTITEMCHANGE	Microsoft® Windows® XP: The mouse entered or left the client area of a button.
BN_CLICKED	The user clicked a button.
BN_DBLCLK or BN_DOUBLECLICKED	The user double-clicked a button.
BN_DISABLE	A button is disabled.
BN_PUSHED or BN_HILITE	The user pushed a button.
BN_KILLFOCUS	The button lost the keyboard focus.
BN_PAINT	The button should be painted.
BN_SETFOCUS	The button gained the keyboard focus.
BN_UNPUSHED or BN_UNHILITE	The button is no longer pushed.

A button sends the **BN_DISABLE**, **BN_PUSHED**, **BN_KILLFOCUS**, **BN_PAINT**, **BN_SETFOCUS**, and **BN_UNPUSHED** notification messages only if it has the BS_NOTIFY style. **BN_DBLCLK** notification messages are sent automatically for BS_USERBUTTON, BS_RADIOBUTTON, and BS_OWNERDRAW buttons. Other button types send **BN_DBLCLK** only if they have the BS_NOTIFY style. All buttons send the **BN_CLICKED** notification message regardless of their button styles.

For automatic buttons, the system changes the push state and paints the button. In this case, the application typically processes only the **BN_CLICKED** and **BN_DBLCLK** notification messages. For buttons that are not automatic, the application typically responds to the notification message by sending a message to change the state of the button.

When the user selects an owner-drawn button, the button sends its parent window a WM_DRAWITEM message containing the identifier of the control to be drawn and information about its dimensions and state.

21.5.3 Button Default Message Processing

The window procedure for the predefined button control window class carries out default processing for all messages that the button control procedure does not process. When the button control procedure returns FALSE for any message, the predefined window procedure checks the messages and performs the default actions listed in the following table.

Message	Default action
BM_CLICK	Sends the button a WM_LBUTTONDOWN and a WM_LBUTTONUP message, and sends the parent window a **BN_CLICKED** notification message.
BM_GETCHECK	Returns the check state of the button.
BM_GETIMAGE	Returns a handle to the bitmap or icon associated with the button or NULL if the button has no bitmap or icon.
BM_GETSTATE	Returns the current check state, push state, and focus state of the button.
BM_SETCHECK	Sets the check state for all styles of radio buttons and check boxes. If the *wParam* parameter is greater than zero for radio buttons, the button is given the WS_TABSTOP style.
BM_SETIMAGE	Associates the specified bitmap or icon handle with the button and returns a handle to the previous bitmap or icon.
BM_SETSTATE	Sets the push state of the button. For owner-drawn buttons, a **WM_DRAWITEM** message is sent to the parent window if the state of the button has changed.
BM_SETSTYLE	Sets the button style. If the low-order word of the *lParam* parameter is TRUE, the button is redrawn.
WM_CHAR	Checks a check box or automatic check box when the user presses the plus (+) or equal (=) keys. Clears a check box or automatic check box when the user presses the minus (–) key.
WM_ENABLE	Paints the button.
WM_ERASEBKGND	Erases the background for owner-drawn buttons. The backgrounds of other buttons are erased as part of the WM_PAINT and **WM_ENABLE** processing.
WM_GETDLGCODE	Returns values indicating the type of input processed by the default button procedure, as shown in the following table.

Button style	Returns
BS_AUTOCHECKBOX	DLGC_WANTCHARS | DLGC_BUTTON
BS_AUTORADIOBUTTON	DLGC_RADIOBUTTON
BS_CHECKBOX	DLGC_WANTCHARS | DLGC_BUTTON
BS_DEFPUSHBUTTON	DLGC_DEFPUSHBUTTON
BS_GROUPBOX	DLGC_STATIC
BS_PUSHBUTTON	DLGC_UNDEFPUSHBUTTON
BS_RADIOBUTTON	DLGC_RADIOBUTTON

Message	Default action
WM_GETFONT	Returns a handle to the current font.
WM_KEYDOWN	Pushes the button if the user presses the SPACEBAR.
WM_KEYUP	Releases the mouse capture for all cases except the TAB key.
WM_KILLFOCUS	Removes the focus rectangle from a button. For push buttons and default push buttons, the focus rectangle is invalidated. If the button has the mouse capture, the capture is released, the button is not clicked, and any push state is removed.
WM_LBUTTONDBLCLK	Sends a **BN_DBLCLK** notification message to the parent window for radio buttons and owner-drawn buttons. For other buttons, a double-click is processed as a **WM_LBUTTONDOWN** message.
WM_LBUTTONDOWN	Highlights the button if the position of the mouse cursor is within the button's client rectangle.
WM_LBUTTONUP	Releases the mouse capture if the button had the mouse capture.
WM_MOUSEMOVE	Performs the same action as **WM_LBUTTONDOWN**, if the button has the mouse capture. Otherwise, no action is performed.
WM_NCCREATE	Turns any BS_OWNERDRAW button into a BS_PUSHBUTTON button.
WM_NCHITTEST	Returns HTTRANSPARENT, if the button control is a group box.
WM_PAINT	Draws the button according to its style and current state.
WM_SETFOCUS	Draws a focus rectangle on the button getting the focus. For radio buttons and automatic radio buttons, the parent window is sent a **BN_CLICKED** notification message.
WM_SETFONT	Sets a new font and optionally updates the window.
WM_SETTEXT	Sets the text of the button. In the case of a group box, the message paints over the preexisting text before repainting the group box with the new text.
WM_SYSKEYUP	Releases the mouse capture for all cases except the TAB key.

The predefined window procedure passes all other messages to the *DefWindowProc* function for default processing.

21.6 List Box

List box items can be represented by text strings, bitmaps, or both. If the list box is not large enough to display all the list box items at once, the list box provides a scroll bar. The user scrolls through the list box items, and applies or removes selection status as necessary. Selection style of a list box item or its visual appearance can be changed in

Operating system metrics. When the user selects or deselects an item, the system sends a notification message to the parent window of the list box.

A dialog box procedure is responsible for initializing and monitoring its child windows, including any list boxes. The dialog box procedure communicates with the list box by sending messages to it and by processing the notification messages sent by the list box.

21.6.1 List Box types and styles

There are two types of list boxes: single-selection (the default) and multiple-selection. In a *single-selection list box*, the user can select only one item at a time. In a *multiple-selection list box*, the user can select more than one item at a time. To create a multiple-selection list box, specify the LBS_MULTIPLESEL or the LBS_EXTENDEDSEL style.

There are many list box styles and window styles that control the appearance and operation of a list box. These styles indicate whether list box items are sorted, arranged in multiple columns, drawn by the application, and so on. The dimensions and styles of a list box are typically defined in a dialog box template included in an application's resources.

To create a list box by using the *CreateWindow* or *CreateWindowEx* function, use the LISTBOX class, appropriate window style constants, and the following style constants to define the list box. After the control has been created, these styles cannot be modified, except as noted.

LBS_DISABLENOSCROLL:
Shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If you do not specify this style, the scroll bar is hidden when the list box does not contain enough items.

LBS_EXTENDEDSEL:
This style allows multiple items to be selected by using the SHIFT key and the mouse or special key combinations.

LBS_HASSTRINGS:
This style specifies that a list box contains items consisting of strings. The list box maintains the memory and addresses for the strings so that the application can use the LB_GETTEXT message to retrieve the text for a particular item. By default, all list boxes except owner-drawn list boxes have this style. You can create an owner-drawn list box either with or without this style.

LBS_MULTICOLUMN:
This style specifies a multi column list box that is scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.

LBS_MULTIPLESEL:
Turns string selection on or off each time the user clicks or double-clicks a string in the list box. The user can select any number of strings.

LBS_NODATA:

This style specifies a no-data list box. Specify this style when the count of items in the list box will exceed one thousand. A no-data list box must also have the LBS_OWNERDRAWFIXED style, but must not have the LBS_SORT or LBS_HASSTRINGS style.

A no-data list box resembles an owner-drawn list box except that it contains no string or bitmap data for an item. Commands to add, insert, or delete an item always ignore any specified item data; requests to find a string within the list box always fail. The system sends the WM_DRAWITEM message to the owner window when an item must be drawn. The **itemID** member of the DRAWITEMSTRUCT structure passed with the **WM_DRAWITEM** message specifies the line number of the item to be drawn. A no-data list box does not send a WM_DELETEITEM message.

LBS_NOINTEGRALHEIGHT:

This style specifies that the size of the list box is exactly the size specified by the application when it created the list box. Normally, the system sizes a list box so that the list box does not display partial items.

LBS_NOREDRAW:

This style specifies that the list box's appearance is not updated when changes are made.To change the redraw state of the control, use the WM_SETREDRAW message.

LBS_NOSEL:

This style specifies that the list box contains items that can be viewed but not selected.

LBS_NOTIFY:

This style notifies the parent window with an input message whenever the user clicks or double-clicks a string in the list box.

LBS_OWNERDRAWFIXED:

This style specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are the same height. The owner window receives a WM_MEASUREITEM message when the list box is created and a **WM_DRAWITEM** message when a visual aspect of the list box has changed.

LBS_OWNERDRAWVARIABLE:

Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a **WM_MEASUREITEM** message for each item in the combo box when the combo box is created and a **WM_DRAWITEM** message when a visual aspect of the combo box has changed.

LBS_SORT

Sorts strings in the list box alphabetically.

LBS_STANDARD

Sorts strings in the list box alphabetically. The parent window receives an input message whenever the user clicks or double-clicks a string. The list box has borders on all sides.

LBS_USETABSTOPS

Enables a list box to recognize and expand tab characters when drawing its strings. You can use the LB_SETTABSTOPS message to specify tab stop positions. The default tab positions are 32 dialog template units apart. Dialog template units are the device-independent units used in dialog box templates. To convert measurements from dialog template units to screen units (pixels), use the MapDialogRect function.

LBS_WANTKEYBOARDINPUT

This style specifies that the owner of the list box receives WM_VKEYTOITEM messages whenever the user presses a key and the list box has the input focus. This enables an application to perform special processing on the keyboard input.

Note: the description of the controls including list box has been taken from Microsoft Help Desk.

21.6.2 Notification Messages from List Boxes

When an event occurs in a list box, the list box sends a notification message to the dialog box procedure of the owner window. List box notification messages are sent when a user selects, double-clicks, or cancels a list box item; when the list box receives or loses the keyboard focus; and when the system cannot allocate enough memory for a list box request. A notification message is sent as a WM_COMMAND message in which the low-order word of the *wParam* parameter contains the list box identifier, the high-order word of *wParam* contains the notification message, and the *lParam* parameter contains the control window handle.

A dialog box procedure is not required to process these messages; the default window procedure processes them.

An application should monitor and process the following list box notification messages.

Notification message	Description
LBN_DBLCLK	The user double-clicks an item in the list box.
LBN_ERRSPACE	The list box cannot allocate enough memory to fulfill a request.
LBN_KILLFOCUS	The list box loses the keyboard focus.
LBN_SELCANCEL	The user cancels the selection of an item in the list box.
LBN_SELCHANGE	The selection in a list box is about to change.
LBN_SETFOCUS	The list box receives the keyboard focus.

21.6.3 Messages to List Boxes

A dialog box procedure can send messages to a list box to add, delete, examine, and change list box items. For example, a dialog box procedure could send an LB_ADDSTRING message to a list box to add an item, and an LB_GETSEL message to determine whether the item is selected. Other messages set and retrieve information about

the size, appearance, and behavior of the list box. For example, the LB_SETHORIZONTALEXTENT message sets the scrollable width of a list box. A dialog box procedure can send any message to a list box by using the SendMessage or SendDlgItemMessage function.

A list box item is often referenced by its *index*, an integer that represents the item's position in the list box. The index of the first item in a list box is zero; the index of the second item is one, and so on.

The following table describes how the predefined list box procedure responds to list box messages.

Message	Response
LB_ADDFILE	Inserts a file into a directory list box filled by the **DlgDirList** function and retrieves the list box index of the inserted item.
LB_ADDSTRING	Adds a string to a list box and returns its index.
LB_DELETESTRING	Removes a string from a list box and returns the number of strings remaining in the list.
LB_DIR	Adds a list of filenames to a list box and returns the index of the last filename added.
LB_FINDSTRING	Returns the index of the first string in the list box that begins with a specified string..
LB_FINDSTRINGEXACT	Returns the index of the string in the list box that is equal to a specified string.
LB_GETANCHORINDEX	Returns the index of the item that the mouse last selected.
LB_GETCARETINDEX	Returns the index of the item that has the focus rectangle.
LB_GETCOUNT	Returns the number of items in the list box.
LB_GETCURSEL	Returns the index of the currently selected item.
LB_GETHORIZONTALEXTENT	Returns the scrollable width, in pixels, of a list box.
LB_GETITEMDATA	Returns the value associated with the specified item.
LB_GETITEMHEIGHT	Returns the height, in pixels, of an item in a list box.
LB_GETITEMRECT	Retrieves the client coordinates of the specified list box item.
LB_GETLOCALE	Retrieves the locale of the list box. The high-order word contains the country/region code and the low-order word contains the language identifier.
LB_GETSEL	Returns the selection state of a list box item.
LB_GETSELCOUNT	Returns the number of selected items in a multiple-selection list box.

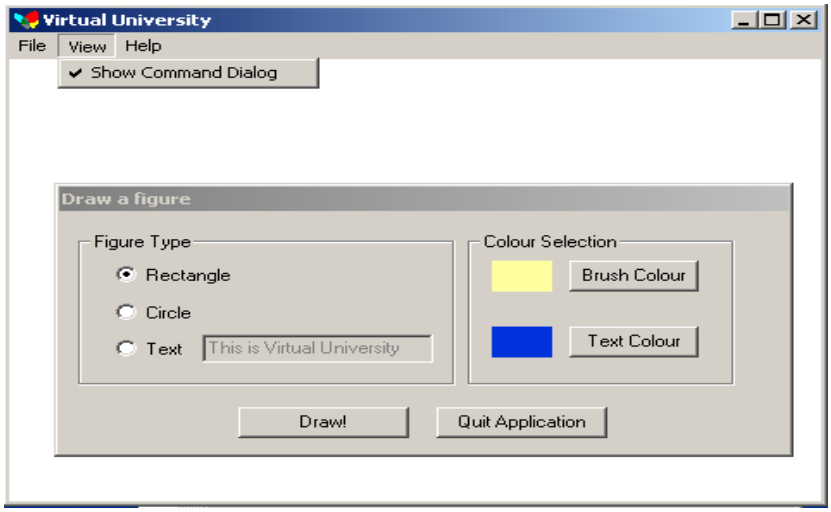
LB_GETSELITEMS	Creates an array of the indexes of all selected items in a multiple-selection list box and returns the total number of selected items.
LB_GETTEXT	Retrieves the string associated with a specified item and the length of the string.
LB_GETTEXTLEN	Returns the length, in characters, of the string associated with a specified item.
LB_GETTOPINDEX	Returns the index of the first visible item in a list box.
LB_INITSTORAGE	Allocates memory for the specified number of items and their associated strings.
LB_INSERTSTRING	Inserts a string at a specified index in a list box.
LB_ITEMFROMPOINT	Retrieves the zero-based index of the item nearest the specified point in a list box.
LB_RESETCONTENT	Removes all items from a list box.
LB_SELECTSTRING	Selects the first string it finds that matches a specified prefix.
LB_SELITEMRANGE	Selects a specified range of items in a list box.
LB_SELITEMRANGEEX	Selects a specified range of items if the index of the first item in the range is less than the index of the last item in the range. Cancels the selection in the range if the index of the first item is greater than the last.
LB_SETANCHORINDEX	Sets the item that the mouse last selected to a specified item.
LB_SETCARETINDEX	Sets the focus rectangle to a specified list box item.
LB_SETCOLUMNWIDTH	Sets the width, in pixels, of all columns in a list box.
LB_SETCOUNT	Sets the number of items in a list box.
LB_SETCURSEL	Selects a specified list box item.
LB_SETHORIZONTALEXTENT	Sets the scrollable width, in pixels, of a list box.
LB_SETITEMDATA	Associates a value with a list box item.
LB_SETITEMHEIGHT	Sets the height, in pixels, of an item or items in a list box.
LB_SETLOCALE	Sets the locale of a list box and returns the previous locale identifier.
LB_SETSEL	Selects an item in a multiple-selection list box.
LB_SETTABSTOPS	Sets the tab stops to those specified in a specified array.
LB_SETTOPINDEX	Scrolls the list box so the specified item is at the top of the visible range.

The predefined list box procedure passes all other messages to *DefWindowProc* for default processing.

21.7 Example Application

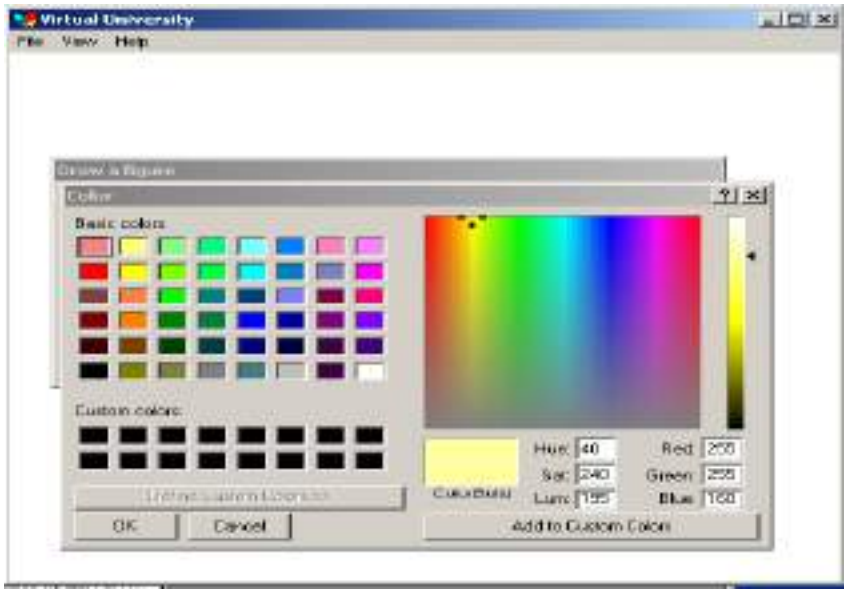
The following components will be used in our application

21.7.1 Modeless Dialogs



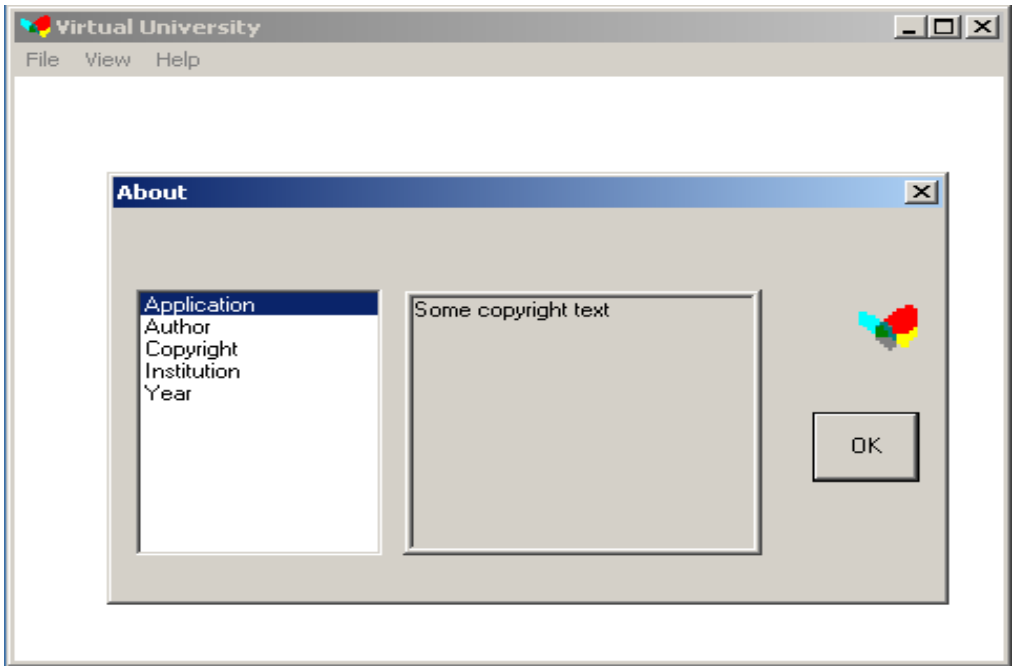
Dialog box is designed in resource edit provided by Visual Studio

21.7.2 Choose Color Dialogs



Choose color is built resource in windows.

21.7.3 About Dialogs



About dialog is designed in resource editor.

21.7.4 Creating Windows used in Application

```
hWndMain = CreateWindow(windowClassName,
windowName,
WS_OVERLAPPEDWINDOW | WS_VISIBLE,
CW_USEDEFAULT, 1, CW_USEDEFAULT, 1,
NULL, NULL, hInstance, NULL
);

if(!hWndMain)
{
        return 0;
}
```

21.7.5 Creating Dialogs

```
hCommandDialog = CreateDialog(hInstance,
MAKEINTRESOURCE(IDD_DIALOG_DRAW),
hWndMain, commandDialogProc
);
```

```
if(!hCommandDialog)
{
	return 0;
}

ShowWindow(hCommandDialog, SW_SHOWNORMAL);

commandDialogShown = TRUE;

CheckMenuItem(GetMenu(hWndMain), ID_VIEW_SHOWCOMMANDDIALOG,
MF_CHECKED | MF_BYCOMMAND);
```

21.7.6 Message Loop

```
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
	if(!IsDialogMessage(hCommandDialog, &msg))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
}
```

21.7.7 Menu Command

```
Case ID_VIEW_SHOWCOMMANDDIALOG:

	if(commandDialogShown)	// already visible?
	{
		ShowWindow(hCommandDialog, SW_HIDE);	// hide it

		CheckMenuItem(GetMenu(hWnd),
		ID_VIEW_SHOWCOMMANDDIALOG, MF_UNCHECKED |
		MF_BYCOMMAND); // uncheck
		commandDialogShown = FALSE;
	}
	else
	{
	}
```

21.7.8 Command Dialog Procedure

```
Static COLORREF textColour, brushColour;

case WM_INITDIALOG:
        CheckDlgButton(hDlg, IDC_RADIO_RECTANGLE, BST_CHECKED);  //
BM_SETCHECK message: check rectangle button

EnableWindow(GetDlgItem(hDlg, IDC_EDIT_TEXT), FALSE);   // disable edit control

SendDlgItemMessage(hDlg, IDC_EDIT_TEXT, EM_LIMITTEXT, TEXT_LIMIT, 0);
        // set text limit

SetWindowText(GetDlgItem(hDlg, IDC_EDIT_TEXT), "This is Virtual University");

brushColour = RGB_BRUSH_COLOR;  //RGB(255, 255, 160)
textColour = RGB_TEXT_COLOR;   //RGB(0, 50, 220)
return TRUE;  // system should set focus

wNotificationCode = HIWORD(wParam);
wID = LOWORD(wParam);
if(wNotificationCode == BN_CLICKED)
{
        switch(wID)
        {
        case IDC_RADIO_RECTANGLE:
                EnableWindow(GetDlgItem(hDlg, IDC_EDIT_TEXT), FALSE);

                // disable edit control similarly in IDC_RADIO_CIRCLE

        case IDC_RADIO_TEXT:
        EnableWindow(GetDlgItem(hDlg, IDC_EDIT_TEXT), TRUE);
        SendDlgItemMessage(hDlg, IDC_EDIT_TEXT, EM_SETSEL, 0, -1);

        SetFocus(GetDlgItem(hDlg, IDC_EDIT_TEXT));

        //Now handlingo of WM_CTLCOLORSTATIC

        case WM_CTLCOLORSTATIC:
        switch(GetDlgCtrlID((HWND)lParam))
        {
        case IDC_STATIC_TEXT_COLOR:
                if(hBrush)       // if some brush was created before
                        DeleteObject(hBrush);
        hBrush = CreateSolidBrush(textColour);       // create a brush
                        return (BOOL)hBrush;
```

```
                    break;

        case IDC_STATIC_BRUSH_COLOR:
                if(hBrush)      // if some brush was created before
                        DeleteObject(hBrush);
                hBrush = CreateSolidBrush(brushColour); // create a brush
                        return (BOOL)hBrush;
                        break;

                default:
                        return FALSE;// perform default message handling
```

21.7.9 Messages Used in Our Application

```
BM_SETCHECK:
wParam: check-state either BST_CHECKED or BST_UNCHECKED

EM_LIMITTEXT: wParam: text length
EM_SETSEL:
wParam: starting pos
lParam: ending pos.
0&-1:All selected, start:-1: current selection deselected
```

21.7.10 The WM_CTRLCOLORSTATIC Message

A static control, or an edit control that is read-only or disabled, sends the **WM_CTLCOLORSTATIC** message to its parent window when the control is about to be drawn. By responding to this message, the parent window can use the specified device context handle to set the text and background colors of the static control.

A window receives this message through its WindowProc function.

```
WM_CTLCOLORSTATIC

    WPARAM wParam
    LPARAM lParam;
```

wParam:
Handle to the device context for the static control window.
lParam:
Handle to the static control.

Return Value:

> If an application processes this message, the return value is a handle to a brush that the system uses to paint the background of the static control.

By default, the DefWindowProc function selects the default system colors for the static control.

Edit controls that are not read-only or disabled do not send the **WM_CTLCOLORSTATIC** message; instead, they send the WM_CTLCOLOREDIT message.

The system does not automatically destroy the returned brush. It is the application's responsibility to destroy the brush when it is no longer needed.

The **WM_CTLCOLORSTATIC** message is never sent between threads; it is sent only within the same thread.

If a dialog box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. If the dialog box procedure returns FALSE, then default message handling is performed. The DWL_MSGRESULT value set by the SetWindowLong function is ignored.

Summary

Windows controls are basic controls that are pre-registered in windows. We have discussed some of them like button, list box and edit box controls. These controls are helpful to display information in a very organized manner in a dialog box or in a window. Edit box control is simple to use. It has few message and notifications messages. Sending message to edit box window we can limit text in edit box, set text and get text etc. Button is another ubiquitous control in windows. Button is used almost in every user interactive application. Button is sent messages like edit box and list box, and also send notification messages to its parent window. List View is another useful control in windows systems. List view control list the items in its window. These items can be selected and clicked on each click list box send notification message to its parent window.

Exercises

1. Create a Medical Store data base form. This form should be a Modal/Modeless dialog box containing all the controls needed for the medical store keeper to enter data.
2. Create Owner draw list box, which has green selection rectangle and white text instead of blue (default) selection rectangle.

Chapter 22
Using Common Dialogs and Windows Controls

22.1 Dialogs (*Continue from the Previous Lecture*)

In this lecture, we will discuss more about the dialog boxes and their commands implementations.

22.2 Command Dialog Procedure

```
case WM_CTLCOLORSTATIC:
        switch(GetDlgCtrlID((HWND)lParam))
        {
        case IDC_STATIC_TEXT_COLOR:
                if(hBrush)      // if some brush was created before
                        DeleteObject(hBrush);
        hBrush = CreateSolidBrush(textColour);      // create a brush
                        return (BOOL)hBrush;
                        break;

        case IDC_STATIC_BRUSH_COLOR:
                if(hBrush)      // if some brush was created before
                        DeleteObject(hBrush);
                hBrush = CreateSolidBrush(brushColour); // create a brush
                        return (BOOL)hBrush;
                        break;

                default:
                        return FALSE;// perform default message handling
```

22.3 Choose Color Dialog

```
ChooseColor(&chooseclr);

typedef struct {
  DWORD       lStructSize;
  HWND        hwndOwner;
  HWND        hInstance;
  COLORREF    rgbResult;
  COLORREF   * lpCustColors;
  DWORD       Flags; CC_RGBINIT | CC_FULLOPEN | CC_ANYCOLOR
  LPARAM      lCustData;
  LPCCHOOKPROC lpfnHook;
  LPCTSTR     lpTemplateName;
} CHOOSECOLOR, *LPCHOOSECOLOR;   return Val???
```

```
case WM_CTLCOLORSTATIC:
        switch(GetDlgCtrlID((HWND)lParam))
        {
```

```
case IDC_BUTTON_BRUSH_COLOR:

if(ShowChooseColorDialog(hDlg, brushColour, &brushColour))
{
        GetClientRect(GetDlgItem(hDlg, IDC_STATIC_BRUSH_COLOR),
        &rect);

        InvalidateRect(GetDlgItem(hDlg, IDC_STATIC_BRUSH_COLOR),
        &rect, TRUE);
}
Break;
```

22.4 Our own defined function ShowChooseColorDialog

```
BOOL ShowChooseColorDialog(HWND Owner, COLORREF initClr, LPCOLORREF
chosenClr)
{
CHOOSECOLOR cc;
        static COLORREF customColors[16];

        memset(&cc, 0, sizeof(cc));

        cc.lStructSize = sizeof(CHOOSECOLOR);
        cc.hwndOwner = hwndOwner;
        cc.rgbResult = initialColor;
        cc.lpCustColors = customColors;
        cc.Flags = CC_RGBINIT | CC_FULLOPEN | CC_ANYCOLOR;

        if(ChooseColor(&cc))// OK pressed
        {
                *chosenColor = cc.rgbResult;
                return TRUE;
        }
        return FALSE;
}
```

Continue from Command Dialog Procedure:

```
case IDC_BUTTON_BRUSH_COLOR:
if(ShowChooseColorDialog(hDlg, brushColour, &brushColour))
{
// REPAINT CONTROL:  send WM_CTLCOLORSTATIC druing repainting

        GetClientRect(GetDlgItem(hDlg, IDC_STATIC_BRUSH_COLOR), &rect);
```

```
        InvalidateRect(GetDlgItem(hDlg, IDC_STATIC_BRUSH_COLOR), &rect,
TRUE);
} Break;
```

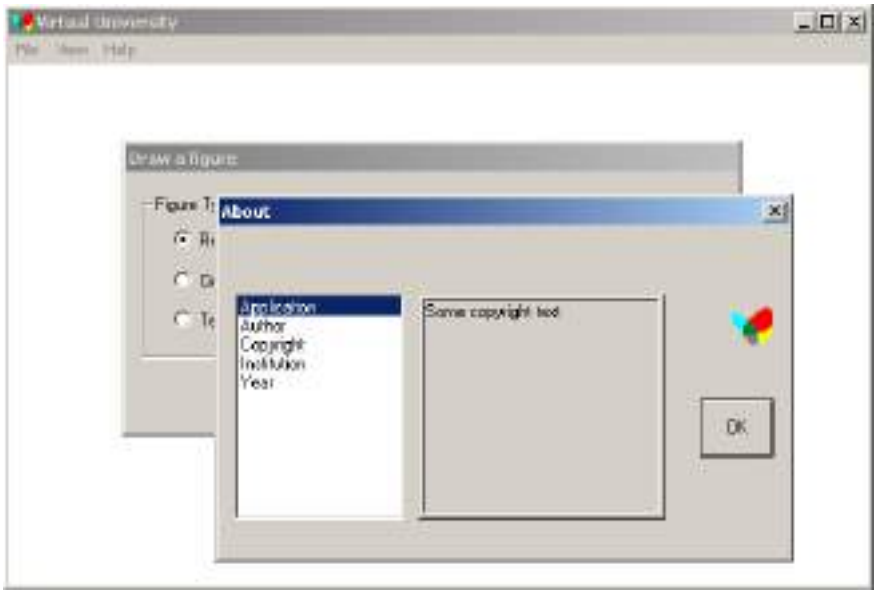
22.5 Command Dialog Procedure (*Drawing*)

```
case IDC_BUTTON_DRAW:
        hDC = GetDC(GetParent(hDlg));
        if(IsDlgButtonChecked(hDlg, IDC_RADIO_RECTANGLE) ==
BST_CHECKED)
        {
        hOwnerBrush = CreateHatchBrush(HS_BDIAGONAL, brushColour);

        hOldBrush = SelectObject(hDC, hOwnerBrush);
        Rectangle(hDC, 10, 10, 200, 200);

        SelectObject(hDC, hOldBrush); // restore old selection
        DeleteObject(hOwnerBrush);
}
```

22.6 The About Box (*Main Window Procedure)*

Now create a Modal Dialog box on the about event.

```
case ID_HELP_ABOUT:
        DialogBox(hAppInstance, MAKEINTRESOURCE(IDD_DIALOG_ABOUT),
hWnd, aboutDialogProc);
```



22.7 About Box Dialog Procedure

```
LPTSTR strings[5][2] = {{"Application", "Lecture 22"},
{"Author", "M. Shahid Sarfraz"},
{"Institution", "Virtual University"},
{"Year", "2003"},
{"Copyright", "2003 Virtual University"}};

case WM_INITDIALOG:
for(i=0; i<5; ++i)
{
        index = SendDlgItemMessage(hDlg,IDC_LIST_ABOUT, LB_ADDSTRING, 0,
        (LPARAM)strings[i][0]);
        SendDlgItemMessage(hDlg, IDC_LIST_ABOUT, LB_SE
        TITEMDATA,index,(LPARAM)strings[i][1])
}

// set current selection to 0
SendDlgItemMessage(hDlg, IDC_LIST_ABOUT, LB_SETCURSEL, 0, 0);

//Check notification messaqges in about dialog box
LPTSTR str;
case WM_COMMAND:
wNotificationCode = HIWORD(wParam);
wID = LOWORD(wParam);

switch(wID)
{
case IDC_LIST_ABOUT:
        if(wNotificationCode == LBN_SELCHANGE)
        {
        index = SendDlgItemMessage(hDlg, wID, LB_GETCURSEL, 0, 0);
        SetDlgItemText(hDlg, IDC_STATIC_ABOUT, strings[0][1]);
        str = (LPTSTR)SendDlgItemMessage(hDlg, IDC_LIST_ABOUT,
        LB_GETITEMDATA, index, 0);
        SetDlgItemText(hDlg, IDC_STATIC_ABOUT, str);
        }
}
```

Summary

We have been studying dialogs from previous two lectures. In this lecture, we have implemented some of the command implementation of dialog boxes. Common dialogs are very much useful in windows. Using common dialogs, you can show user to choose colors, files and printer, etc. Dialog resources are easy to use and easier to handle. Controls can be displayed on the dialogs. Dialogs by default set the font and dimensions of the controls. Dialogs are used in many areas like configuration of hardware devices,

internet connections, properties and database configurations. Another important dialogs are called property sheets. This property sheet enables you to select any category from the tabs.

Exercises

1. Create a Medical Store data base form. This form should be a Modal/Modeless dialog box containing all the controls needed for the medical store keeper to enter data. This form should handle all the controls notification messages. Save the data, entered in a dialog box controls, in a file.
2. Create Owner draw combo box, which has green selection rectangle and white text instead of blue (default) selection rectangle.

Chapter 23

Common Controls

23.1 Overview of Windows Common Controls

A control is a child window an application uses in conjunction with another window to perform simple input and output (I/O) tasks. Controls are most often used within dialog boxes, but they can also be used in other windows. Controls within dialog boxes provide the user with the means to type text, choose options, and direct a dialog box to complete its action. Controls in other windows provide a variety of services, such as letting the user choose commands, view status, and view and edit text. The user control overviews discuss how to use these controls.

The following table lists the Windows controls.

Control	Description
Animation	An animation control is a window that displays an Audio-Video Interleaved (AVI) clip.
Button	Button controls typically notify the parent window when the user chooses the control.
Combo Box	Combo box controls are a combination of list boxes and edit controls, letting the user choose and edit items.
ComboBoxEx	**ComboBoxEx Controls** are an extension of the combo box control that provides native support for item images.
Date and Time Picker	A date and time picker (DTP) control provides a simple and intuitive interface through which to exchange date and time information with a user.
Drag List Box	**Drag List Boxes** are a special type of list box that enables the user to drag items from one position to another.
Edit	Edit controls let the user view and edit text.
Flat Scroll Bar	Flat scroll bars behave just like standard scroll bars except that you can customize their appearance to a greater extent than standard scroll bars.
Header	A header control is a window that is usually positioned above columns of text or numbers. It contains a title for each column, and it can be divided into parts.
Hot Key	A hot key control is a window that enables the user to enter a combination of keystrokes to be used as a hot key.
Image Lists	An image list is a collection of images of the same size, each of which can be referred to by its index.
IP Address Controls	An Internet Protocol (IP) address control allows the user to enter an IP address in an easily understood format.
List Box	List box controls display a list from which the user can select one or more items.
List-View	A list-view control is a window that displays a collection of items. The control provides several ways to arrange and display the items.
Month Calendar	A month calendar control implements a calendar-like user interface.

Pager	A pager control is a window container that is used with a window that does not have enough display area to show all of its content.
Progress Bar	A progress bar is a window that an application can use to indicate the progress of a lengthy operation.
Property Sheets	A property sheet is a window that allows the user to view and edit the properties of an item.
ReBar	Rebar controls act as containers for child windows. An application assigns child windows, which are often other controls, to a rebar control band.
Rich Edit	Rich Edit controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects.
Scroll Bars	Scroll bars let the user choose the direction and distance to scroll information in a related window.
Static	Static controls often act as labels for other controls.
Status Bars	A status bar is a horizontal window at the bottom of a parent window in which an application can display various kinds of status information.
SysLink	A SysLink control provides a convenient way to embed hypertext links in a window.
Tab	A tab control is analogous to the dividers in a notebook or the labels in a file cabinet. By using a tab control, an application can define multiple pages for the same area of a window or dialog box.
Toolbar	A toolbar is a control window that contains one or more buttons. Each button, when clicked by a user, sends a command message to the parent window.
ToolTip	ToolTips are hidden most of the time. They appear automatically, or pop up, when the user pauses the mouse pointer over a tool.
Trackbar	A trackbar is a window that contains a slider and optional tick marks. When the user moves the slider, using either the mouse or the direction keys, the trackbar sends notification messages to indicate the change.
Tree-View	A tree-view control is a window that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk.
Up-Down	An up-down control is a pair of arrow buttons that the user can click to increment or decrement a value, such as a scroll position or a number displayed in a companion control.

23.2 Common control Library

Most common controls belong to a window class defined in the common control DLL. The window class and the corresponding window procedure define the properties, appearance, and behavior of the control. To ensure that the common control DLL is loaded, include the *InitCommonControlsEx* function in your application. You create a common control by specifying the name of the window class when calling the *CreateWindowEx* function or by specifying the appropriate class name in a dialog box template.

DLL Versions

All 32-bit versions of Windows include common controls DLL, Comctl32.dll. However, this DLL has been updated several times since it was first introduced. Each successive version supports the features and application programming interface (API) of earlier versions. However, each new version also contains a number of new features and a correspondingly larger API. Applications must be aware of which version of Comctl32.dll is installed on a system, and only use the features and API that the DLL supports.

Because new versions of the common controls were distributed with Microsoft Internet Explorer, the version of Commctl32.dll that is present is commonly different from the version that was shipped with the operating system. It may actually be several versions more recent. It is thus not enough for your application to know which operating system it is running on. It must directly determine which version of Comctl32.dll is present.

23.3 Common control Styles

CCS_ADJUSTABLE:
This style enables a toolbar's built-in customization features, which enable the user to drag a button to a new position or to remove a button by dragging it off the toolbar. In addition, the user can double-click the toolbar to display the **Customize Toolbar** dialog box, which enables the user to add, delete, and rearrange toolbar buttons.

CCS_BOTTOM:
Causes the control to position itself at the bottom of the parent window's client area and sets the width to be the same as the parent window's width. Status windows have this style by default.

CCS_LEFT:
This style causes the control to be displayed vertically on the left side of the parent window.

CCS_NODIVIDER:
This style prevents a two-pixel highlight from being drawn at the top of the control.

CCS_NOMOVEX:
This style causes the control to resize and move itself vertically, but not horizontally, in response to a WM_SIZE message. If CCS_NORESIZE is used, this style does not apply.

CCS_NOMOVEY:
This style causes the control to resize and move itself horizontally, but not vertically, in response to a **WM_SIZE** message. If CCS_NORESIZE is used, this style does not apply. Header windows have this style by default.

CCS_NOPARENTALIGN:
This style prevents the control from automatically moving to the top or bottom of the parent window. Instead, the control keeps its position within the parent window despite changes to the size of the parent. If CCS_TOP or CCS_BOTTOM is also used, the height is adjusted to the default, but the position and width remain unchanged.

CCS_NORESIZE:
This style prevents the control from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height specified in the request for creation or sizing.

CCS_RIGHT:
This style causes the control to be displayed vertically on the right side of the parent window.

CCS_TOP:
This style causes the control to position itself at the top of the parent window's client area and sets the width to be the same as the parent window's width. Toolbars have this style by default.

CCS_VERT:
This style causes the control to be displayed vertically.

23.4 Initialize Common Controls

For initialization common controls there are two functions available:

- InitCommonControls()
- InitCommonControlsEx()

23.4.1 InitCommonControls Function

Registers and initializes the common control window classes.

According to the Microsoft documentation this little function is obsolete. New applications should use the InitCommonControlsEx function. So you should not use this function.

```
void InitCommonControls(VOID);
```

This little function does not return anything.

23.4.2 InitCommonControlsEx Function

Registers specific common control classes from the common control dynamic-link library (DLL).

```
BOOL InitCommonControlsEx(
    LPINITCOMMONCONTROLSEX lpInitCtrls
);
```

lpInitCtrls: Pointer to an INITCOMMONCONTROLSEX structure that contains information specifying which control classes will be registered.

Return Value
Returns TRUE if successful, or FALSE otherwise.

The effect of each call to **InitCommonControlsEx** is cumulative. For example, if **InitCommonControlsEx** is called with the ICC_UPDOWN_CLASS flag, then is later called with the ICC_HOTKEY_CLASS flag, the result is that both the up-down and hot key common control classes are registered and available to the application.

23.4.2.1 INITCOMMONCONTROLSEX Structure

This structure carries information used to load common control classes from the dynamic-link library (DLL). This structure is used with the InitCommonControlsEx function.

```
typedef struct tagINITCOMMONCONTROLSEX {
    DWORD dwSize;
    DWORD dwICC;
} INITCOMMONCONTROLSEX, *LPINITCOMMONCONTROLSEX;
```

dwSize:

Size of the structure, in bytes.

dwICC:

Set of bit flags that indicate which common control classes will be loaded from the DLL. This value can be a combination of the following:

ICC_ANIMATE_CLASS: Load animate control class.

ICC_BAR_CLASSES: Load toolbar, status bar, trackbar, and ToolTip control classes.

ICC_COOL_CLASSES: Load rebar control class.

ICC_DATE_CLASSES: Load date and time picker control class.

ICC_HOTKEY_CLASS: Load hot key control class.

ICC_INTERNET_CLASSES: Load IP address class.

ICC_LINK_CLASS: Load a hyperlink control class.

ICC_LISTVIEW_CLASSES: Load list-view and header control classes.

ICC_NATIVEFNTCTL_CLASS: Load a native font control class.

ICC_PAGESCROLLER_CLASS: Load pager control class.

ICC_PROGRESS_CLASS: Load progress bar control class.

ICC_STANDARD_CLASSES: Load one of the intrinsic User32 control classes. The user controls include button, edit, static, listbox, combobox, and scrollbar.

ICC_TAB_CLASSES: Load tab and ToolTip control classes.

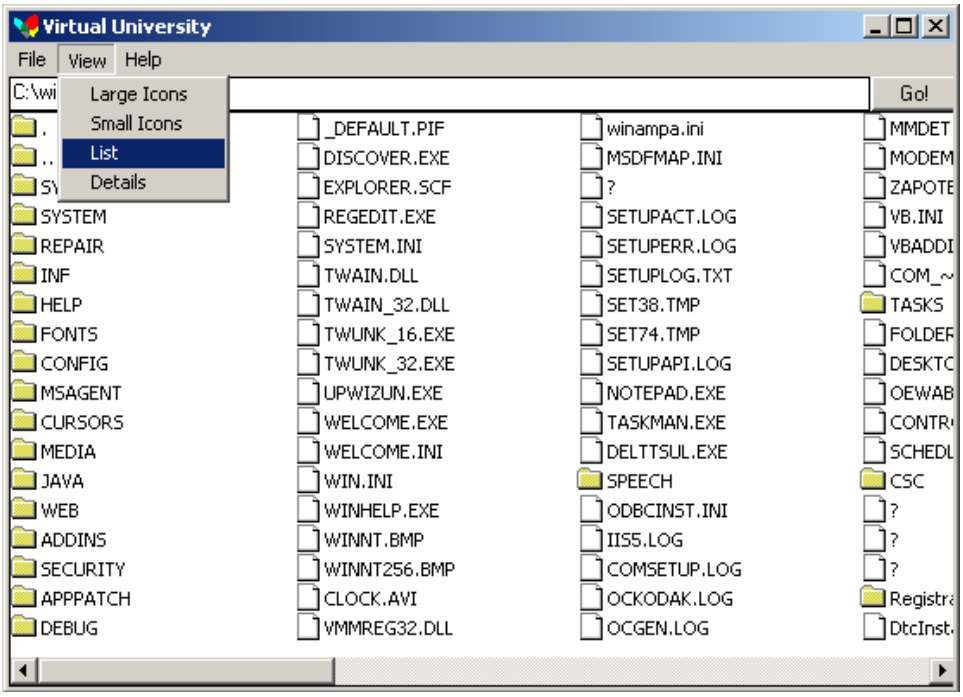
ICC_TREEVIEW_CLASSES: Load tree-view and ToolTip control classes.

ICC_UPDOWN_CLASS: Load up-down control class.

ICC_USEREX_CLASSES: Load ComboBoxEx class.

ICC_WIN95_CLASSES: Load animate control, header, hot key, list-view, progress bar, status bar, tab, ToolTip, toolbar, trackbar, tree-view, and up-down control classes.

23.5 List View



23.6 Today's Goal

Today we are going to create a List Box. This list box will be explorer style list box. In this list box you can see large, small, list, report styles.

23.7 Image List

An image list is a collection of images of the same size, each of which can be referred to by its index.

23.8 ImageList_Create Function

```
HIMAGELIST ImageList_Create(
    int cx,
    int cy,
    UINT flags,
    int cInitial,
    int cGrow
);
```

cx:
Width, in pixels, of each image.

Cy:

Height, in pixels, of each image.

Flags:

Set of bit flags that specify the type of image list to create. This parameter can be a combination of the following values, but it can include only one of the ILC_COLOR values.

ILC_COLOR:

Use the default behavior if none of the other ILC_COLOR* flags is specified. Typically, the default is ILC_COLOR4, but for older display drivers, the default is ILC_COLORDDB:

ILC_COLOR4:

Use a 4-bit (16-color) device-independent bitmap (DIB) section as the bitmap for the image list.

ILC_COLOR8:

Use an 8-bit DIB section. The colors used for the color table are the same colors as the halftone palette.

ILC_COLOR16:

Use a 16-bit (32/64k-color) DIB section.

ILC_COLOR24:

Use a 24-bit DIB section.

ILC_COLOR32:

Use a 32-bit DIB section.

ILC_COLORDDB:

Use a device-dependent bitmap.

ILC_MASK:

Use a mask. The image list contains two bitmaps, one of which is a monochrome bitmap used as a mask. If this value is not included, the image list contains only one bitmap.

ILC_MIRROR:

Microsoft® Windows® can be mirrored to display languages such as Hebrew or Arabic that read right-to-left. If the image list is created on a mirrored version of Windows, then the images in the lists are mirrored, that is, they are flipped so they display from right to left. Use this flag on a mirrored version of Windows to instruct the image list not to automatically mirror images.

ILC_PERITEMMIRROR

cInitial:

This member is number of images that the image list initially contains.

cGrow:

This member is a number of images by which the image list can grow when the system needs to make room for new images. This parameter represents the number of new images that the resized image list can contain.

23.9 ImageList_AddIcon Function

```
int ImageList_AddIcon(

    HIMAGELIST himl,
    HICON hicon
);
```

himl:
Handle to the image list. If this parameter identifies a masked image list, the macro copies both the image and mask bitmaps of the icon or cursor. If this parameter identifies a nonmasked image list, the macro copies only the image bitmap.

Hicon:
Handle to the icon or cursor that contains the bitmap and mask for the new image.

Return Value:
Returns the index of the new image if successful, or -1 otherwise.

Because the system does not save *hicon*, you can destroy it after the macro returns if the icon or cursor was created by the CreateIcon function. You do not need to destroy *hicon* if it was loaded by the LoadIcon function; the system automatically frees an icon resource when it is no longer needed.

23.10 ImageList_ReplaceIcon Function

```
int ImageList_ReplaceIcon(
    HIMAGELIST himl,
    int i,
    HICON hicon
);
```

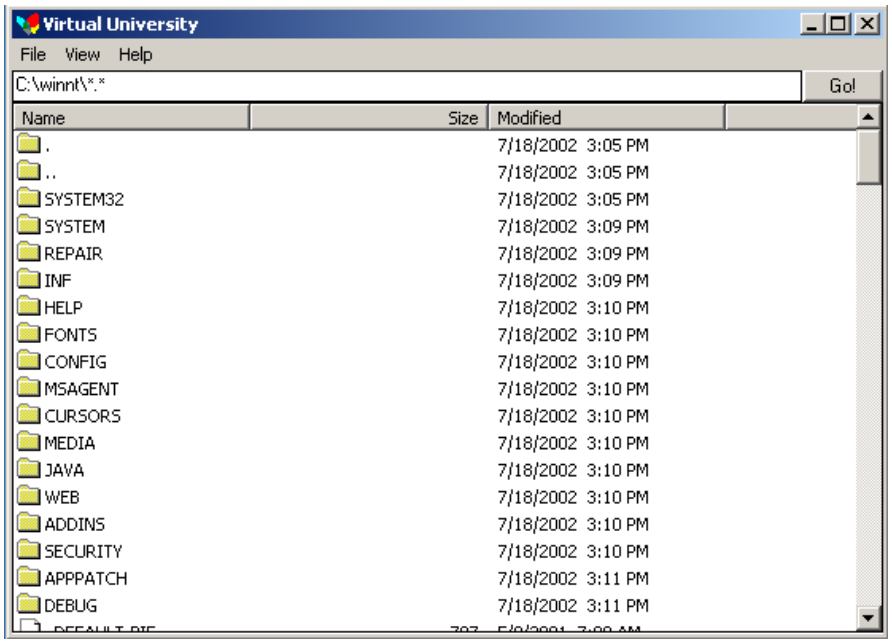
himl:
Handle to the image list.

i:
Index of the image to replace. If *i* is -1, the function appends the image to the end of the list.

Hicon:
Handle to the icon or cursor that contains the bitmap and mask for the new image.

Return Value:
Returns the index of the image if successful, or -1 otherwise.

23.11 Screen Shot of an Example Application



23.12 Creating List View Control

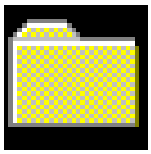
```
#define ID_LISTVIEW                 5

hWndListView = CreateWindow(WC_LISTVIEW,
 "Window Name",
 WS_TABSTOP | WS_CHILD | WS_BORDER | WS_VISIBLE | LVS_AUTOARRANGE |
          LVS_REPORT,
     10, 10, 350, 280, hWndMain, (HMENU)ID_LISTVIEW,
          hInstance, NULL
);

if(!hWndListView)
{
          return 0;
}
```

Creating Image List

```
hLarge = ImageList_Create(GetSystemMetrics(SM_CXICON),
GetSystemMetrics(SM_CYICON), ILC_MASK, 1, 1);
hSmall = ImageList_Create(GetSystemMetrics(SM_CXSMICON),
GetSystemMetrics(SM_CYSMICON), ILC_MASK, 1, 1);
```

```
hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON_FOLDER));
ImageList_AddIcon(hLarge, hIcon);
ImageList_AddIcon(hSmall, hIcon);
hIcon = LoadIcon(.. MAKEINTRESOURCE(IDI_ICON_FILE));
```

23.13 Windows Default Folder Icon



Folder.ico

23.14 Add Image List

```
ListView_SetImageList(hWndListView, hLarge, LVSIL_NORMAL);
   ListView_SetImageList(hWndListView, hSmall, LVSIL_SMALL);

HIMAGELIST ListView_SetImageList(
   HWND hwnd,
   HIMAGELIST himl,
   int iImageList  type of IL: LVSIL_NORMAL | LVSIL_SMALL | LVSIL_STATE
);
```

23.15 Add column to List View

```
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
        lvc.cx = COL_WIDTH;

        for(i=0; i<3; ++i)
        {
                lvc.iSubItem = i;
                lvc.fmt = alignments[i];
                lvc.pszText = columnHeadings[i];
                if(ListView_InsertColumn(hWndListView, i, &lvc) == -1)
                return 1;
        }
```

23.16 Add an Item

```
/* add an item with 3 subitems = 4 columns */

lvi.state = 0;              // no state: cut, focussed, selected etc.
lvi.stateMask = 0;      // no state specified: cut, focussed, selected etc.
lvi.lParam = (LPARAM)1234;          // item specific data

do
{
        lvi.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM | LVIF_STATE;
        lvi.iItem = itemNo++;           // which item it refers to
        lvi.iSubItem = 0;       // refers to an ITEM
        lvi.iImage = (findFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) ?
0 : 1; // proper image
        lvi.pszText = findFileData.cFileName;

        // add the item
        if(ListView_InsertItem(hWndListView, &lvi) == -1)
        return 0;
```

23.17 Add Sub Item for this Item

```
lvi.mask = LVIF_TEXT;
lvi.iSubItem = 1;
(findFileData.nFileSizeHigh * (MAXDWORD+1)) + findFileData.nFileSizeLow;
if(findFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
        wsprintf(buf, "");
else
        wsprintf(buf, "%10lu", findFileData.nFileSizeLow);

        lvi.pszText = buf;
        if(ListView_SetItem(hWndListView, &lvi) == -1)
                return 1;
```

23.18 Find First File

```
hFind= FindFirstFile(DEFAULT_PATH, &findFileData);
        if(hFind == INVALID_HANDLE_VALUE)
        {
                MessageBox(NULL, "Error calling FindFirstFile", "Error", MB_OK);
                return 0;
        }
```

23.19 Add Column to List View

```
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
	lvc.cx = COL_WIDTH;

	for(i=0; i<3; ++i)
	{
		lvc.iSubItem = i;
		lvc.fmt = alignments[i];
		lvc.pszText = columnHeadings[i];
		if(ListView_InsertColumn(hWndListView, i, &lvc) == -1)
			return 0;
	}
```

23.20 Last Modified Date of File

```
FileTimeToLocalFileTime(&findFileData.ftLastWriteTime, &fileTime);
	FileTimeToSystemTime(&fileTime, &systemTime);

	strcpy(strAMPM, systemTime.wHour>=12 ? "PM" : "AM");
	if(systemTime.wHour>=12)
		systemTime.wHour -= 12;
	if(!systemTime.wHour)
		systemTime.wHour = 12;

	wsprintf(buf, "%d/%d/%d %2d:%02d %s", systemTime.wMonth, systemTime.wDay,
systemTime.wYear, systemTime.wHour, systemTime.wMinute, strAMPM);
	lvi.iSubItem = 2;
	lvi.pszText = buf;
	if(ListView_SetItem(hWndListView, &lvi) == -1)
		return 1;
```

23.21 Modified List View control

```
VOID SetView(HWND hwndListView, DWORD dwStyle)
{
	DWORD dwCurrentStyle;

	dwCurrentStyle = GetWindowLong(hwndListView, GWL_STYLE);
	SetWindowLong(hwndListView, GWL_STYLE, (dwCurrentStyle &
~LVS_TYPEMASK) | dwStyle);
}
```

Summary

Common Controls are the part of Microsoft Windows Graphics Operating System. Almost all the WYSIWYG application use Common Controls for their compatibility and user friendliness with windows. In this lecture, we studied about common controls, their styles and behavior. We also created an application which best demonstrates the List View control of common controls. Common controls include controls like page controls, tree controls, list view controls that is modified from windows original control, button control that is also modified from windows original controls, data and time picker control, status bar, progress bar, rebar controls. These all controls reside in common controls library and the library has shipped with many versions. Before using the library you must check the valid version of the library because different version of library contains different controls properties.

Exercises

1. Create Tree control and show all the files hierarchy.

Chapter 24

Dynamic Link Libraries

24.1 What Is a Process

A running application that consists of a private virtual address space, code, data, and other operating-system resources, such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the context of the process.

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

24.2 Memory Management Basics

Each process on 32-bit Microsoft® Windows® has its own virtual address space that enables addressing up to 4 gigabytes of memory. All threads of a process can access its virtual address space. However, threads cannot access memory that belongs to another process which protects a process from being corrupted by another process.

24.2.1 Virtual Address Space

The virtual addresses used by a process do not represent the actual physical location of an object in memory. Instead, the system maintains a **page map** for each process, which is an internal data structure used to translate virtual addresses into corresponding physical addresses. Each time a thread references an address, the system translates the virtual address to a physical address.

The virtual address space is divided into partitions as follows: The 2 GB partition in low memory (0x00000000 through 0x7FFFFFFF) is available to the process, and the 2 GB partition in high memory (0x80000000 through 0xFFFFFFFF) is reserved for the system.

24.2.2 Virtual Address Space and Physical storage

The virtual address space of each process is much larger than the total physical memory available to all processes. To increase the size of physical storage, the system uses the disk for additional storage. The total amount of storage available to all executing processes is the sum of the physical memory and the free space on disk available to the *paging file*, a disk file used to increase the amount of physical storage. Physical storage and the virtual address space of each process are organized into *pages*, units of memory, whose size depends on the host computer. For example, on x86 computers the host page size is 4 kilobytes.

To maximize its flexibility in managing memory, the system can move pages of physical memory to and from a paging file on disk. When a page is moved in physical memory, the system updates the page maps of the affected processes. When the system needs space in physical memory, it moves the least recently used pages of physical memory to the paging file. Manipulation of physical memory by the system is completely transparent to applications which operate only in their virtual address spaces.

24.2.3 Page State

The pages of a process's virtual address space can be in one of the following states.

State	Description
Free	The page is neither committed nor reserved. The page is not accessible to the process. It is available to be committed, reserved, or simultaneously reserved and committed. Attempting to read from or write to a free page results in an access violation exception. A process can use the **VirtualFree** or **VirtualFreeEx** function to release reserved or committed pages of its address space, returning them to the free state.
Reserved	The page has been reserved for future use. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. It is available to be committed. A process can use the **VirtualAlloc** or **VirtualAllocEx** function to reserve pages of its address space and later to commit the reserved pages. It can use **VirtualFree** or **VirtualFreeEx** to decommit committed pages and return them to the reserved state.
Committed	Physical storage is allocated for the page, and access is controlled by a memory protection option. The system initializes and loads each committed page into physical memory only during the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. A process can use **VirtualAlloc** or **VirtualAllocEx** to allocate committed pages. These functions can commit pages that are already committed. The **GlobalAlloc** and **LocalAlloc** functions allocate committed pages with read/write access.

24.3 Memory Protection

Memory that belongs to a process is implicitly protected by its private virtual address space. In addition, Windows provides memory protection using the virtual memory hardware. The implementation of this protection varies with the processor. For example,

code pages in the address space of a process can be marked read-only and protected from modification by user-mode threads.

The following table lists the memory-protection options provided by Windows. You must specify one of the following values when allocating or protecting a page in memory.

Value	Meaning
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY	Gives copy-on-write protection to the committed region of pages. **Windows Me/98/95:** This flag is not supported.

The following are modifiers that can be used in addition to the options provided in the previous table, except as noted.

Value	Meaning
PAGE_GUARD	Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages thus act as a one-time access alarm.

When an access attempt leads the system to turn off guard page status, the underlying page protection takes over.

If a guard page exception occurs during a system service, the service typically returns a failure status indicator.

This value cannot be used with PAGE_NOACCESS.

PAGE_NOCACHE

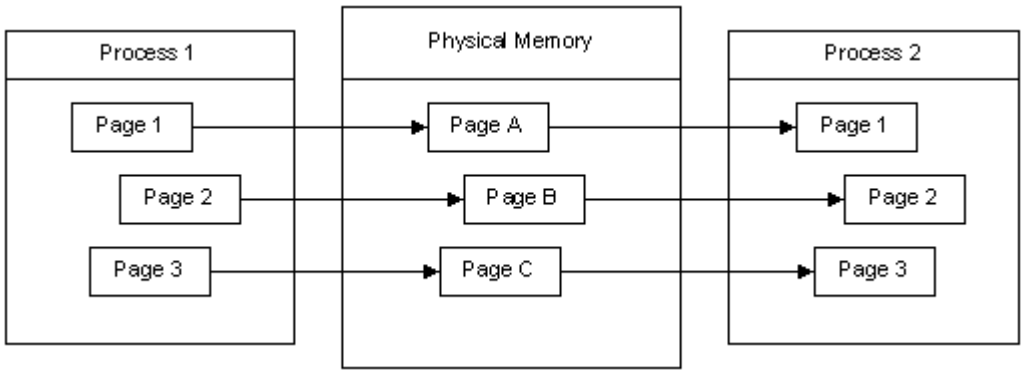
Does not allow caching of the committed regions of pages in the CPU cache. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching.

This value cannot be used with PAGE_NOACCESS.

Copy-on-Write Protection

Copy-on-write protection is an optimization that allows multiple processes to map their virtual address spaces such that they share a physical page until one of the processes modifies the page. This is part of a technique called *lazy evaluation*, which allows the system to conserve physical memory and time by not performing an operation until absolutely necessary.

For example, suppose two processes load pages from the same DLL into their virtual memory spaces. These virtual memory pages are mapped to the same physical memory pages for both processes. As long as neither of the processes writes to these pages, they can map to and share the same physical pages as shown in the following diagram.



If Process 1 writes to one of these pages, the contents of the physical page are copied to another physical page and the virtual memory map is updated for Process 1. Both processes now have their own instance of the page in physical memory. Therefore, it is

not possible for one process to write to a shared physical page and for the other process to see the changes.

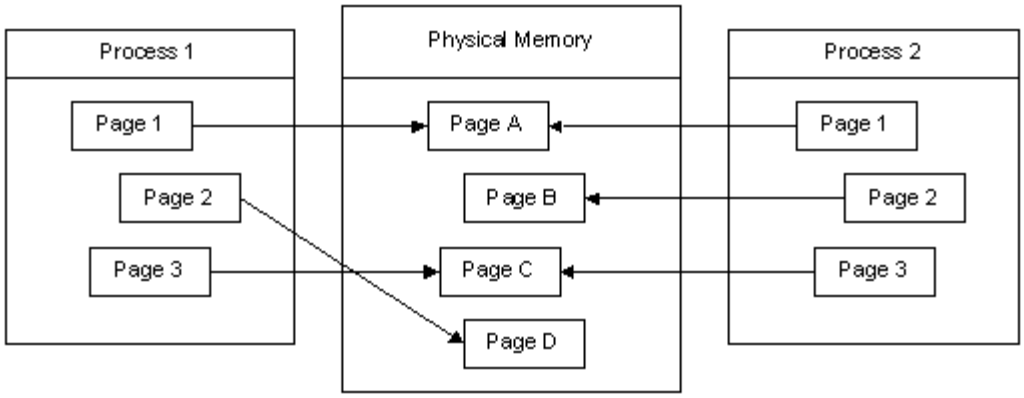


Figure 1

Loading Applications and DLLs

When multiple instances of the same Windows-based application are loaded, each instance is run in its own protected virtual address space. However, their instance handles (*hInstance*) typically have the same value. This value represents the base address of the application in its virtual address space. If each instance can be loaded into its default base address, it can map to and share the same physical pages with the other instances, using copy-on-write protection. The system allows these instances to share the same physical pages until one of them modifies a page. If for some reason one of these instances cannot be loaded in the desired base address, it receives its own physical pages.

DLLs are created with a default base address. Every process that uses a DLL will try to load the DLL within its own address space at the default virtual address for the DLL. If multiple applications can load a DLL at its default virtual address, they can share the same physical pages for the DLL. If for some reason a process cannot load the DLL at the default address, it loads the DLL elsewhere. Copy-on-write protection forces some of the DLL's pages to be copied into different physical pages for this process, because the fixups for jump instructions are written within the DLL's pages, and they will be different for this process. If the code section contains many references to the data section, this can cause the entire code section to be copied to new physical pages.

24.4 What is a Thread

A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

24.4.1 Multitasking

A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor *time slice* to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors. However, you must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

24.5 Linking the Compiled Code

What is compiled .OBJ code?

Compiled Object file contains the reference of the unresolved symbols.
Linker links the library and paste the actual code from the libraries to the executable code that is called *static Linking*;
Linking at run time is called Dynamic Linking.
Dynamic Link Libraries (DLLs) are linked dynamically.

Libraries are statically linked to the code and unresolved symbols. When a programs loads in memory and run then it would need dynamic link libraries that have the symbols and resolved addresses.

24.6 Dynamic Link Libraries

A *dynamic-link library* (DLL) is a module that contains functions and data that can be used by another module (application or DLL).

A DLL can define two kinds of functions: exported and internal. The exported functions are intended to be called by other modules, as well as from within the DLL where they are defined. Internal functions are typically intended to be called only from within the DLL where they are defined. Although a DLL can export data, its data is generally used only by its functions. However, there is nothing to prevent another module from reading or writing that address.

DLLs provide a way to modularize applications so that functionality can be updated and reused more easily. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

The Windows application programming interface (API) is implemented as a set of dynamic-link libraries, so any process that uses the Windows API uses dynamic linking.

Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

Types of Dynamic Linking

There are two methods for calling a function in a DLL:

- In *load-time dynamic linking*, a module makes explicit calls to exported DLL functions as if they were local functions. This requires you to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.
- In *run-time dynamic linking*, a module uses the **LoadLibrary** or **LoadLibraryEx** function to load the DLL at run time. After the DLL is loaded, the module calls the **GetProcAddress** function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by **GetProcAddress**.

DLLs and Memory Management

Every process that loads the DLL maps it into its virtual address space. After the process loads the DLL into its virtual address, it can call the exported DLL functions.

The system maintains a per-thread reference count for each DLL. When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero (run-time dynamic linking only), the DLL is unloaded from the virtual address space of the process.

Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:

- The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
- The DLL uses the stack of the calling thread and the virtual address space of the calling process.
- The DLL allocates memory from the virtual address space of the calling process.

24.7 DLL Entry Point

The **DllMain** function is an optional entry point into a dynamic-link library (DLL). If the function is used, it is called by the system when processes and threads are initialized and terminated, or upon calls to the **LoadLibrary** and **FreeLibrary** functions.

DllMain is a placeholder for the library-defined function name. You must specify the actual name you use when you build your DLL. For more information, see the documentation included with your development tools.

```
BOOL WINAPI DllMain(
  HINSTANCE hinstDLL,  /*Handle to the instance of the library*/
  DWORD fdwReason,     /*reason of loading and unloading
  LPVOID lpvReserved   /*future use or no use des. By Microsoft*/
);
```

hinstDLL: Handle to the DLL module. The value is the base address of the DLL. The **HINSTANCE** of a DLL is the same as the **HMODULE** of the DLL, so *hinstDLL* can be used in calls to functions that require a module handle.
fdwReason: Indicates why the DLL entry-point function is being called. This parameter can be one of the following values.

Value	Meaning
DLL_PROCESS_ATTACH	The DLL is being loaded into the virtual address space of the current process as a result of the process starting up or as a result of a call to **LoadLibrary**. DLLs can use this opportunity to initialize any instance data or to use the **TlsAlloc** function to allocate a thread local storage (TLS) index.
DLL_THREAD_ATTACH	The current process is creating a new thread. When this occurs, the system calls the entry-point function of all DLLs currently attached to the process. The call is made in the context of the new thread. DLLs can use this opportunity to initialize a TLS slot for the thread. A thread calling the DLL entry-point function with DLL_PROCESS_ATTACH does not call the DLL entry-point function with DLL_THREAD_ATTACH. Note that a DLL's entry-point function is called with this value only by threads created after the DLL is loaded by the process. When a DLL is loaded using **LoadLibrary**, existing threads do not call the entry-point function of the newly loaded DLL.
DLL_THREAD_DETACH	A thread is exiting cleanly. If the DLL has stored a pointer to allocated memory in a TLS slot, it should use this opportunity to free the memory. The system calls the entry-point function of all currently loaded DLLs

	with this value. The call is made in the context of the exiting thread.
DLL_PROCESS_DETACH	The DLL is being unloaded from the virtual address space of the calling process as a result of unsuccessfully loading the DLL, termination of the process, or a call to **FreeLibrary**. The DLL can use this opportunity to call the **TlsFree** function to free any TLS indices allocated by using **TlsAlloc** and to free any thread local data. Note that the thread that receives the DLL_PROCESS_DETACH notification is not necessarily the same thread that received the DLL_PROCESS_ATTACH notification.

lpvReserved:

If *fdwReason* is DLL_PROCESS_ATTACH, *lpvReserved* is NULL for dynamic loads and non-NULL for static loads.

If *fdwReason* is DLL_PROCESS_DETACH, *lpvReserved* is NULL if **DllMain** has been called by using **FreeLibrary** and non-NULL if **DllMain** has been called during process termination.

Return Values:

When the system calls the **DllMain** function with the DLL_PROCESS_ATTACH value, the function returns TRUE if it succeeds or FALSE if initialization fails. If the return value is FALSE when **DllMain** is called because the process uses the **LoadLibrary** function, **LoadLibrary** returns NULL. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.) If the return value is FALSE when **DllMain** is called during process initialization, the process terminates with an error. To get extended error information, call **GetLastError**.

When the system calls the **DllMain** function with any value other than DLL_PROCESS_ATTACH, the return value is ignored.

During initial process startup or after a call to **LoadLibrary**, the system scans the list of loaded DLLs for the process. For each DLL that has not already been called with the DLL_PROCESS_ATTACH value, the system calls the DLL's entry-point function. This call is made in the context of the thread that caused the process address space to change, such as the primary thread of the process or the thread that called **LoadLibrary**. Access to the entry point is serialized by the system on a process-wide basis.

There are cases in which the entry-point function is called for a terminating thread even if the entry-point function was never called with DLL_THREAD_ATTACH for the thread:

- The thread was the initial thread in the process, so the system called the entry-point function with the DLL_PROCESS_ATTACH value.
- The thread was already running when a call to the **LoadLibrary** function was made, so the system never called the entry-point function for it.

When a DLL is unloaded from a process as a result of an unsuccessful load of the DLL, termination of the process, or a call to **FreeLibrary**, the system does not call the DLL's entry-point function with the DLL_THREAD_DETACH value for the individual threads of the process. The DLL is only sent a DLL_PROCESS_DETACH notification. DLLs can take this opportunity to clean up all resources for all threads known to the DLL. However, if the DLL does not successfully complete a DLL_PROCESS_ATTACH notification, the DLL does not receive either a DLL_THREAD_DETACH or DLL_PROCESS_DETACH notification.

Warning The entry-point function should perform only simple initialization or termination tasks. It must not call the **LoadLibrary** or **LoadLibraryEx** function (or a function that calls these functions), because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, the entry-point function must not call the **FreeLibrary** function (or a function that calls **FreeLibrary**), because this can result in a DLL being used after the system has executed its termination code.

It is safe to call other functions in Kernel32.dll, because this DLL is guaranteed to be loaded in the process address space when the entry-point function is called. It is common for the entry-point function to create synchronization objects such as critical sections and mutexes, and use TLS. Do not call the registry functions, because they are located in Advapi32.dll. If you are dynamically linking with the C run-time library, do not call **malloc**; instead, call **HeapAlloc**.

Calling imported functions other than those located in Kernel32.dll may result in problems that are difficult to diagnose. For example, calling User, Shell, and COM functions can cause access violation errors, because some functions in their DLLs call **LoadLibrary** to load other system components. Conversely, calling those functions during termination can cause access violation errors because the corresponding component may already have been unloaded or uninitialized.

Because DLL notifications are serialized, entry-point functions should not attempt to communicate with other threads or processes. Deadlocks may occur as a result.

24.8 DLL Exports and DLL Imports

The export table How to export and import code (functions) in a DLLs
__declspec(dllimport) int i;
__declspec(dllexport) void function(void);

24.9 DLL Function and calling function from in it

LoadLibrary loads a library in process address space.

```
HMODULE LoadLibrary(
 LPCTSTR lpFileName //file name of module
);
```

FreeLibrary free the library that was loaded previously by LoadLibrary function.

```
FreeLibrary(hModule)
```

Now we call function from library using GetProcAddress. GetProcAddress returns the address of the function.

```
FARPROC GetProcAddress(
  HMODULE hModule,    // handle to DLL module
  LPCSTR lpProcName   // function name
);
```

Summary

Dynamic link libraries are the windows executables but these cannot be executed by writing its name on command line or double clicking on it. These libraries contain separate modules that load and run in any process address space. Thread is the execution unit in a Process. A process can have more than one thread. There are two types of dynamic linking load time dynamic linking and run time dynamic linking. In load time dynamic linking, a module makes explicit calls to exported DLL functions as if they were local functions and in run time dynamic linking Load library function is used to load the library and Get procedure address functions are called to get the address of the function from loaded library. DLL can export functions in the form definition files. In definition file we can provide ordinal as well. Ordinal is a number that is used to locate the function instead of function name.

Exercises

1. Create a dynamic link library and make a function which displays only message box.
2. Call the function from above library in your executable module the linking must be dynamic linking.

Chapter 25

Threads and DLL's

25.1 Import Libraries (.lib)

Import Library is statically linked to Executable module.

Example of Import libraries in windows are:

- Kernel32.lib
- User32.lib
- Gdi32.lib

Important System DLLs are

- Kernel32.dll
- User32.dll
- Gdi32.dll

25.2 Calling Conventions

Functions used in DLL's are normally use *__stdcall* calling convention. *__stdcall* calling convention is a standard calling convention used by the APIs in Windows. This calling convention cleans the stack after returning the called procedure automatically. No extra code is needed to clean out stack. *__stdcall* calling convention pushes the arguments in stack from right to left order.

25.3 Variable Scope in DLL

Variables defined in DLL have scope in memory until the DLL is loaded. After unloading, the variable scope is vanished. Locally defined variables are accessed within the DLL only. The variables that are set to export variables can be accessed outside the DLL if the DLL is statically linked.

Variables can be shared across multiple processes by making the separate data section as following.

```
#pragma data_seg( [ [ { push | pop }, ] [ identifier, ] ] [ "segment-
name" [, "segment-class" ] )
```

Specifies the data segment where initialized variables are stored in the .obj file. OBJ files can be viewed with the dumpbin application. The default segment in the .obj file for initialized variables is .data. Variables initialized to zero are considered uninitialized and are stored in .bss.

data_seg with no parameters resets the segment to .data.

push (optional)
Puts a record on the internal compiler stack. A **push** can have an *identifier* and *segment-name*.

pop (optional)
Removes a record from the top of the internal compiler stack.

identifier (optional)
When used with **push**, assigns a name to the record on the internal compiler stack. When used with **pop**, pops records off the internal stack until *identifier* is removed; if *identifier* is not found on the internal stack, nothing is popped.

identifier enables multiple records to be popped with a single **pop** command.

"segment-name" (optional)
The name of a segment. When used with **pop**, the stack is popped and *segment-name* becomes the active segment name.

Example

```
// pragma_directive_data_seg.cpp
int h = 1;                      // stored in .data
int i = 0;                      // stored in .bss
#pragma data_seg(".my_data1")
int j = 1;                      // stored in "my_data1"

#pragma data_seg(push, stack1, ".my_data2")
int l = 2;                      // stored in "my_data2"

#pragma data_seg(pop, stack1)   // pop stack1 off the stack
int m = 3;                      // stored in "stack_data1"

int main() {
}
```

Data allocated using **data_seg** does not retain any information about its location.

#pragma comment(linker, “/SECTION: seg_data1, RWS”)

`/SECTION:`*`name`*`,[E][R][W][S][D][K][L][P][X][,ALIGN=#]`

The /SECTION option changes the attributes of a section, overriding the attributes set when the .obj file for the section was compiled.

A section in a portable executable (PE) file is roughly equivalent to a segment or the resources in a new executable (NE) file. Sections contain either code or data. Unlike segments, sections are blocks of contiguous memory with no size constraints. Some sections contain code or data that your program declared and uses directly, while other data sections are created for you by the linker and library manager (lib.exe) and contain information vital to the operating system.

Do not use the following names, as they will conflict with standard names. For example, .sdata is used on RISC platforms:

- .arch
- .bss
- .data
- .edata
- .idata
- .pdata
- .rdata
- .reloc
- .rsrc
- .sbss
- .sdata
- .srdata
- .text
- .xdata

Specify one or more attributes for the section. The attribute characters, listed below, are not case sensitive. You must specify all attributes that you want the section to have; an omitted attribute character causes that attribute bit to be turned off. If you do not specify R, W, or E, the existing read, write, or executable status remains unchanged.

The meanings of the attribute characters are shown below.

Character	Attribute	Meaning
E	Execute	The section is executable
R	Read	Allows read operations on data
W	Write	Allows write operations on data
S	Shared	Shares the section among all processes that load the image
D	Discardable	Marks the section as discardable
K	Cacheable	Marks the section as not cacheable
L	Preload	VxD only; marks the section as preload
P	Pageable	Marks the section as not pageable
X	Memory-resident	VxD only; marks the section as memory-resident

K and P are peculiar in that the section flags that correspond to them are in the negative sense. If you specify one of them on the .text section (/SECTION:.text,K), there will be no difference in the section flags when you run DUMPBIN with the /HEADERS option; it was already implicitly cached. To remove the default, specify /SECTION:.text,!K and DUMPBIN will reveal section characteristics, including "Not Cached."

A section in the PE file that does not have E, R, or W set is probably invalid.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

25.4 Resource Only DLL

Resource Only DLL contains only resource of different language and local types. Resource only DLLs do not contain Entry Point or any DllMain Function.

Use of resource-only DLL is for internationalization.

25.5 DLL Versions

Version information makes it easier for applications to install files properly and enables setup programs to analyze files currently installed. The version-information resource contains the file's version number, intended operating system, and original file name.

You can use the version information functions to determine where a file should be installed and identify conflicts with currently installed files. These functions enable you to avoid the following problems:

- installing older versions of components over newer versions
- changing the language in a mixed-language system without notification
- installing multiple copies of a library in different directories
- copying files to network directories shared by multiple users

The version information functions enable applications to query a version resource for file information and present the information in a clear format. This information includes the file's purpose, author, version number, and so on.

You can add version information to any files that can have Microsoft® Windows® resources, such as dynamic-link libraries (DLLs), executable files, or font files. To add the information, create a VERSIONINFO Resource and use the resource compiler to compile the resource.

25.6 Get File Version Info

The **GetFileVersionInfo** function retrieves version information for the specified file.

```
BOOL GetFileVersionInfo(
```

```
    LPTSTR lptstrFilename,      //file name whose version is
to get*/

    DWORD dwHandle,       /*unused*/
    DWORD dwLen,          /*length of the given buffer*/
    LPVOID lpData         /* buffer*/
);
```

lptstrFilename: Pointer to a null-terminated string that specifies the name of the file of interest. If a full path is not specified, the function uses the search sequence specified by the LoadLibrary function.

dwHandle: This parameter is ignored.

dwLen: Specifies the size, in bytes, of the buffer pointed to by the *lpData* parameter.

> Call the GetFileVersionInfoSize function first to determine the size, in bytes, of a file's version information. The *dwLen* member should be equal to or greater than that value.
>
> If the buffer pointed to by *lpData* is not large enough, the function truncates the file's version information to the size of the buffer.

lpData: Pointer to a buffer that receives the file-version information.

> You can use this value in a subsequent call to the VerQueryValue function to retrieve data from the buffer.

Return Value:

> If the function succeeds, the return value is nonzero.
> If the function fails, the return value is zero. To get extended error information, call GetLastError.

Call the **GetFileVersionInfoSize** function before calling the **GetFileVersionInfo** function. To retrieve information from the file-version information buffer, use the **VerQueryValue** function.

25.7 Threads

25.7.1 Threads and Message Queuing

Message Queue is created when every any GDI function call is made or sendmessage or post message function calls are made. Message Queue can be attached to every thread either it is User interface thread or worker threads.

User Interface threads always a message queue.

Worker threads are initially without message queue.

User Interface threads are those threads which are attached any GUI component such as window.

When a process start at least one thread is running that first thread is called primary thread other threads can made, these threads will, then, be called secondary threads.

25.7.2 Creating Secondary Thread

For creating thread we can use following functions:

_beginthread() and _endthread()
This function is a 'C' runtime concept from UNIX system
These functions no longer have place in Win32 systems.

The CreateThread API

In windows systems CreateThread API is used to create a thread in a process. Every thread has its own thread procedure.

- Threads can be stopped and exited using ExitThread API call.
- Thread enters into running state after creating it. For thread not to be run automatically gives the CREATE_SUSPENDED flag in CreateThread API.
- Threads can be suspended or resumes after their creations by:
- SuspendThread and ResumeThread.

25.7.3 Thread Advantages

Using threads has the following advantages:

1. Threads can be used to start another activity parallel. E.g. saving file on disk, automatically while you are typing.
2. Perform different calculations parallel.

25.7.4 Thread Disadvantages

Threads major disadvantage is that they make the system slow because thread uses the time sharing concept that is another name multitasking. A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor *time slice* to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to

another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors.

Note: You must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

Summary

Multitasking Operating systems are useful to run applications simultaneously. Threads and processes are the key features of Operating systems. In this lecture we studied about variable sharing in DLLs, variable scope in DLLs, DLL Versioning, Resource only DLLs, Threads and their advantages and disadvantages. Many Threads can work better than using single thread sometime.

Exercises

1. Create a dynamic link library and make a function which displays only message box. Export the functions using __dllexport.
2. Call the function from above library in your executable module. The linking must be static linking and use __dllimport.

Chapter 26
Threads and Synchronization

26.1 Thread's Creation

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

lpThreadAttributes: Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

> The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator.

dwStackSize: Initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable.

lpStartAddress: Pointer to the application-defined function to be executed by the thread and represents the starting address of the thread.

lpParameter: Pointer to a variable to be passed to the thread.

dwCreationFlags: Flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation.

lpThreadId: Pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

Return value: If the function succeeds, the return value is a handle to the new thread.

> If the function fails, the return value is NULL.

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2028 threads. If you reduce the default stack size, you can create more threads. However,

your application will have better performance if you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

The new thread handle is created with the THREAD_ALL_ACCESS access right. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread. If the thread impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's TokenDefaultDacl owner or members.

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the **ExitThread** function. Use the **GetExitCodeThread** function to get the thread's return value.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **GetThreadPriority** and **SetThreadPriority** functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

Do not create a thread while impersonating another user. The call will succeed, however the newly created thread will have reduced access rights to itself when calling **GetCurrentThread**. The access rights granted are derived from the access rights that the impersonated user has to the process. Some access rights including THREAD_SET_THREAD_TOKEN and THREAD_GET_CONTEXT may not be present, leading to unexpected failures.

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.

- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the static C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called. Note that this is not a problem with the C run-time in a DLL.

26.2 Thread's Example

```
enum Shape { RECTANGLE, ELLIPSE };
DWORD WINAPI drawThread(LPVOID shape);
SYSTEMTIME st;

hThread1 = CreateThread(NULL, 0,
drawThread,
(LPVOID)RECTANGLE, CREATE_SUSPENDED,
&dwThread1
);

hThread2 = CreateThread(NULL, 0,
drawThread, (LPVOID)ELLIPSE,
CREATE_SUSPENDED, &dwThread2
);

hDC = GetDC(hWnd);
hBrushRectangle=CreateSolidBrush(RGB(170,220,160));
hBrushEllipse = CreateHatchBrush(HS_BDIAGONAL,RGB(175,180,225));

InitializeCriticalSection(&cs);

srand( (unsigned)time(NULL) );
ResumeThread(hThread2);
ResumeThread(hThread1);
```

26.2.1 Thread Procedure

```
DWORD WINAPI drawThread(LPVOID type)
{
        int i;

        if((enum Shape)type == RECTANGLE)
        {
                for(i=0; i<10000; ++i)
```

```
                {
                        EnterCriticalSection(&cs);
                        SelectObject(hDC, hBrushRectangle);
Rectangle(hDC, 50, 1, rand()%300, rand()%100);
                        GetLocalTime(&st);
        LeaveCriticalSection(&cs);
                        Sleep(10);
                }
        }
```

26.3 Synchronization

Using threads we can use lot of shared variables. These shared variables maybe used by a single thread further more these variables may also be used and changed by several parralle threads. If there are several threads operating at the same time then a particular DC handle can be used in one of the threads only. If we want to use a single DC handle in more then one thread, we use synchronization objects. Synchronization objects prevent other threads to use the shared data at the same.

To synchronize access to a resource, use one of the *synchronization objects* in one of the *wait functions*. The state of a synchronization object is either *signaled* or *nonsignaled*. The wait functions allow a thread to block its own execution until a specified nonsignaled object is set to the signaled state.

26.3.1 Overlapped Input and Output

You can perform either synchronous or asynchronous (or overlapped) I/O operations on files, named pipes, and serial communications devices. The **WriteFile**, **ReadFile**, **DeviceIoControl**, **WaitCommEvent**, **ConnectNamedPipe**, and **TransactNamedPipe** functions can be performed either synchronously or asynchronously. The **ReadFileEx** and **WriteFileEx** functions can be performed asynchronously only.

When a function is executed synchronously, it does not return until the operation has been completed. This means that the execution of the calling thread can be blocked for an indefinite period while it waits for a time-consuming operation to finish. Functions called for overlapped operation can return immediately, even though the operation has not been completed. This enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks. For example, a single thread can perform simultaneous I/O operations on different handles, or even simultaneous read and write operations on the same handle.

To synchronize its execution with the completion of the overlapped operation, the calling thread uses the **GetOverlappedResult** function or one of the *wait functions* to determine when the overlapped operation has been completed. You can also use the **HasOverlappedIoCompleted** macro to poll for completion.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle.

Overlapped operations require a file, named pipe, or communications device that was created with the FILE_FLAG_OVERLAPPED flag. To call a function to perform an overlapped operation, the calling thread must specify a pointer to an **OVERLAPPED** structure. If this pointer is NULL, the function return value may incorrectly indicate that the operation completed. The system sets the state of the event object to nonsignaled when a call to the I/O function returns before the operation has been completed. The system sets the state of the event object to signaled when the operation has been completed.

When a function is called to perform an overlapped operation, it is possible that the operation will be completed before the function returns. When this happens, the results are handled as if the operation had been performed synchronously. If the operation was not completed, however, the function's return value is FALSE, and the **GetLastError** function returns ERROR_IO_PENDING.

A thread can manage overlapped operations by either of two methods:

- Use the **GetOverlappedResult** function to wait for the overlapped operation to be completed.
- Specify a handle to the **OVERLAPPED** structure's manual-reset event object in one of the *wait functions* and then call **GetOverlappedResult** after the wait function returns. The **GetOverlappedResult** function returns the results of the completed overlapped operation, and for functions in which such information is appropriate, it reports the actual number of bytes that were transferred.

When performing multiple simultaneous overlapped operations, the calling thread must specify an **OVERLAPPED** structure with a different manual-reset event object for each operation. To wait for any one of the overlapped operations to be completed, the thread specifies all the manual-reset event handles as wait criteria in one of the multiple-object *wait functions*. The return value of the multiple-object wait function indicates which manual-reset event object was signaled, so the thread can determine which overlapped operation caused the wait operation to be completed.

If no event object is specified in the **OVERLAPPED** structure, the system signals the state of the file, named pipe, or communications device when the overlapped operation has been completed. Thus, you can specify these handles as synchronization objects in a wait function, though their use for this purpose can be difficult to manage. When performing simultaneous overlapped operations on the same file, named pipe, or communications device, there is no way to know which operation caused the object's state to be signaled. It is safer to use a separate event object for each overlapped operation.

26.3.2 Asynchronous Procedure Call

An asynchronous procedure call (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function. APCs made by the system are called "kernel-mode APCs." APCs made by an application are called "user-mode APCs." A thread must be in an alertable state to run a user-mode APC.

Each thread has its own APC queue. An application queues an APC to a thread by calling the **QueueUserAPC** function. The calling thread specifies the address of an APC function in the call to **QueueUserAPC**. The queuing of an APC is a request for the thread to call the APC function.

When a user-mode APC is queued, the thread to which it is queued is not directed to call the APC function unless it is in an alertable state. A thread enters an alertable state when it calls the **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**, or **WaitForSingleObjectEx** function. Note that you cannot use **WaitForSingleObjectEx** to wait on the handle to the object for which the APC is queued. Otherwise, when the asynchronous operation is completed, the handle is set to the signaled state and the thread is no longer in an alertable wait state, so the APC function will not be executed. However, the APC is still queued, so the APC function will be executed if you call another alertable wait function.

Note that the **ReadFileEx**, **SetWaitableTimer**, and **WriteFileEx** functions are implemented using an APC as the completion notification callback mechanism.

26.3.3 Critical Section

Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization (a processor-specific test and set instruction). Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. There is no guarantee about the order in which threads will obtain ownership of the critical section; however, the system will be fair to all threads. Unlike a mutex object, there is no way to tell whether a critical section has been abandoned.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type CRITICAL_SECTION. Before the threads of the process can use it, initialize the critical section by using the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function.

A thread uses the **EnterCriticalSection** or **TryEnterCriticalSection** function to request ownership of a critical section. It uses the **LeaveCriticalSection** function to release ownership of a critical section. If the critical section object is currently owned by another thread, **EnterCriticalSection** waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the *wait functions* accept a specified time-out interval. The **TryEnterCriticalSection** function attempts to enter a critical section without blocking the calling thread.

Once a thread owns a critical section, it can make additional calls to **EnterCriticalSection** or **TryEnterCriticalSection** without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. To release its ownership, the thread must call **LeaveCriticalSection** once for each time that it entered the critical section.

A thread uses the **InitializeCriticalSectionAndSpinCount** or **SetCriticalSectionSpinCount** function to specify a spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin *dwSpinCount* times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

When a critical section object is owned, the only other threads affected are those waiting for ownership in a call to **EnterCriticalSection**. Threads that are not waiting are free to continue running.

26.4 Wait Functions

The *wait functions* to allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

There are four types of wait functions:

- single-object
- multiple-object
- alertable
- registered

Single-object Wait Functions

The **SignalObjectAndWait**, **WaitForSingleObject**, and **WaitForSingleObjectEx** functions require a handle to one synchronization object. These functions return when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The **SignalObjectAndWait** function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

Multiple-object Wait Functions

The **WaitForMultipleObjects**, **WaitForMultipleObjectsEx**, **MsgWaitForMultipleObjects**, and **MsgWaitForMultipleObjectsEx** functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

- The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The **MsgWaitForMultipleObjects** and **MsgWaitForMultipleObjectsEx** function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue.

For example, a thread could use **MsgWaitForMultipleObjects** to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue. The thread can use the **GetMessage** or **PeekMessage** function to retrieve the input.

When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to nonsignaled.

Alertable Wait Functions

The **MsgWaitForMultipleObjectsEx**, **SignalObjectAndWait**, **WaitForMultipleObjectsEx,** and **WaitForSingleObjectEx** functions differ from the

other wait functions in that they can optionally perform an *alertable wait operation*. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread. For more information about alertable wait operations and I/O completion routines. See Synchronization and Overlapped Input and Output. For more information about APCs, see Asynchronous Procedure Calls that is already described in our above section *Synchronization*.

Registered Wait Functions

The **RegisterWaitForSingleObject** function differs from the other wait functions in that the wait operation is performed by a thread from the thread pool. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool.

By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call the **UnregisterWaitEx** function to cancel the operation. To specify that a wait operation should be executed only once, set the *dwFlags* parameter of **RegisterWaitForSingleObject** to WT_EXECUTEONLYONCE.

Wait Functions and Synchronization Objects

The wait functions can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. Wait functions can modify the states of synchronization objects as follows:

- The count of a semaphore object decreases by one, and the state of the semaphore is set to nonsignaled if its count is zero.
- The states of mutex, auto-reset event, and change-notification objects are set to nonsignaled.
- The state of a synchronization timer is set to nonsignaled.
- The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

Wait Functions and Creating Windows

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than the other wait functions.

26.5 Synchronization Objects

A *synchronization object* is an object whose handle can be specified in one of the *wait functions* to coordinate the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.

The following object types are provided exclusively for synchronization.

Type	Description
Event	Notifies one or more waiting threads that an event has occurred.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource.
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource.
Waitable timer	Notifies one or more waiting threads that a specified time has arrived.

Though available for other uses, the following objects can also be used for synchronization.

Object	Description
Change notification	Created by the **FindFirstChangeNotification** function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree.
Console input	Created when a console is created. The handle to console input is returned by the **CreateFile** function when CONIN$ is specified, or by the **GetStdHandle** function. Its state is set to signaled when there is unread input in the console's input buffer, and set to nonsignaled when the input buffer is empty.
Job	Created by calling the **CreateJobObject** function. The state of a job object is set to signaled when all its processes are terminated because the specified end-of-job time limit has been exceeded.
Memory resource notification	Created by the **CreateMemoryResourceNotification** function. Its state is set to signaled when a specified type of change occurs within physical memory.
Process	Created by calling the **CreateProcess** function. Its state is set to nonsignaled while the process is running, and set to signaled when the process terminates.
Thread	Created when a new thread is created by calling the **CreateProcess**, **CreateThread**, or **CreateRemoteThread** function. Its state is set to nonsignaled while the thread is running, and set to signaled when the thread terminates.

In some circumstances, you can also use a file, named pipe, or communications device as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the event object set in the **OVERLAPPED** structure. It is safer to use the event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

26.5.1 Mutex Object

The **CreateMutex** function creates or opens a named or unnamed mutex object.

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,/*null if default security
attributes*/
  BOOL bInitialOwner, /*is the current thread is the initialize owner*/
  LPCTSTR lpName       /*name of the named mutex object*/
);
```

lpMutexAttributes: Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpMutexAttributes* is NULL, the handle cannot be inherited.

> The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new mutex. If *lpMutexAttributes* is NULL, the mutex gets a default security descriptor. The ACLs in the default security descriptor for a mutex come from the primary or impersonation token of the creator.

bInitialOwner: If this value is TRUE and the caller created the mutex, the calling thread obtains initial ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex.

lpName: Pointer to a null-terminated string specifying the name of the mutex object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

> If *lpName* matches the name of an existing named mutex object, this function requests the MUTEX_ALL_ACCESS access right. In this case, the *bInitialOwner* parameter is ignored because it has already been set by the creating process. If the *lpMutexAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.
>
> If *lpName* is NULL, the mutex object is created without a name.
>
> If *lpName* matches the name of an existing event, semaphore, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\).

Return Values:

If the function succeeds, the return value is a handle to the mutex object. If the named mutex object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS. Otherwise, the caller created the mutex.

The handle returned by **CreateMutex** has the MUTEX_ALL_ACCESS access right and can be used in any function that requires a handle to a mutex object.

Any thread of the calling process can specify the mutex-object handle in a call to one of the *wait functions*. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a mutex object is signaled when it is not owned by any thread. The creating thread can use the *bInitialOwner* flag to request immediate ownership of the mutex. Otherwise, a thread must use one of the wait functions to request ownership. When the mutex's state is signaled, one waiting thread is granted ownership, the mutex's state changes to nonsignaled, and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses the **ReleaseMutex** function to release its ownership.

The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execution. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **ReleaseMutex** once for each time that the mutex satisfied a wait.

Two or more processes can call **CreateMutex** to create the same named mutex. The first process actually creates the mutex, and subsequent processes open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, you should set the *bInitialOwner* flag to FALSE; otherwise, it can be difficult to be certain which process has initial ownership.

Multiple processes can have handles of the same mutex object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the **CreateProcess** function can inherit a handle to a mutex object if the *lpMutexAttributes* parameter of **CreateMutex** enabled inheritance.

- A process can specify the mutex-object handle in a call to the **DuplicateHandle** function to create a duplicate handle that can be used by another process.
- A process can specify the name of a mutex object in a call to the **OpenMutex** or **CreateMutex** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

26.6 Thread Example Using Mutex Object

```
hThread1= CreateThread(NULL, 0, drawThread, (LPVOID)RECTANGLE,
CREATE_SUSPENDED, &dwThread1);

hThread2 = .. .. ..

hBrushRectangle = CreateSolidBrush(RGB(170,220,160));
hBrushEllipse=CreateHatchBrush(HS_BDIAGONAL,RGB(175,180,225));

hMutex=CreateMutex(NULL, 0, NULL);

srand( (unsigned)time(NULL) );
ResumeThread(hThread2);

for(i=0; i<10000; ++i)
{
Switch(WaitForSingleObject(hMutex, INFINITE))
{
        case WAIT_OBJECT_0:
        SelectObject(hDC, hBrushRectangle);
  Rectangle(hDC, 50, 1, rand()%300, rand()%100);
        GetLocalTime(&st);
ReleaseMutex(hMutex);
Sleep(10);

};
```

26.7 Checking if the previous application is running

Using Named Mutex object you can check the application instance whether it is already running or not. Recreating the named mutex open the previous mutex object but set last error to ERROR_ALREADY_EXIST. You can check GetLastError if it is ERROR_ALREADY_EXIST, then it is already running.

26.8 Event Object

The **CreateEvent** function creates or opens a named or unnamed event object.

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,    /*null for the default
security */
  BOOL bManualReset,   /*manual reset or automatically reset its state*/
  BOOL bInitialState,  /*set initialize state signaled or unsignalled*/
  LPCTSTR lpName       /* nanme of the event object*/
);
```

lpEventAttributes: Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpEventAttributes* is NULL, the handle cannot be inherited.

> The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new event. If *lpEventAttributes* is NULL, the event gets a default security descriptor. The ACLs in the default security descriptor for an event come from the primary or impersonation token of the creator.

bManualReset: If this parameter is TRUE, the function creates a manual-reset event object which requires use of the **ResetEvent** function set the state to nonsignaled. If this parameter is FALSE, the function creates an auto-reset event object, and system automatically resets the state to nonsignaled after a single waiting thread has been released.

bInitialState: If this parameter is TRUE, the initial state of the event object is signaled; otherwise, it is nonsignaled.

lpName: Pointer to a null-terminated string specifying the name of the event object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

> If *lpName* matches the name of an existing named event object, this function requests the EVENT_ALL_ACCESS access right. In this case, the *bManualReset* and *bInitialState* parameters are ignored because they have already been set by the creating process. If the *lpEventAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.
>
> If *lpName* is NULL, the event object is created without a name.
>
> If *lpName* matches the name of an existing semaphore, mutex, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Return Values:

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS.

The handle returned by **CreateEvent** has the EVENT_ALL_ACCESS access right and can be used in any function that requires a handle to an event object.

Any thread of the calling process can specify the event-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The initial state of the event object is specified by the *bInitialState* parameter. Use the **SetEvent** function to set the state of an event object to signaled. Use the **ResetEvent** function to reset the state of an event object to nonsignaled.

When the state of a manual-reset event object is signaled, it remains signaled until it is explicitly reset to nonsignaled by the **ResetEvent** function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object, can be released while the object's state is signaled.

When the state of an auto-reset event object is signaled, it remains signaled until a single waiting thread is released; the system then automatically resets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled.

Multiple processes can have handles of the same event object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the **CreateProcess** function can inherit a handle to an event object if the *lpEventAttributes* parameter of **CreateEvent** enabled inheritance.
- A process can specify the event-object handle in a call to the **DuplicateHandle** function to create a duplicate handle that can be used by another process.
- A process can specify the name of an event object in a call to the **OpenEvent** or **CreateEvent** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

26.8.1 Using Event Object (*Example*)

Applications use event objects in a number of situations to notify a waiting thread of the occurrence of an event. For example, overlapped I/O operations on files, named pipes, and communications devices use an event object to signal their completion.

In the following example, an application uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. First, the master thread uses the **CreateEvent** function to create a manual-reset event object. The master thread sets the event object to non-signaled when it is writing to the buffer and then resets the object to signaled when it has finished writing. Then it creates several reader threads and an auto-reset event object for each thread. Each reader thread sets its event object to signaled when it is not reading from the buffer.

```
#define NUMTHREADS 4

HANDLE hGlobalWriteEvent;

void CreateEventsAndThreads(void)
{
    HANDLE hReadEvents[NUMTHREADS], hThread;
    DWORD i, IDThread;

    // Create a manual-reset event object. The master thread sets
    // this to nonsignaled when it writes to the shared buffer.

    hGlobalWriteEvent = CreateEvent(
        NULL,         // no security attributes
        TRUE,         // manual-reset event
        TRUE,         // initial state is signaled
        "WriteEvent"  // object name
        );

    if (hGlobalWriteEvent == NULL)
    {
        // error exit
    }

    // Create multiple threads and an auto-reset event object
    // for each thread. Each thread sets its event object to
    // signaled when it is not reading from the shared buffer.

    for(i = 1; i <= NUMTHREADS; i++)
    {
        // Create the auto-reset event.
        hReadEvents[i] = CreateEvent(
            NULL,     // no security attributes
            FALSE,    // auto-reset event
            TRUE,     // initial state is signaled
            NULL);    // object not named

        if (hReadEvents[i] == NULL)
        {
```

```
            // Error exit.
        }

        hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE) ThreadFunction,
            &hReadEvents[i],  // pass event handle
            0, &IDThread);
        if (hThread == NULL)
        {
            // Error exit.
        }
    }
}
```

Before the master thread writes to the shared buffer, it uses the **ResetEvent** function to set the state of hGlobalWriteEvent (an application-defined global variable) to nonsignaled. This blocks the reader threads from starting a read operation. The master then uses the **WaitForMultipleObjects** function to wait for all reader threads to finish any current read operations. When **WaitForMultipleObjects** returns, the master thread can safely write to the buffer. After it has finished, it sets hGlobalWriteEvent and all the reader-thread events to signaled, enabling the reader threads to resume their read operations.

```
VOID WriteToBuffer(VOID)
{
    DWORD dwWaitResult, i;

    // Reset hGlobalWriteEvent to nonsignaled, to block readers.

    if (! ResetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Wait for all reading threads to finish reading.

    dwWaitResult = WaitForMultipleObjects(
        NUMTHREADS,   // number of handles in array
        hReadEvents,  // array of read-event handles
        TRUE,         // wait until all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // All read-event objects were signaled.
        case WAIT_OBJECT_0:
            // Write to the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitProcess(0);
    }
```

```
    // Set hGlobalWriteEvent to signaled.

    if (! SetEvent(hGlobalWriteEvent) )
    {
        // Error exit.
    }

    // Set all read events to signaled.
    for(i = 1; i <= NUMTHREADS; i++)
        if (! SetEvent(hReadEvents[i]) )
        {
            // Error exit.
        }
}
```

Before starting a read operation, each reader thread uses **WaitForMultipleObjects** to wait for the application-defined global variable hGlobalWriteEvent and its own read event to be signaled. When **WaitForMultipleObjects** returns, the reader thread's auto-reset event has been reset to nonsignaled. This blocks the master thread from writing to the buffer until the reader thread uses the **SetEvent** function to set the event's state back to signaled.

```
VOID ThreadFunction(LPVOID lpParam)
{
    DWORD dwWaitResult;
    HANDLE hEvents[2];

    hEvents[0] = *(HANDLE*)lpParam;  // thread's read event
    hEvents[1] = hGlobalWriteEvent;

    dwWaitResult = WaitForMultipleObjects(
        2,            // number of handles in array
        hEvents,      // array of event handles
        TRUE,         // wait till all are signaled
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // Both event objects were signaled.
        case WAIT_OBJECT_0:
            // Read from the shared buffer.
            break;

        // An error occurred.
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitThread(0);
    }
    // Set the read event to signaled.
    if (! SetEvent(hEvents[0]) )
    {
        // Error exit.
    }
}
```

26.9 Semaphore Object

The **CreateSemaphore** function creates or opens a named or unnamed semaphore object.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);
```

lpSemaphoreAttributes: Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpSemaphoreAttributes* is NULL, the handle cannot be inherited.

> The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new semaphore. If *lpSemaphoreAttributes* is NULL, the semaphore gets a default security descriptor. The ACLs in the default security descriptor for a semaphore come from the primary or impersonation token of the creator.

lInitialCount: Initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to *lMaximumCount*. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the **ReleaseSemaphore** function.

lMaximumCount: Maximum count for the semaphore object. This value must be greater than zero.

lpName: Pointer to a null-terminated string specifying the name of the semaphore object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

> If *lpName* matches the name of an existing named semaphore object, this function requests the SEMAPHORE_ALL_ACCESS access right. In this case, the *lInitialCount* and *lMaximumCount* parameters are ignored because they have already been set by the creating process. If the *lpSemaphoreAttributes* parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.
>
> If *lpName* is NULL, the semaphore object is created without a name.
>
> If *lpName* matches the name of an existing event, mutex, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Return Values:

> If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, the function returns a handle to the existing object and **GetLastError** returns ERROR_ALREADY_EXISTS.
>
> If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

The handle returned by **CreateSemaphore** has the SEMAPHORE_ALL_ACCESS access right and can be used in any function that requires a handle to a semaphore object.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a semaphore object is signaled when its count is greater than zero, and nonsignaled when its count is equal to zero. The *lInitialCount* parameter specifies the initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one. Use the **ReleaseSemaphore** function to increment a semaphore's count by a specified amount. The count can never be less than zero or greater than the value specified in the *lMaximumCount* parameter.

Multiple processes can have handles of the same semaphore object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the **CreateProcess** function can inherit a handle to a semaphore object if the *lpSemaphoreAttributes* parameter of **CreateSemaphore** enabled inheritance.
- A process can specify the semaphore-object handle in a call to the **DuplicateHandle** function to create a duplicate handle that can be used by another process.
- A process can specify the name of a semaphore object in a call to the **OpenSemaphore** or **CreateSemaphore** function.

Use the **CloseHandle** function to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

26.10 Thread Local Storage (TLS)

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process may allocate locations in which to store thread-specific data. Dynamically bound (run-time) thread-specific data is supported by way of the TLS API (TlsAlloc, TlsGetValue, TlsSetValue, TlsFree). Win32 and the Visual C++ compiler, now support statically bound (load-time) per-thread data in addition to the existing API implementation.

API Implementation for TLS

Thread Local Storage is implemented through the Win32 API layer as well as the compiler. For details, see the Win32 API documentation for *TlsAlloc*, *TlsGetValue*, *TlsSetValue*, and *TlsFree*.

The Visual C++ compiler includes a keyword to make thread local storage operations more automatic, rather than through the API layer. This syntax is described in the next section, Compiler Implementation for TLS.

Compiler Implementation for TLS

To support TLS, a new attribute, **thread**, has been added to the C and C++ languages and is supported by the Visual C++ compiler. This attribute is an extended storage class modifier, as described in the previous section. Use the **__declspec** keyword to declare a **thread** variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

Summary

In this lecture, we studied about Threads and synchronization. To synchronize access to a resource, use one of the *synchronization objects* in one of the *wait functions*. The state of a synchronization object is either *signaled* or *nonsignaled*. The wait functions allow a thread to block its own execution until a specified nonsignaled object is set to the signaled state. Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process. Another synchronization object is semaphore, events and mutex. Threads with synchronization problems have the best use in network applications.

Exercises

1. Create Thread to find factorial of any number.

Chapter 27
Network Programming Part I

27.1 Introduction

Following are the some of the concept of packet information. These concepts will be used in network programming.

- IP addresses and ports
- The structure of an IP packet
- Protocol
- Connection-oriented vs. datagram protocols
- IP, TCP and UDP
- HTTP, other wrapper protocols

27.2 Well known Protocols

Following are the well known protocols used today.

Ports	Name
80 25 110 43 53 21	http SMTP POP3 WHOIS DNS FTP

27.3 DNS (Domain Name Systems)

Domain Name System (DNS), the locator service of choice in Microsoft® Windows®, is an industry-standard protocol that locates computers on an IP-based network. IP networks such as the Internet and Windows networks rely on number-based addresses to process information. Users however, are better at remembering letter-based addresses, so it is necessary to translate user-friendly names http://www.vu.edu.pk into addresses that the network can recognize (**203.215.177.33**).

Domain Name System, DNS, is an industry-standard protocol used to locate computers on an IP-based network. Users are better at remembering friendly names, such as www.microsoft.com or msdn.microsoft.com, than number-based addresses, such as 207.46.131.137.

IP networks, such as the Internet and Microsoft® Windows® 2000 networks rely on number-based addresses to ferry information throughout the network; therefore, it is necessary to translate user-friendly names (www.microsoft.com) into addresses that

the network can recognize (207.46.131.137). DNS is the service of choice in Windows 2000 to locate such resources and translate them into IP addresses.

DNS is the primary locator service for Active Directory, and therefore, DNS can be considered a base service for both Windows 2000 and Active Directory. Windows 2000 provides functions that enable application programmers to use DNS functions such as programmatically making DNS queries, comparing records, and looking up names.

27.4 Well known host names on the internet

- www.vu.edu.pk 203.215.177.33
- www.yahoo.com 64.58.76.179
- www.most.gov.pk 66.96.232.41
- www.pak.gov.pk 66.197.42.253
- www.google.com 216.239.53.100
- www.whois.net 128.121.95.59

27.5 Windows Sockets

Windows Sockets (Winsock) enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data across the wire, independent of the network protocol being used. With Winsock, programmers are provided access to advanced Microsoft® Windows® networking capabilities such as multicast and Quality of Service (QOS).

Winsock follows the Windows Open System Architecture (WOSA) model; it defines a standard service provider interface (SPI) between the application programming interface (API), with its exported functions and the protocol stacks. It uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Windows in Windows Sockets 1.1, with which Windows Sockets 2 applications are backward compatible. Winsock programming previously centered on TCP/IP. Some programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API adds functions where necessary to handle several protocols.

27.6 Basic Sockets Operations

The following are the basic operations performed by both server and client systems.

1. Create an unbound socket
2. Binding Server
3. Connecting Client
4. Listen
5. Accept
6. Send
7. Receive

27.7 Windows Socket Library

File	Purpose
ws2_32.dll	Main WinSock 2 DLL
wsock32.dll	For WinSock 1.1 support, 32-bit applications
mswsock.dll	MS extensions to WinSock
winsock.dll	For WinSock 1.1 support, 16-bit applications
ws2help.dll	WinSock2 helper
ws2tcpip.dll	WinSock 2 helper for TCP/IP stacks

These files are windows socket libraries.

27.8 WinSock Initialization

The **WSAStartup** function initiates use of WS2_32.DLL by a process.

```
int WSAStartup(
  WORD wVersionRequested,      /*MAKEWORD(2,2)*/
  LPWSADATA lpWSAData          /*POINTER TO THE WSADATA structure
);
```

wVersionRequested: Highest version of Windows Sockets support that the caller can use. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.
lpWSAData: Pointer to the **WSADATA** data structure that is to receive details of the Windows Sockets implementation.

Return Values: The **WSAStartup** function returns zero if successful. Otherwise, it returns one of the error codes listed in the following.
An application cannot call **WSAGetLastError** to determine the error code as is normally done in Windows Sockets if **WSAStartup** fails. The WS2_32.DLL will *not* have been loaded in the case of a failure so the client data area where the last error information is stored could not be established.

Error code	Meaning
WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The *lpWSAData* is not a valid pointer.

The **WSAStartup** function *must* be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling **WSAStartup**.

In order to support future Windows Sockets implementations and applications that can have functionality differences from the current version of Windows Sockets, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** and the WS2_32.DLL indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSAStartup**, the WS2_32.DLL examines the version requested by the application. If this version is equal to or higher than the lowest version supported by the DLL, the call succeeds and the DLL returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested*. The WS2_32.DLL then assumes that the application will use *wVersion* If the *wVersion* parameter of the **WSADATA** structure is unacceptable to the caller, it should call **WSACleanup** and either search for another WS2_32.DLL or fail to initialize.

It is legal and possible for an application written to this version of the specification to successfully negotiate a higher version number version. In that case, the application is only guaranteed access to higher-version functionality that fits within the syntax defined in this version, such as new Ioctl codes and new behavior of existing functions. New functions may be inaccessible. To get full access to the new syntax of a future version, the application must fully conform to that future version, such as compiling against a new header file, linking to a new library, or other special cases.

This negotiation allows both a WS2_32.DLL and a Windows Sockets application to support a range of Windows Sockets versions. An application can use WS2_32.DLL if there is any overlap in the version ranges. The following table shows how **WSAStartup** works with different applications and WS2_32.DLL versions.

App versions	**DLL versions**	***wVersion requested***	***wVersion***	***wHigh version***	**End result**
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	2.0	2.0	2.0	2.0	use 2.0

Example Code

The following code fragment demonstrates how an application that supports only version 2.2 of Windows Sockets makes a **WSAStartup** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
```

Once an application or DLL has made a successful **WSAStartup** call, it can proceed to make other Windows Sockets calls as needed. When it has finished using the services of the WS2_32.DLL, the application or DLL must call **WSACleanup** to allow the WS2_32.DLL to free any resources for the application.

Details of the actual Windows Sockets implementation are described in the **WSADATA** structure.

An application or DLL can call **WSAStartup** more than once if it needs to obtain the **WSADATA** structure information more than once. On each such call the application can specify any version number supported by the DLL.

An application must call one **WSACleanup** call for every successful **WSAStartup** call to allow third-party DLLs to make use of a WS2_32.DLL on behalf of an application.

This means, for example, that if an application calls **WSAStartup** three times, it must call **WSACleanup** three times. The first two calls to **WSACleanup** do nothing except decrement an internal counter; the final **WSACleanup** call for the task does all necessary resource deallocation for the task.

WinSock version: high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

Summary

Socket is important in an inter-process communication. Sockets are more reliable and secure. Socket version 2 is used these days. In windows, sockets are started using WSAStartup API. WSAStartup API starts and initializes Windows Sockets. Domain Name System (DNS), the locator service of choice in Microsoft® Windows®, is an industry-standard protocol that locates computers on an IP-based network.

Exercises

1. Study internet protocols yourself.

Chapter 28
Network Programming Part II

28.1 WinSock Server Socket Functions

Bind:

The **bind** function associates a local address with a socket.

```
int bind(
  SOCKET s,                    //socket descriptor */
  const struct sockaddr* name, /* sockaddr structure */ /*read the

compatibility problem statements by the use of IPv4 and IPv6*/
/*connect Virtual University resource for the updated IPv6
informations*/

  int namelen
);
```

s: Descriptor identifying an unbound socket.
name: Address to assign to the socket from the **sockaddr** structure.
namelen: Length of the value in the *name* parameter, in bytes.

Return Value: If no error occurs, **bind** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WSAStartup** call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because **setsockopt** option SO_BROADCAST is not enabled.
WSAEADDRINUSE	A process on the computer is already bound to the same fully-qualified address and the socket has not been marked to allow address reuse with SO_REUSEADDR. For example, the IP address and port are bound in the af_inet case). (See the SO_REUSEADDR socket option under **setsockopt**.)
WSAEADDRNOTAVAIL	The specified address is not a valid address for this computer.
WSAEFAULT	The *name* or *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the *name* parameter contains an incorrect address format for the associated address family, or the first two bytes of the memory block specified by

	name does not match the address family associated with the socket descriptor *s*.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	The descriptor is not a socket.

The **bind** function is used on an unconnected socket before subsequent calls to ***connect*** or ***listen*** functions. It is used to bind to either connection-oriented (stream) or connectionless (datagram) sockets. When a socket is created with a call to the **socket** function, it exists in a namespace (address family), but it has no name assigned to it. Use the **bind** function to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts when using the Internet address family:

- The address family.
- A host addresses.
- A port number that identifies the application.

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **sockaddr** structure. It is cast this way for Windows Sockets 1.1 compatibility. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first 2 bytes in this block (corresponding to the **sa_family** member of the **sockaddr** structure) must contain the address family that was used to create the socket. Otherwise, an error WSAEFAULT occurs.

If an application does not care what local address is assigned, specify the manifest constant value ADDR_ANY for the **sa_data** member of the *name* parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of multihomed hosts (that is, hosts that have more than one network interface and address).

For TCP/IP, if the port is specified as zero, the service provider assigns a unique port to the application with a value between 1024 and 5000. The application can use **getsockname** after calling **bind** to learn the address and the port that has been assigned to it. If the Internet address is equal to INADDR_ANY, **getsockname** cannot necessarily supply the address until the socket is connected, since several addresses can be valid if the host is multihomed. Binding to a specific port number other than port 0 is discouraged for client applications, since there is a danger of conflicting with another socket already using that port number.

Note *When using* ***bind*** *with the SO_EXCLUSIVEADDR or SO_REUSEADDR socket option, the socket option must be set prior to executing* ***bind*** *to have any affect.*

Sockaddr

The sockaddr structure varies depending on the protocol selected. Except for the *sa_family* parameter, sockaddr contents are expressed in network byte order.

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a sockaddr structure. It is presented in this manner for Windows Sockets compatibility. The actual structure is interpreted differently in the context of different address families. The only requirements are that the first **u_short** is the address family and the total size of the memory buffer in bytes is *namelen*.

The structures below are used with IPv4 and IPv6, respectively. Other protocols use similar structures.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
struct sockaddr_in6 {
        short   sin6_family;
        u_short sin6_port;
        u_long  sin6_flowinfo;
        struct  in6_addr sin6_addr;
        u_long  sin6_scope_id;
};
struct sockaddr_in6_old {
        short   sin6_family;
        u_short sin6_port;
        u_long  sin6_flowinfo;
        struct  in6_addr sin6_addr;
};
```

Host and network byte-ordering: htonl(), htons(), ntohl(), ntohs()

gethostbyname

The **gethostbyname** function retrieves host information corresponding to a host name from a host database.

```
struct hostent* FAR gethostbyname(
  const char* name
);
```

name: Pointer to the null-terminated name of the host to resolve.

Return Value: If no error occurs, **gethostbyname** returns a pointer to the **hostent** structure described above. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling **WSAGetLastError**.

Error code	Meaning
WSANOTINITIALISED	A successful **WSAStartup** call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failure.
WSANO_RECOVERY	A nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

The **gethostbyname** function returns a pointer to a **hostent** structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

The **gethostbyname** function cannot resolve IP address strings passed to it. Such a request is treated exactly as if an unknown host name were passed. Use **inet_addr** to convert an IP address string the string to an actual IP address, then use another function, **gethostbyaddr**, to obtain the contents of the **hostent** structure.

If null is provided in the *name* parameter, the returned string is the same as the string returned by a successful **gethostname** function call.

Note: *The gethostbyname function does not check the size of the name parameter before passing the buffer. In improperly sized name parameters, heap corruption can occur.*

Connect

The **connect** function establishes a connection to a specified socket.

```
int connect(
  SOCKET s,
  const struct sockaddr* name,
  int namelen
);
```

s: Descriptor identifying an unconnected socket.

name: Name of the socket in the **sockaddr** structure to which the connection should be established.

namelen: Length of *name*, in bytes

Return Values: If no error occurs, **connect** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError**.

On a blocking socket, the return value indicates success or failure of the connection attempt.

With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, **connect** will return SOCKET_ERROR, and **WSAGetLastError** will return WSAEWOULDBLOCK. In this case, there are three possible scenarios:

- Use the **select** function to determine the completion of the connection request by checking to see if the socket is writeable.
- If the application is using **WSAAsyncSelect** to indicate interest in connection events, then the application will receive an FD_CONNECT notification indicating that the **connect** operation is complete (successfully or not).
- If the application is using **WSAEventSelect** to indicate interest in connection events, then the associated event object will be signaled indicating that the **connect** operation is complete (successfully or not).

Until the connection attempt completes on a nonblocking socket, all subsequent calls to **connect** on the same socket will fail with the error code WSAEALREADY, and WSAEISCONN when the connection completes successfully. Due to ambiguities in version 1.1 of the Windows Sockets specification, error codes returned from **connect** while a connection is already pending may vary among implementations. As a result, it is not recommended that applications use multiple calls to connect to detect connection completion. If they do, they must be prepared to handle WSAEINVAL and WSAEWOULDBLOCK error values the same way that they handle WSAEALREADY, to assure robust execution.

The **connect** function is used to create a connection to the specified destination. If socket *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

For connection-oriented sockets (for example, type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the namespace of the socket.

Note: *If a socket is opened, a* ***setsockopt*** *call is made, and then a* ***sendto*** *call is made, Windows Sockets performs an implicit* ***bind*** *function call.*

When the socket call completes successfully, the socket is ready to send and receive data. If the address member of the structure specified by the *name* parameter is all zeroes, **connect** will return the error WSAEADDRNOTAVAIL. Any attempt to reconnect an active connection will fail with the error code WSAEISCONN.

For connection-oriented, nonblocking sockets, it is often not possible to complete the connection immediately. In such a case, this function returns the error WSAEWOULDBLOCK. However, the operation proceeds.

When the success or failure outcome becomes known, it may be reported in one of two ways, depending on how the client registers for notification.

- If the client uses the **select** function, success is reported in the writefds set and failure is reported in the exceptfds set.
- If the client uses the functions **WSAAsyncSelect** or **WSAEventSelect**, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure.

For a connectionless socket (for example, type SOCK_DGRAM), the operation performed by **connect** is merely to establish a default destination address that can be used on subsequent **send**/ **WSASend** and **recv**/ **WSARecv** calls. Any datagrams received from an address other than the destination address specified will be discarded. If the address member of the structure specified by *name* is all zeroes, the socket will be disconnected. Then, the default remote address will be indeterminate, so **send**/ **WSASend** and **recv**/ **WSARecv** calls will return the error code WSAENOTCONN. However, **sendto**/ **WSASendTo** and **recvfrom**/ **WSARecvFrom** can still be used. The default destination can be changed by simply calling **connect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **connect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must use **setsockopt** to enable the SO_BROADCAST option. Otherwise, **connect** will fail with the error code WSAEACCES.

When a connection between sockets is broken, the sockets should be discarded and recreated. When a problem develops on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

28.2 Sending or receiving from server

Send

The **send** function sends data on a connected socket.

```
int send(
  SOCKET s,
  const char* buf,
  int len,
  int flags
);
```

s: Descriptor identifying a connected socket.
buf: Buffer containing the data to be transmitted.
len: Length of the data in *buf*, in bytes
flags: Indicator specifying the way in which the call is made.

Return Values: If no error occurs, **send** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a value of SOCKET_ERROR is returned

The **send** function is used to write outgoing data on a connected socket. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by using **getsockopt** to retrieve the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol, the error WSAEMSGSIZE is returned, and no data is transmitted.

The successful completion of a **send** does not indicate that the data was successfully delivered.

If no buffer space is available within the transport system to hold the data to be transmitted, **send** will block unless the socket has been placed in nonblocking mode. On nonblocking stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both client and server computers. The **select**, **WSAAsyncSelect** or **WSAEventSelect** functions can be used to determine when it is possible to send more data.

Calling **send** with a zero *len* parameter is permissible and will be treated by implementations as successful. In such cases, **send** will return zero as a valid value. For message-oriented sockets, a zero-length transport datagram is sent.

The *flags* parameter can be used to influence the behavior of the function beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only. Also see DECnet Out-Of-band data for a discussion of this topic).

Recv

The **recv** function receives data from a connected or bound socket.

```
int recv(
  SOCKET s,
  char* buf,
  int len,
  int flags
);
```

s: Descriptor identifying a connected socket.
buf: Buffer for the incoming data.
len: Length of *buf*, in bytes
flags: Flag specifying the way in which the call is made.

Return Values: If no error occurs, **recv** returns the number of bytes received. If the connection has been gracefully closed, the return value is zero. Otherwise, a value of SOCKET_ERROR is returned,

The **recv** function is used to read incoming data on connection-oriented sockets, or connectionless sockets. When using a connection-oriented protocol, the sockets must be connected before calling **recv**. When using a connectionless protocol, the sockets must be bound before calling **recv**.

The local address of the socket must be known. For server applications, use an explicit **bind** function or an implicit **accept** or **WSAAccept** function. Explicit binding is discouraged for client applications. For client applications, the socket can become bound implicitly to a local address using **connect**, **WSAConnect**, **sendto**, **WSASendTo**, or **WSAJoinLeaf**.

For connected or connectionless sockets, the **recv** function restricts the addresses from which received messages are accepted. The function only returns messages from the

remote address specified in the connection. Messages from other addresses are (silently) discarded.

For connection-oriented sockets (type SOCK_STREAM for example), calling **recv** will return as much information as is currently available—up to the size of the buffer specified. If the socket has been configured for in-line reception of OOB data (socket option SO_OOBINLINE) and OOB data is yet unread, only OOB data will be returned. The application can use the **ioctlsocket** or **WSAIoctl SIOCATMARK** command to determine whether any more OOB data remains to be read.

For connectionless sockets (type SOCK_DGRAM or other message-oriented sockets), data is extracted from the first enqueued datagram (message) from the destination address specified by the **connect** function.

If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **recv** generates the error WSAEMSGSIZE. For unreliable protocols (for example, UDP) the excess data is lost; for reliable protocols, the data is retained by the service provider until it is successfully read by calling **recv** with a large enough buffer.

If the socket is connection oriented and the remote side has shut down the connection gracefully, and all data has been received, a **recv** will complete immediately with zero bytes received. If the connection has been reset, a **recv** will fail with the error WSAECONNRESET.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

Value	Meaning
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. The function subsequently returns the amount of data that can be read in a single call to the **recv** (or **recvfrom**) function, which may not be the same as the total amount of data queued on the socket. The amount of data that can actually be read in a single call to the **recv** (or **recvfrom**) function is limited to the data size written in the **send** or **sendto** function call.
MSG_OOB	Processes OOB data. (See DECnet Out-of-band data for a discussion of this topic.)

28.3 Difference between server and client socket calls

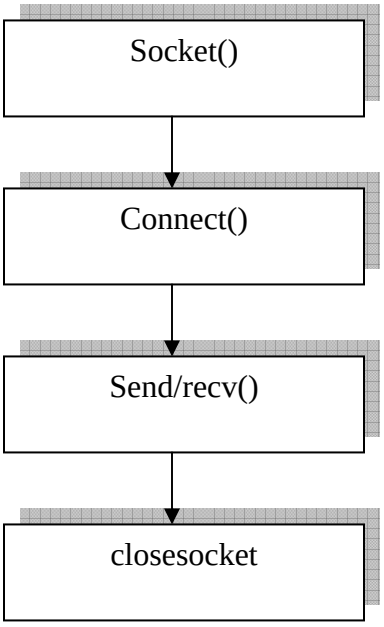


Figure 1 Client Connection

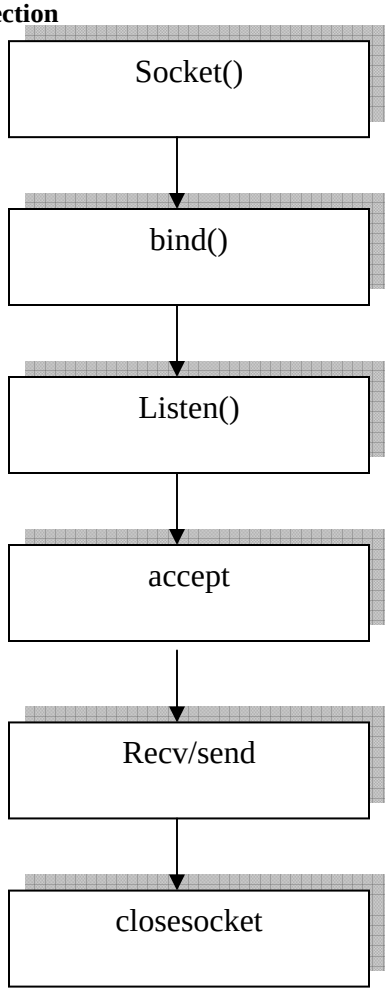


Figure 2 Server Connection

28.4 Listen

The **listen** function places a socket in a state in which it is listening for an incoming connection.

```
int listen(
  SOCKET s,
  int backlog
);
```

s: Descriptor identifying a bound, unconnected socket.

Backlog: Maximum length of the queue of pending connections. If set to SOMAXCONN, the underlying service provider responsible for socket *s* will set the backlog to a maximum reasonable value. There is no standard provision to obtain the actual backlog value.

Return Values: If no error occurs, **listen** returns zero. Otherwise, a value of SOCKET_ERROR is returned.

To accept connections, a socket is first created with the **socket** function and bound to a local address with the **bind** function, a backlog for incoming connections is specified with listen, and then the connections are accepted with the **accept** function. Sockets that are connection oriented those of type SOCK_STREAM for example, are used with **listen**. The socket *s* is put into passive mode where incoming connection requests are acknowledged and queued pending acceptance by the process.

The **listen** function is typically used by servers that can have more than one connection request at a time. If a connection request arrives and the queue is full, the client will receive an error with an indication of WSAECONNREFUSED.

If there are no available socket descriptors, **listen** attempts to continue to function. If descriptors become available, a later call to **listen** or **accept** will refill the queue to the current or most recent backlog, if possible, and resume listening for incoming connections.

An application can call **listen** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new backlog value, the excess pending connections will be reset and dropped.

28.5 Accept

The **accept** function permits an incoming connection attempt on a socket.

```
SOCKET accept(
  SOCKET s,            /*socket descriptor*/
```

```
  struct sockaddr* addr,       /*sockaddr structure*/
  int* addrlen                 /*string length returned*/
);
```

s: Descriptor identifying a socket that has been placed in a listening state with the **listen** function. The connection is actually made with the socket that is returned by **accept**.

addr: Optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family that was established when the socket from the **sockaddr** structure was created.

Addrlen: Optional pointer to an integer that contains the length of *addr*.

Return Values: If no error occurs, **accept** returns a value of type **SOCKET** that is a descriptor for the new socket. This returned value is a handle for the socket on which the actual connection is made. Otherwise, a value of INVALID_SOCKET is returned

The **accept** function extracts the first connection on the queue of pending connections on socket *s*. It then creates and returns a handle to the new socket. The newly created socket is the socket that will handle the actual connection; it has the same properties as socket *s*, including the asynchronous events registered with the **WSAAsyncSelect** or **WSAEventSelect** functions.

The **accept** function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking. If the socket is marked as nonblocking and no pending connections are present on the queue, **accept** returns an error as described in the following. After the successful completion of **accept** returns a new socket handle, the accepted socket cannot be used to accept more connections. The original socket remains open and listens for new connection requests.

The parameter *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned.

The **accept** function is used with connection-oriented socket types such as SOCK_STREAM. If *addr* and/or *addrlen* are equal to NULL, then no information about the remote address of the accepted socket is returned.

28.6 WinSock Example Application

A client showing simple communication to either our own small server, or some server on the internet, e.g. WHOIS servers, HTTP server, time service etc.

A small utility that synchronizes system time with a source on the internet, accounting for transmission-delays

Screen shot of our application.

28.7 Example Application

```
Int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
        WSADATA wsaData;
        HOSTENT *ptrHostEnt;
        struct sockaddr_in serverSockAddr;   // the address of the socket to connect to

        int abc;

        // try initialising the windows sockets library
        if(WSAStartup( MAKEWORD(1,1), &wsaData))    // request WinSock ver 1.1
        {
                MessageBox(NULL, "Error initialising sockets library.", "WinSock
Error", MB_OK | MB_ICONSTOP);
                return 1;
        }


/*Get host name */
if(!(ptrHostEnt = gethostbyname(WHOIS_SERVER_NAME)))
        {
                MessageBox(NULL, "Could not resolve WHOIS server name.",
"WinSock Error", MB_OK | MB_ICONSTOP);
```

```
                WSACleanup();
                return 1;
        }

serverSockAddr.sin_family = AF_INET;      // fill the address structure with appropriate
values
serverSockAddr.sin_port = htons(WHOIS_PORT);  // MUST convert to network
byte-order
        memset(serverSockAddr.sin_zero, 0, sizeof(serverSockAddr.sin_zero));
        memcpy(&serverSockAddr.sin_addr.S_un.S_addr, ptrHostEnt->h_addr_list[0],
sizeof(unsigned long));

        clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if(clientSocket == INVALID_SOCKET)
        {
                MessageBox(NULL, "Error creating client socket.", "WinSock Error",
MB_OK | MB_ICONSTOP);
                WSACleanup();
                return 1;
        }

/*Start Connection*/

if(connect(clientSocket, (struct sockaddr *)&serverSockAddr, sizeof(serverSockAddr)))
        {

                abc =  WSAGetLastError();

                MessageBox(NULL, "Error connecting to WHOIS server.", "WinSock
Error", MB_OK | MB_ICONSTOP);
                WSACleanup();
                return 1;
        }

        if(DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG_MAIN),
NULL, mainDialogProc) == 1)
                MessageBox(NULL, "Error occurred while sending data to WHOIS
server.", "WinSock Error", MB_OK | MB_ICONSTOP);

        WSACleanup();

        return 0;
}
```

```
BOOL CALLBACK mainDialogProc(HWND hDlg, UINT message, WPARAM
wParam, LPARAM lParam)
{
	int wID, wNotificationCode;
	char domainName[MAX_DOMAIN_LEN+2+1]; // accomodate CR/LF/NULL
	char result[BUFFER_SIZE], *startOfBuffer = result;
	int bytesReceived;

	switch(message)
	{
	case WM_INITDIALOG:
		SendDlgItemMessage(hDlg, IDC_EDIT_DOMAIN, EM_LIMITTEXT,
MAX_DOMAIN_LEN, 0);
		return TRUE;
		break;

	case WM_COMMAND:
		wNotificationCode = HIWORD(wParam);
		wID = LOWORD(wParam);
		switch(wID)
		{
case IDC_BUTTON_SEND:
			EnableWindow(GetDlgItem(hDlg, IDC_BUTTON_SEND),
FALSE);	// disable for 2nd use
			GetDlgItemText(hDlg, IDC_EDIT_DOMAIN,
(LPSTR)domainName, MAX_DOMAIN_LEN+1);
			strcpy(domainName+strlen(domainName), "\r\n");
			if(send(clientSocket, (const char *)domainName,
strlen(domainName), 0) == SOCKET_ERROR)
				EndDialog(hDlg, 1);
			else
			{

				bytesReceived = recv(clientSocket, startOfBuffer,
				BUFFER_SIZE-1, 0); // -1 for NULL
				while(bytesReceived > 0)// 0:close
					//SOCKET_ERROR:error
					{
						startOfBuffer += bytesReceived;	//
				//move it forward
				bytesReceived = recv(clientSocket, startOfBuffer,
				BUFFER_SIZE-(startOfBuffer-result)-1, 0); // -1 for NULL
					}

			if(startOfBuffer != result)	// something received
			*startOfBuffer = NULL;	// NULL terminate
```

```
                        else
                                strcpy(result, "Null Response");

                        SetDlgItemText(hDlg, IDC_EDIT_RESULT, result);
                }

                break;

        case IDCANCEL:
                EndDialog(hDlg, 0);
                break;
        }
        return TRUE;
        break;

    default:
        return FALSE;
    }
    return TRUE;
}
```

Summary

In this lecture, we studied about WinSock functions. These functions include connect, recv, send, accept, bind, gethotbyname, etc. we saw the difference between client socket connection and server socket connection. And finally we made application that is whoisserver. This application tells that the name is registered name or not. If the name is registered, then we cannot register it again.

Note: These lectures explain only IPv4, this protocol is being replaced by IPv6. New resource should use IPv6. For New Internet Protocol version and programming using IPv6, connect to Virtual University resource online.

Exercises

1. Create a simple socket client server application that uses stream socket and TCP/IP protocols. On connecting, the client server must show message that client has been connected.

Chapter 29

Network Programming Part III

29.1 Lecture Goal

This lecture goal is to develop a little Web Server.
This Web Server will serve HTTP requests, sent via a Web Browser using following URLs:

```
http://www.vu.edu.pk/default.html
http://www.vu.edu.pk/index.asp
http://www.vu.edu.pk/win32.html
http://www.vu.edu.pk/courses/win32.html
```

29.2 Uniform Resource Locator (URL)

```
Anatomy of a URL (Uniform Resource Locator):
http://www.vu.edu.pk/courses/win32.html
http://  protocol
www.vu.edu.pk    Web Server

courses/win32.html     location of file on server

Or http://www.vu.edu.pk:80/.../....

:80 is the specifies Port Number to use for connection
```

29.3 HTML

HTML stands for Hyper Text Mark-up Language.
This language contains text-formatting information e.g. font faces, font colors, font sizes, alignment etc. and also contains *HyperLinks*: text that can be clicked to go to another HTML document on the Internet. HTML tags are embedded within normal text to make it hypertext.

29.4 Web Browser

HTTP Client – Web Browser examples are:
Microsoft Internet Explorer
Netscape Navigator

These web servers connect to your HTTP web server, requests a document, and displays in its window

29.5 HTTP

HTTP is a Stateless protocol.

- No information or “state” is maintained about previous HTTP requests
- Easier to implement than state-aware protocols

29.6 MIME

MIME stands for Multi-purpose Internet Mail Extensions.

MIME contains encoding features, added to enable transfer of binary data, e.g. images (GIF, JPEG etc.) via mail. Using MIME encoding HTTP can now transfer complex binary data, e.g. images and video.

29.7 RFC

Short for Request for Comments, a series of notes about the Internet, started in 1969 (when the Internet was the ARPANET). An Internet Document can be submitted to the IETF by anyone, but the IETF decides if the document becomes an RFC. Eventually, if it gains enough interest, it may evolve into an Internet standard.
HTTP version 1.1 is derived from HTTP/1.1, Internet RFC 2616, Fielding, et al. Each RFC is designated by an RFC number. Once published, an RFC never changes. Modifications to an original RFC are assigned a new RFC number.

29.8 Encoding and Decoding

HTTP is a Text Transport Protocol
Transferring binary data over HTTP needs Data Encoding and Decoding because binary characters are not permitted Similarly some characters are not permitted in a URL, e.g. SPACE. Here, URL encoding is used

29.9 Encoding Example Escape Sequence

Including a Carriage Return / Line feed in a string
printf(“Line One\nThis is new line”);

Including a character in a string not found on our normal keyboards
printf(“The funny character \xB2”);

29.10 Virtual Directory

Represents the Home Directory of a Web Server

IIS (Internet Information Server) has c:\inetpub\wwwroot\ as its default Home Directory

Here, /courses/ either corresponds to a Physical Directory c:\inetpub\wwwroot\courses OR Virtual Directoy

In a Web Server, we may specify that /courses/ will represent some other physical directory on the Web Server like D:\MyWeb\. Then /courses/ will be a Virtual Directory. In Windows2000 and IIS 5.0 (Internet Information Server), a folder's "Web Sharing…" is used to create a Virtual Directory for any folder.

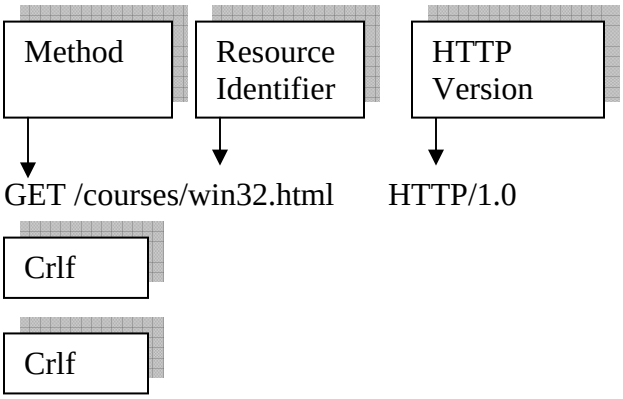
29.11 Web Browser Fetches a pages

- http://www.vu.edu.pk/courses/win32.html

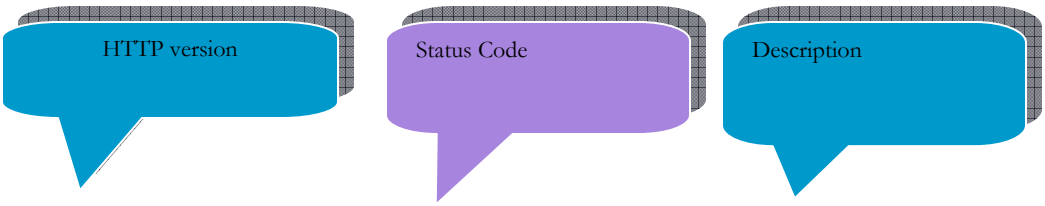
- Hostname/DNS lookup for www.vu.edu.pk to get IP address
- HTTP protocol uses port 80.
- Connect to port 80 of the IP address discovered above!

- Request the server for /courses/win32.html

29.12 HTTP Client Request



Request line is followed by 2 Carriage-Return /Line-feed sequences



HTTP/1.1 200 OK }Status Line

Content-type: text/html
Content-Length:2061 *Headers delimited by CR/LF sequence*

Crlf

Actual data follows the headers

29.13 File Extension and MIME

File extensions are non-standard across different platforms and cannot be used to determine the type of contents of any file.

Different common MIME types

image/gif	GIF image
image/jpeg	JPEG image
text/html	HTML document
text/plain	plain text

In an HTTP response, a Web Server tells the browser MIME type of data being sent

MIME type is used by the browser to handle the data appropriately i.e. show an image, display HTML etc.

MIME:

MIME: Multi-purpose Internet Mail Extensions MIME Encoding features were added to enable transfer of binary data, e.g. images (GIF, JPEG etc.) via mail. Using MIME encoding HTTP can now transfer complex binary data, e.g. images and video

29.14 MIME Encoding

MIME: Short for Multipurpose Internet Mail Extensions, a specification for formatting non-ASCII messages so that they can be sent over the Internet.

Enables us to send and receive graphics, audio, and video files via the Internet mail system.

There are many predefined MIME types, such as GIF graphics files and PostScript files. It is also possible to define your own MIME types.

In addition to e-mail applications, Web browsers also support various MIME types. This enables the browser to display or output files that are not in HTML format.

MIME was defined in 1992 by the Internet Engineering Task Force (IETF). A new version, called S/MIME, supports encrypted messages.

29.15 HTTP Status codes

404 Not Found
- requested document not found on this server

200 OK
- request secceeded, requested object later in this message

400 Bad Request
- request message not understood by server

302 Object Moved
- requested document has been moved to some other location

29.16 HTTP Redirection

HTTP/1.1 302 Object Moved
Location: http://www.vu.edu.pk

crlf

Most browsers will send another HTTP request to the new location, i.e. http://www.vu.edu.pk
This is called Browser Redirection

29.17 HTTP Request per 1 TCP/IP Connection

HTML text is received in one HTTP request from the Web Server

Browser reads all the HTML web page and paints its client area according to the HTML tags specified. Browser generates one fresh HTTP request for each image specified in the HTML file

29.18 Server Architecture

Our server architecture will be based upon the following points

- Ability to serve up to 5 clients simultaneously
- Multi-threaded HTTP Web Server
- 1 thread dedicated to accept client connections
- 1 thread per client to serve HTTP requests
- 1 thread dedicated to perform termination housekeeping of communication threads
- Use of Synchronization Objects

Many WinSock function calls e.g. accept() are blocking calls
Server needs to serve up 5 clients simultaneously. Using other WinSock blocking calls, need to perform termination tasks for asynchronously terminating communication threads.

Summary

In this lecture, we studied some terms and their jobs. We studied HTTP (hyper text transfer protocol) which is used to transfer text data across the net work. We also studied HTML that is hyper text markup language which is simply a text script. Html is loaded in web browser and web browser translates the text and executes instruction written in form of text. For transferring media like image data and movie data, we overviewed MIME.

Note: For example and more information connect to Virtual University resource Online.

Exercises

1. Create a chat application. Using that application, you should be able to chat with your friend on network.

Chapter 30
Network Programming Part IV

30.1 Server Architecture

Server architecture will be based on:

- Dialog-based GUI application
- Most of the processing is at back-end
- Running on TCP port 5432 decimal

30.2 HTTP Web Server Application

Initialize Windows Sockets Library

```
if(WSAStartup(MAKEWORD(1,1), &wsaData))
{
        … … …
        return 1;
}
```

```
//Get machine's hostname and IP address

gethostname(hostName, sizeof(hostName));
ptrHostEnt = gethostbyname(hostName);

//Fill the socket address with appropriate values

serverSocketAddress.sin_family = AF_INET;
serverSocketAddress.sin_port = htons(SERVER_PORT);
… … …
memcpy(&serverSocketAddress.sin_addr.S_un.S_addr,  ptrHostEnt->h_addr_list[0],
sizeof(unsigned long));
```

Create the server socket

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(serverSocket == INVALID_SOCKET)
{
        … … …
        WSACleanup();
        return 1;
}
```

Bind the socket

```
if(bind(serverSocket, (struct sockaddr *)&serverSocketAddress,
sizeof(serverSocketAddress)))
{
        … … …
        WSACleanup();
        return 1;
}
```

Put the socket in listening mode

```
if(listen(serverSocket, MAX_PENDING_CONNECTIONS))
{
        … … …
        WSACleanup();
        return 1;
}
```

Here is the time to accept client connections

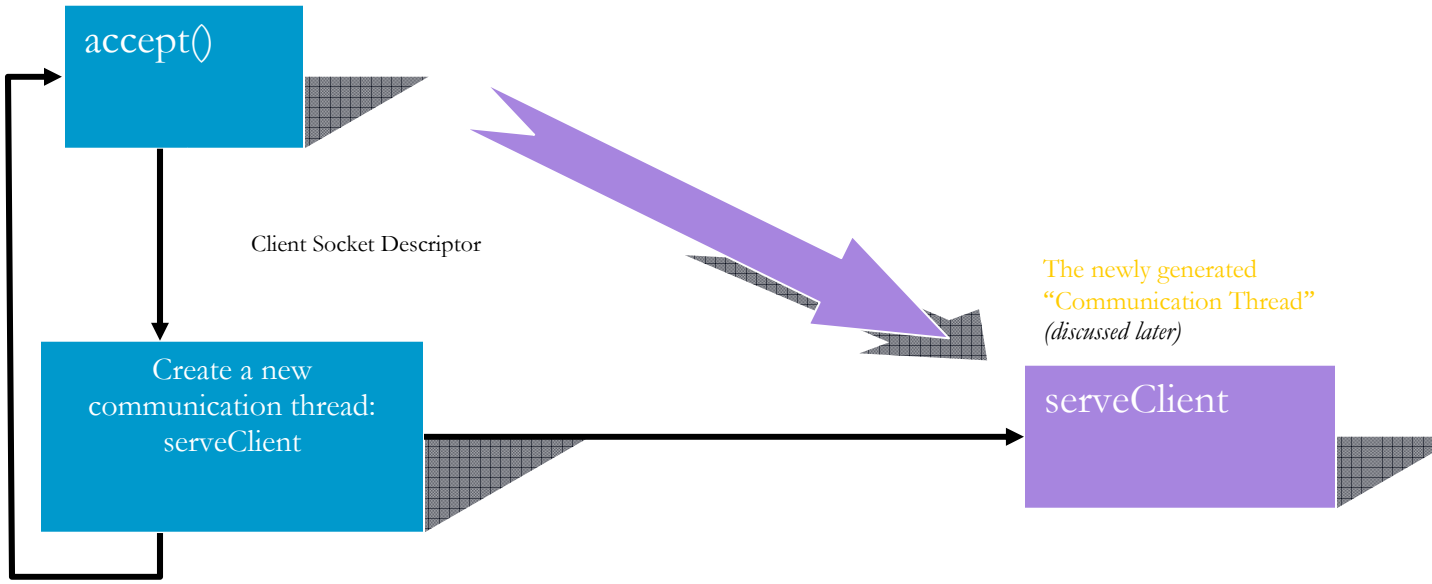
Create a thread that will call accept() in a loop to accept multiple client connections

```
hAcceptingThread = CreateThread(
                NULL,
                0,
                (LPTHREAD_START_ROUTINE)
                        acceptClientConnections,
                NULL,
                CREATE_SUSPENDED,
                &dwAcceptingThread);
```

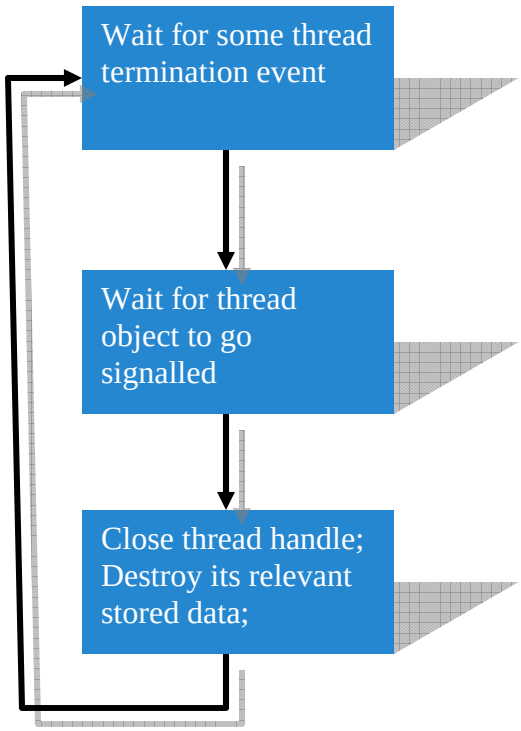
Create a thread to do termination house-keeping when some communication thread terminates.

```
hTerminatingThread = CreateThread(
        NULL,
        0,
        (LPTHREAD_START_ROUTINE)
                terminateCommunicationThreads,
        NULL,
        CREATE_SUSPENDED,
        &dwTerminatingTThread);
```

Accept Client Connection (*Thread Routine*)



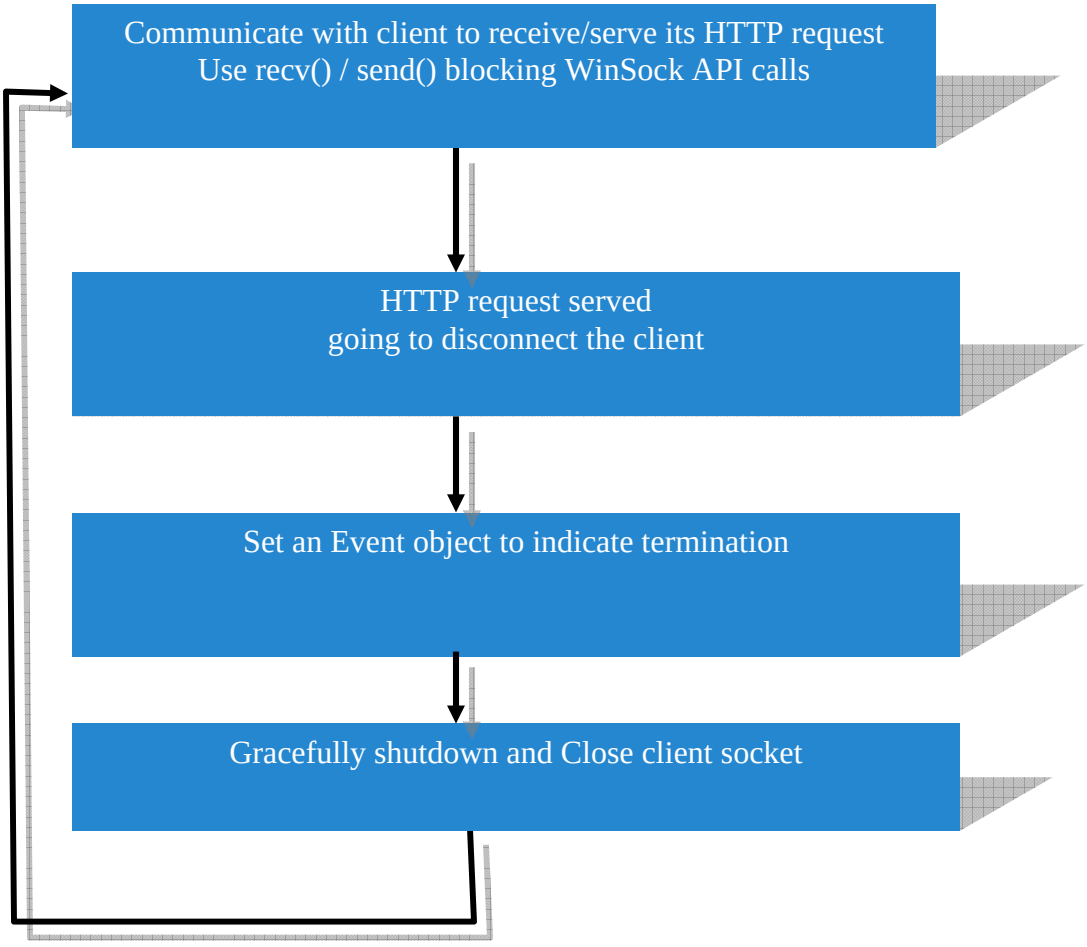
Terminate communication threads (thread routine)



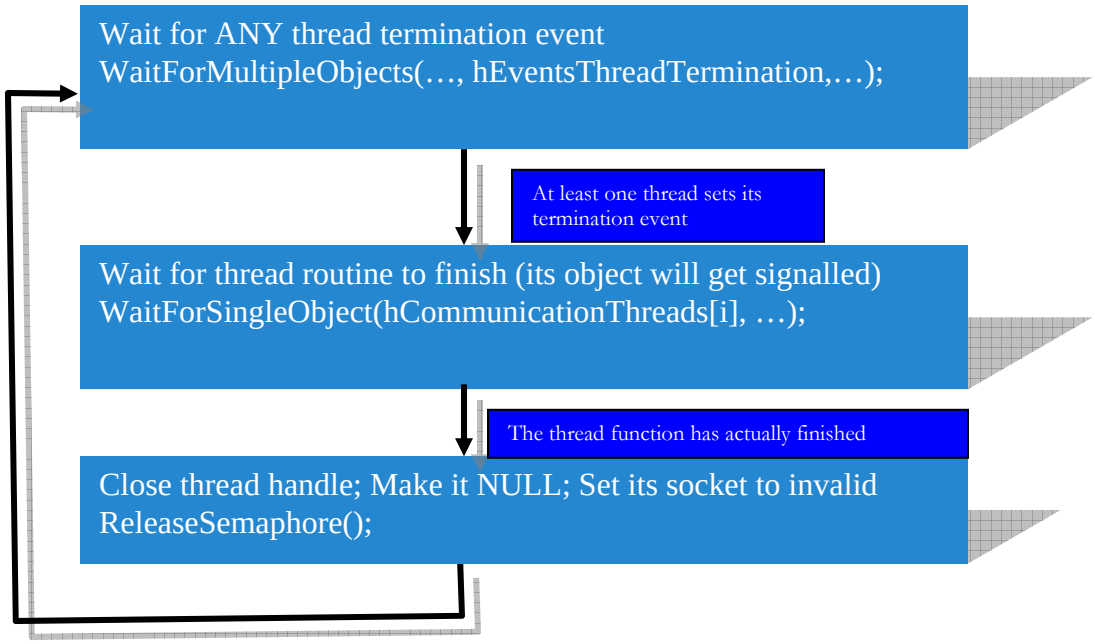
Application Variables and constants

```
#define MAX_CLIENTS       5
SOCKET clientSockets[MAX_CLIENTS];

HANDLE hCommunicationThreads[MAX_CLIENTS];
DWORD dwCommunicationThreads[MAX_CLIENTS];

HANDLE hAcceptingThread;
DWORD dwAcceptingThread;

HANDLE hTerminatingThread;
DWORD dwTerminatingTThread;
```

servClient Communication thread routine



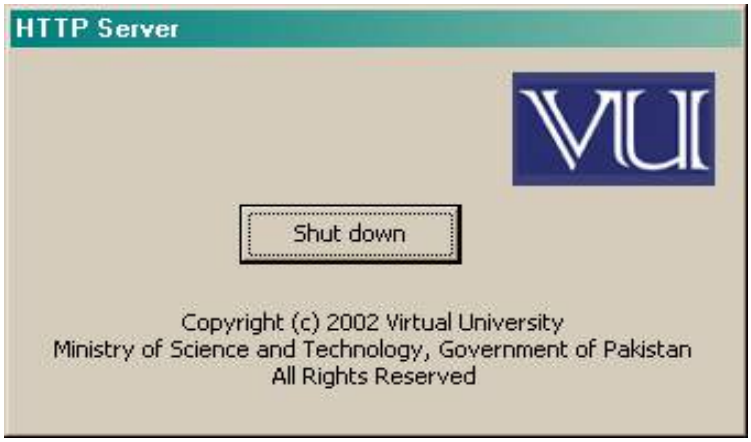
***terminateCommunicationThreads* thread routine**



Thread Procedures Summary

acceptClientConnections
- to accept client connection
•terminateCommunicationThreads
• - to do housekeeping when communication threads terminate
•serveClient
- to do actual communication to receive and serve an HTTP request

30.1 Server Shut down user interface



30.3 Variable Initialization

```
for(i=0; i<MAX_CLIENTS; ++i)
{
  clientSockets[i] = INVALID_SOCKET;

  hCommunicationThreads[i] = NULL;
  dwCommunicationThreads[i] = 0;
  hEventsThreadTermination[i] = NULL;
}
```

30.4 Initialize WinSock Library

```
if(WSAStartup(MAKEWORD(1,1), &wsaData))
{
        MessageBox(NULL,
        "Error initialising sockets library.",
        "WinSock Error",
        MB_OK | MB_ICONSTOP);
        return 1;
}
```

30.5 Win32 Error Codes

int WSAGetLastError(void);
- get error code for the last unsuccessful Windows Sockets operation

DWORD GetLastError(VOID);
- retrieve calling threads last-error code

30.6 HTTP Web Server Application

Get machine's hostname and IP address

```
gethostname(hostName, sizeof(hostName));
ptrHostEnt = gethostbyname(hostName);
Fill the socket address with appropriate values
serverSocketAddress.sin_family = AF_INET;
serverSocketAddress.sin_port = htons(SERVER_PORT);
… … …
memcpy(&serverSocketAddress.sin_addr.S_un.S_addr,
ptrHostEnt->h_addr_list[0], sizeof(unsigned long));
```

Create the server socket

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(serverSocket == INVALID_SOCKET)
{
   … … …
   WSACleanup();
   return 1;
}
```

Bind the socket

```
if(bind(serverSocket,(struct sockaddr *)&serverSocketAddress,
sizeof(serverSocketAddress)))
{
  … … …
 WSACleanup();
 return 1;
}
```

Put the socket in listening mode

```
if(listen(serverSocket, MAX_PENDING_CONNECTIONS))
{
      … … …
     SACleanup();
     return 1;
}
```

Here is the time to accept client connections

Limiting Maximum Concurrent connections

Create an unnamed semaphore object with MAX_CLIENTS as initial/maximum count

```
hSemaphoreMaxClients = CreateSemaphore(NULL,
MAX_CLIENTS,
MAX_CLIENTS, NULL
);
```

“I am dying…”, the thread said

Create an array of non-signalled event objects

```
for(i=0; i<MAX_CLIENTS; i++)
```

```
hEventsThreadTermination[i] =  CreateEvent(NULL, FALSE, FALSE, NULL);
```

Create the connection-accepting thread

```
hAcceptingThread = CreateThread( NULL, 0, (LPTHREAD_START_ROUTINE)
acceptClientConnections, NULL, CREATE_SUSPENDED, &dwAcceptingThread);
```

Create the termination house-keeping thread

```
hTerminatingThread = CreateThread(… … …);
```

Display the dialog

```
DialogBox(…, …, …, mainDialogProc);
```

Main Dialog Proc

```
case WM_INITDIALOG:
ResumeThread(hAcceptingThread);
ResumeThread(hTerminatingThread);
return TRUE;
break;
```

Handling the server shut-down button

```
case IDC_BUTTON_SHUTDOWN:
//Perform any shut-down tasks that my be necessary
EndDialog(hDlg, 0);
break;
```

Accept Client Connections Thread Routine

Start of the loop to accept client connections Wait for semaphore count to go non-zero

```
dwWaitResult = WaitForSingleObject(
hSemaphoreMaxClients, INFINITE);

switch(dwWaitResult)
{
case WAIT_OBJECT_0:
We can accept more connections here because  semaphore object is signaled
clientSocket = accept(… … …);

clientSocket = accept(… … …);
```

Connection accepted! Look for the first empty slot to save the new socket descriptor

```
for(i=0; i<MAX_CLIENTS; i++)
{
  if(clientSockets[i] == INVALID_SOCKET)
  break;
}

nextClientIndex = i;
clientSockets[nextClientIndex]=clientSocket;
```

nextClientIndex is used as an index in ALL arrays to store information relevant to this new client connection

```
clientSockets[nextClientIndex]=clientSocket;

hCommunicationThreads[nextClientIndex] = CreateThread(…, …, serveClient,
//thread procedure
(LPVOID)nextClientIndex, thread parameter
CREATE_SUSPENDED,
…);
```

Index for this client in all arrays is passed to this thread routine

```
DWORD WINAPI serveClient(LPVOID clientNumber)
{
char msg[2046] = "";

Receiving an HTTP request from browser
recv( clientSockets[(UINT)clientNumber], msg,2046,0);

//nextClientIndex is used as an index in ALL arrays to store information relevant to this
//new client connection

clientSockets[nextClientIndex]=clientSocket;

hCommunicationThreads[nextClientIndex] = CreateThread(…, …, serveClient,
(LPVOID)nextClientIndex,thread parameter, CREATE_SUSPENDED,…);
```

Sample Request

Request parsing: understanding what the client has demanded GET /courses/win32.html HTTP/1.0
Assume F:\ is your server's home directory, and \courses\is not a virtual directory, server should return the file
F:\courses\win32.html

HTTP Redirection
Redirecting the client irrespective of the HTTP request!

The string in the #define directive is assumed to be on a single line
#define RESPONSE
"HTTP/1.1 302 Object Moved\r\n
Location: http://www.vu.edu.pk\r\n\r\n"

Sending the hard-coded HTTP response back to browser

```
send(clientSockets[(UINT)clientNumber],
RESPONSE,
sizeof(RESPONSE),
0);
```

Using Port Numbers

There is no compulsion to build all HTTP Web Servers to run on port 80. These are 'suggested' port numbers for a Win32 developer
Standard servers do run on port 80. Our HTTP Web Server may also need to run on port 80 if put it to public use

Returning HTML Document

#define directive is assumed to be on a single line
#define RESPONSE "HTTP/1.0 200 OK\r\n

Content-type: text/html\r\n
Content-length: 1325\r\n\r\n"
Send the hard-coded HTTP status and headers

```
send(clientSockets[(UINT)clientNumber], RESPONSE, sizeof(RESPONSE), 0);
//Now sends the whole file using character I/O of standard C runtime
ch = fgetc(fptr);
while(!feof(fptr))  {
send(clientSockets[(UINT)clientNumber], &ch, 1, 0);
ch = fgetc(fptr);
}
```

***terminateCommunicationThreads* thread routine**

Wait for some thread to set a termination event

```
dwWaitResult = WaitForMultipleObjects(MAX_CLIENTS, hEventsThreadTermination,
FALSE, INFINITE);
//Get the array index
```

```
threadIndex = dwWaitResult - WAIT_OBJECT_0;
//Wait for the thread to actually terminate

WaitForSingleObject( hCommunicationThreads[threadIndex], INFINITE);
Close handles and set variables to initial values again

CloseHandle(hCommunicationThreads[threadIndex]);
hCommunicationThreads[threadIndex] = NULL;
clientSockets[threadIndex] = INVALID_SOCKET;

//Resource freed, increase the semaphore value

ReleaseSemaphore(hSemaphoreMaxClients, 1, NULL);
```

A Flawed Web Server

Fixed sized arrays waste memory and lack run-time flexibility One event per thread to signify termination: *WaitForMultipleObjects* cannot wait on more than a certain number of objects e.g. 64 on x86 under NT.

Dynamic Web Content

Server blindly dumps HTML files to the clients. This is ‘static content’.

Server reads file and modifies its output e.g.

%%time%% replaced with current system time
Every 2 clients connected at different instants of time will receive different content.
This is ‘dynamic content’.
%%time%% may be called a tag

Microsoft Active Server Pages
Macromedia ColdFusion
Tags are not sent to the client. These are processed by the server and the resulting output is sent to the browser.

CGI

CGI is Common Gateway Interface. Win32 executable execute by the server. All browser request data is available at stdin (read using scanf() etc.) and all output sent to stdout (output using printf etc.) is sent to the browser instead of the server screen.

Summary

In this lecture, we designed a web server which listens on port 80 and can receive requests from the clients and send message to the client. Our server supports maximum five clients at a time.

Note: For more on Windows Programming, connect to the Virtual University resource online. Examples, source codes can be found online.

Exercises

1. Practice to create such applications as explained in this lecture and in previous lectures with different ideas.