WINDOWS I/O

Managing Input

- The task of capturing user input, processing it, and displaying output, becomes one of responding to events.
- Windows will generate an input message for each input event, such as mouse movement or a keystroke.
- Recall that each message structure contains an identifier for the message and a set of parameters - **lParam** and **wParam**.
- The contents of these parameters would be things such as: cursor or mouse coordinates, keypress values, or other system-related values such as character size.
- Message types are identified by their prefixes. For example, the **BM_SETCHECK** message is used to place or remove a checkmark for button controls.
- The **BM** prefix indicates that the message falls under the category of button control messages.
- The following table provides the various categories and their prefixes.

CATEGORY	PREFIX
Button Control	BM
ComboBox	CB
Edit Control	EM
Input	WM
List Box	LB
Owner Draw Control	WM
System Messages	WM
Window Management	WM

- Keypresses generate the **WM_CHAR** message which the keyboard device driver passes is to the active window.
- The keyboard messages are placed in the Windows system message queue.
- The messages are then sent to the appropriate application message queue.
- Finally, the messages in an application's message queue are retrieved by the application's **WinMain()** function.
- From the message loop they are dispatched to the registered window procedure or the default window procedure for the window class.
- Question is: How does Windows know which application it should send keyboard input messages?

- The answer lies in sending the keyboard messages to the application with the "**input focus**".

- The **input focus** is simply the active window or stated another way, the application with the **highlighted title bar** has the input focus.

Keyboard Messages

- Keyboard event generated messages fall into two categories.

Keystroke Messages

- Generated whenever a key is pressed or released.

- Pressing the '**X**' key, for example, generates two keystroke messages: one for pressing it and another for releasing it.

Keystroke System Messages

- These are messages which are important to Windows, from keys such as **Alt**, **Control**, **Shift**, **Esc**, **Insert**, **Delete**, the function keys, etc.

- For example the '**Alt - Tab**' combination is used to switch from one application to another.

Keystroke Non-system Messages

- These correspond to alphanumeric characters.

- The following table summarizes the messages generated for both the above categories:

	Key Pressed	**Key Released**
Nonsystem Keystroke	WM_KEYDOWN	WM_KEYUP
System Keystroke	WM_SYSKEYDOWN	WM_SYSKEYUP

- A keystroke message contains important information about the keystroke in its two 32-bit variables, **wParam** and **lParam**.

- The **lParam** variable contains information such as:
    - **Repeat Count** (Bits 0 - 15): Number of keystrokes.
    - **OEM scan code** (Bits 16 - 23): Generated by the hardware (for example, the value passed back to the program in the AH register during a BIOS Interrupt 16H call.
    - **Extended Key Flag** (Bit 24): Set to 1 if the keystroke originated from **the IBM Enhanced Keyboard** (separate numeric keypad and function keys across the top).
    - **Context Code** (Bit 29): Set to 1 if the Alt key is pressed. Always 1 for WM_SYSKEYDOWN and WM_SYSKEYUP messages.
    - **Previous Key state** (Bit 30): Set 0 if the key was previously up and 1 if it was down.
    - **Transition code** (Bit 31): Set to 0 if the key is being pressed and 1 if the key is being released.
- The **wParam** parameter of the keystroke message contains the **virtual key code** (device independent code) that identifies the key that was pressed.
- The **wparam** is the important one for our applications since it identifies the key that was pressed.

- The virtual key codes have names defined in the winuser.h (automatically included in windows.h) header file. For example, the **Ctrl-Break** sequence has the **hex value 03**, and it is defined as **VK_CANCEL** in **Windows.h**.
- These values are often cryptic and difficult to work with so it is preferable to use **translated messages**, that is, convert them to **character messages**.

Character Messages

- Generated whenever a combination of keystrokes results in a displayable character.
- This message results from the use of the **TranslateMessage** function in the message loop of the application.
- Just as in the keystroke case, there are two types of character messages as well: System and Nonsystem messages.
- Most applications process **character messages** instead of keystroke messages.
- The **TranslateMessage** function is used to translate keystroke messages into character messages.
- As a result a **WM_CHAR** message is generated and dispatched to the message handler.

- Character messages also contain the **lParam** and **wParam** variables.
- **lParam** has the same meaning as that for keystroke messages.
- **wParam** is used to get the **ASCII code** for the character.
- Managing keyboard input in Windows becomes a matter of responding to keystroke (**WM_KEYDOWN**) and character (**WM_CHAR**) messages.
- The character messages sent to the window procedure are framed between keystroke messages.
- For example pressing the lower case '**a**' will result in the following **3 messages**:

	Message	Key or Code
1	**WM_KEYDOWN**	Virtual key A
2	**WM_CHAR**	ASCII code for 'a'
3	**WM_KEYUP**	Virtual key A

- On the other hand, pressing the upper case '**A**' (with the Shift key) will result in the following **5 messages**:

	Message	Key or Code
1	**WM_KEYDOWN**	Virtual key VK_SHIFT
2	**WM_KEYDOWN**	Virtual key A
3	**WM_CHAR**	ASCII code for 'A'
4	**WM_KEYUP**	Virtual key A
5	**WM_KEYUP**	Virtual key VK_SHIFT

Writing Text to a Window

- Windows must first establish a link between your application and the screen or acquire a **Device Context** (**DC**).

- The **DC** is an **output path** from the **application** to the **client area** of a window via a **device driver**.

- The **GetDC()** function is called to retrieve a handle of a display device context (DC) for the client area of the specified window. The display device context can then be used in subsequent function calls to draw in the client area of the window.

  HDC GetDC (HWND hWnd);

  - **hWnd :** The handle to the window whose device context is to be retrieved.

- The **ReleaseDC()** function releases a device context, freeing it for use by other applications. This must be done so that the finite number of available DCs will be exhausted.

  int ReleaseDC (HWND hwnd, HDC hdc);

  - **hwnd :** Identifies the window whose device context is to be released.
  - **hdc : I**dentifies the device context to be released.

- Note that it is very important that you release the device context because there are only a limited number of device contexts available. Failing to release a device context can cause Windows to behave unpredictably.

- The **TextOut()** function is used to write a character string at the specified location in the client area, using the currently selected font.

  BOOL TextOut (HDC hdc, int nXStart, int nYStart, LPCTSTR lpString,
  int cbString);

  hdc

  Identifies the device context.

  nXStart

Specifies the logical x-coordinate (in terms of pixels) of the point at which Windows will output the string.

nYStart

Specifies the logical x-coordinate (in terms of pixels) of the point at which Windows will output the string.

lpString

Points 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 number of characters in the string.

- By default the upper-left corner of the client area of the window is location (0, 0). The X values increases to the right and the Y value increases downward.
- The characters typed by the user have to be first converted to a string that is **one character long** and then displayed.
- The **sprintf()** function accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string buffer.

  int sprintf(char *buffer, const char *format[, argument, ...]);
- The example Windows program we have been using has now been modified to display user keypresses.
- Note that the **TextOut()** function will not advance to a new line when you enter a newline character, nor will it respond to tabs. These must be performed by your program.
- Windows is a graphics-based system where characters are typically of different sizes. This means that if you enter a character such as "**w**" and then enter "**i**", part of the previous character will still be displayed.
- Therefore this method of text output to a window is crude. There are other more sophisticated techniques that will be shown later.

Processing the WM_PAINT Message

- In the previous example, if you enter some text and then minimize the window, the characters that were previously displayed will not be displayed after the window is restored.
- Also if the window is overwritten by under window and then uncovered, the characters will not be redisplayed.
- The reason for this is that Windows does not keep track of what a window contains. It is your program's responsibility to maintain the current window contents.
- Each time the contents of a window require updating, your program will be sent a **WM_PAINT** message.
- The **WM_PAINT** message informs a window procedure that part of the window's client area needs updating. This usually occurs when an overlapping window is moved or closed.
- Therefore in our example we must add a **WM_PAINT** case to our switch statement inside the window function.
- Now in order to acquire and release a device context in response to a **WM_PAINT** message, the window function must call **BeginPaint** and **EndPaint** respectively.
- All other output messages can be handled with **GetDC** and **ReleaseDC**.
- The device context is acquired using:

```
HDC BeginPaint (
        HWND hwnd,  // handle to window
        PPAINTSTRUCT *lpPaint   // pointer to structure
);
```

- If the function succeeds, the return value is the handle to a display device context for the specified window. Returns NULL if it fails.
- Upon return, the structure pointed to by ***lpPaint*** will contain information that your program can use to repaint the window.

- 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;
  BOOL fErase;
  RECT rcPaint;
  BOOL fRestore;
  BOOL fIncUpdate;
  BYTE rgbReserved[32];
  } PAINTSTRUCT;

  - **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.

  - **rcPaint**

  Specifies a **RECT** structure that specifies the upper left and lower right corners of the rectangle in which the painting is requested.

  - **fRestore**

  Reserved; used internally by the system.

  - **fIncUpdate**

  Reserved; used internally by the system.

  - **rgbReserved**

  Reserved; used internally by the system.

- It is very important to understand that a **DC** obtained using **BeginPaint** must be released only through a call to **EndPaint**.

- Also **BeginPaint** must only be used when a **WM_PAINT** message is being processed.

Mouse Input

- Each mouse event, such as a click, double click, dragging and mouse movement, causes a message to be sent to the window in which the event occurred.
- A window receiving a mouse message does not have to be active or have the input focus.
- Mouse messages occur in two stages. Windows generates a hit test message.
- Hit test messages are used to determine where a mouse event has occurred. Once that has been determined, a mouse message is generated.
- The message, **WM_NCHITTEST**, is sent to the window that contains the cursor when the mouse moves, or when a mouse button is pressed or released.
- The **lParam** parameter of this message holds the x and y coordinates of the cursor when the message was sent.
- Usually, the **WM_NCHITTEST** message is sent to the **DefWindowPro**c function.
- The return value of the **DefWindowProc** is a code indicating if the mouse hit was in the system area, such as the borders, the title bar or the client area.
- Windows uses the **WM_NCHITTEST** message to generate all other mouse messages.
- For example, a double click will generate a series of **WM_NCHITTEST** messages. Because the mouse is positioned over the system menu box, **DefWindowProc** returns a value of **HTSYSMENU** and Windows puts a **WM_NCLBUTTONDBLCLK** message in the message queue with **wParam** equal to **HTSYSMENU**.
- The **lParam** parameter of each mouse message contains the x and y coordinates of the mouse position. The coordinates are relative to the upper-left corner of the window.
- The low-order word of the **lParam** contains the horizontal coordinate of the mouse position.
- The high-order word contains the vertical position.
- The **wParam** parameter contains a value that indicates the status of the various virtual keys, specifically the three mouse buttons, the control key, and the shift key.

Managing Output

- Windows uses **the Graphics Device Interface** (**GDI**) to display output on a wide variety of devices.
- The process of displaying output always begins by obtaining a device context. A device context is a data structure (**PAINTSTRUCT**) that Windows uses to save a set of attributes for a device.
- An application will first retrieve a handle to the device context and use that handle for all subsequent **GDI** functions.
- So far we have been using very basic text output with arbitrary and fixed spacing between characters and lines.
- This approach is inadequate for more sophisticated applications because Windows uses proportional character fonts and adjustable font sizes.
- The main function for outputting text to the client area of a window is TextOut() as we have seen. This function simply displays a string a specified location without any formatting or interpretation of characters such as return/linefeed sequences.
- Therefore the program must have some means of determining the dimensions and other attributes of a selected font.
- The following API function that obtains information about the current font:

```
BOOL GetTextMetrics(
    HDC hdc,            // handle to device context
    LPTEXTMETRIC lptm   // pointer to text metrics structure
);
```

- This function fills the specified buffer with the metrics for the currently selected font.

- The **TEXTMETRIC** structure contains basic information about a physical font. All sizes are given in logical units; that is, they depend on the current mapping mode of the display context.

```
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BCHAR tmFirstChar;
    BCHAR tmLastChar;
    BCHAR tmDefaultChar;
    BCHAR tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

- The two important members for our use here are:

- **tmHeight**
    - Specifies the **height** (ascent + descent) of characters.
- **tmExternalLeading**
    - Specifies the amount of extra leading (space) that the application adds between rows.

- By adding these two values together we can obtain the total number of vertical units between the start of one line of text and the start of the next.

- In order for our program to display one string after another on the same line, it must have knowledge of where the previous string ended.

- This implies that you must have some way of determining the length of the string by calculating how many characters it contains.

- Using a function such as **strlen()** is not meaningful here because characters in this environment will have varying widths.

- Windows provides the following API function to solve this problem:

```
BOOL GetTextExtentPoint32 (
  HDC hdc,          // handle to device context
  LPCTSTR lpString,  // pointer to text string
  int cbString,      // number of characters in string
  LPSIZE lpSize      // pointer to structure for string size
);
```

- The argument ***lpSize*** is a pointer to a SIZE structure in which the dimensions of the string are to be returned.

- The **SIZE** structure specifies the width and height of a rectangle.

```
typedef struct tagSIZE {
    LONG cx;        // Specifies the rectangle's width.
    LONG cy;        // Specifies the rectangle's height.
} SIZE;
```

- The **GetTextExtentPoint32** function computes the width and height of the specified string of text.

- Upon successful return of the function call, the **cx** member will contain the length of the string.

- This value can be used to determine the location of the endpoint of the string when displayed.

- To display another string on the same line, we can continue where the previous string left off.