Important Instructions to examiners:

1) The answers should be examined by key words and not as word-to-word as given in themodel answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may tryto assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given moreImportance (Not applicable for subject English and Communication Skills).
4) While assessing figures, examiner may give credit for principal components indicated in thefigure. The figures drawn by candidate and model answer may vary. The examiner may givecredit for anyequivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constantvalues may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgement on part of examiner of relevantanswer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalentconcept.

1. a) Attempt any three of the following: Marks 12

a) Write any four function for string operation.

(Any four function-1 Mark Each)

Ans: String functions defined in windows calculate string lengths, copy strings, concatenate string and compare strings.:-

```
ILength=Lstrlen(pString)
pString=Lstrcpy (pString1, pString2);
pString=Lstrcpun(pString1, pString2,iCount);
pString=Lstrcat (pString1, pString2);
iComp=Lstrcmp(pString1, pString2);
iComp=Lstrcmpi(pString1, pString2);
```

These work much the same as their C library equivalents. They accept wide-character strings if the UNICODE identifier is defined and regular strings if not.

b) Explain GDI object.

(Any four objects-1 Mark each)

Ans:

The six GDI objects are pens, brushes, bitmaps, regions, fonts, and palettes. Except for palettes, all of these objects are selected into the device context using Select Object.

(1) **Bitmap:** A bitmap is an array of bits in which one or more bits correspond to each display pixel. They can be used bitmaps to represent images and can use them to create brushes

(2) **Bush:** A brush defines a bit mapped patterns of pixels that is used to fill areas with color.

(3) **Font:** A font is a complete collection of characters of a particular type face and a particular size. Fonts arc generally stored on disk as resources and some are device specific

(4) **Palette:** A palette is a color mapping interface that allows an application to take full advantage of the color capability of an output device without interfacing with other applications.

c) Explain any four caret functions.

(Any Four caret function-1 Mark each)

Ans: When you type text into a program, generally a little underline, vertical bar, or box called as Caret (Not the mouse Cursor) shows you where the next character you type will appear on the screen.

- *CreateCaret* Creates a caret associated with a window.
- *SetCaretPos* Sets the position of the caret within the window.
- *ShowCaret* Shows the caret.
- *HideCaret* Hides the caret.
- *DestroyCaret* Destroys the caret.

CreateCaret Creates a caret associated with a window.

SetCaretPos Sets the position of the caret within the window.

ShowCaret Shows the caret.

HideCaret Hides the caret.

DestroyCaret Destroys the caret.

```
case WM_SETFOCUS:
    // create and show the caret
     CreateCaret (hwnd, NULL, cxChar, cyChar);
            SetCaretPos (xCaret * cxChar, yCaret * cyChar);
            ShowCaret (hwnd);
```

```
case WM_KILLFOCUS:
        // hide and destroy the caret
         HideCaret(hwnd);
         DestroyCaret();
```

There are also functions to get the current caret position (GetCaretPos) and to get and set the caret blink time (GetCaretBlinkTime and SetCaretBlinkTime).

d) Explain alternate capture solution.

(*Question not clear. For any appropriate legible answer 4 Marks to be given.)*

Ans: In a normally implemented Windows drawing program involving mouse capture one may begin by depressing the left mouse button to indicate one corner of a rectangle and then drag the mouse. The program draws an outlined rectangle with the opposite corner at the current mouse position. When one releases the mouse, the program fills in the rectangle. But when we press the left mouse button within client area and then move the cursor outside the window, the program stops receiving WM_MOUSEMOVE messages. Now release the button. The program doesn't get that WM_BUTTONUP message because the cursor is outside the client area. Move the cursor back within client area. The window procedure still thinks the button is pressed. This is not good. The program doesn't know what's going on.

This is the type of problem for which mouse capturing was invented. If the user is dragging the mouse, it should be no big deal if the cursor drifts out of the window for a moment. The program should still be in control of the mouse.

Capturing the mouse is easier than baiting a mousetrap. You need only call

```
SetCapture (hwnd) ;
```

After this function call Windows sends all mouse messages to the window procedure for the window whose handle is hwnd. The mouse messages always come through as client-area messages, even when the mouse is in a nonclient area of the window. The lParam parameter still indicates the position of the mouse in client-area coordinates. These x and y coordinates,

however, can be negative if the mouse is to the left of or above the client area. When you want to release the mouse, call

ReleaseCapture () ;

which will returns things to normal.

In the 32-bit versions of Windows, mouse capturing is a bit more restrictive than it was in earlier versions of Windows. Specifically, if the mouse has been captured, and if a mouse button is not currently down, and if the mouse cursor passes over another window, the window underneath the cursor will receive the mouse messages rather than the window that captured the mouse. This is necessary to prevent one program from messing up the whole system by capturing the mouse and not releasing it.

To avoid problems, your program should capture the mouse only when the button is depressed in your client area. You should release the capture when the button is released.

b) Attempt <u>any one</u> of the following: Marks 6

a) What is device context? Explain how to obtain a handle for DC.

(Description of DC- 3 Marks, How to Obtain- 3 Marks)

Ans: When you want to draw on a graphics output device such as the screen or printer, you must first obtain a handle to a device context (or DC). In giving your program this handle, Windows is giving you permission to use the device. You then include the handle as an argument to the GDI functions to identify to Windows the device on which you wish to draw.

The device context contains many "attributes" that determine how the GDI functions work on the device. Ex: font isn't specified in TextOut function.

Five attributes of the device context affect the appearance of lines. i.e. Current pen position, pen, background mode, background color and drawing mode.

Handle to device context is object in two ways depending upon whether WM_PAINT message is being processed or some other message is being processed. The functions involved are BeginPaint, EndPaint, GetDC, ReleaseDC.

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
        HDC hdc;
        .
        .
  switch (message)
  {
  case WM_CREATE:
                hdc = GetDC(hwnd);
                .
                .//functions that use DC
                .
    ReleaseDC(hwnd, hdc);
    return 0;

  case WM_PAINT :
    hdc = BeginPaint(hwnd, &ps);
                .
                .//functions that use DC
                .
    EndPaint (hwnd, &ps);
    return 0;
      .
      .//Process other messages
      .
      }
      //Call the default window procedure for unprocessed messages
      return DefWindowProc (hwnd, message, wParam, lParam);
      }
```

b) State and explain GDI primitives.

(State- 2 Marks and primitives-1 Mark each)

Ans: The types of graphics to be display on the screen or the printer can themselves be divided into several categories. which are called primitives

GDI primitives:

(1) **Lines and curves:** Line are the foundation of an:- vector graphics drawing system. GDI supports straight lines, rectangles ellipses

(2) **Filled areas:** A series or lines or curves encloses an area. we can use the enclose area for filling with the current GDI brush object.

(3) **Bitmaps:** A bitmap is a rectangular array of bits that correspond 10 the pixels of a display device.

(4) **Text:** Text is not quite represented as mathematical equation as other aspects of computer graphics like polyline, curves etc.

2. Attempt any four of the following: Marks 16

a) What is difference between O.S. and API?

(Any 4 points- 1 Mark each)

Ans: API is set of function calls whereas OS is the interface between the user and the computer. OS makes use of the API to get its work done.

Any operating system has a constructively defined set of API (Application Program interface). An API consist all the function calls that an application program can make of an operating system, as well as definitions of associated data types and structures.

A Windows programmer with experience in Windows 98 would find the source code for a Windows l.0 program very familiar. One way the API has changed, has been in enhancements. Windows 1.0 supported fewer than 450 function calls; today there are thousands many more API

programs. The biggest change in the Windows API and its syntax came about during the switch from a 16-bit architecture to a 32·bit architecture, Versions 1.0 through 3.1 or Windows used the so-called segmented memory mode of the 16-bit, Intel 8086, 8088, and 286 microprocessors, a mode that was also supported for compatibility purposes in the 32-hit Intel microprocessors beginning with the 386.

The microprocessor register size in this mode was 16 bits, and hence the C int data type was also 16 bits wide. In the segmented memory model, memory addresses were formed from two components a 16-bit segment pointer and a 16-bit offset pointer. From the programmer's perspective, this was quite messy and involved differentiating between long, or far, pointers and short, or near pointers. Beginning in Windows NT and Windows 95, Windows supported a 32-bit flat memory model using the 32-bit modes of the Intel 386, 486, and Pentium processors.

The C int data type was promoted to a 32-bit value. Programs written for 32-bit versions of Windows use simple 32-bit pointer values that address a flat linear address space. The API for the 16-bit versions for Windows (Windows 1.0 through Windows 3.1) is now known as Win l6. The API for the 32-bit versions of Windows (Windows 95, Windows 98, and all versions of Windows NT) is now known as Win32.

b) How GDI function call can be classified?

(For four classes-4 Marks)

Ans: The several hundred function calls that comprise GDI can be classified in several broad groups:

- Functions that get (or create) and release (or destroy) a device context As we saw in earlier chapters, you need a handle to a device context in order to draw. The BeginPaint and EndPaint functions (although technically a part of the USER module rather than the GDI module) let you do this during the WM_PAINT message, and GetDC and ReleaseDC functions let you do this during other messages. We'll examine some other functions regarding device contexts shortly.

- Functions that obtain information about the device context The GetTextMetrics function to obtain information about the dimensions of the font currently selected in the device context.
- Functions that draw something we use the TextOut function to display some text in the client area of the window. Other GDI functions let us draw lines and filled areas.

- Functions that set and get attributes of the device context An "attribute" of the device context determines various details regarding how the drawing functions work. For example, you can use SetTextColor to specify the color of any text you draw using TextOut or other text output functions. We used SetTextAlign to tell GDI that the starting position of the text string in the TextOut function should be the right side of the string rather than the left, which is the default. All attributes of the device context have default values that are set when the device context is obtained. For all Set functions, there are Get functions that let you obtain the current device context attributes.
- Functions that work with GDI "objects" Here's where GDI gets a bit messy. First an example: By default, any lines you draw using GDI are solid and of a standard width. You may wish to draw thicker lines or use lines composed of a series of dots or dashes. The line width and this line style are not attributes of the device context. Instead, they are characteristics of a "logical pen." You can think of a pen as a collection of bundled attributes. You create a logical pen by specifying these characteristics in the CreatePen, CreatePenIndirect, or ExtCreatePen function. Although these functions are considered to be part of GDI, unlike most GDI functions they do not require a handle to a device context. The functions return a handle to a logical pen. To use this pen, you "select" the pen handle into the device context. The current pen selected in the device context is considered an attribute of the device context. From then on, whatever lines you draw use this pen. Later on, you deselect the pen object from the device context and destroy the object. Destroying the pen is necessary because the pen definition

  occupies allocated memory space. Besides pens, you also use GDI objects for creating brushes that fill enclosed areas, for fonts, for bitmaps, and for other aspects of GDI.

c) State the foreign language keyboard problem.

(*For the following answer-4 Marks)*

Ans: The Foreign-Language Keyboard Problem refers to the problem where keyboard layout changing is not supported by GDI leading to incorrect symbols being generated when keys are pressed.

Example: A program may display incorrect characters—when running the English version of Windows with the Russian or Greek keyboard layouts installed, when running the Greek

version of Windows with the Russian or German keyboard layouts installed, and when running the Russian version of Windows with the German, Russian, or Greek keyboards installed.

d) Explain the process of Emulating the mouse with the keyboard.
(*For the following answer-4 Marks*)

Ans: Emulating the mouse with the keyboard.

1. A keyboard interface is attached to the program to winproc procedure.
2. Adding a keyboard interface to a program that uses the mouse cursor for pointing purposes require that it must take care about displaying and moving the mouse cursor.
3. Even if a mouse device is not installed , windows can still display the mouse cursor. Windows maintains something called “display count” for this cursor .
4. If a mouse is installed , the display count is initially 0, if not, the display count is initially -1.
5. The mouse cursor is diapl;ayed only if the display count is non-negative . the display count is incremented by calling ShowCursor(TRUE); and decremented by calling ShowCursor(FALSE);.
6. The mouse is installed or not is not determined without using ShowCursor command.
7. The cursor position is obtained by calling GetCursorPos(&pt);where pt is the point structure.
8. The cursor position is set by SetCursorPos(x,y); in both x and y are screen co-ordinates and not the client co-ordinates. The lparam parameter in WM_KEYDOWN indicates that the key is pressed.

e) Write down the function to change button text and button colors.
(For function name- 1 Mark and for syntax-1 Mark)

Ans: You can change the text in a button (or in any other window) by calling SetWindowText:

- SetWindowText (hwnd, pszString) ;
- where hwnd is a handle to the window whose text is being changed and pszString is a pointer to a null-terminated string. For a normal window, this text is the text of the caption bar. For a button control, it's the text displayed with the button.

f) Explain radio button and group box.
(*Radio button-2 Marks and Group box-2 Marks)*

Ans: Radio buttons:

1. Radio buttons do not work as toggles that is when clicked on, a radio button a second time, its state remains unchanged .
2. The radio button looks very much like a check box except that it contains a little circle rather than a box. A heavy dot within the circle indicates that the radio button has been checked.
3. The radio button has the window style BS_RADIOBUTTON or BS_AUTORADIOBUTTON, but the later is used only in dialog boxes. When receiving a WM _ COMMAND message from a radio button, it should display its check by sending it a BM_SETCHECK message with wParam equal to 1 :

   SendMessage (hwndButton, BM_SETCHECK, 1, 0);
4. For all other radio buttons in the same group, it can tum off the. checks by sending them BM_SETCHECK messages with wParam equal to 0 :

   SendMessage (hwndButton, BM_SETCHECK, 0, 0);

Group boxes:

1. The group box, which has the BS_GROUPBOX style, is all oddity ill the button class.
2. 11 neither processes mouse or keyboard input nor send WM COMMAND messages to its parent.
3. The group box is a rectangular outline with its window text at the top. Group boxes are often used to enclose other button controls.

3. Attempt any four of the following: Marks 16

a) How to clip with Rectangles and Regions?

(*For the following answer-4 Marks*)

Ans: You can use regions for drawing or for clipping.

You use a region for clipping (that is, restricting drawing to a specific part of your client area) by selecting the region into the device context.

Clipping with Rectangles and Regions

The InvalidateRect function invalidates a rectangular area of the display and generates a WM_PAINT message. For example, you can use the InvalidateRect function to erase the client area and generate a WM_PAINT message:

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

You can obtain the coordinates of the invalid rectangle by calling GetUpdateRect, and you can validate a rectangle of the client area using the ValidateRect function.

When you receive a WM_PAINT message, the coordinates of the invalid rectangle are available from the PAINTSTRUCT structure that is filled in by the BeginPaint function. This invalid rectangle also defines a "clipping region." You cannot paint outside the clipping region.

Windows has two functions similar to InvalidateRect and ValidateRect that work with regions rather than rectangles:

```
InvalidateRgn (hwnd, hRgn, bErase) ;
```

and

```
ValidateRgn (hwnd, hRgn) ;
```

When you receive a WM_PAINT message as a result of an invalid region, the clipping region will not necessarily be rectangular in shape.

You can create a clipping region of your own by selecting a region into the device context using either

```
SelectObject (hdc, hRgn) ;
```

or

```
SelectClipRgn (hdc, hRgn) ;
```

A clipping region is assumed to be measured in device coordinates.

GDI makes a copy of the clipping region, so you can delete the region object after you select it in the device context. Windows also includes several functions to manipulate this clipping region, such as ExcludeClipRect to exclude a rectangle from the clipping region, IntersectClipRect to create a new clipping region that is the intersection of the previous clipping region and a rectangle, and OffsetClipRgn to move a clipping region to another part of the client area.

b) Write a program to print a "Hello World" in C++.

(*For the following answer-4 Marks*)

Ans:

```
#include <windows.h>

//# indicates directive

//Directives are direct instructions to the compiler

//Macros are also directives

//Directives are executed before compilation

//include tells VC++ that the program has identifiers defined in the header file

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)

//WinMain is like main() of C or C++. Also called as program entry point.

//Its the starting point of execution of windows program

//It is called by the operating system along with arguments

//Its prototype is defined in winbase.h

//WINAPI refers to calling convention. It is a macro. It is defined in windef.h

//When WINAPI means _stdcall (standard calling convention) the following holds:-

/*

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

*/

//HINSTANCE is the name of user defined type and hInstance is variable

//h prefix means handle which is a 32-bit pointer used for reference. Its type is normally void *

//Instance refers to the application that is running

//hInstance contains address of memory where application data and functions are stored

//hPrevInstance is null in Win2000 and above otherwise it contains handle to previous instance of the application

//PSTR is pointer to string (char *).

//sz stands for String terminated with Zero

//szCmdLine contains command line arguments without the exe name (command with arguments can be specified using

// Start->Run option or from the dos prompt)

//iCmdShow contains number that indicates whether window is to be shown in maximized, minimized or restored form

//Its value can be SW_MAXIMIZE, SW_MINIMIZE or SW_RESTORE defined in winuser.h

{

MessageBox (NULL, "Hello World!", "My first program...", MB_ICONINFORMATION | MB_OK);

//(handle to owner window, text in message box, message box title, message box style i.e. icon and buttons)

//| is a binary or operator

return 0;

}

c) State the importance of system and non-system keystroke.

(*System keystroke – 2 Marks, Non system keystroke – 2 Marks*)

Ans: System Keystrokes are usually generated for keys typed in combination with the Alt key.

These keystrokes invoke options on the program's menu or system menu, or they are used for system functions such as switching the active window (Alt-Tab or Alt-Esc) or for system menu accelerators (Alt in combination with a function key such as Alt-F4 to close an application).

Programs usually ignore the WM_SYSKEYUP and WM_SYSKEYDOWN messages and pass them to DefWindowProc.

For all four keystroke messages, wParam is a virtual key code that identifies the key being pressed or released and lParam contains other data pertaining to the keystroke.

Virtual Key code identifies the key being pressed or released.

The virtual key codes you use most often have names beginning with VK_ defined in the WINUSER.H header file.

Ex:

VK_CANCEL for Ctrl+Break.

VK_LBUTTON for mouse left button.

VK_SHIFT for the shift key.

d) How will you use child window for hit testing?

(*For the following answer-4 Marks*)

Ans: Some programs (for example, the Windows Paint program) divide the client area into several smaller logical areas.

The Paint program has an area at the left for its icon-based tool menu and an area at the bottom for the color menu.

When Paint hit-tests these two areas, it must take into account the location of the smaller area within the entire client area before determining the actual item being selected by the user.

In reality, Paint simplifies both the drawing and hit-testing of these smaller areas through the use of "child windows."

The child windows divide the entire client area into several smaller rectangular regions.

Each child window has its own window handle, window procedure, and client area.

Each child window procedure receives mouse messages that apply only to its own window.

The lParam parameter in the mouse message contains coordinates relative to the upper left corner of the client area of the child window, not relative to the client area of the "parent" window.

Child windows used in this way can help you structure and modularize your programs.

If the child windows use different window classes, each child window can have its own window procedure.

The different window classes can also define different background colors and different default cursors.

d) List the types of scrollbars and what is the difference between them.

(List -2 Marks, Difference- 2 Marks)

Ans: There is a difference between "window scroll bars" and "scroll bar controls." Window scroll bars, appear at the right and bottom of the window. You add window scroll bars to a window by including the identifier WS_VSCROLL or WS_HSCROLL or both in the window style when creating the window.

Scroll bar controls, are child windows that can appear anywhere in the client area of the parent window. You create child window scroll bar controls by using the predefined window class "scrollbar" and one of the two scroll bar styles SBS_VERT and SBS_HORZ.

Unlike the button controls (and the edit and list box controls to be discussed later), scroll bar controls do not send WM_COMMAND messages to the parent window. Instead, they send WM_VSCROLL and WM_HSCROLL messages, just like window scroll bars. When processing the scroll bar messages, you can differentiate between window scroll bars and scroll bar controls by the lParam parameter. It will be 0 for window scroll bars and the scroll bar window handle for scroll bar controls. The high and low words of the wParam parameter have the same meaning for window scroll bars and scroll bar controls.

Unlike window scroll bars which have a fixed width, Windows uses the full rectangle dimensions given in the CreateWindow call (or later in the MoveWindow call) to size the scroll bar controls.

4 A) Attempt <u>any three</u> of the following Marks 12

a) Explain message box function.
(*For the following answer-4 Marks*)

Ans: The MessageBox function creates, displays, and operates a message box.

The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

MessageBox (NULL, "Hello World!", "My first program...", MB_ICONINFORMATION | MB_OK);

//(handle to owner window, text in message box, message box title, message box style i.e. icon and buttons)

///| is a binary or operator

Value	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_CANCELTRYCONTINUE	Windows 2000: The message box contains three push buttons: Cancel, Try Again, Continue.Use this message box type instead of MB_ABORTRETRYIGNORE.
MB_HELP	Windows 95/98, Windows NT 4.0 and later: Adds a Help button to the message box.When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner.
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.
MB_ICONEXCLAMATION, MB_ICONWARNING	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION, MB_ICONASTERISK	An icon consisting of a lowercase letter i in a circle appears in the message box.

MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	A stop-sign icon appears in the message box.

If the function succeeds, the return value is one of the following menu-item values.

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDTRYAGAIN	Try Again button was selected.
IDYES	Yes button was selected.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

b) Why queue is used for keystroke messages?
(*For the following answer-4 Marks*)

Ans: Windows and the keyboard device driver translate the hardware scan codes into formatted messages. However, these messages are not placed in an application's message queue right away. Instead, Windows stores these messages in something called the system message queue. The system message queue is a single message queue maintained by Windows specifically for the preliminary storage of user input from the keyboard and the mouse. Windows will take the next message from the system message queue and place it in an application's message queue only when a Windows application has finished processing a previous input.

The reasons for this two-step process—storing messages first in the system message queue and then passing them to the application message queue—involves synchronization. As we just learned, the window that is supposed to receive keyboard input is the window with the input focus. A user can be typing faster than an application can handle the keystrokes, and a particular keystroke might have the effect of switching focus from one window to another. Subsequent keystrokes should then go to another window.

c) What are the contents of IParam and WParam in case of client mouse message?
(Iparam – 2 Marks, Wparam – 2 Marks)

Ans: The value of lParam contains the position of the mouse.

The low word is the x-coordinate, and the high word is the y-coordinate relative to the upper left corner of the client area of the window.

x = LOWORD (lParam) ;

y = HIWORD (lParam) ;

The value of wParam indicates the state of the mouse buttons and the Shift and Ctrl keys.

To test wParam use these bit masks:

MK_LBUTTON	Left button is down
MK_MBUTTON	Middle button is down
MK_RBUTTON	Right button is down

MK_SHIFT Shift key is down

MK_CONTROL Ctrl key is down

Example: for WM_LBUTTONDOWN message, if the value

wparam & MK_SHIFT is TRUE (nonzero), then the Shift key was down when the left button was pressed.

d) Explain "Window Procedure" with an example.
(*For the following answer-4 Marks*)

Ans: This is the place where real action occurs.

This procedure determines what is displayed and how inputs are handled.

This procedure is generally not called directly.

It can have switch case structure.

```
#include <windows.h>
//# indicates directive
//Directives are direct instructions to the compiler
//Macros are also directives
//Directives are executed before compilation
//include tells VC++ that the program has identifiers defined in the header file

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
//This is prototype declaration or forward declaration
//It is the first line of function definition with semicolon
//Tells VC++ to look for the function definition anywhere in the program
```

//Without prototype; calling a function before definition will result in "function not defined" compiler error

//WndProc function is defined after definition of WinMain

const char g_szClassName[] = "myWindowClass";

//This is our window class name that will be registered and used in CreateWindowEx function

//It is an array of char i.e. a string

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)

//WinMain is like main() of C or C++

//Its the starting point of execution of windows program

//It is called by the operating system along with arguments

//Its prototype is defined in winbase.h

//WINAPI refers to calling convention. It is a macro. It is defined in windef.h

//When WINAPI means _stdcall (standard calling convention) the following holds:-

/*

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

*/

//HINSTANCE is the name of user defined type and hInstance is variable

//h prefix means handle which is a 32-bit pointer used for reference. Its type is normally void *

//Instance refers to the application that is running

//hInstance contains address of memory where application data and functions are stored

//hPrevInstance is null in Win2000 and above otherwise it contains handle to previous instance of the application

//PSTR is pointer to string (char *).

//sz stands for String terminated with Zero

//szCmdLine contains command line arguments without the exe name (command with arguments can be specified using

// Start->Run option or from the dos prompt)

//iCmdShow contains number that indicates whether window is to be shown in maximized, minimized or restored form

//Its value can be SW_MAXIMIZE, SW_MINIMIZE or SW_RESTORE defined in winuser.h

{

WNDCLASSEX wc;

//WNDCLASSEX structure contains window class information

HWND hwnd;

//Variable that will store handle to window

MSG msg;

//Variable to contain message information

wc.cbSize = sizeof(WNDCLASSEX);

//cbSize should contain the size of the structure itself

wc.style = 0;

//style specifies the class style

wc.lpfnWndProc = WndProc;

//WndProc is pointer to WndProc() function

wc.cbClsExtra = 0;

//Specifies the number of extra bytes to allocate following the window-class structure

wc.cbWndExtra = 0;

//Specifies the number of extra bytes to allocate following the window instance

wc.hInstance = hInstance;

//Handle to application instance

wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);

//LoadIcon function gives handle to the icon that the int const IDI_APPLICATION represents

//hIcon member of wc refers to the icon shown when Alt + Tab is pressed

//First parameter of LoadIcon = 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

wc.hCursor = LoadCursor (NULL, IDC_CROSS);

//First parameter = 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

//Cursor refers to mouse pointer

wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);

//Handle to the class background brush. This member can be a handle to the physical brush to be used for

// painting the background, or it can be a color value. A color value must be one of the following standard

// system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert

// it to HBRUSH type

```
wc.lpszMenuName = NULL;

//Pointer to a null-terminated character string that specifies the resource name of the class menu

wc.lpszClassName = g_szClassName;

//Our window class name

wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

//Handle to icon that is shown in upper-left corner of application or in the taskbar


if (!RegisterClassEx(&wc))

//The RegisterClassEx function registers a window class for subsequent use in calls to the CreateWindowEx function

{

    MessageBox (NULL, "Failed to register class !", "Error!", MB_ICONEXCLAMATION | MB_OK);

    //(handle to owner window, text in message box, message box title, message box style i.e. icon and buttons)

    //| is a binary or operator

    return 0;

}


hwnd = CreateWindowEx(

    WS_EX_CLIENTEDGE,      //extended window style. Specifies that a window has a border with a sunken edge.

    g_szClassName,         //registered class name

    "The title of my program",    //window name. This will be shown in the title bar

    WS_OVERLAPPEDWINDOW,   //window style. Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION,
```

// WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles

CW_USEDEFAULT, //horizontal position of window. CW_USEDEFAULT makes the system select default position

CW_USEDEFAULT, //vertical position of window

240, //window width

120, //window height

NULL, //handle to parent or owner window

NULL, //menu handle or child identifier

hInstance, //handle to application instance

NULL //window-creation data

);

//If the function succeeds, the return value is a handle to the new window.

if(hwnd == NULL)

{

MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);

return(0);

}

ShowWindow (hwnd, iCmdShow);//sets the specified window's show state. Second parameter can be SW_HIDE, SW_RESTORE

UpdateWindow (hwnd);//updates the client area of the specified window by sending a WM_PAINT message to the window

```
        while (GetMessage (&msg, NULL, 0, 0) > 0)
        //retrieves a message from the calling thread's message queue
        //(message information, handle to window, first message, last message)
        //handle to window = NULL means it retrieves messages for any window that belongs to the
        calling thread
        {
                TranslateMessage (&msg);//translates virtual-key messages into character messages.
                                        //The character messages are posted to the calling thread's message
        queue, to be read
                DispatchMessage (&msg);//dispatches a message to a window procedure
        }

        return msg.wParam;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
//(handle to window, message identifier, first message parameter, second message parameter)
{
        switch (msg)
        {
                case WM_CLOSE:     //WM_CLOSE message is sent as a signal that a window or an
application should terminate
                                //An application can prompt the user for confirmation, prior to destroying a
window,
                                //by processing the WM_CLOSE message and calling the DestroyWindow
function only if the user confirms the choice
```

```
                DestroyWindow(hwnd);        //destroys the specified window. Send
WM_DESTROY

                break;


            case WM_DESTROY://The WM_DESTROY message is sent when a window is being
destroyed. It is sent to the window

                                    //procedure of the window being destroyed after the window
is removed from the screen

                PostQuitMessage(0);

                //The PostQuitMessage function indicates to the system that a thread has made a
request to terminate (quit).

                //0 Specifies an application exit code. This value is used as the wParam parameter
of the WM_QUIT message

                //It causes the GetMessage function to return zero.

                break;

            default:

                return DefWindowProc (hwnd, msg, wParam, lParam);

                //The DefWindowProc function calls the default window procedure to provide
default processing for any window

                //messages that an application does not process. The return value is the result of the
message processing

        }


        return(0);

}
```

B) Attempt any one of the following: Marks 6

a) Explain " Printf" in windows programming with an example.

(*For the following answer-4 Marks*)

Ans: Windows has no concept of standard input and standard output. You can use fprintf in a Windows program, but not printf.

Equivalent of printf ("The sum of %i and %i is %i", 5, 3, 5+3) ; will be

```
char szBuffer [100] ;

sprintf (szBuffer, "The sum of %i and %i is %i", 5, 3, 5+3) ;

MessageBox (NULL, szBuffer, "", 0);
```

b) What is the mapping modes of GDI?

(*Any four mapping modes-1 Mark each*)

Ans: Mapping mode is one attribute of device context.

Four other device context attributes—the window origin, the viewport origin, the window extents, and the viewport extents—are closely related to the mapping mode attribute.

Specified coordinate values are "logical units." Windows translates the logical units into "device units," or pixels.

The mapping mode also implies an orientation of the x-axis and the y-axis; that is, it determines whether values of x increase as you move toward the left or right side etc.

Mapping Mode	Logical Unit	x-axis	y-axis
MM_TEXT	Pixel	Right	Down
MM_LOMETRIC	0.1 mm	Right	Up
MM_HIMETRIC	0.01 mm	Right	Up

MM_LOENGLISH	0.01 in.	Right	Up
MM_HIENGLISH	0.001 in.	Right	Up
MM_TWIPS	1/1440 in.	Right	Up
MM_ISOTROPIC	Arbitrary (x = y)	Selectable	Selectable
MM_ANISOTROPIC	Arbitrary (x !=y)	Selectable	Selectable

The words METRIC and ENGLISH refer to popular systems of measurement; LO and HI are "low" and "high" and refer to precision. "Twip" is a fabricated word meaning "twentieth of a point." A point is a unit of measurement in typography that is approximately 1/72 inch but that is often assumed in graphics programming to be exactly 1/72 inch. A "twip" is 1/20 point and hence 1/1440 inch.

"Isotropic" means identical in all directions.

SetMapMode (hdc, iMapMode) ;

iMapMode = GetMapMode (hdc) ;

The default mapping mode is MM_TEXT.

5. Attempt any four of the following: Marks 16

a) Differentiate between Win16 and Win32 API.

(Each point-1 Mark)

Ans:

WIN16 API	WIN 32 API
Win16 supports 16 bits value coordinate points	Win32 supports 32 bits value coordinate points
Win32 function calls return a two dimensional coordinate points packed in 32 bit integer	Win32 it is not possible
Win16 no support for Ruin32 Application	Win32 ensure the win 16 API compatibility and new applications also
Win16 functions calls go through translation layer	Win32 function calls are processed through by operating system

b) Write a program to draw a line.

(Correct program-4 Marks, Examiner shall consider other program too)

Ans:

```
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
static TCHAR szAppName[] = TEXT ("LineDemo") ;
HWND hwnd ;
MSG msg ;
WNDCLASS wndclass ;
wndclass.style = CS_HREDRAW | CS_VREDRAW ;
```

```
wndclass.lpfnWndProc = WndProc ;
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = szAppName ;
if (!RegisterClass (&wndclass))
{
MessageBox (NULL, TEXT ("Program requires Windows NT!"),
szAppName, MB_ICONERROR) ;
return 0 ;
}
hwnd = CreateWindow (szAppName, TEXT ("Line Demonstration"),
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT,
NULL, NULL, hInstance, NULL) ;
ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;
while (GetMessage (&msg, NULL, 0, 0))
{
TranslateMessage (&msg) ;
DispatchMessage (&msg) ;
}
return msg.wParam ;
}
```

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

```
{
static int cxClient, cyClient ;
HDC hdc ;
PAINTSTRUCT ps ;
switch (message)
{
case WM_SIZE:
cxClient = LOWORD (lParam) ;
cyClient = HIWORD (lParam) ;
return 0 ;
case WM_PAINT:
hdc = BeginPaint (hwnd, &ps) ;
MoveToEx (hdc, 0, 0, NULL) ;
LineTo (hdc, cxClient, cyClient) ;
MoveToEx (hdc, 0, cyClient, NULL) ;
LineTo (hdc, cxClient, 0) ;


EndPaint (hwnd, &ps) ;
return 0 ;
case WM_DESTROY:
PostQuitMessage (0) ;
```

```
return 0 ;

}

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

}
```

c) Describe Truetype and big fonts.
(True Type -2 Marks, Big Fonts-2 Marks)

Ans:

TrueType and Big Fonts

The bitmap fonts that we've been using (with the exception of the fonts in the Japanese version of Windows) contain a maximum of 256 characters. This is to be expected, because the format of the bitmap font file goes back to the early days of Windows when character codes were assumed to be mere 8-bit values. That's why when we use the SYSTEM_FONT or the SYSTEM_FIXED_FONT, there are always some characters from some languages that we can't display properly. (The Japanese system font is a bit different because it's a double-byte character set; most of the characters are actually stored in TrueType Collection files with a filename extension of .TCC.)

TrueType fonts can contain more than 256 characters. Not all TrueType fonts have more than 256 characters, but the ones shipped with Windows 98 and Windows NT do. Or rather, they do if you've installed Multilanguage support. In the Add/Remove Programs applet of the Control Panel, click the Windows Setup tab and make sure Multilanguage Support is checked. This multilanguage support involves five character sets: Baltic, Central European, Cyrillic, Greek, and Turkish. The Baltic character set is used for Estonian, Latvian, and Lithuanian. The Central European character set is used for Albanian, Czech, Croatian, Hungarian, Polish, Romanian, Slovak, and Slovenian. The Cyrillic character set is used for Bulgarian, Belarusian, Russian, Serbian, and Ukrainian.

The TrueType fonts shipped with Windows 98 support those five character sets, plus the Western European (ANSI) character set that is used for virtually all other languages except those in the Far East (Chinese, Japanese, and Korean). TrueType fonts that support multiple character

sets are sometimes referred to as "big fonts." The word "big" in this context does not refer to the size of the characters, but to their quantity.

You can take advantage of big fonts even in a non-Unicode program, which means that you can use big fonts to display characters in several different alphabets. However, you need to go beyond the GetStockObject function in obtaining a font to select into a device context.

The functions CreateFont and CreateFontIndirect create a logical font, similar to the way CreatePen creates a logical pen and CreateBrush creates a logical brush. CreateFont has 14 arguments that describe the font you want to create. CreateFontIndirect has one argument, but that argument is a pointer to a LOGFONT structure, which has 14 fields that correspond to the arguments of the CreateFont function.

d) How to get, set, show and hide the cursor?
(Each correct function/syntax-1 Mark)

Ans:

Get:-

One can obtain the cursor position by calling function:-

GetCursorPos (&pt);

where *pt* is a POINT structure. The function fills in the POINT fields with the *x* and *y* coordinates of the mouse.

Set:-

You can set the cursor position by using

SetCursorPos (x, y);

In cases, the *x* and *y* values are screen coordinates, not client-area coordinates

Show:-

One can show the cursor by calling function

ShowCursor(true);

Even if a mouse device is not installed, Windows can still display a mouse cursor. Windows maintains something called a "display count" for this cursor. If a mouse is installed, the display count is initially 0; if not, the display count is initially -1. The mouse cursor is displayed only if the display count is non-negative. You can increment the display count by calling above function.

Hide:-

One can hide the cursor by calling function

ShowCursor(false);

Even if a mouse device is not installed, Windows can still display a mouse cursor. Windows maintains something called a "display count" for this cursor. If a mouse is installed, the display count is initially 0; if not, the display count is initially -1. The mouse cursor is displayed only if the display count is non-negative. You can increment the display count by calling above function

e) Write a function for visible and Enabled buttons.

(For each function-1 Mark, Any 4 functions are expected)

Ans:

To receive mouse and keyboard input, a child window must be both visible (displayed) and enabled. When a child window is visible but not enabled, Windows displays the text in gray rather than black.

If you don't include WS_VISIBLE in the window class when creating the child window, the child window will not be displayed until you make a call to *ShowWindow*:

ShowWindow (hwndChild, SW_SHOWNORMAL) ;

But if you include WS_VISIBLE in the window class, you don't need to call *ShowWindow*. However, you can hide the child window by this call to *ShowWindow*:

ShowWindow (hwndChild, SW_HIDE) ;

You can determine if a child window is visible by a call to

IsWindowVisible (hwndChild) ;

You can also enable and disable a child window. By default, a window is enabled. You can disable it by calling

EnableWindow (hwndChild, FALSE) ;

For button controls, this call has the effect of graying the button text string. The button no longer responds to mouse or keyboard input. This is the best method for indicating that a button option is currently unavailable. You can re-enable a child window by calling

EnableWindow (hwndChild, TRUE) ;

You can determine whether a child window is enabled by calling

IsWindowEnabled (hwndChild) ;

f) Write down a function for putting strings in the list box.

(For description-1 Mark, for correct program-3 Marks)

Ans:

After you've created the list box, the next step is to put text strings in it. You do this by sending messages to the list box window procedure using the *SendMessage* call. The text strings are generally referenced by an index number that starts at 0 for the topmost item. In the examples that follow, *hwndList* is the handle to the child window list box control, and *iIndex* is the index value. In cases where you pass a text string in the *SendMessage* call, the *lParam* parameter is a pointer to a null-terminated string.

In most of these examples, the *SendMessage* call can return LB_ERRSPACE (defined as -2) if the window procedure runs out of available memory space to store the contents of the list box. *SendMessage* returns LB_ERR (-1) if an error occurs for other reasons and LB_OKAY (0) if the operation is successful. You can test *SendMessage* for a nonzero value to detect either of the two errors.

If you use the LBS_SORT style (or if you are placing strings in the list box in the order that you want them to appear), the easiest way to fill up a list box is with the LB_ADDSTRING message:

SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) szString) ;

If you do not use LBS_SORT, you can insert strings into your list box by specifying an index value with LB_INSERTSTRING:

SendMessage (hwndList, LB_INSERTSTRING, iIndex, (LPARAM) szString) ;

For instance, if *iIndex* is equal to 4, *szString* becomes the new string with an index value of 4 the fifth string from the top because counting starts at 0. Any strings below this point are pushed down. An *iIndex* value of -1 adds the string to the bottom. You can use LB_INSERTSTRING with list boxes that have the LBS_SORT style, but the list box contents will not be re-sorted.

```
#include <windows.h>
#define ID_LIST 1
#define ID_TEXT 2
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
static TCHAR szAppName[] = TEXT ("Environ") ;
HWND hwnd ;
MSG msg ;
WNDCLASS wndclass ;
wndclass.style = CS_HREDRAW | CS_VREDRAW ;
wndclass.lpfnWndProc = WndProc ;
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = szAppName ;
if (!RegisterClass (&wndclass))
{
MessageBox (NULL, TEXT ("This program requires Windows NT!"),
szAppName, MB_ICONERROR) ;
return 0 ;
}
hwnd = CreateWindow (szAppName, TEXT ("Environment List Box"),
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT,
NULL, NULL, hInstance, NULL) ;
ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;
```

```
while (GetMessage (&msg, NULL, 0, 0))
{
TranslateMessage (&msg) ;
DispatchMessage (&msg) ;
}
return msg.wParam ;
}

void FillListBox (HWND hwndList)
{
int iLength ;
TCHAR * pVarBlock, * pVarBeg, * pVarEnd, * pVarName ;
pVarBlock = GetEnvironmentStrings () ; // Get pointer to environment block
while (*pVarBlock)
{
if (*pVarBlock != `=`) // Skip variable names beginning with `=`
{
pVarBeg = pVarBlock ; // Beginning of variable name
while (*pVarBlock++ != `=`) ; // Scan until `=`
pVarEnd = pVarBlock - 1 ; // Points to `=` sign
iLength = pVarEnd - pVarBeg ; // Length of variable name
// Allocate memory for the variable name and terminating
// zero. Copy the variable name and append a zero.
pVarName = calloc (iLength + 1, sizeof (TCHAR)) ;
CopyMemory (pVarName, pVarBeg, iLength * sizeof (TCHAR)) ;
pVarName[iLength] = `\0' ;
// Put the variable name in the list box and free memory.
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) pVarName) ;
free (pVarName) ;
}
while (*pVarBlock++ != `\0') ; // Scan until terminating zero
```

```
}
FreeEnvironmentStrings (pVarBlock) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam,
  LPARAM lParam)
{
static HWND hwndList, hwndText ;
int iIndex, iLength, cxChar, cyChar ;
TCHAR * pVarName, * pVarValue ;
switch (message)
{
case WM_CREATE :
cxChar = LOWORD (GetDialogBaseUnits ()) ;
cyChar = HIWORD (GetDialogBaseUnits ()) ;
// Create listbox and static text windows.
hwndList = CreateWindow (TEXT ("listbox"), NULL,
WS_CHILD | WS_VISIBLE | LBS_STANDARD,
cxChar, cyChar * 3,
cxChar * 16 + GetSystemMetrics (SM_CXVSCROLL),
cyChar * 5,
hwnd, (HMENU) ID_LIST,
(HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
NULL) ;
hwndText = CreateWindow (TEXT ("static"), NULL,
WS_CHILD | WS_VISIBLE | SS_LEFT,
cxChar, cyChar,
GetSystemMetrics (SM_CXSCREEN), cyChar,
hwnd, (HMENU) ID_TEXT,
(HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
NULL) ;
```

```
FillListBox (hwndList) ;
return 0 ;
case WM_SETFOCUS :
SetFocus (hwndList) ;
return 0 ;
case WM_COMMAND :
if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_SELCHANGE)
{
// Get current selection.
iIndex = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) + 1 ;
pVarName = calloc (iLength, sizeof (TCHAR)) ;
SendMessage (hwndList, LB_GETTEXT, iIndex, (LPARAM) pVarName) ;
// Get environment string.
iLength = GetEnvironmentVariable (pVarName, NULL, 0) ;
pVarValue = calloc (iLength, sizeof (TCHAR)) ;
GetEnvironmentVariable (pVarName, pVarValue, iLength) ;
// Show it in window.
SetWindowText (hwndText, pVarValue) ;
free (pVarName) ;
free (pVarValue) ;
}
return 0 ;
case WM_DESTROY :
PostQuitMessage (0) ;
return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

6. Attempt <u>any four</u> of the following: Marks 16

a) Explain the problem associated with ASCII code.

(For each problem-1 Mark, Any Four problems are expected)

Ans:

- The word ASCII is indicated by the first word of the acronym and is truly an American standard and it isn't even good enough for those countries where English is spoken. There is no symbol for British pound. English uses the Latin (or Roman) alphabet.
- ASCII has no provision for letters with accent marks i.e. diacritics.
- This was obviously not the best solution to internationalization because there is no guarantee of consistency.
- It contains only 256 symbols which doesn't include foreign languages.

b) What is a purpose of function keys on keyboard? Draw layout for any one type of keyboard.

(For purpose of function keys on the keyboard-3 Marks, layout-1 Mark)

Ans:

Decimal	*Hex*	*WINUSER.H Identifier*	*Required ?*	*IBM-Compatible Keyboard*
112-121	70-79	VK_F1 through VK_F10	X	Function keys F1 through F10
122-135	7A-87	VK_F11 through VK_F24		Function keys F11 through F24
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

Some other virtual key codes are defined, but they are reserved for keys specific to nonstandard keyboards or for keys most commonly found on mainframe terminals. Check /Platform SDK/User Interface Services/User Input/Virtual-Key Codes for a complete list.

c) Describe Non-client area mouse message.
(Description -3 Marks, Mouse button generated messages-1 Mark)

Ans:

If the mouse is outside a window's client area but within the window, Windows sends the window procedure a "nonclient-area" mouse message. The nonclient area of a window includes the title bar, the menu, and the window scroll bars.

You do not usually need to process nonclient-area mouse messages. Instead, you simply pass them on to DefWindowProc so that Windows can perform system functions. In this respect, the nonclient-area mouse messages are similar to the system keyboard messages WM_SYSKEYDOWN, WM_SYSKEYUP, and WM_SYSCHAR.

The nonclient-area mouse messages parallel almost exactly the client-area mouse messages. The message identifiers include the letters "NC" to indicate "nonclient." If the mouse is moved within a nonclient area of a window, the window procedure receives the message WM_NCMOUSEMOVE. The mouse buttons generate these messages:

Button	***Pressed***	***Released***	***Pressed (Second Click)***
Left	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
Middle	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
Right	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

The wParam and lParam parameters for nonclient-area mouse messages are somewhat different from those for client-area mouse messages. The wParam parameter indicates the nonclient area where the mouse was moved or clicked. It is set to one of the identifiers beginning with HT (standing for "hit-test") that are defined in the WINUSER.H.

The lParam parameter contains an x-coordinate in the low word and a y-coordinate in the high word. However, these are screen coordinates, not client-area coordinates as they are for client-area mouse messages. For screen coordinates, the upper-left corner of the display area has x and y values of 0. Values of x increase as you move to the right, and values of y increase as you move down the screen.

d) Explain WM-CTLCOLORBIN message.

(Description-3 Marks, list of various function- 1 Mark)

Ans:

We've seen how we can adjust our client area color and text color to the background colors of the buttons. Can we adjust the colors of the buttons to the colors we prefer in our program? Well, in theory, yes, but in practice, no. What you probably don't want to do is use SetSysColors to change the appearance of the buttons. This will affect all programs currently running under Windows; it's something users would not appreciate very much.

A better approach (again, in theory) is to process the WM_CTLCOLORBTN message. This is a message that button controls send to the parent window procedure when the child window is about to paint its client area. The parent window can use this opportunity to alter the colors that the child window procedure will use for painting. (In 16-bit versions of Windows, a message named WM_CTLCOLOR was used for all controls. This has been replaced with separate messages for each type of standard control.)

When the parent window procedure receives a WM_CTLCOLORBTN message, the wParam message parameter is the handle to the button's device context and lParam is the button's window handle. When the parent window procedure gets this message, the button control has already obtained its device context. When processing a WM_CTLCOLORBTN message in your window procedure, you:

- Optionally set a text color using SetTextColor
- Optionally set a text background color using SetBkColor
- Return a brush handle to the child window

In theory, the child window uses the brush for coloring a background. It is your responsibility to destroy the brush when it is no longer needed.

Here's the problem with WM_CTLCOLORBTN: Only the push buttons and owner-draw buttons send WM_CTLCOLORBTN to their parent windows, and only owner-draw buttons respond to the parent window processing of the message using the brush for coloring the background. This is fairly useless because the parent window is responsible for drawing owner-draw buttons anyway.

e) Explain check box and its use.

(Description-3 Marks, Use of Check box-1 Mark)

Ans:

A check box is a square box with text; the text usually appears to the right of the check box. (If you include the BS_LEFTTEXT style when creating the button, the text appears to the left; you'll probably want to combine this style with BS_RIGHT to right-justify the text.) Check boxes are usually incorporated in an application to allow a user to select options. The check box commonly functions as a toggle switch: clicking the box once causes a check mark to appear; clicking again toggles the check mark off.

The two most common styles for a check box are BS_CHECKBOX and BS_AUTOCHECKBOX. When you use the BS_CHECKBOX style, you must set the check mark yourself by sending the control a BM_SETCHECK message. The *wParam* parameter is set to 1 to create a check mark and to 0 to remove it. You can obtain the current check state of the box by sending the control a BM_GETCHECK message. You might use code like this to toggle the X mark when processing a WM_COMMAND message from the control:

SendMessage ((HWND) lParam, BM_SETCHECK, (WPARAM)

!SendMessage ((HWND) lParam, BM_GETCHECK, 0, 0), 0) ;

Notice the '!' operator in front of the second *SendMessage* call. The *lParam* value is the child window handle that is passed to your window procedure in the WM_COMMAND message. When you later need to know the state of the button, send it another BM_GETCHECK message. Or you can retain the current check state in a static variable in your window procedure. You can also initialize a BS_CHECKBOX check box with a check mark by sending it a BM_SETCHECK message:

SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;

For the BS_AUTOCHECKBOX style, the button control itself toggles the check mark on and off. Your window procedure can ignore WM_COMMAND messages. When you need the current state of the button, send the control a BM_GETCHECK message:

iCheck = (int) SendMessage (hwndButton, BM_GETCHECK, 0, 0) ;

The value of iCheck is TRUE or nonzero if the button is checked and FALSE or 0 if not. The other two check box styles are BS_3STATE and BS_AUTO3STATE. As their names indicate, these styles can display a third state as well a gray color within the check box which

occurs when you send the control a WM_SETCHECK message with *wParam* equal to 2. The gray color indicates to the user that the selection is indeterminate or irrelevant.

The check box is aligned with the rectangle's left edge and is centered within the top and bottom dimensions of the rectangle that were specified during the *CreateWindow* call. Clicking anywhere within the rectangle causes a WM_COMMAND message to be sent to the parent. The minimum height for a check box is one character height. The minimum width is the number of characters in the text, plus two.

Uses:-

1. Checkboxes are used for giving multiple options/choices
2. The user can select multiple option of same categories