Programming in Graphical Environment

Windows API Lecture 2

Paweł Aszklar
`pawel.aszklar@pw.edu.pl`

Faculty of Mathematics and Information Science
Warsaw Univeristy of Technology

Warsaw 2024

Windows Messages

- Windows programming event driven
- Events and queries represented as messages
- System generates messages in response to user input, changes in the system, etc.
- Applications can use the same mechanism for in-process and interprocess communication
- Messages will be either:
  - placed in *message queue* (*queued messages*)
  - passed directly (*sent messages*) to window's class message handling callback (*window procedure*).

Message Properties

- Properties:

`HWND hWnd`	Intended recipient of the message
	Note: can be `nullptr` for messages messages concerning the whole process
`UINT message`	Message type identifier
`WPARAM wParam`	Message-specific parameter (`0` if unused)
`LPARAM lParam`	Message-specific parameter (`0` if unused)

- Meaning of `lParam`, `wParam` depends on message type (check docs!)
- Parameters can be values, bitflags, bitfields, pointer to structures etc.
- Optional properties (only queued messages):

`DWORD time`	Time when message was generated (in milliseconds from system start)
`POINT pt`	Mouse position when message was generated (in screen coordinates)

- Queued messages retrieved as `MSG` structure

Message Types

Message type ranges:

`0 - 0x3FFF`
`0 - WM_USER-1`
	System-defined messages (shouldn't be used for custom communication)

`0x4000 - 0x7FFF`
`WM_USER - WM_USER+16383`
`WM_USER - WM_APP-1`
	For custom communication with private window class (some used by system control classes, so shouldn't be used for custom application-wide communication)

`0x8000 - 0xBFFF`
`WM_APP - WM_APP+20479`
	For custom communication of any type

`0xD000 - 0xFFFF`	Custom system-wide messages, identifier provided by `RegisterWindowMessageW` (won't be constant, same string results in same identifier for every process)
`>0xFFFF`	Reserved

Note: Hundreds system messages defined in docs. Relevant types will be mentioned for each topic.

Message Type Examples

Messages sent on window creation
(e.g. during CreateWindowExW, etc.):

- WM_NCCREATE, WM_CREATE:
  - same parameters, similar effect
  - WM_CREATE after window created, WM_NCCREATE earlier
  - lParam points to CREATESTRUCTW, containing parameters of CreateWindowExW.
  - returning -1 from window procedure cancels window creation (CreateWindowExW returns nullptr) and destroys the window.

```
struct CREATESTRUCTW
{
    LPVOID      lpCreateParams;
    HINSTANCE   hInstance;
    HMENU       hMenu;
    int         cx;
    int         cy;
    int         x;
    int         y;
    LONG        style;
    LPCWSTR     lpszName;
    LPCWSTR     lpszClas;
    DWORD       dwExStyle;
};
```

- WM_GETMINMAXINFO, WM_NCCALCSIZE: relate to window size and position (discussed later)
- Message order undocumented, except WM_NCCREATE before WM_CREATE
- In practice: WM_GETMINMAXINFO, WM_NCCREATE, WM_NCCALCSIZE, WM_CREATE
- Other messages sent if WS_VISIBLE style set (refer to ShowWindow discussion)

Message Type Examples

Messages <u>sent</u> on window destruction (i.e. during `DestroyWindow` or cancelled window creation):

- `WM_DESTROY` is sent:
  - after window is hidden
  - before destruction of child/owned windows
  - will not be sent if window creation cancelled on `WM_NCCREATE`
- `WM_NCDESTROY`:
  - sent after child/owned windows are destroyed
  - usually the last message window receives
- Other messages can be sent before if window was visible, active, had focus etc. (refer to `ShowWindow` discussion)

Message Type Examples

Other examples:

- `WM_SYSCOMMAND` received when chooses command from system menu, caption buttons, etc.
    - `lParam` — command type, e.g. `SC_CLOSE` to close window
- `WM_CLOSE` received when window is to be closed
    - source: x button; Close option in system menu, taskbar; program closed from task manager; etc.
    - can be used to display confirmation window, destroy or hide window etc.
    - `DefWindowProcW` destroys window by default, don't pass the message to prevent it
- `WM_QUIT` queued (posted) message indicating application (thread) should exit
    - Generated by calling `PostQuitMessage` with an exit code
    - Usually posted in response to `WM_DESTROY`
    - `wParam` contains exit code value
    - no recipient window (`hWnd` is `nullptr`)

Window Procedure

```
LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

- Messages intended for a window will usually be passed to window procedure callback associated with its class.
- Parameters — message properties
- Other properties (will only work for queued messages)
  - `GetMessagePos` — mouse position
    (use `GET_X_LPARAM` and `GET_Y_LPARAM` from `<windowsx.h>` to extract coordinates)
  - `GetMessageTime` — message timestamp
  - `GetMessageExtraInfo`, `SetMessageExtraInfo` — access application-defined message property
- Return value depends on message type (check docs!) — usually 0 to indicate message handled

Window Procedure

```
LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

- `switch` statement, some dictionary structure, etc. (or combination thereof) can be used to associate handler code with specific message type
- For unhandled messages, pass parameters to and return result of `DefWindowProcW`
- Some messages need to be forwarded to `DefWindowProcW` even if processed by application, e.g. `WM_NCACTIVATE` (check docs!)
- Exception safety: window procedure must not throw exceptions. They will either be suppressed or terminate the program immediately, without propagating further up the call stack.

Example — Object-oriented Approach

- Win32 is a C API, however, creation of OOP wrapper isn't difficult
- `window` class will wrap `HWND`
- Common Win32 window class $\rightarrow$ `static` member functions
- Note: window procedure must be a free function or a `static` member function
  - Internal Win32 window data contains pointer-sized field to be used by application
  - Access: `GetWindowLongPtrW`, `SetWindowLongPtrW` with `GWLP_USERDATA` offset
  - Can be used to store `window` instance pointer
  - `static` window procedure can retrieve it and call a non-`static` member function
  - Some messages will be sent before `CreateWindowExW` returns `HWND`
  - Solution: Pass `this` as `lpParam`, retrieve it from `WM_NCCREATE`

Example — Object-oriented Approach

```
//window.h
#pragma once
#include <utility>
#include <string>
class window
{
    static bool is_class_registered(HINSTANCE, LPCWSTR);
    static void register_class(HINSTANCE, LPCWSTR);
    HWND m_hWnd;
public:
    static LRESULT window_proc(HWND, UINT, WPARAM, LPARAM);
    virtual LRESULT window_proc(UINT, WPARAM, LPARAM);
    window() : m_hWnd { nullptr } { }
    window(const window&) = delete;
    window(window&& other) : m_hWnd { nullptr } { *this = std::move(other); }
    window(HINSTANCE, const std::wstring&);
    window& operator=(const window&) = delete;
    window& operator=(window&& other) { std::swap(m_hWnd); return *this; }
    operator HWND() const { return m_hWnd; }
    virtual ~window();
};
```

Example — Object-oriented Approach

```
//window.cpp
#include "window.h"
bool window::is_class_registered(HINSTANCE hInst, LPCWSTR cName)
{
    WNDCLASSEXW wcx;
    return GetClassInfoExW(hInst, cName, &wcx);
}
void window::register_class(HINSTANCE hInst, LPCWSTR cName)
{
    WNDCLASSEXW wcx{};
    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_VREDRAW|CS_HREDRAW;
    wcx.lpfnWndProc = window_proc;
    wcx.hCursor = LoadCursorW(nullptr, IDC_ARROW);
    wcx.hbrBackground = static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    wcx.lpszClassName = cName;
    RegisterClassExW(&wcx);
}
```

Example — Object-oriented Approach

```
//window.cpp cont'd
window::window(HINSTANCE hInst, const std::wstring& title)
    : m_hWnd { nullptr }
{
    LPCWSTR className = L"My Window Class";
    if (!is_class_registered(hInst, className))
        register_class(hInst, className);
    CreateWindowExW(0, className, title.c_str(),
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        nullptr, nullptr, hInst, reinterpret_cast<LPVOID>(this));
    //m_hWnd will be set on WM_NCCREATE
}
window::~window()
{
    if (m_hWnd)
        DestroyWindow(m_hWnd);
}
```

Example — Object-oriented Approach

```cpp
//window.cpp cont'd
LRESULT window::window_proc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    window *w = nullptr;
    if (msg == WM_NCCREATE) {
        auto pcs = reinterpret_cast<LPCREATESTRUCTW>(lParam);
        w = reinterpret_cast<window*>(pcs->lpCreateParams);
        SetWindowLongPtrW(hWnd, GWLP_USERDATA, reinterpret_cast<LONG_PTR>(w));
        w->m_hWnd = hWnd;
    } else w = reinterpret_cast<window*>(GetWindowLongPtrW(hWnd, GWLP_USERDATA));
    if (w) {
        auto r = w->window_proc(msg, wParam, lParam);
        if (msg == WM_NCDESTROY) {
            w->m_hWnd = nullptr;
            SetWindowLongPtrW(hWnd, GWLP_USERDATA, 0);
        }
        return r;
    }
    return DefWindowProcW(hWnd, msg, wParam, lParam);
}
```

Example — Object-oriented Approach

```
//window.cpp cont'd
LRESULT window::window_proc(UINT msg, WPARAM wParam, LPARAM lParam)
{
    //handle window logic
    //Example:
    switch (msg)
    {
    case WM_CLOSE:
        DestroyWindow(m_hWnd);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(EXIT_SUCCESS);
        return 0;
    }
    return DefWindowProcW(m_hWnd, msg, wParam, lParam);
}
```

Example — Object-oriented Approach

Custom window data:

- Preferred: static member fields of `window` or derived class (obvious!)
- Inside of internal Win32 window data:
  - request memory with `cbWndExtra` when registering class
  - access with `GetWindowLongPtrW`, `SetWindowLongPtrW` with non-negative offset
- Inside of internal Win32 window class data:
  - request memory with `cbClsExtra` when registering class
  - access with `GetClassLongPtrW`, `SetClassLongPtrW` with non-negative offset

Customised behaviour:

- Modify non-`static` `window_proc`
- Override `window_proc` in derived class
  (remember to forward call to base class for unhandled messages!)

Note: For the sake of brevity error checking, exception safety, etc. were ignored.
Check subject website for an example of a more careful implementation!

Message Routing

Note: Following discussion only considers routing messages to windows on a single thread that created them. See ▸ Appendix A and ▸ Appendix B for more detail.

Sending messages:

- Refers to calling window procedure directly
- Used for messages that need to be processed by window procedure and/or when system/calling function needs to examine the result

Queued messages:

- Refers to messages added to a queue
- Used for event notifications, messages that don't require immediate processing

Order and method in which messages arrive to window procedure should not be relied on

Sent Messages

- Certain system messages are sent, e.g. `WM_ACTIVATE`, `WM_SETFOCUS`
- Some API functions send messages, e.g. `WM_CREATE` from `CreateWindowExW` or `WM_DESTROY` from `DestroyWindow`
- Explicitly sending messages:
  - `SendMessageW`
  - `SendMessageTimeoutW`
  - `SendNotifyMessageW`
  - `SendMessageCallbackW`

  When sending messages from thread to associated window, behaviour of all four is the same.
- Use `InSendMessage`, `InSendMessageEx` (e.g. in window procedure) to check if processing sent message.

Message Queue

- Queue created automatically for any thread that needs it
- Messages for windows created by a thread are always added to its message queue
- Size of queue limited — messages might be dropped if queue is full
- Order of messages in a queue not guaranteed unless clearly documented (check docs!)
- However, in practice messages ordered by priority (from highest to lowest)
  - *Posted* messages — added by an application directly or indirectly via API function call
  - *Input* messages — usually added by the system in response to user input or system internal events
  - *Low priority* messages — of which there are three,
    in order of importance: `WM_QUIT`, `WM_PAINT`, `WM_TIMER`

  Messages of the same priority are handled in FIFO order. For more details see ▸ Appendix A
- Multiple instances of some messages (namely `WM_MOUSEMOVE`, `WM_NCMOUSEMOVE`, low priority messages) might be coalesced into one, with properties reflecting the last instance added

Message Queue

Posting messages:

- Adding *posted* messages to queue explicitly: `PostMessageW`, `PostThreadMessageW`
- Some API functions post messages as well, e.g. `TranslateMessage`
- Don’t post input and low priority messages:
  - They will behave like posted messages
  - Additional operation system performs when normally adding them to queue will not happen
  - Use functions listed below to synthesise them instead
  - Application must not post `WM_QUIT`

Synthesising input and low priority messages:

- `WM_QUIT` — use `PostQuitMessage` (unfortunately named)
- `WM_PAINT` — use `InvalidateRect`, `InvalidateRgn` (discussed later)
- input messages — use `SendInput` (also discussed later, and also unfortunately named)

Message Loop

- Each thread with a message queue must continuously query for and remove pending messages (*message pumping*)
- Otherwise system might consider thread hung and replace its windows with *ghost windows*
- Usually each UI thread contains a message loop
- Some API functions might pump messages internally, e.g. `MessageBoxW`, `SendMessageW`

Message Loop — Example

```
...
window w{ hInst, L"Hello World" };
MSG msg{ };
BOOL gmResult;
while ((gmResult = GetMessageW(&msg, nullptr, 0, 0)) != 0)  {
    if(gmResult == -1)
        return EXIT_FAILURE;
    TranslateMessage(&msg);
    DispatchMessageW(&msg);
}
return msg.wParam;
...
```

Retrieving Messages

```
BOOL GetMessageW(LPMSG lpMsg, HWND hWnd, UINT msgMin, UINT msgMax);
```

- Removes and returns a message (matching filters) from thread's queue
- Blocks until such message available
- Filtering based on recipient window
  - if `hWnd==0`, no filtering;
  - if `hWnd==-1`, only messages whose `msg.hWnd==0`;
  - otherwise, only messages for specific window.
- Filtering based on message type
  - if both `msgMin` and `msgMax` are 0, no filtering;
  - otherwise, only messages with type in range `[msgMin, msgMax]` (inclusive).
  - `WM_KEYFIRST`, `WM_KEYLAST` or `WM_MOUSEFIRST`, `WM_MOUSELAST` for keyboard, mouse messages.
- Returns:
- `0` if `WM_QUIT` was retrieved;
- `-1` on error;
- non-zero otherwise

Retrieving Messages

```
BOOL PeekMessageW(LPMSG lpMsg, HWND hWnd, UINT msgMin, UINT msgMax, UINT flags)
```

- Retrieves message (matching filters) from thread's queue (but doesn't block)
- Filters as in `GetMessageW`, additional filters via `flags`.
- Message not removed from queue, unless `PM_REMOVE` flag set.
- Returns `0` if no matching messages found, non-zero otherwise.

```
BOOL WaitMessage()
```

- Blocks until new message is available in queue, returning `0` on error
- Only messages added after last `GetMessageW`, `PeekMessageW`, `WaitMessage`, etc. call are not considered new.
- `GetMessageW` works as combination of `PeekMessageW` and `WaitMessage`

Additional related functions:

`GetQueueStatus`, `MsgWaitForMultipleObjects`, `MsgWaitForMultipleObjectsEx`

Processing Messages

- `LRESULT DispatchMessageW(const MSG* msg)`
  - Passes message to appropriate callback (timer callback, window procedure).
  - Returns callback result.
- `BOOL TranslateMessage(const MSG* msg)`
  - Translates key messages (`WM_KEYDOWN`, `WM_KEYUP`, etc.) to characters, posts `WM_CHAR` if necessary.
  - Doesn’t modify message.
  - Don’t call if virtual keys used for other purpose (`TranslateAcceleratorW`, `IsDialogMessageW`).
- `int TranslateAcceleratorW(HWND hWnd, HACCEL hAccTable, LPMSG msg)`
  - Call if window has accelerator table (list of shortcuts) — accelerators will be discussed later
  - Translates virtual keys to shortcuts, posts `WM_COMMAND`, `WM_SYSCOMMAND` if necessary
  - If returned value is non-zero, message is processed — don’t pass it to `DispatchMessageW`, etc.
- `BOOL IsDialogMessageW(HWND hDlg, LPMSG msg)`
  - Call for modeless dialogs (can be used with any window containing controls) — controls and dialog boxes will be discussed later
  - Processes messages enabling keyboard navigation between controls in a window. Other messages are translated/dispatched.
  - If returned value is non-zero, message is processed — don’t pass it to `DispatchMessageW`, etc.

Processing Messages — Example

```
...
//assuming:
//hDlg - only modeless dialog window
//hMain - only window with accelerator table
//hAT - handle to main window's accelerator table
MSG msg{ };
BOOL gmResult;
while ((gmResult = GetMessageW(&msg, nullptr, 0, 0)) != 0)  {
    if(gmResult == -1)
        return EXIT_FAILURE;
    if (IsDialogMessageW(hDlg, &msg) == 0)
    {
        if (TranslateAcceleratorW(hMain, hAT, &msg) == 0)
        {
            TranslateMessage(&msg);
            DispatchMessageW(&msg);
        }
    }
}
...
return msg.wParam;
```

Long running operations

Messages should not wait more than few seconds in the queue.
Long running operations should not stall the message loop.

- If waiting for system objects (Processes, Threads, Mutexes, I/O etc.): `MsgWaitForMultipleObjects`, `MsgWaitForMultipleObjectsEx`
- If operation can be paused frequently: `PeekMessageW`
- In general: move operation to separate thread, post messages to UI thread (or use other synchronisation mechanisms) to update progress, present results (see ▸ Appendix B)

```
...
bool done = false;
MSG msg{ };
while (!done) {
    if (PeekMessageW(&msg, nullptr,
            0, 0, PM_REMOVE)) {
        if(msg.message == WM_QUIT)
            done = true;
        else {
            TranslateMessage(&msg);
            DispatchMessageW(&msg);
        }
    }
    else {
        //Continue operation
        ...
    }
}
...
```

Timers

`UINT_PTR SetTimer(HWND hWnd, UINT_PTR id, UINT timeout, TIMERPROC callback)`

- Creates a new timer or restarts an existing one
- `hWnd`,`id` identify a window timer (global timer if `hWnd` is `nullptr`). All timers have non-zero `id`.
- If timer exists, it is restarted with new timeout value and callback.
- Otherwise new timer is created. For global timer, new ID is generated and returned (ignoring `id`), for window timer ID is `id`. Pass `0` as `id` to guarantee a new timer.
- `timeout` in milliseconds before timer elapses (repeated until timer destroyed)
- Each time timer elapses, `WM_TIMER` will be added to the queue (multiples can coalesce).
- If `callback` is not `nullptr` it will be called (by `DispatchMessageW` or `DefWindowProcW`).
- Returns ID of the timer or `0` on error.

`BOOL KillTimer(HWND hWnd, UINT_PTR id)`

- Destroys an existing timer
- Doesn't remove `WM_TIMER` already in the queue.

Timers

`WM_TIMER`

- Properties:
  - `wParam` — timer ID
  - `lParam` — callback address specified in `SetTimer` call. If not `nullptr` `DispatchMessageW` will call it instead of window procedure.
- Inaccurate and unreliable (low priority message, coalescence of duplicates in queue)
- Don't rely on `timeout` value passed to `SetTimer`, use `time` member of `MSG`, `GetMessageTime`, etc. (see next slide)

```
void TimerProc(HWND hWnd, UINT msg, UINT_PTR id, DWORD time)
```

- Timer callback prototype
- `hWnd`,`id` — timer ID and associated window
- `msg` — message type (`WM_TIMER`)
- `time` — time when message was removed from queue (milliseconds from system start)

Timing

At any point current time (from system start) can be acquired:

- `DWORD GetTickCount()` — in *ms*
- `BOOL QueryPerformanceCounter(LARGE_INTEGER *count)` — high resolution (sub-μs)
- `BOOL QueryPerformanceFrequency(LARGE_INTEGER *frequency)` — resolution of above timestamp (counts per second)

Other time related functions:

- Time in UTC: `GetSystemTime`, `GetSystemTimeAdjustment`, `SetSystemTime`
- Time in local time-zone: `GetLocalTime`, `SetLocalTime`
- Formatting time: `GetTimeFormatEx`
- Waitable Timers (for use with `MsgWaitForMultipleObjects`, etc.): `CreateWaitableTimerW`, `CreateWaitableTimerExW`, `CloseHandle`, etc.
- Multimedia timers: `timeGetTime`, `timeSetEvent`, `timeKillEvent`, etc. (events run in separate thread!)

Animation

Special case of long running operation:

- Simple action performed time and again (or at intervals)
- Also applies to any short operation repeated over a period of time

Animation using timers

- Only for extra animations
- Unreliable (due to low message priority) for animation central to program functionality (playing multimedia, etc.)

Animation using general approach w/ `PeekMessageW`

- Often wasteful if animation update is very short
  - Little point frequency above monitor refresh-rate
  - No visible change due to rounding (e.g. displaying movie frame, position at nearest pixel, etc.)
- Can be combined with short `Sleep` to animate on interval (remember to subtract frame drawing time)

Other approaches: animation using waitable timers, sending messages from separate thread, etc.

Window Types

Reminder of highly confusing terminology used in documentation:

Overlapped window	without `WS_POPUP` and `WS_CHILD` styles, can overlap unrelated windows
Pop-up window	with `WS_POPUP` style, can overlap unrelated windows
Top-level window	Rarely used, overlapped or pop-up window.
Child window	with `WS_CHILD` style, must have and is contained within parent's client area

Relationship types (established e.g. by passing `hWndParent` to `CreateWindowExW`):

Parent-Child	Only possible if window has `WS_CHILD` style. Its parent can itself be a child window.
Owner-Owned	Windows without `WS_CHILD` become *owned*. Window specified as "parent" is the *owner*.

Desktop Window

- Root of window hierarchy - in certain contexts acts as owner of top-level windows without explicit owner.
- Retrieved by: `GetDesktopWindow`
- Setting desktop as owner/parent explicitly should be avoided, especially for child windows.
- Not actually the thing with wallpaper and icons, use `GetShellWindow` for that.

Related functions:

- `SystemParametersInfoW` can be used access some desktop properties:
  - `SPI_GETDESKWALLPAPER`, `SPI_SETDESKWALLPAPER`, `SPI_SETDESKPATTERN`,
  - `SPI_GETWORKAREA`, `SPI_SETWORKAREA`,
  - etc.
- `GetThreadDesktop`, `EnumDesktopsW`, etc.

Relationships Between Windows

Establishing relationship:

- Passing parent/owner as `hWndParent` to `CreateWindowExW`
- `BOOL SetParent(HWND hWndChild, HWND hWndNewParent)`
  - Sets new parent/owner for child/owned window
  - Pass `nullptr` as `hWndNewParent` to break relationship (`WS_CHILD` style of child needs to be removed separately)
  - Re-parenting generally should be avoided
- Child window cannot be set as an owner. If specified as such, its most immediate top-level ancestor is used instead.
- Avoid relationships between windows from different threads or processes (see ▸ Appendix B)
- When creating child window, `hMenu` parameter is instead used to set child ID
- Child ID can later be changed by `SetWindowLongPtrW` with `GWLP_ID`

Relationships Between Windows — Navigation

Navigation up the hierarchy:

- `HWND GetParent(HWND hWnd)`, returns:
  - parent/owner for child and owned pop-up windows
  - `nullptr` for all other windows
- `HWND GetAncestor(HWND hWnd, UINT flag)`, depending on `flag` returns:

`GA_PARENT`	Parent of a child window (or desktop window for others)
`GA_ROOT`	Top-level (pop-up/overlapped) windows – window itself (regardless if owned); Child windows – first top-level window up the chain of parents
`GA_ROOTOWNER`	Child or owned pop-up windows – first overlapped or unowned pop-up window up the chain Other windows – window itself.

Relationships Between Windows — Navigation

Navigation (mostly) down/sideways in the hierarchy:

- `HWND GetWindow(HWND hWnd, UINT cmd)`

`GW_OWNER`	Owner	`GW_CHILD`	First child
`GW_HWNDPREV`	Previous (same type)	`GW_HWNDNEXT`	Next (same type)
`GW_HWNDFIRST`	First (same type)	`GW_HWNDLAST`	Last (same type)
`GW_ENABLEDPOPUP`	First enabled owned pop-up window, or the window itself		

  - Types of windows: top-most; non-topmost top-level; siblings with the same parent.
  - Windows in one type enumerated in Z Order.

Relationships Between Windows — Navigation

Navigating relationships:

- Retrieving immediate child from point (relative to top-left corner of parent's client area): `ChildWindowFromPoint`, `ChildWindowFromPointEx`, `RealChildWindowFromPoint`
- `BOOL IsChild(HWND hWndParent, HWND hWndChild)`
  - Checks if window is direct or indirect child of a parent
  - Checks only up the chain of parent windows (stops at first top-level window)
- `BOOL EnumChildWindows(HWND parent, WNDENUMPROC enumProc, LPARAM lParam)`
  - Enumerates child windows of `parent` (including indirect descendants)
  - Each, along with `param` (application-defined parameter), passed to `enumProc`
  - `BOOL EnumProc(HWND child, LPARAM param)` — return `TRUE` to continue enumeration.
  - Preferred over calling `GetWindow` in a loop.

Other related function: `EnumWindows`, `EnumDesktopWindows`, `EnumThreadWindows`, `FindWindowW`, `WindowFromPoint` etc.

Message-Only Windows

Creating a window with `HWND_MESSAGE` as parent creates *message-only window*

- Never visible,
- Doesn't receive messages unless explicitly sent/posted to it,
- Not considered top-level (regardless of styles, child of hidden system window),
- Thus, doesn't receive broadcast message, can't be enumerated with `EnumWindows`, etc.

Window Show State — Focus

(Keyboard) Focus:

- Only one of thread's windows can have focus
- It will receive keyboard input from thread's message queue
- `GetFocus` to find it, `SetFocus` to change (only within thread)
- `WM_SETFOCUS`, `WM_KILLFOCUS`
    - Sent to window receiving/losing focus
    - `wParam` — handle to the other window (if of the same thread)
    - Do not activate or disable windows while processing `WM_KILLFOCUS` — possible deadlocks!
- Newly shown top-level windows by default receive focus

Window Show State — Active

Active window:

- Top-level window which has focus or whose (direct or indirect) child has focus
- `GetActiveWindow` to find it, `SetActiveWindow` to change (only within thread)
- `WM_NCACTIVATE`, `WM_ACTIVATE` sent to deactivated and activated windows
- on `WM_ACTIVATE`, `DefWindowProcW` sets focus to activated window
- Newly shown top-level windows by default are activated
- Activated window receives focus

Window Show State — Foreground

Foreground window:

- Top-level window user is working with
- System passes keyboard input to its thread’s message queue
- Process that created foreground window is a foreground process (has higher priority)
- `GetForegroundWindow` to find it.
- `SetForegroundWindow` — activates window and brings it to foreground
- Stealing foreground from another process has some restrictions (check docs!)
- When foreground status of a process changes, `WM_ACTIVATEAPP` is sent to each top-level window.

Window Show State — Disabled

Disabled window:

- Disabling window
  - blocks mouse input for it and its children (redirected parent if it itself is a child)
  - kills its focus (but ignores focus in children – might still receive keyboard input)
  - does not deactivate window, should be done manually
- Controlled by `WS_DISABLED` window style. Use `IsWindowEnabled` to check state.
- `BOOL EnableWindow(HWND hWnd, BOOL enable)` to change state
  - If state changes as a result, sends `WM_ENABLE`
  - Additionally if window becomes disabled, sends `WM_CANCELMODE`
- Inactive disabled top-level window can be activated
  - Programmatically: `SetActiveWindow`
  - From taskbar, through Alt + ⇆, etc., make sure it owns another visible window (a.k.a modal window) to prevent it

If activated, receives focus normally!

Window Show State — Visible

Window visibility:

- Controlled by `WS_VISIBLE` window style
- `IsWindowVisible` checks windows visibility
- `WM_SHOWWINDOW` if visibility changes (and other — regarding window location, repainting, etc.)
- Hiding window hides all children, removes taskbar button (if present), deactivates window, removes focus, etc.
- `ShowWindow` and number of other functions to change visibility (see next slides).
- `ShowOwnedPopups` can hide all pop-ups owned by a window, then show them back up

Window Show State — Minimized, Maximized, Normal

- Controlled by styles:
  - if `WS_MINIMIZE` set — minimized,
  - otherwise if `WS_MAXIMIZE` set — maximized,
  - otherwise window in normal, tracked size
- `IsIconic`, `IsZoomed` check if window is minimized, maximized
- Minimizing window
  - Minimizes owned windows first (recursively,)
  - Moves and resizes to empty client area (off screen if window on taskbar, still technically *visible*)
  - Kill focus (window and descendants), deactivates window
- Restoring minimized window reverts above, back to normal/maximized state it had before.
- `WM_QUERYOPEN` send to minimized window if it's about to be restored. Return `FALSE` to cancel.
- Maximized window fills workspace area (if working maximize button present) or entire screen (otherwise). Window border removed in maximized state.
- Restoring maximized window returns it to normal size/position
- `WM_SYSCOMMAND` sent when state changed by user (`SC_MINIMIZE`, `SC_MAXIMIZE`, `SC_RESTORE`)

Window Show State — ShowWindow Function

```
BOOL ShowWindow(HWND hWnd, int showCmd)
```

Modifies visibility, active, maximized, minimized state:

`SW_SHOWDEFAULT`	use `showCmd` passed to `wWinMain`
`SW_HIDE`	hide window
`SW_SHOW`	activate and show w/ current size/position
`SW_SHOWNA`	show w/ current size/position (don't activate)
`SW_SHOWNORMAL`	activate and show w/ normal size/position
`SW_SHOWNOACTIVATE`	show w/ normal size/position (don't activate)
`SW_MINIMIZE`	minimize window
`SW_MAXIMIZE`	maximize window
`SW_RESTORE`	activate and show, if minimized/maximized, return to normal size/position
`SW_SHOWMAXIMIZED`	activate and show maximized
`SW_SHOWMAXIMIZED`	activate and show minimized
`SW_SHOWMINNOACTIVE`	show minimized (don't activate)

First time function is called (for what OS determines to be main window) `showCmd` might be ignored

Window Show State — Window Placement

```
BOOL GetWindowPlacement(HWND hWnd,
                        WINDOWPLACEMENT *p);
BOOL SetWindowPlacement(HWND hWnd,
                        const WINDOWPLACEMENT *p);
```

```
struct WINDOWPLACEMENT
{
    UINT   length;
    UINT   flags;
    UINT   showCmd;
    POINT  ptMinPosition;
    POINT  ptMaxPosition;
    RECT   rcNormalPosition;
    RECT   rcDevice;
};
```

Can be used to check or set:

- Show state (like `ShowWindow`)
- Top-left window corner when minimized/maximized, former ignored if minimized to taskbar, can't set the latter
- Normal (tracked) size and position

Note: Positions in (see later slides ▸ here for definitions)

- parent's client coordinates – child windows
- screen coordinates — top-level windows with `WS_EX_TOOLWINDOW` style
- workspace coordinates – other windows (Warning! Other positioning functions use screen coordinates for those)

Other related functions: `SetWindowPos`, `DeferWindowPos`

Window Show State — Initial State

State set by `CreateWindowExW`:

- Controlled by `WS_VISIBLE`, `WS_MINIMIZE`, `WS_MAXIMIZE`, `WS_DISABLED`
- If `WS_VISIBLE`, `ShowWindow` is called, with `showCmd`:
  - If `x==CW_USEDEFAULT && y!=CW_USEDEFAULT`, value of `y` is used
  - `SW_SHOW` otherwise
- Styles of a window can be modified by `SetWindowLongPtrW` with `GWL_STYLE`, `GWL_EXSTYLE`

Coordinates

- Position as (x, y) pair in *device* units (usually pixels, but not always — high DPI displays)
- *x* increases left to right, *y* increases top to bottom
- Position depends on coordinates used:

screen coordnates	Origin in top-left corner of main display
client coordinates	Origin in top-left corner of window's client area
workspace coordinates	Origin in top-left corner of workspace area, i.e. screen area excluding taskbar and any other desktop toolbars

- Top-level window — position in screen coordinates
- Child window — position in parent's client coordinates
- Conversions: `ScreenToClient`, `ClientToScreen`
- Workspace area:
  - `GetWindowPlacement`, `SetWindowPlacement`
  - For main display: `SystemParametersInfoW` with `SPI_GETWORKAREA`, `SPI_SETWORKAREA`
  - For other displays: `GetMonitorInfoW`

Position and Size

Window keeps track of:

- Normal (tracking) size and position, i.e. when window is not minimized/maximized
- Current size and position — changes between normal, minimized, maximized size/position depending on current window state.
- Z-Order (order in which windows overlap each other)

Size includes any non-client elements (border, caption bar, menu, scroll bars)

Initial Position and Size

- Tracking size set by `x`, `y`, `nWidth`, `nHeight` passed to `CreateWindowExW`
- Screen coords for top-level windows, parent's client coords for child windows.
- If `x==CW_USEDEFAULT`, position selected by OS
- If `nWidth==CW_USEDEFAULT`, size selected by OS
- `CW_USEDEFAULT` only for overlapped windows, otherwise values are `0`

To create window with specific client area size:

- `AdjustWindowRectEx`, passing intended window styles
- Use `GetSystemMetrics` with `SM_CXVSCROLL`, `SM_CYHSCROLL` to account for scrollbars

Z-Order

- Order in which windows are drawn (i.e. which is visible on top of another).
- Affected by topmost flag (`WS_EX_TOPMOST`).
    - Only top-level windows can be topmost.
    - Topmost window can only own topmost windows (inverse not true)
    - Specifying topmost owner mark window as topmost.
    - Marking as topmost also marks owned windows (transitive).
    - Marking as non-topmost makes owner and owned windows non-topmost (transitive)
- Desktop acts as owner of unowned windows

Z-Order Rules

Top-level windows:

- Owned windows are always above owner
- All topmost windows are above non-topmost
- Giving window a topmost owner marks it as topmost
- Placing window above/below another moves there all related windows (*), moved group retains relative order
- Placing window above topmost window makes it topmost
- Placing window below non-topmost window makes it non-topmost

Child windows:

- Always on parent's client area, below any other top-level window that overlaps it
- Can only be positioned relative to siblings

* owner and owned by moved window (transitive) with shared topmost state, up to (but not including) common root of window being moved and the one it is placed above/below.

Z-Order Functions

Navigation:

- GetWindow (see previous slide ▸ here) — navigates in z-order windows in the same group:
  - Children of the same parent
  - Non-topmost top-level window
  - Topmost top-level windows
- GetTopWindow — shortcut for GetWindow w/ GW_CHILD, nullptr handle retrieves top-level windows.
- GetNextWindow — same as GetWindow

Changing Z-Order:

- Activating window places it to top of it's group (topmost or non-topmost)
- BringWindowToTop — places window on top of other windows in it's group, activates it if top-level.
- SetWindowPos, DeferWindowPos — see next slides.

Position and Size Functions

Check window size and position

- `GetWindowRect` — in screen coords (top-level) or parent's client coords (child)
- `GetClientRect` — client area rectangle in window's own client coords (i.e. $(x, y) = (0, 0)$)
- `GetWindowPlacement` — see previous slide ▸ here

Set window size and position:

- `MoveWindow` — modify position and size (can also force repaint)
- `SetWindowPlacement` — see previous slide ▸ here
- `SetWindowPos` — activate window, change its size, position, z-order, visibility
- `BeginDeferWindowPos`, `DeferWindowPos`, `EndDeferWindowPos` — change multiple windows simultaneously (avoid flicker)

User change:

- Position — dragging caption bar (if present)
- Size — dragging sizing border (if present)

Size and position change when minimizing, maximizing, restoring

SetWindowPos

```
BOOL SetWindowPos(HWND wnd, HWND insAfter, int x, int y, int cx, int cy, UINT flags)
```

- `x, y, cx, cy` — new position, size
- `insAfter` — new Z-Order

`HWND_BOTTOM`	Bottom of z-order (looses top-most)
`HWND_TOP`	Top of z-order
`HWND_TOPMOST`	Adds top-most
`HWND_NOTOPMOST`	Above all non-topmost windows
window handle	Below specified window

- Inactive window is by default also activated (which may further change Z-Order)

SetWindowPos

```
BOOL SetWindowPos(HWND wnd, HWND insAfter, int x, int y, int cx, int cy, UINT flags)
```

Flags:

SWP_NOMOVE	don’t change position, ignores `x` and `y`
SWP_NOSIZE	don’t resize, ignores `cx` and `cy`
SWP_NOZORDER	don’t change Z-Order, ignores `insAfter`
SWP_NOACTIVATE	don’t activate the window
SWP_NOOWNERZORDER	Z-Order change will not affect window’s owner
SWP_HIDEWINDOW SWP_SHOWWINDOW	change visibility state
SWP_FRAMECHANGED	applies new styles to window frame (it’s redrawn if needed)
SWP_DRAWFRAME	forces a redraw of window’s frame
SWP_NOSENDCHANGING	don’t send `WM_WINDOWPOSCHANGING`

Position and Size Messages

- WM_WINDOWPOSCHANGING, WM_WINDOWPOSCHANGED
  - Sent each time size, position, visibility, z-order changes (regardless of source)
  - lParam points to WINDOWPOS structure (fields correspond to SetWindowPos parameters)
  - On WM_WINDOWPOSCHANGING change fields to affect the outcome
- WM_MOVE, WM_SIZE
  - lParam — new position/size of window
  - wParam (WM_SIZE only) — indicates change in minimized/maximized state
  - Sent by DefWindowProcW on WM_WINDOWPOSCHANGED (thus can be prevented)
  - Legacy messages, more efficient to handle WM_WINDOWPOSCHANGED directly
- WM_GETMINMAXINFO
  - Sent before window changes size/position (usually by DefWindowProcW on WM_WINDOWPOSCHANGING, but not only)
  - lParam — points to GETMINMAXINFO
  - Modify it to change window's maximized position, size and minimum, maximum tracked size
- WM_NCCALCSIZE — Sent to calculate window's client area (see docs!)

Position and Size Messages

User dragging:

- `WM_SYSCOMMAND` w/ `SC_MOVE` or `SC_SIZE` sent once at start, cause `DefWindowProcW` to enter a modal loop
- `WM_ENTERSIZEMOVE` and `WM_EXITSIZEMOVE` sent once at the loop's start and end (respectively)
- `WM_MOVING`, `WM_SIZING`
  - Sent periodically while dragging
  - `lParam` — points to `RECT` with window's new location
  - Modify to change the outcome
  - `wParam` (`WM_SIZING` only) — edge/corner being dragged
- `WM_WINDOWPOSCHANGING`, `WM_WINDOWPOSCHANGED` also arrive while dragging, as with any change to size or position

Window and Window Class Data

- Use `GetClassLongPtrW`, `SetClassLongPtrW` to access, change any class info passed in `WNDCLASSEXW` upon registering, e.g.:

`GCL_STYLE`	class styles
`GCLP_HBRBACKGROUND`	background brush
`GCLP_WNDPROC`	window procedure

- Use `GetWindowLongPtrW`, `SetWindowLongPtrW` to access, change window data, e.g.:

`GWL_STYLE`, `GWL_EXSTYLE`	window styles
`GWLP_ID`	child window ID
`GWLP_WNDPROC`	window procedure

- Windows might require manual frame redraw on style changes, see previous slide ▸ here
- Replacing window procedure creates a subclass.
  - Change in class only affects windows created afterwards.
  - Change in window only affects that window.
  - New procedure should pass unhandled messages to the old one using `CallWindowProcW`

End of Windows API Lecture 2

Thank you for listening! ☺

Message Queue

Sent messages behaves differently depending on the owner thread of recipient window:

Sender thread	Window procedure called directly
Other thread	Message added to it’s incoming message queue

Messages stored in thread’s message queue as (depending on type):

Incoming messages	Sent from other threads
Posted messages	Posted by any thread
Input messages	Generated by system from input devices
Special flags	general: *quit*, *mouse moved*; each window: *repaint*; each timer: *elapsed*

Queued Messages

Incoming sent messages:

- Never returned by: `GetMessageW`, `PostMessageW`, etc.
- Processed directly inside calls to API functions (invisibly to the caller): `GetMessageW`, `PeekMessageW`, cross-thread `SendMessageW`, etc.
- Processing not affected by filters of `GetMessageW`, `PeekMessageW`, etc.

Input messages:

- Source: mouse, keyboard, *raw* input (if requested, mouse, keyboard, gamepads, etc.)
- Mouse movement only sets *mouse moved* flag
- Appending another input message clears that flag and inserts `WM_MOUSEMOVE` or `WM_NCMOUSEMOVE` before it.
- If mouse move message is inserted after another of the same type, they coalesce i.e. older one removed, parameters combined

Retrieving Messages

Messages retrieval functions process them based on priority:

1. Incoming messages

   Note: delivered directly (e.g. to window procedure), another lower-priority message will be returned
2. Posted messages
3. Generated `WM_QUIT` (*quit* flag)
4. Input messages (if required and *mouse moved* flag set, `WM_MOUSEMOVE` or `WM_NCMOUSEMOVE` is generated)
5. Generated `WM_PAINT` (any window's *repaint* flag)
6. Generated `WM_TIMER` (any timer's *elapsed* flag)

Generated Messages

Created on demand if:

- No other, higher-priority message exists
- Their respective flag is set
- They match given message filter
- Exception: `WM_QUIT` disregards filters

If created, but not removed (e.g. no `PM_REMOVE` in `PeekMessageW`, input messages list not empty when *mose move* messages created)

- `WM_PAINT`, `WM_TIMER` added to posted messages list
- `WM_NCMOUSEMOVE`, `WM_MOUSEMOVE` added to input messages list
- `WM_QUIT` never added anywhere (ignores lack of `PM_REMOVE`)

PeekMessageW in Pseudocode

Note: error checking and some flags omitted, not entirely inaccurate, but based on observable behaviour

```
BOOL PeekMessageW(LPMSG msg, HWND hWnd, UINT msgMin, UINT msgMax, UINT flags)
{
    Deliver all pending incoming sent messages;
    if (existing posted message matches filter) {
        *msg = that message;
        if (flags & PM_REMOVE) Remove it from posted messages list;
        return TRUE;
    }
    if (existing input message, excl. last mouse move, matches filter) {
        *msg = that message;
        if (flags & PM_REMOVE) Remove it from input message list;
        return TRUE;
    }
    ...
}
```

PeekMessageW in Pseudocode

```
BOOL PeekMessageW(LPMSG msg, HWND hWnd, UINT msgMin, UINT msgMax, UINT flags)
{
    ...
    if (quit queue flag set) {
        clear quit queue flag;
        *msg = a WM_QUIT message;
        return TRUE;
    }
    if (mouse moved queue flag set && mouse move message matches filter) {
        append WM_MOUSEMOVE or WM_NCMOUSEMOVE
            to the input messages list (with coalescing);
        *msg = first mouse move message in the input list
        if (flags & PM_REMOVE) Remove it from input message list
        return TRUE;
    }
    ...
}
```

PeekMessageW in Pseudocode

```
BOOL PeekMessageW(LPMSG msg, HWND hWnd, UINT msgMin, UINT msgMax, UINT flags)
{
    ...
    if (thread's window needs repainting && WM_PAINT matches filter) {
        //Doesn't clear windows repaint flag
        *msg = a WM_PAINT message;
        if (!(flags && PM_REMOVE)) append that message to posted messages list
        return TRUE;
    }
    if (thread's timer elapsed && WM_TIMER matches filter) {
        clear timer's elapsed flag
        *msg = a WM_TIMER message;
        if (!(flags && PM_REMOVE)) append that message to posted messages list
        return TRUE;
    }
    return FALSE;
}
```

Posting Cross-Thread Messages

```
BOOL PostMessageW(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
```

- Posts message to `hWnd`'s owner thread's message queue

```
BOOL PostThreadMessageW(DWORD threadId, UINT msg, WPARAM wParam)
```

- Posts message to thread's message queue (`GetCurrentThreadId`, `GetWindowThreadId`)
- Thread must have a message queue (possible, since no `HWND` required — in that case, before posting, call e.g. `PeekMessageW` once in the recipient thread to force queue creation)

Sending Cross-Thread Messages

```
LRESULT SendMessageTimeoutW(HWND hWnd, UINT msg,WPARAM wParam,
                            LPARAM lParam, UINT flags, UINT timeout)
```

- Sends message to `hWnd` (processed on it's owner thread).
- Blocks until message processes or timeout expires.
- Some available flags:

`SMTO_NORMAL`	calling thread processes incoming messages while waiting
`SMTO_BLOCK`	prevent calling thread from processing incoming messages
`SMTO_ABORTIFHUNG`	returns early if receiving thread is *not responding*
`SMTO_NOTIMEOUTIFNOTHUNG`	ignores timeout if receiving thread is *responding*

- If timeout expires before receiving thread starts processing the message, it is remove.
- However, it can timeout after message processing started (impossible to cancel message).

```
LRESULT SendMessageW(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
```

- Same as `SendMessageTimeoutW` with `INFINITE` timeout and `SMTO_NORMAL`.

Sending Cross-Thread Messages

```
BOOL SendMessageCallbackW(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam,
                          SENDASYNCPROC callback, ULONG_PTR data)
```

- Sends message to `hWnd` (processed on it’s owner thread).
- Returns immediately
- When message is processed, callback is executed, passing the result and `data` (on sender thread, sender must have a message loop)
- Callback prototype:
  `void SendAsyncProc(HWND hWnd, UINT msg, ULONG_PTR data, LRESULT result)`

```
BOOL SendNotifyMessageW(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
```

- Same as `SendMessageCallbackW` with `nullptr` callback

Broadcasting Messages

Sending/posting message broadcasts:

- Use `HWND_BROADCAST` as `hWnd`.
- Recipients: all top-level windows in the system (including disabled, invisible unowned windows)
- Only registered messages should be broadcasted.
- In `SendMessageTimeoutW` timeout applies to each recipient separately (i.e. function can wait up to timeout times number of recipients.)
- In `SendMessageCallbackW` callback is called for each recipient window.
- `BroadcastSystemMessageW`, `BroadcastSystemMessageExW` for more options for posting/sending broadcasted messages.

Synchronisation

Asynchronous calls (don’t wait): `PostMessageW`, `SendMessageCallbackW`, `SendNotifyMessageW`

- Will fail for system-defined messages whose parameters contain pointers.
- For custom messages, program should provide proper marshalling, synchronisation, etc.
- `SendMessageTimeoutW` can timeout while other thread is in the middle of processing, be careful about freeing resources, etc.

Passing result back to sender:

- When window procedure exits on recipient thread, system sends internal message to sender.
- Recipient can call `ReplyMessage` to provide result (and possibly unblock sender) early
- On sender thread those internal messages are processed to provide result of `SendMessageW`, `SendMessageTimeoutW` or run callback of `SendMessageCallbackW`

Cross-Thread Input Attachment

Thread group

- Threads that share input message queue
- Created (or expanded) by:
  - Establishing cross-thread parent/child or owner/owned relationships
  - Calling `AttachThreadInput` (can also break attachment)
- Share input state (keyboard state, active window, focus, etc.)
- Input messages are synchronised, thread will not receive input messages if:
  - First input message belongs to another thread
  - Another thread has received input message and hasn't indicated that processing was done (next call to `GetMessageW`, `PeekMessageW`, `SendMessageW`, etc.)
- Can lead to difficult to debug deadlocks due to input synchronisation, thus should be avoided (see here and here)
- Attaching to thread of a non-cooperating process is a bad idea (see here)