

WinAPI in ASM

WINAPI IN ASSEMBLY INTRODUCTION	2
DISPLAYING A MESSAGEBOX	18
CONSOLE INPUT.....	25
CHECKING FOR ERRORS	26
SINGLE CHARACTER INPUT	27
CONSOLE OUTPUT	34
FILE HANDLING.....	40
WRITEFILE AND SETFILEPOINTER	45
CONSOLE WINDOW MANIPULATION.....	52
SETTING TEXT COLOR.....	60
TIME, WINAPI AND ASSEMBLY.....	63
CALLING 64-BIT WINAPI FUNCTION IN MASM.....	72
DYNAMIC MEMORY	76
x86 MEMORY MANAGEMENT.....	81

WINAPI IN ASSEMBLY INTRODUCTION

When a Windows application starts up, it can launch in one of two ways:
either as a **console application** or as a **graphical (windowed) application**.

In our project files, we tell the linker that we want a console-based program by using this option with the LINK command:

```
/SUBSYSTEM:CONSOLE
```

A console program looks a lot like the old MS-DOS window—but with some modern upgrades we'll get into soon.

Under the hood, the console includes:

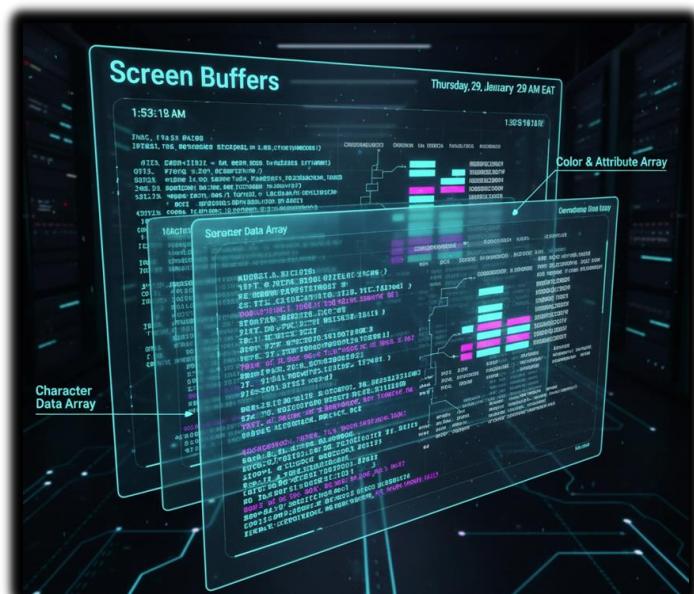
One input buffer, which stores input records. These records describe things like:

- Keyboard input
- Mouse clicks
- User actions such as resizing the console window

One or more screen buffers, which are two-dimensional arrays holding:

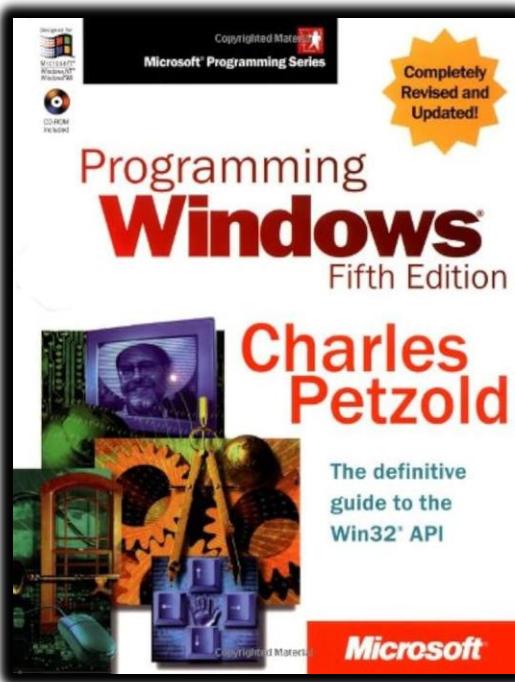
- Character data
- Color and attribute information

These screen buffers control how text actually appears inside the console window.



Reference Material (a.k.a. where the real answers live)

- I've put Win32 API reference info in **my own GitHub repository**. If you don't see it, just email me and I'll send you a ZIP file.
- **Charles Petzold's book** is another classic reference. Fair warning: it's old-school—huge, technical, no colors, no hand-holding. If you enjoy reading 1000-page technical manuals without falling asleep, it's for you. If you want the physical book... well, more power to you—go buy it 😊



Key Takeaways from This Section

- This section introduces **a small subset of Win32 API functions**, using simple examples.
It's meant as a starting point, not a complete reference.
- For full documentation, the **Microsoft MSDN website** is the authoritative source. When searching, make sure to filter for "**Platform SDK**".
- The sample programs include lists of function names found in:
 - ⊕ kernel32.lib
 - ⊕ user32.libThese are provided mainly for reference.
- Win32 API functions often rely on **named constants**, such as:

In Microsoft's MSDN Library documentation, the trailing "A" or "W" is typically omitted from function names.

In the program include files provided with this book, functions such as WriteConsoleA are redefined as follows:

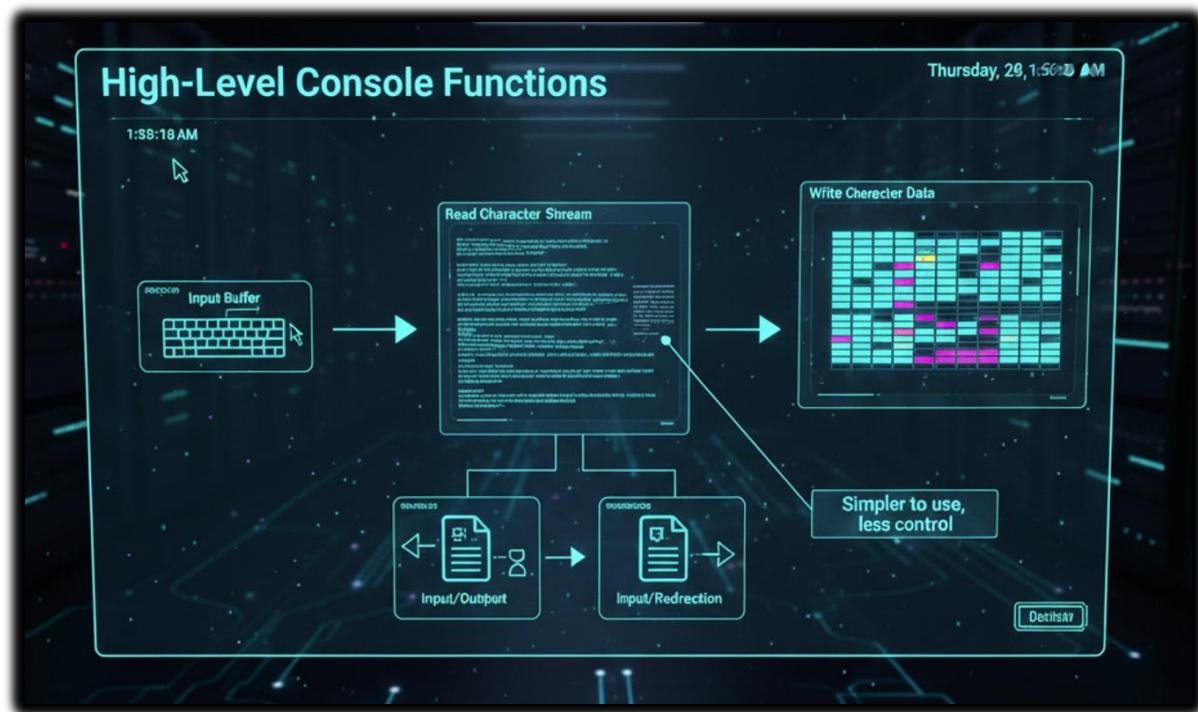
```
WriteConsole EQU <WriteConsoleA>
```

This definition allows you to call WriteConsole using a generic function name, without explicitly specifying the character set.

There are two levels of console access, each balancing ease of use and control:

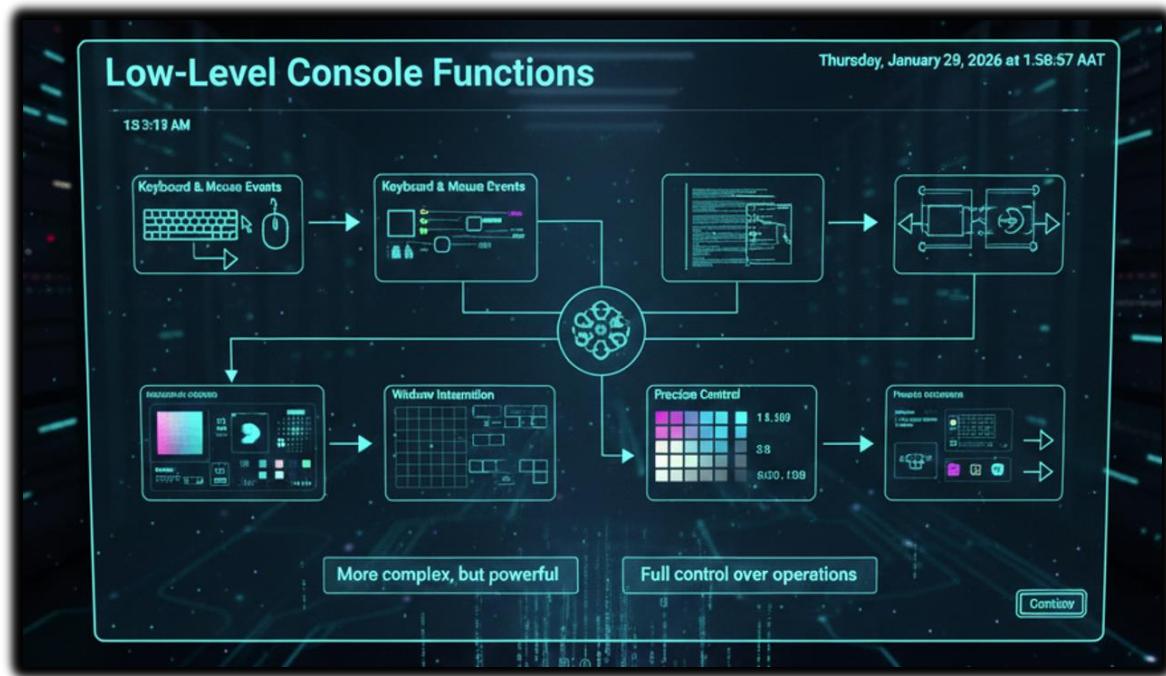
High-Level Console Functions

- Read a stream of characters from the console's input buffer.
- Write character data to the console's screen buffer.
- Support input and output redirection to and from text files.
- Simpler to use, but offer less control over console behavior.



Low-Level Console Functions

- Provide detailed information about keyboard and mouse events.
- Capture user interactions with the console window (such as dragging and resizing).
- Allow precise control over the console window's size, position, and text colors.
- More complex, but offer full control over console operations.



Windows Data Types

MASM provides its own versions of the standard MS-Windows data types. The mappings are pretty straightforward, once you see how they line up.

MASM ↔ Windows Type Mappings

MS-Windows Type	MASM Type	Description
BOOL, BOOLEAN	DWORD	A boolean value (TRUE or FALSE)
BYTE	BYTE	An 8-bit unsigned integer
CHAR	BYTE	An 8-bit Windows ANSI character

In other words, these MASM types are effectively equivalent to their Windows counterparts:

```
DWORD = BOOL = BOOLEAN  
BYTE = CHAR
```

So, if you see a Windows function expecting a **BOOL**, using a **DWORD** in MASM is perfectly correct.

A Note About **HANDLEs** (Important!)

One thing that often surprises people:
the Windows **HANDLE** type is **also just a DWORD**.

That means a single **HANDLE** variable can store a reference to many different kinds of objects, such as:

- A window
- A file
- A memory region
- Other system objects

The meaning of the handle depends entirely on **which API function returned it**—not on the data type itself.

Here is an example of how to declare and use a **HANDLE** variable in MASM:

```
07 handleVariable: DWORD  
08  
09 ; Get a handle to the console window.  
10 invoke GetConsoleWindow, handleVariable  
11  
12 ; Use the handle to write a message to the console.  
13 invoke WriteConsole, handleVariable, addr message, length message, bytesWritten, NULL  
14  
15 ; Close the handle to the console window.  
16 invoke CloseHandle, handleVariable
```

The **SmallWin.inc** include file contains constant definitions, text equates, and function prototypes used in Win32 API programming.

It is automatically included in programs through **Irvine32.inc**.

The file defines several Win32 data types, including the **HANDLE** type.

Below are examples demonstrating how to use the **SmallWin.inc** include file:

```
20 ; Get a handle to the standard input handle.  
21 invoke GetStdHandle, STD_INPUT_HANDLE, handleVariable  
22  
23 ; Get a handle to the standard output handle.  
24 invoke GetStdHandle, STD_OUTPUT_HANDLE, handleVariable  
25  
26 ; Get a handle to the standard error handle.  
27 invoke GetStdHandle, STD_ERROR_HANDLE, handleVariable
```

SmallWin.inc and Win32 Data Types (MASM-Friendly Overview)

The SmallWin.inc include file exists to make **Win32 API programming in MASM less painful**.

It defines common Windows data types, structures, constants, and function prototypes so you don't have to reinvent the wheel every time.

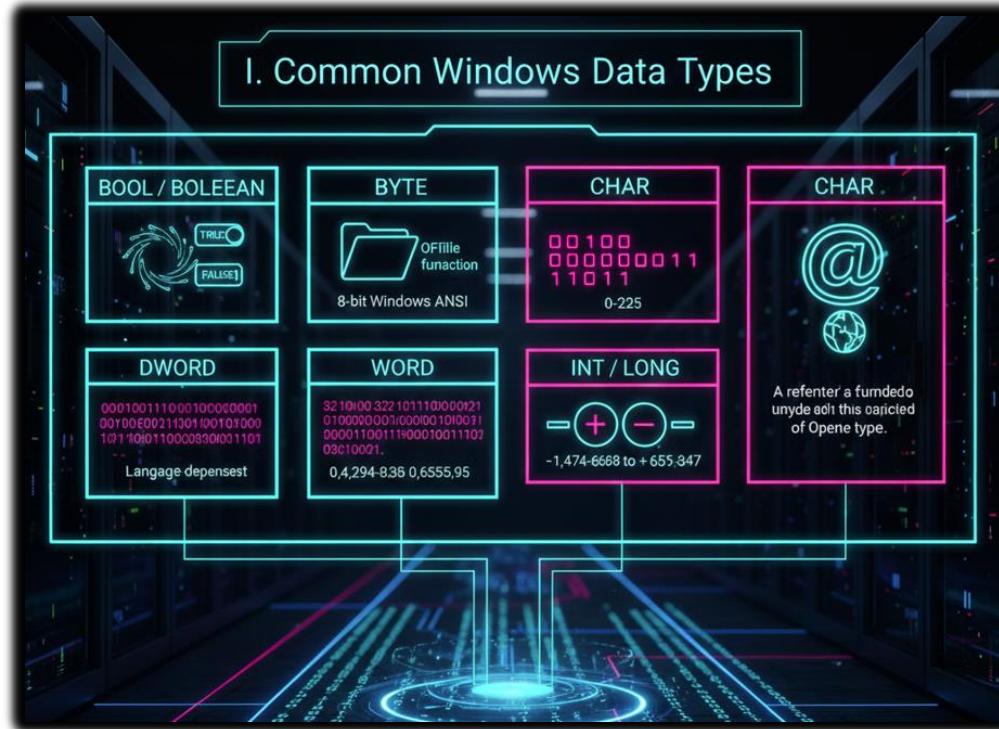
It's automatically included when you use Irvine32.inc, so in most MASM projects, you already have access to it.

I. Common Windows Data Types

Here's what the most common Win32 data types actually mean, without the legalese:

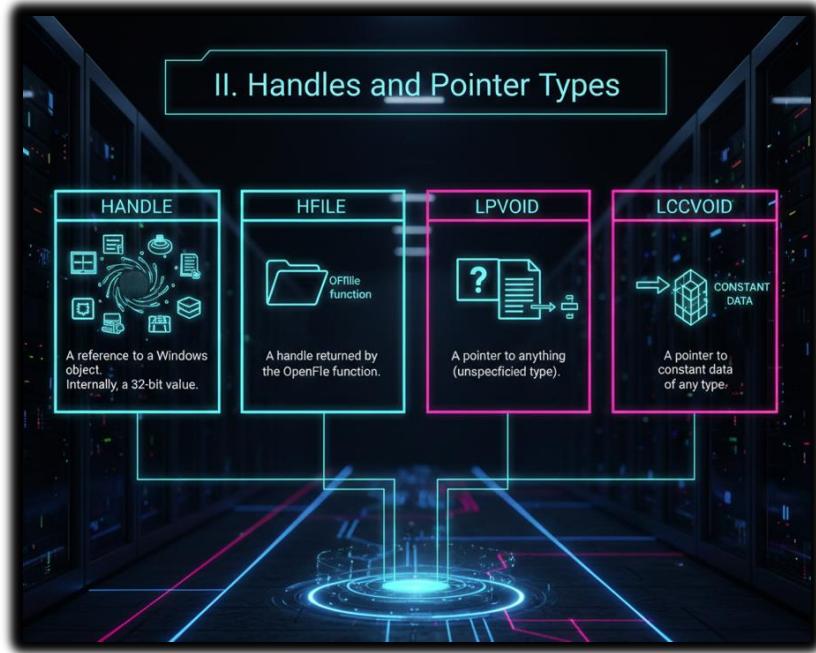
- **BOOL / BOOLEAN**
A simple true-or-false value.
- **BYTE**
An 8-bit unsigned integer (0–255).
- **CHAR**
An 8-bit Windows ANSI character.
This is what older (non-Unicode) Windows programs use for text. The exact character set depends on language and region.
- **DWORD**
A 32-bit unsigned integer (0–4,294,967,295).
- **WORD**
A 16-bit unsigned integer (0–65,535).

- **INT / LONG**
A 32-bit signed integer
(-2,147,483,648 to +2,147,483,647).
INT and LONG are the same size in Win32.
- **UINT**
A 32-bit unsigned integer, functionally the same as DWORD.



II. Handles and Pointer Types

- **HANDLE**
A reference to a Windows object (window, file, memory region, etc.).
Internally, it's just a 32-bit value, but *what it refers to* depends on how you got it.
- **HFILE**
A handle returned by the OpenFile function.
- **LPVOID**
A pointer to *anything* (unspecified type).
- **LPCVOID**
A pointer to constant data of any type.



III. String and Text Pointer Types

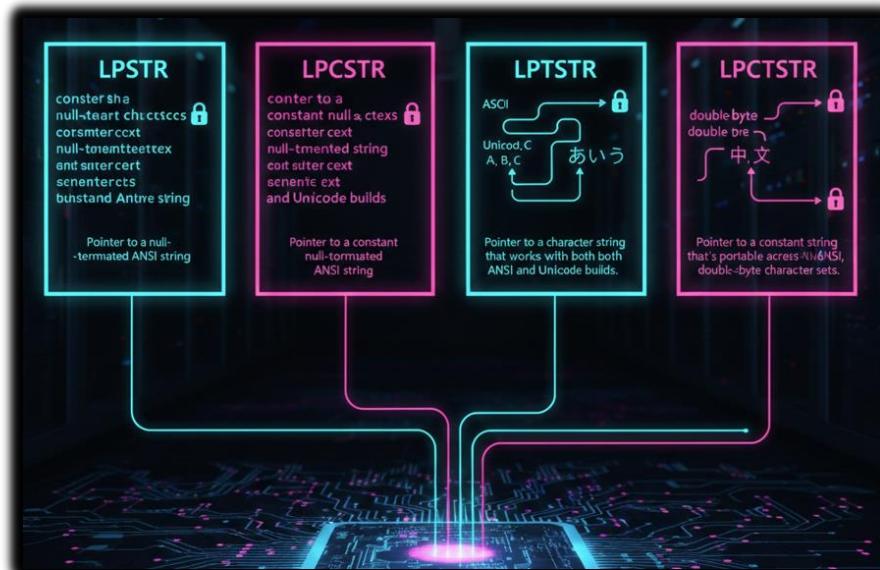
These show up everywhere in the Windows API:

LPSTR - Pointer to a null-terminated ANSI string.

LPCSTR - Pointer to a constant null-terminated ANSI string.

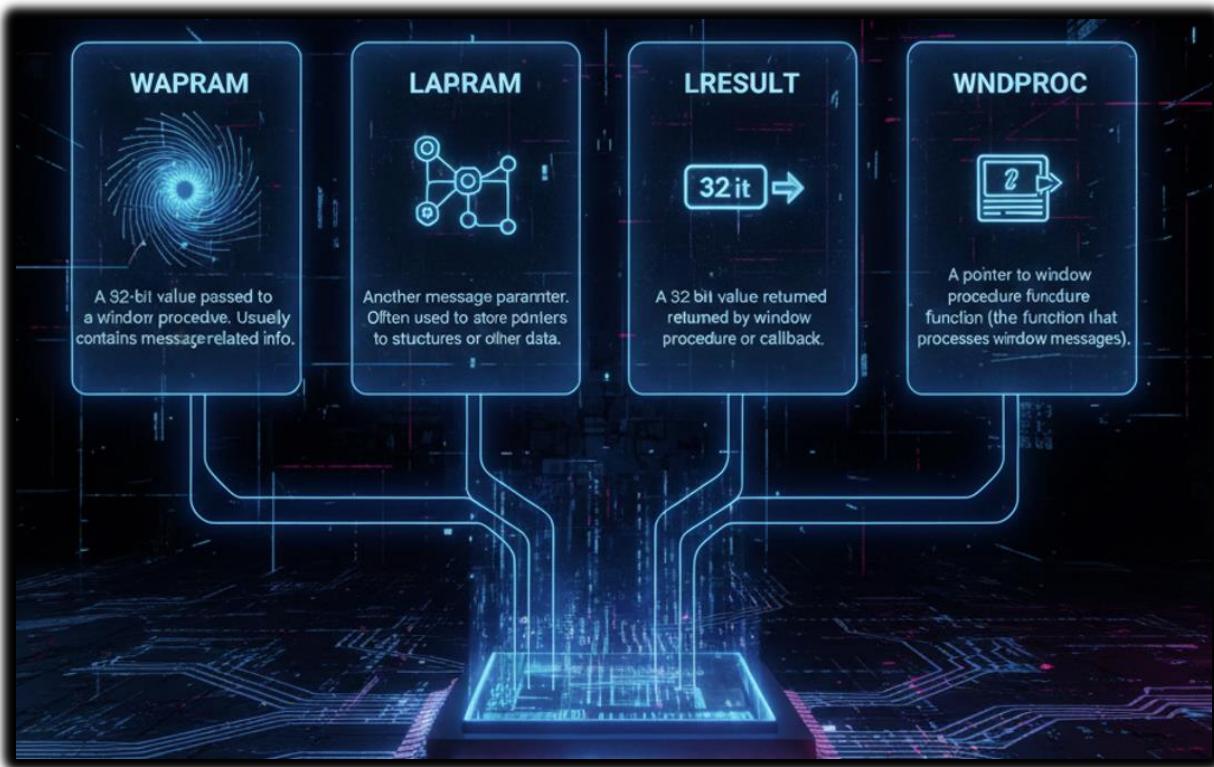
LPTSTR - Pointer to a character string that works with both ANSI and Unicode builds.

LPCTSTR - Pointer to a constant string that's portable across ANSI, Unicode, and double-byte character sets (used in languages like Chinese and Japanese).



IV. Message and Callback Types

- **WPARAM**
A 32-bit value passed to a window procedure.
Usually contains message-related info.
- **LPARAM**
Another message parameter.
Often used to store pointers to structures or other data.
- **LRESULT**
A 32-bit value returned by a window procedure or callback.
- **WNDPROC**
A pointer to a window procedure function (the function that processes window messages).



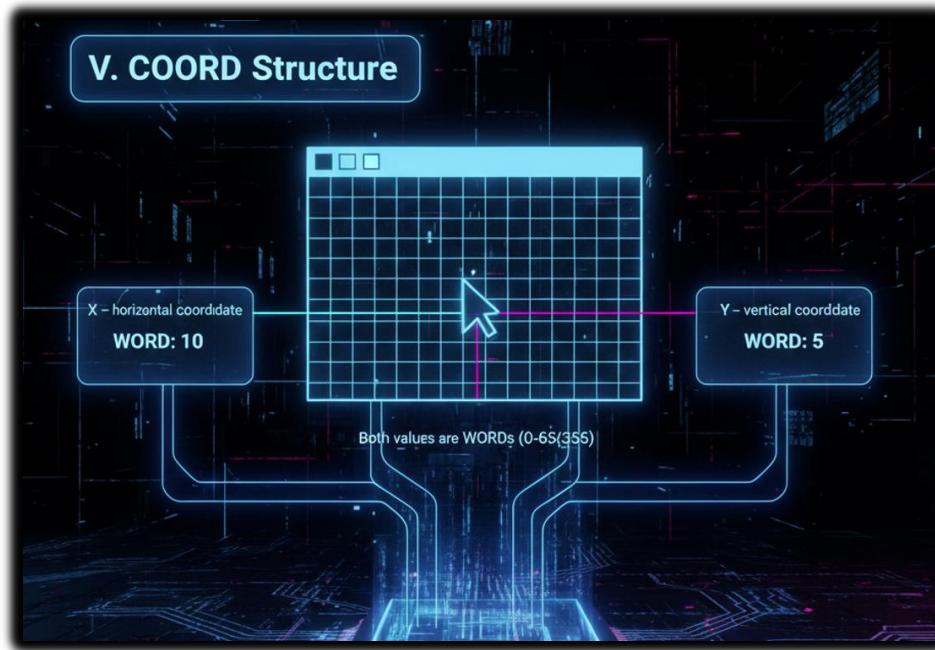
Special Structures

I. COORD Structure

Used to store a position in the console window.

- X – horizontal coordinate
- Y – vertical coordinate

Both values are WORDs.



II. SYSTEMTIME Structure

Stores detailed date and time information:

- wYear – year
- wMonth – month
- wDayOfWeek – day of the week
- wDay – day of the month
- wHour – hour
- wMinute – minute
- wSecond – second
- wMilliseconds – milliseconds

II. SYSTEMTIME Structure



Console Handles (The Backbone of Console I/O)

Console handles are **32-bit values** that identify console devices.

Win32 console functions use these handles to perform input and output.

The Three Standard Console Handles

- **STD_INPUT_HANDLE** – keyboard input
- **STD_OUTPUT_HANDLE** – console display output
- **STD_ERROR_HANDLE** – error message output

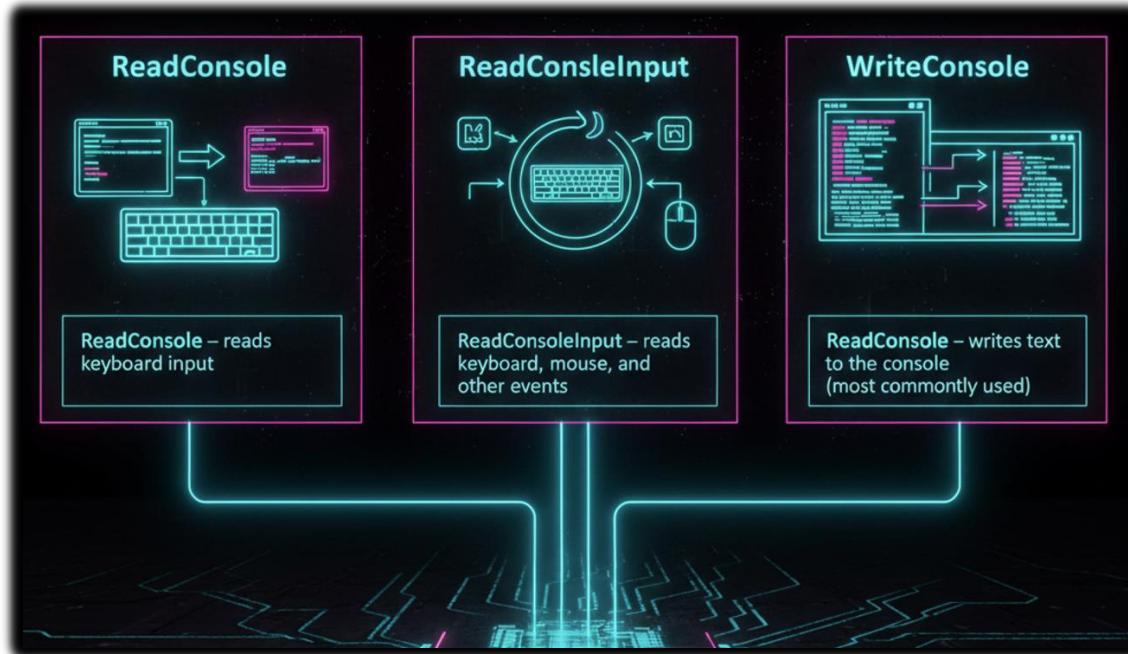
You retrieve these using: **GetStdHandle**

Once you have a handle, you can use it to read input, write output, or control the console's behavior.

Common Console I/O Functions

I. Input and Output

- **ReadConsole** – reads keyboard input
- **ReadConsoleInput** – reads keyboard, mouse, and other events
- **WriteConsole** – writes text to the console (most commonly used)



II. Screen Buffer Access

- **CreateConsoleScreenBuffer** – creates a new screen buffer
- **SetConsoleActiveScreenBuffer** – switches which buffer is visible
- **ReadConsoleOutput** – reads characters and colors from the screen
- **WriteConsoleOutput** – writes characters and colors to the screen

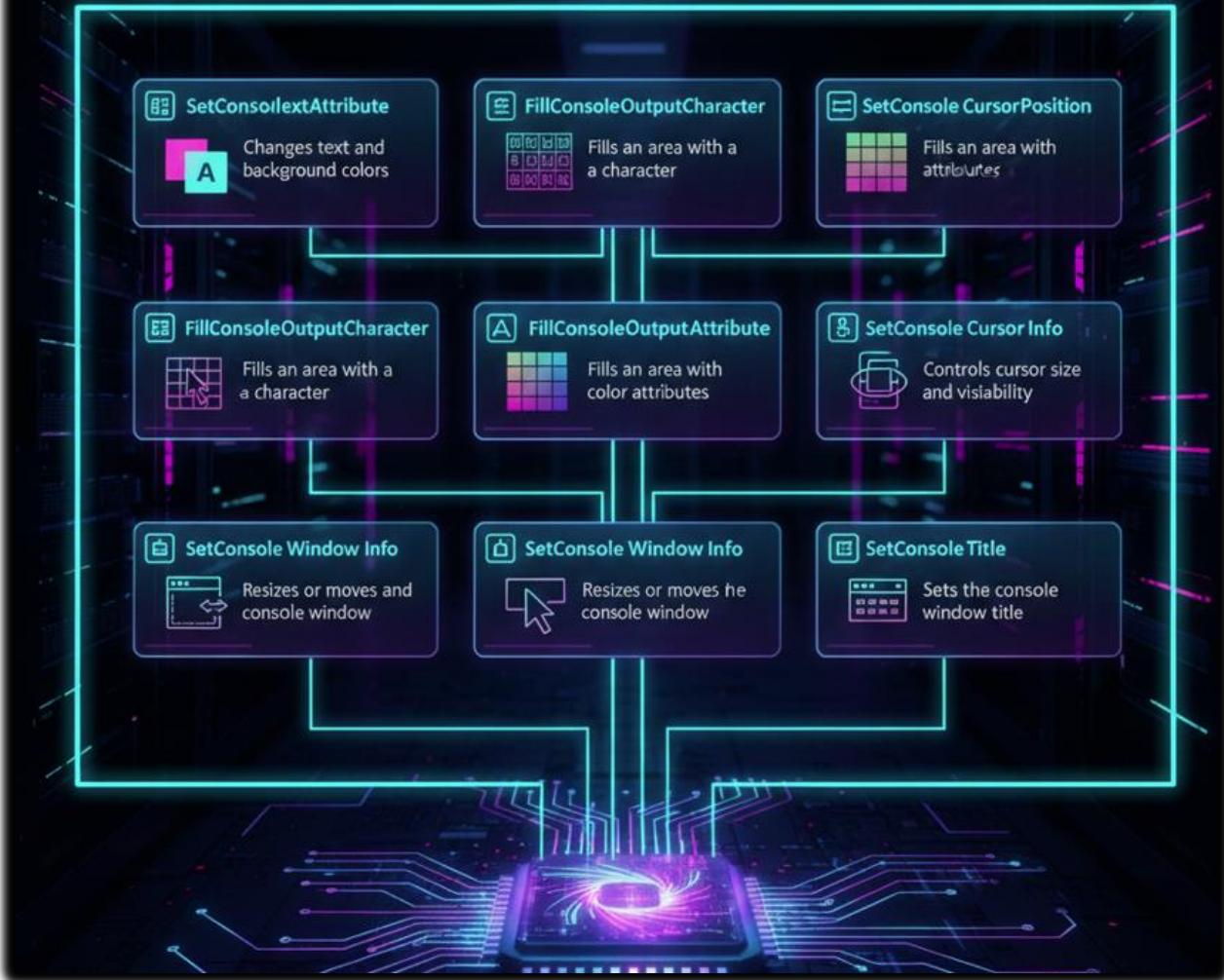
Windows Console Buffer Management		
API FUNCTION	RETURN / PARAMS	CORE PURPOSE
<code>CreateConsoleScreenBuffer</code>	HANDLE Access, Share, Security, Flags	Allocates a private off-screen virtual grid. Essential for "Page Flipping" to prevent flicker.
<code>WriteConsoleOutput</code>	BOOL hBuffer, CHAR_INFO[], Size, Coord, Region	The "Blitter." Blasts a rectangular block of characters and colors from an array directly into memory.
<code>SetConsoleActiveScreenBuffer</code>	BOOL hConsoleOutput	The "Flip." Instantly swaps the visible window to display the prepared off-screen buffer.



III. Console Appearance and Behavior

- **SetConsoleTextAttribute** – changes text and background colors
- **FillConsoleOutputCharacter** – fills an area with a character
- **FillConsoleOutputAttribute** – fills an area with color attributes
- **SetConsoleCursorPosition** – moves the cursor
- **SetConsoleCursorInfo** – controls cursor size and visibility
- **SetConsoleWindowInfo** – resizes or moves the console window
- **SetConsoleTitle** – sets the console window title

II. Console Appearance and Behavior

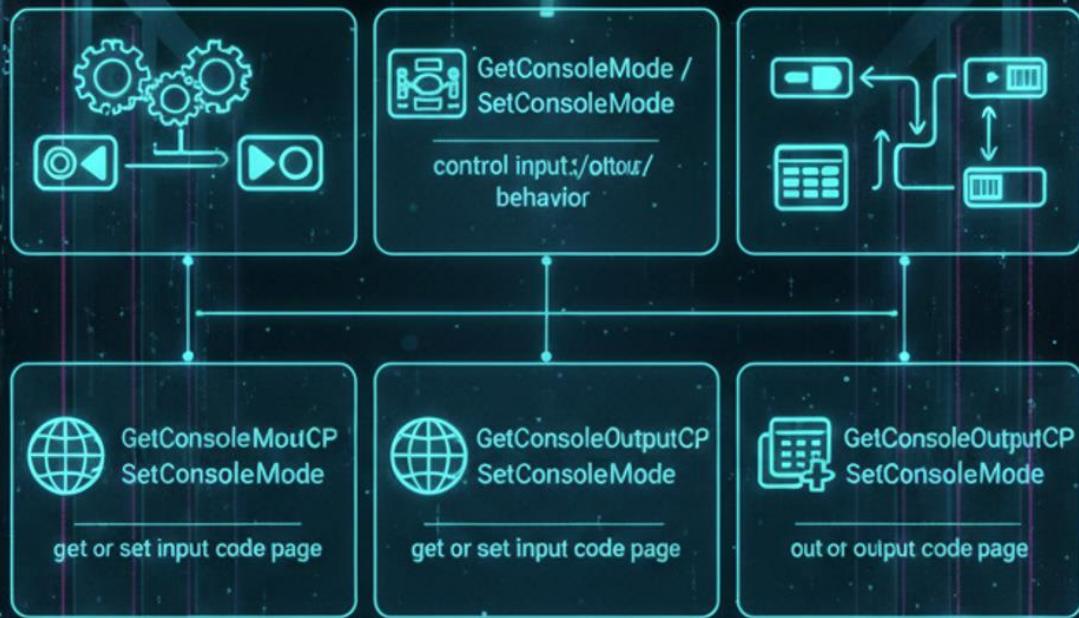


IV. Console Modes and Code Pages

- `GetConsoleMode` / `SetConsoleMode` – control input/output behavior
- `GetConsoleCP` / `SetConsoleCP` – get or set input code page
- `GetConsoleOutputCP` / `SetConsoleOutputCP` – get or set output code page

These are especially important for international character support.

IV. Console Modes and Code Pages

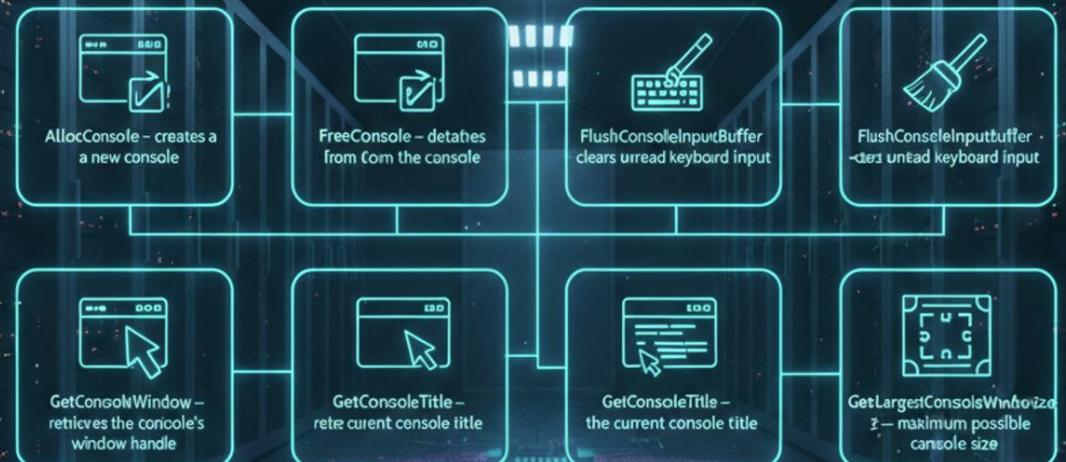


Especially important for international character support

V. Console Management and Control

- **AllocConsole** – creates a new console
- **FreeConsole** – detaches from the console
- **FlushConsoleInputBuffer** – clears unread keyboard input
- **GetConsoleWindow** – retrieves the console's window handle
- **GetConsoleTitle** – retrieves the current console title
- **GetLargestConsoleWindowSize** – maximum possible console size

V. Console Management and Control



VI. Console Events and Signals

- **SetConsoleCtrlHandler** – registers a handler function
- **HandlerRoutine** – your custom handler function
- **GenerateConsoleCtrlEvent** – sends control signals (close, terminate, etc.)

VI. Console Events and Signals



VII. Why This Matters

Console handles and Win32 console functions are **the core of console-based Windows programs**.

Once you understand how handles, buffers, and modes fit together, you can:

- Build responsive console apps
- Control text layout and color
- Handle keyboard and mouse input
- Manage multiple screen buffers

In short: **this is where simple DOS-style programs turn into real Windows applications.**

DISPLAYING A MESSAGEBOX

In Win32 console applications, you can use the **MessageBoxA** function to display a message box to the user.

The MessageBoxA function requires **four parameters**:

hWnd

- A handle to the window that owns the message box.
- If this parameter is NULL, the message box is created as a top-level window.

lpText

- A pointer to the text displayed inside the message box.

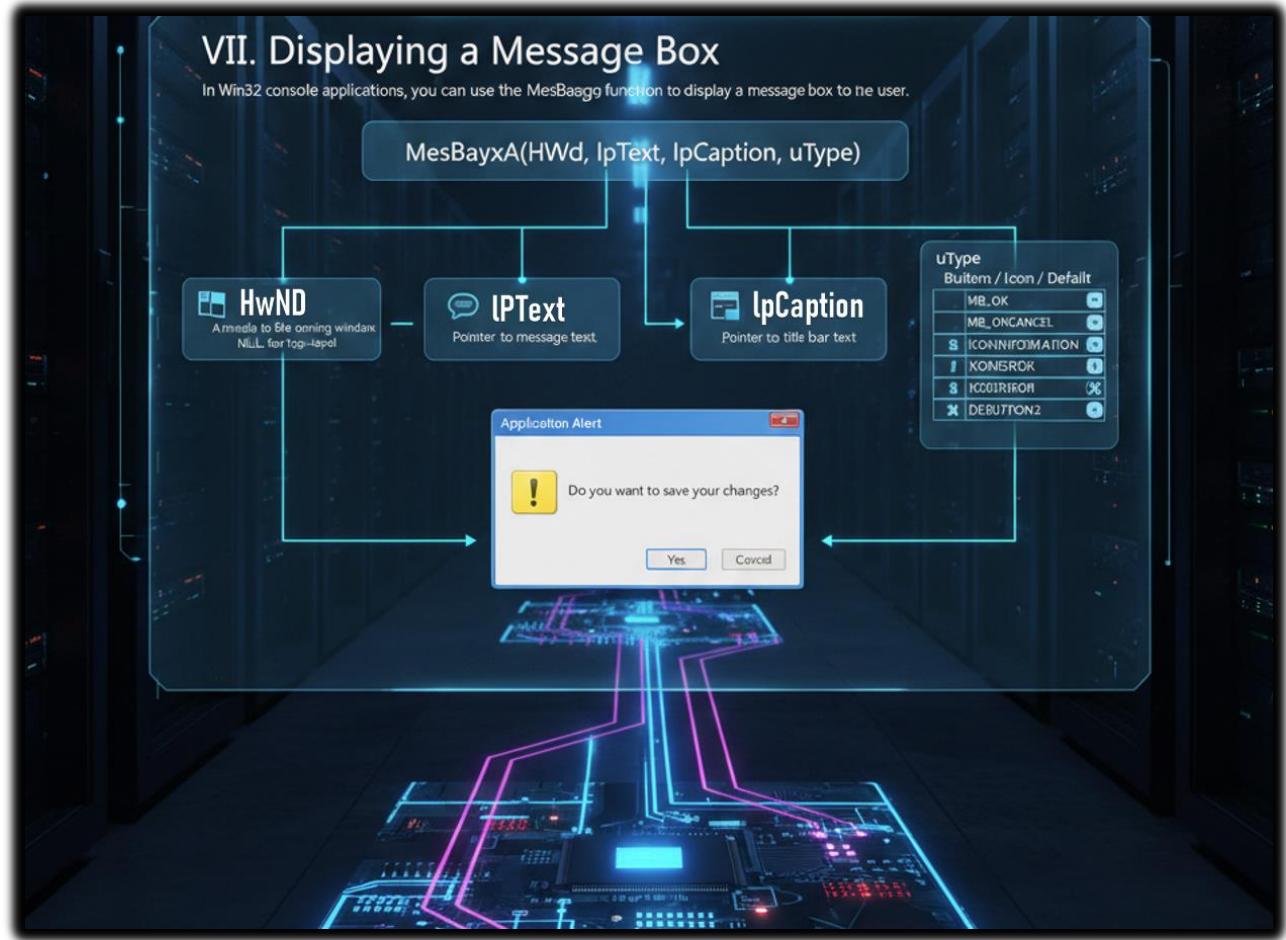
lpCaption

- A pointer to the text displayed in the message box's title bar.

uType

- A bit-mapped integer that specifies the type of message box.
- This parameter controls:
 - The buttons displayed (e.g., OK, Yes/No)
 - The icon displayed
 - The default selected button

The following table lists some of the possible values for the **uType** parameter:



We learnt all this in the Windows API notes.

MessageBoxA: uType Flag Reference

WinUser.h

CONSTANT (SYMBOLIC)	INTEGER	BEHAVIOR DESCRIPTION
MB_OK	0	Default. Displays a message box with one OK button.
MB_OKCANCEL	1	Displays OK and Cancel buttons.
MB_ABORTRETRYIGNORE	2	Displays Abort , Retry , and Ignore buttons.
MB_YESNOCANCEL	3	Displays Yes , No , and Cancel buttons.
MB_YESNO	4	Displays Yes and No buttons.
MB_RETRYCANCEL	5	Displays Retry and Cancel buttons.
MB_CANCELTRYCONTINUE	6	Displays Cancel , Try Again , and Continue .
MB_ICONSTOP	16	Displays a Stop-sign icon (Error).
MB_ICONQUESTION	32	Displays a Question-mark icon.
MB_ICONEXCLAMATION	48	Displays an Exclamation-point icon (Warning).
MB_ICONINFORMATION	64	Displays an Information-symbol icon.

The **values in the table** are commonly used as **integer constants** for the **uType** parameter when creating message boxes.

The **uType** parameter determines the **type and behavior** of the message box.

In your program, you can use these constants like this:

```
30 import ctypes
31
32 # Display a message box with an OK button
33 ctypes.windll.user32.MessageBoxW(0, 'This is an example', 'MessageBox', 0x00000000)
34
35 # Display a message box with Yes and No buttons
36 ctypes.windll.user32.MessageBoxW(0, 'Question?', 'MessageBox', 0x00000004)
37
38 # Display a message box with an exclamation-point icon
39 ctypes.windll.user32.MessageBoxW(0, 'Warning!', 'MessageBox', 0x00000030)
```

Or

```
45 #include <windows.h>
46
47 int main() {
48     ;Display a message box with an OK button
49     MessageBoxW(NULL, L"This is an example", L"MessageBox", MB_OK);
50
51     ;Display a message box with Yes and No buttons
52     int result = MessageBoxW(NULL, L"Question?", L"MessageBox", MB_YESNO);
53
54     ;Check the user's response
55     if (result == IDYES) {
56         ;User clicked Yes
57     } else if (result == IDNO) {
58         ;User clicked No
59     }
60
61     ;Display a message box with an exclamation-point icon
62     MessageBoxW(NULL, L"Warning!", L"MessageBox", MB_ICONEXCLAMATION);
63
64     return 0;
65 }
```

In this C program, different **message box types** are used, including:

- MB_OK – displays a single OK button
- MB_YESNO – displays Yes and No buttons
- MB_ICONEXCLAMATION – displays an exclamation icon

To specify the message box type, pass the corresponding **integer constant** as the **third parameter** to the MessageBoxW function.

The program can **check the user's response** to the Yes and No buttons by examining the value returned by the function.

- For example, IDYES indicates the user clicked Yes.
- IDNO indicates the user clicked No.

Or

```

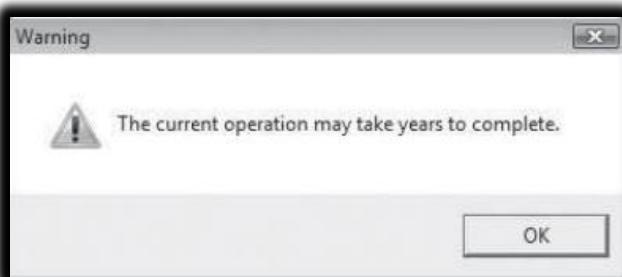
070 #include <windows.h>
071
072 int main() {
073     ;Display a message box with an OK button
074     MessageBoxW(NULL, L"This is an example", L"MessageBox", 0);
075
076     ;Display a message box with Yes and No buttons
077     int result = MessageBoxW(NULL, L"Question?", L"MessageBox", 4);
078
079     ;Check the user's response
080     if (result == 6) {
081         ;User clicked Yes
082     } else if (result == 7) {
083         ;User clicked No
084     }
085
086     ;Display a message box with an exclamation-point icon
087     MessageBoxW(NULL, L"Warning!", L"MessageBox", 48);
088
089     return 0;
090 }
```

- In this code, **integer values** are used directly to specify message box types.
- Examples of integer values and their corresponding constants:
 - ⚠ 0 → MB_OK (single OK button)
 - ⚠ 4 → MB_YESNO (Yes and No buttons)
 - ⚠ 48 → MB_ICONEXCLAMATION (exclamation icon)
- You can use these values to create message boxes with the **desired type and behavior** in your C program.

Demonstration Program

Program that demonstrates some capabilities of the MessageBoxA function.

The first function call displays a warning message:

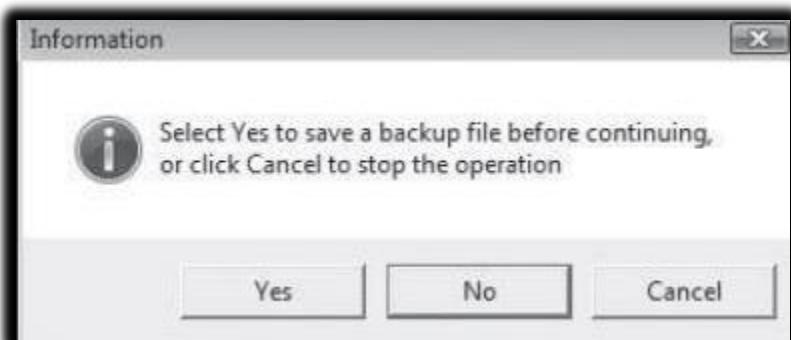


The second function call displays a question icon and Yes/No buttons.

If the user selects the Yes button, the program could use the return value to select a course of action:



The third function call displays an information icon with three buttons:



The fourth function call displays a stop icon with an OK button:



```

; Demonstrate MessageBoxA (MessageBox.asm)
; Shows how to create message boxes with different icons and buttons in Win32 API

INCLUDE Irvine32.inc

.data
; ----- Strings for MessageBox captions and messages -----
captionW BYTE "Warning",0
warningMsg BYTE "The current operation may take years to complete.",0

captionQ BYTE "Question",0
questionMsg BYTE "A matching user account was not found.",0dh,0ah,"Do you wish to continue?",0

captionC BYTE "Information",0
infoMsg BYTE "Select Yes to save a backup file before continuing.",0dh,0ah,"or click Cancel to stop the operation",0

captionH BYTE "Cannot View User List",0
haltMsg BYTE "This operation not supported by your user account.",0

.code
main PROC
    ; ----- Warning message box (Exclamation icon + OK button) -----
    INVOKE MessageBox, NULL, ADDR warningMsg, ADDR captionW, MB_OK + MB_ICONEXCLAMATION

    ; ----- Question message box (Question icon + Yes/No buttons) -----
    INVOKE MessageBox, NULL, ADDR questionMsg, ADDR captionQ, MB_YESNO + MB_ICONQUESTION

    ; Check if the user clicked "Yes" (eax holds the return value)
    cmp eax, IDYES
    ; You could add logic here to handle the "Yes" response if needed

    ; ----- Information message box (Information icon + Yes/No/Cancel buttons) -----
    ; Default button is set to No (MB_DEFBUTTON2)
    INVOKE MessageBox, NULL, ADDR infoMsg, ADDR captionC, MB_YESNOCANCEL + MB_ICONINFORMATION + MB_DEFBUTTON2

    ; ----- Stop message box (Stop icon + OK button) -----
    INVOKE MessageBox, NULL, ADDR haltMsg, ADDR captionH, MB_OK + MB_ICONSTOP

    ; Exit the program
    exit
main ENDP
END main

```

Includes Irvine32.inc which provides macros like INVOKE and constants for Win32 API.

.data section defines captions and messages for each message box, null-terminated.

main PROC is the entry point where all message boxes are displayed:

- MessageBox is called with parameters: hWnd, message, caption, and type (uType).
- uType combines **buttons** + **icons** + optional **default button**.

The **return value in eax** indicates which button the user clicked:

- IDYES → Yes
- IDNO → No
- IDCANCEL → Cancel

After all message boxes are shown, the program exits cleanly.

CONSOLE INPUT

```
140 ; Read From the Console (ReadConsole.asm)
141 INCLUDE Irvine32.inc
142
143 BufSize = 80
144
145 .data
146 buffer BYTE BufSize DUP(?), 0, 0
147 stdInHandle HANDLE ?
148 bytesRead DWORD ?
149
150 .code
151 main PROC
152     ; Get handle to standard input
153     INVOKE GetStdHandle, STD_INPUT_HANDLE
154     mov stdInHandle, eax
155
156     ; Wait for user input
157     INVOKE ReadConsole, stdInHandle, ADDR buffer, BufSize, ADDR bytesRead, 0
158
159     ; Display the buffer
160     mov esi, OFFSET buffer
161     mov ecx, bytesRead
162     mov ebx, TYPE buffer
163     call DumpMem
164
165     exit
166 main ENDP
167
168 END main
```

In Win32 console programming, there is a **console input buffer** that stores **input event records** (keystrokes, mouse movements, and mouse clicks).

High-level input functions, like ReadConsole, process these events and return a **stream of characters** to the program.

ReadConsole function:

- Reads text input from the console and stores it in a buffer.
- Parameters include:
 - ⊕ Console input handle
 - ⊕ Pointer to a character buffer
 - ⊕ Number of characters to read
 - ⊕ Pointer to store the count of characters read
 - ⊕ Reserved parameter (must be 0)

Example program workflow:

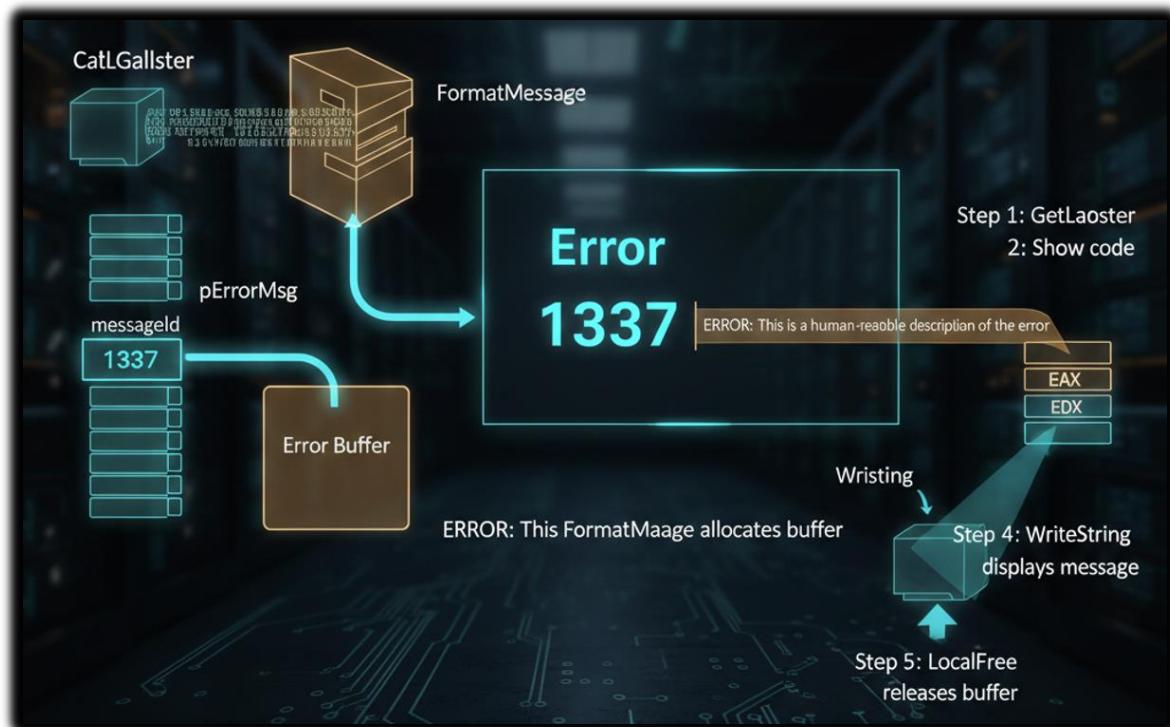
- Defines **buffer size** (BufSize) and declares necessary data variables:
 - ⊕ Buffer for storing input
 - ⊕ Handle for standard input (stdInHandle)
 - ⊕ Variable for number of bytes read (bytesRead)
- Retrieves **standard input handle** using GetStdHandle.
- Calls **ReadConsole** to read input from the user.
- Parameters include standard input handle, buffer, max characters to read, pointer for characters read, and reserved value 0.

After reading input, the program displays the buffer content using **DumpMem** (from Irvine32 library), showing both **hexadecimal and ASCII representations**.

The program can display **all user input**, including end-of-line characters (0Dh and 0Ah) from pressing Enter.

CHECKING FOR ERRORS

Read the **GetLastError.asm** file, no need to yap 😊

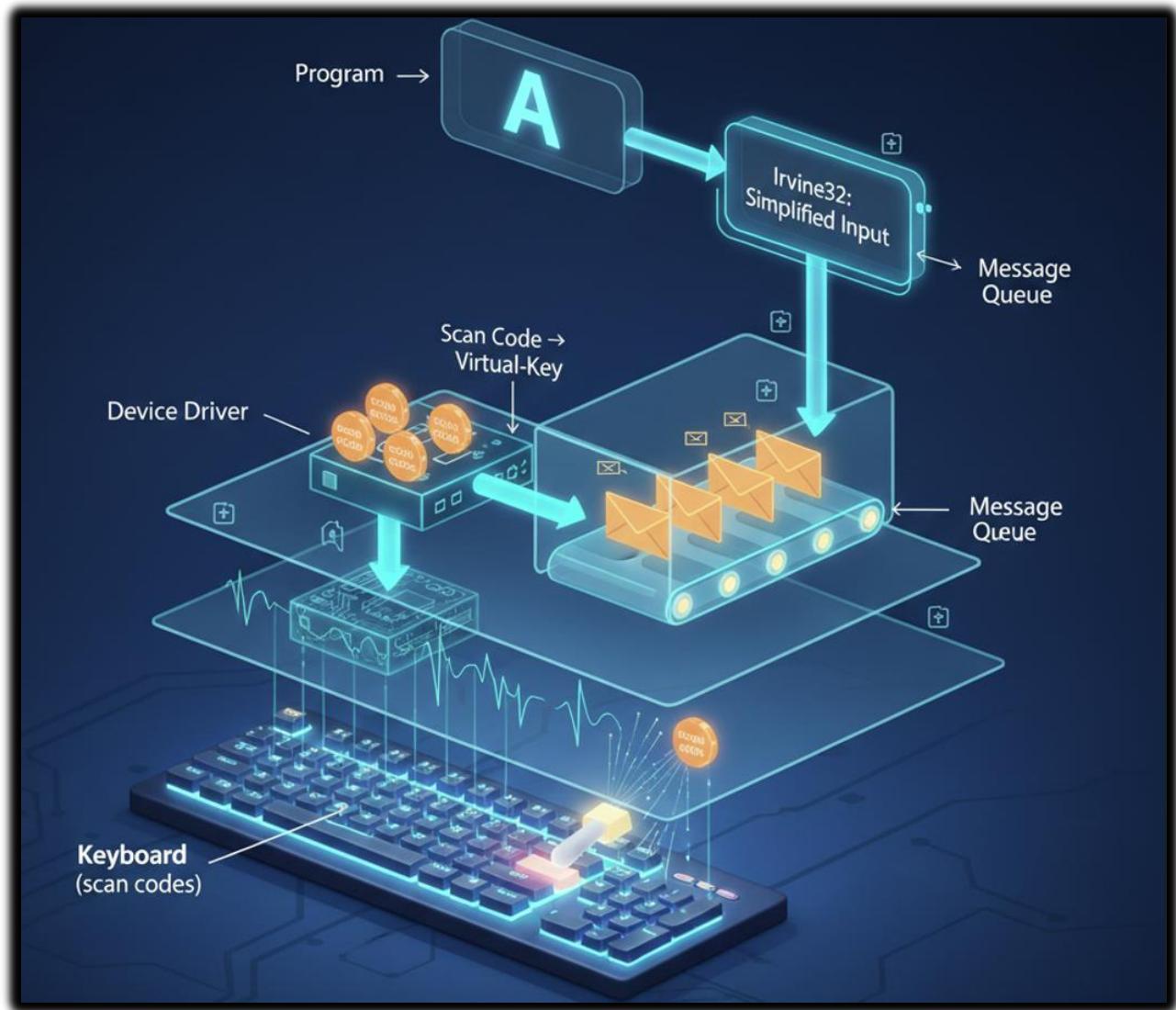


SINGLE CHARACTER INPUT

Single-Character Input and Irvine32 Keyboard Procedures

When working in **console mode** on Windows, reading a single character from the keyboard involves a few layers: the keyboard device driver, scan codes, virtual-key codes, and the program's message queue.

Here's a breakdown of how it works and how the Irvine32 library helps simplify it.



How Keyboard Input Works in Windows

1. Key Press

When you press a key, the keyboard device driver sends an **8-bit scan code** to the computer. This code tells the system which physical key was pressed.

2. Key Release

Releasing the key sends a second scan code.

3. Translation to Virtual-Key Codes

Windows converts these scan codes into **16-bit virtual-key codes**.

Virtual-key codes are **device-independent**, meaning they identify the key's purpose (like VK_UP for the Up Arrow) rather than its physical location.

4. Message Queue

Windows creates a message containing the scan code, virtual-key code, and related info.

This message is placed in the program's **message queue**.

5. Program Access

The console program eventually retrieves the message using the **console input handle**.

Irvine32 Keyboard Procedures

The **Irvine32 library** provides two main procedures to make keyboard input much easier:

ReadChar

Waits for a key to be typed and returns its ASCII value in the AL register.

.ReadKey

Checks for a keypress **without waiting**.

- If no key is pressed, the **Zero Flag** is set.
- If a key is pressed:
 - ✚ Zero Flag is clear
 - ✚ AL contains either an ASCII code or zero (for special keys)
 - ✚ Upper halves of EAX and EDX are overwritten

Handling Special Keys with ReadKey

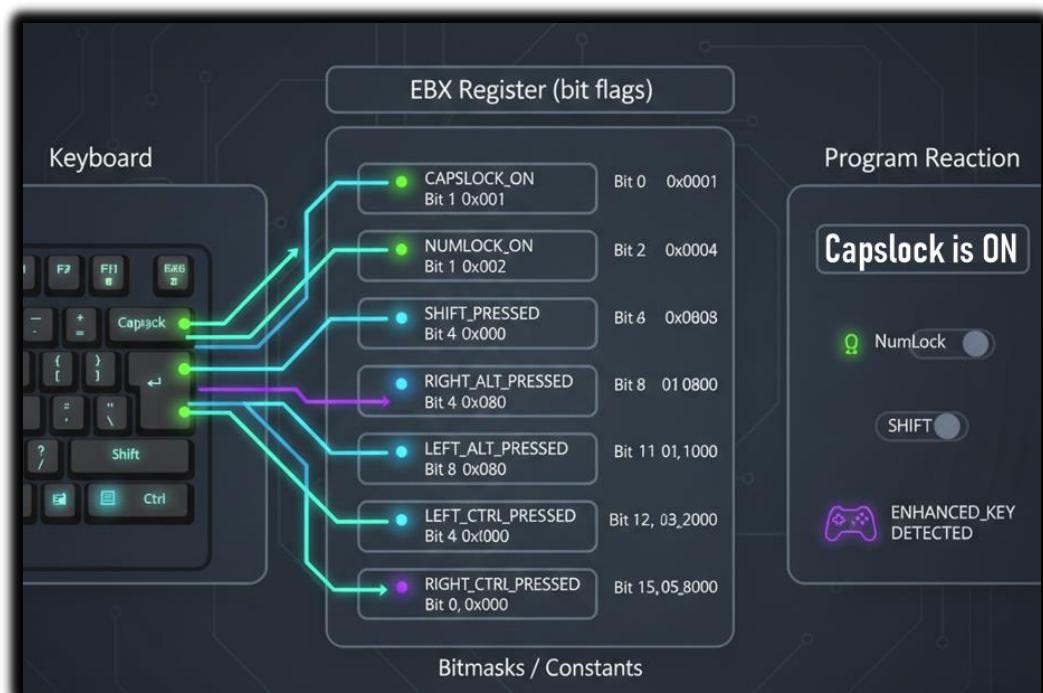
- If AL is zero, the key is a **special key** (like a function key or arrow key).
- AH contains the **scan code** for the key.
- DX contains the **virtual-key code**.
- EBX contains **control key state information**.

Control Key State Values in EBX

You can check EBX to see which control keys are active:

- CAPSLOCK_ON – CapsLock is on
- NUMLOCK_ON – NumLock is on
- SCROLLLOCK_ON – Scroll Lock is on
- SHIFT_PRESSED – Shift key is pressed
- LEFT_ALT_PRESSED / RIGHT_ALT_PRESSED – Alt key pressed
- LEFT_CTRL_PRESSED / RIGHT_CTRL_PRESSED – Ctrl key pressed
- ENHANCED_KEY – Key is enhanced (special function)

These constants let you **respond to user actions** precisely.



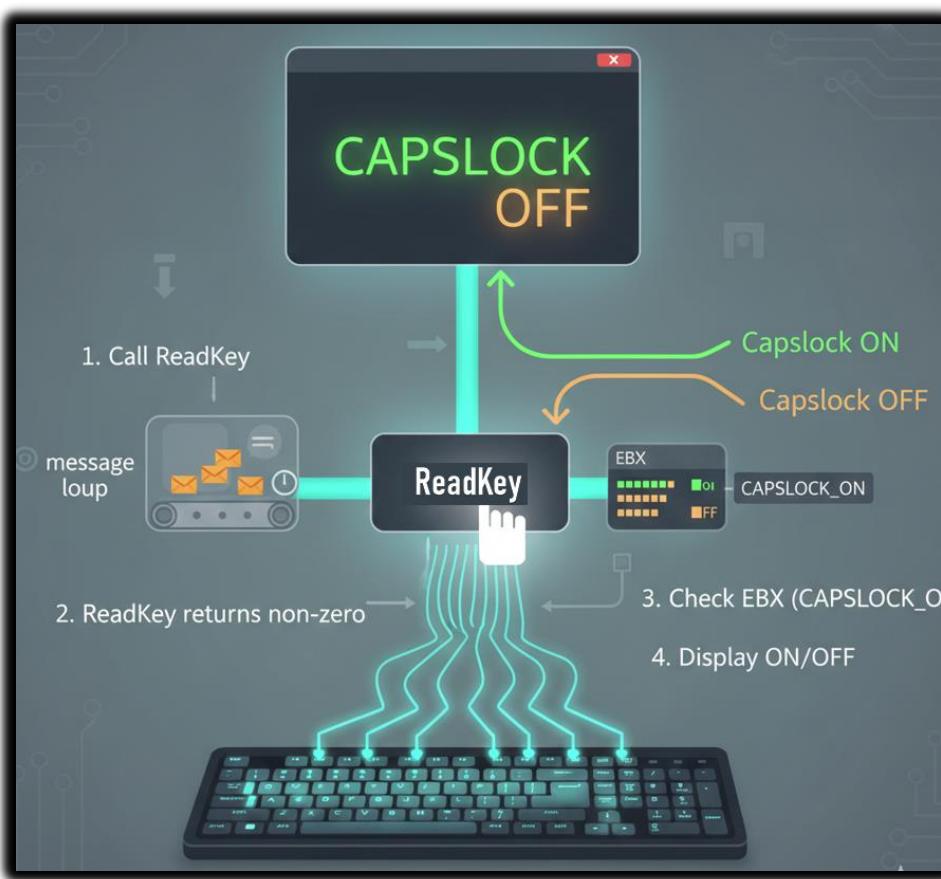
Testing Keyboard Input

Using ReadKey

You can write a simple test program:

1. Call ReadKey (optionally with a delay to allow Windows to process its message loop).
2. If a key is pressed (ReadKey returns non-zero):
 - ⊕ Check EBX using constants like CAPSLOCK_ON
 - ⊕ Display whether CapsLock (or other keys) is ON or OFF

This helps verify that your keyboard input handling works correctly.



Using GetKeyState

- GetKeyState is a **Win32 API function** that lets you test **the current state of any individual key**.
- Pass it a **virtual-key code** (like VK_CAPITAL for CapsLock or VK_NUMLOCK for NumLock).
- It returns the key's state (pressed, toggled, etc.) so your program can react accordingly.



Summary

- **ReadChar** → wait for a single ASCII character
- **ReadKey** → check immediately for keypress, including special keys
- **EBX & control key constants** → lets you detect Shift, Ctrl, Alt, CapsLock, and more
- **GetKeyState** → check the state of any key at any time

Using these procedures, your assembly programs can **easily handle single-character input**, detect special keys, and respond to the state of control keys in real time.

Table 11-4: Testing Keys with GetKeyState

PHYSICAL KEY	VIRTUAL KEY SYMBOL	BIT TO TEST IN EAX
NumLock	<code>VK_NUMLOCK</code>	0
Scroll Lock	<code>VK_SCROLL</code>	0
Left Shift	<code>VK_LSHIFT</code>	15
Right Shift	<code>VK_RSHIFT</code>	15
Left Ctrl	<code>VK_LCONTROL</code>	15
Right Ctrl	<code>VK_RCONTROL</code>	15
Left Menu (Alt)	<code>VK_LMENU</code>	15
Right Menu (Alt)	<code>VK_RMENU</code>	15

Note: Bit 15 indicates if the key is currently **down**. Bit 0 indicates if the key is **toggled** (on/off), which is why it is used for lock keys like NumLock.

GetKeyState returns a value in **EAX**, which can be tested to determine the state of a key.

The program demonstrates checking the **NumLock** and **Left Shift** keys.

To check **NumLock**:

- Call GetKeyState with `VK_NUMLOCK`.
- Test **bit 0 (lowest bit) of AL**.
- If set, **NumLock is ON**.

To check **Left Shift**:

- Call GetKeyState with `VK_LSHIFT`.
- Test **bit 31 (highest bit) of EAX**.
- If set, the **Left Shift key is currently pressed**.

The program displays **messages** based on the test results to report the state of the keys.

```
227 ;Testing Keyboard Input with ReadKey (TestReadkey.asm)
228 INCLUDE Irvine32.inc
229 INCLUDE Macros.inc
230
231 .code
232 main PROC
233 L1:
234     mov eax, 10      ; Delay for message processing
235     call Delay
236     call ReadKey    ; Wait for a keypress
237     jz L1
238
239     test ebx, CAPSLOCK_ON
240     jz L2
241     mWrite <"CapsLock is ON", 0dh, 0ah>
242     jmp L3
243 L2:
244     mWrite <"CapsLock is OFF", 0dh, 0ah>
245 L3:
246     exit
247 main ENDP
248 END main
```

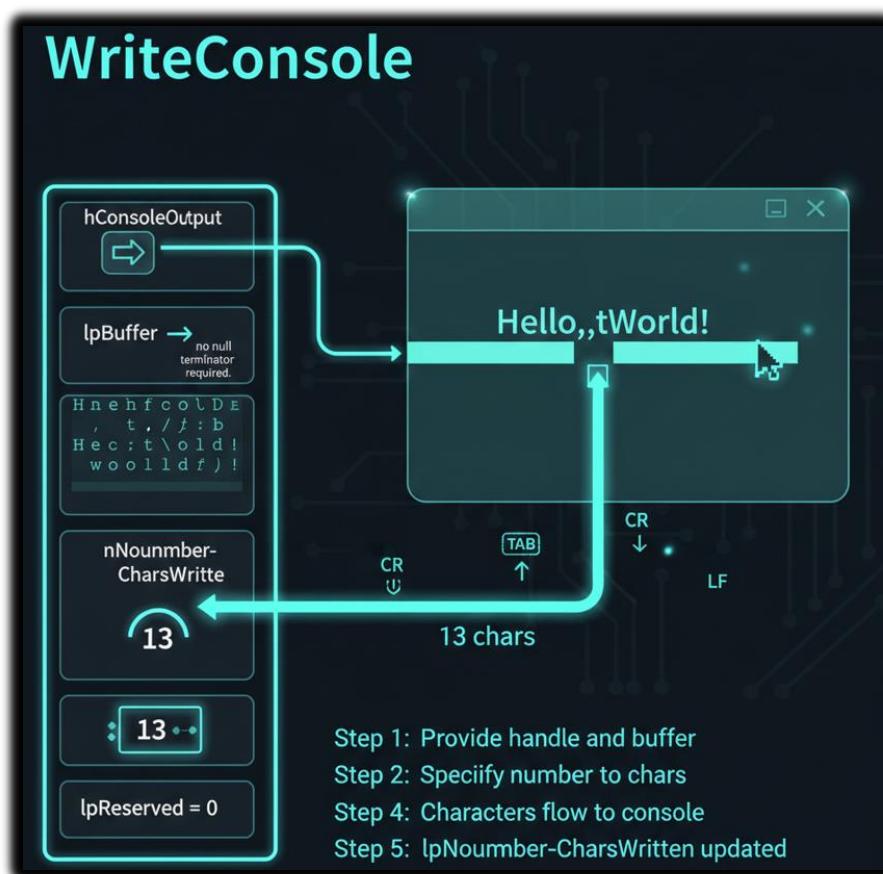
Program 2:

```
251 ;GetKeyState
252 INCLUDE Irvine32.inc
253 INCLUDE Macros.inc
254
255 .code
256 main PROC
257     INVOKE GetKeyState, VK_NUMLOCK
258     test al, 1
259     .IF !Zero?
260         mWrite <"The NumLock key is ON", 0dh, 0ah>
261     .ENDIF
262
263     INVOKE GetKeyState, VK_LSHIFT
264     test eax, 80000000h
265     .IF !Zero?
266         mWrite <"The Left Shift key is currently DOWN", 0dh, 0ah>
267     .ENDIF
268
269     exit
270 main ENDP
271 END main
```

CONSOLE OUTPUT

I. WriteConsole Function

- Used to **write a string to the console** at the current cursor position.
- **Parameters:**
 - ⊕ hConsoleOutput: Handle to the console output stream.
 - ⊕ lpBuffer: Pointer to the array of characters to write.
 - ⊕ nNumberOfCharsToWrite: Number of characters to write.
 - ⊕ lpNumberOfCharsWritten: Pointer to receive the number of characters actually written.
 - ⊕ lpReserved: Not used; set to 0.
- Advances the cursor just past the last character written.
- Can handle ASCII control characters (tabs, carriage returns, line feeds).
- The string **does not need to be null-terminated**.



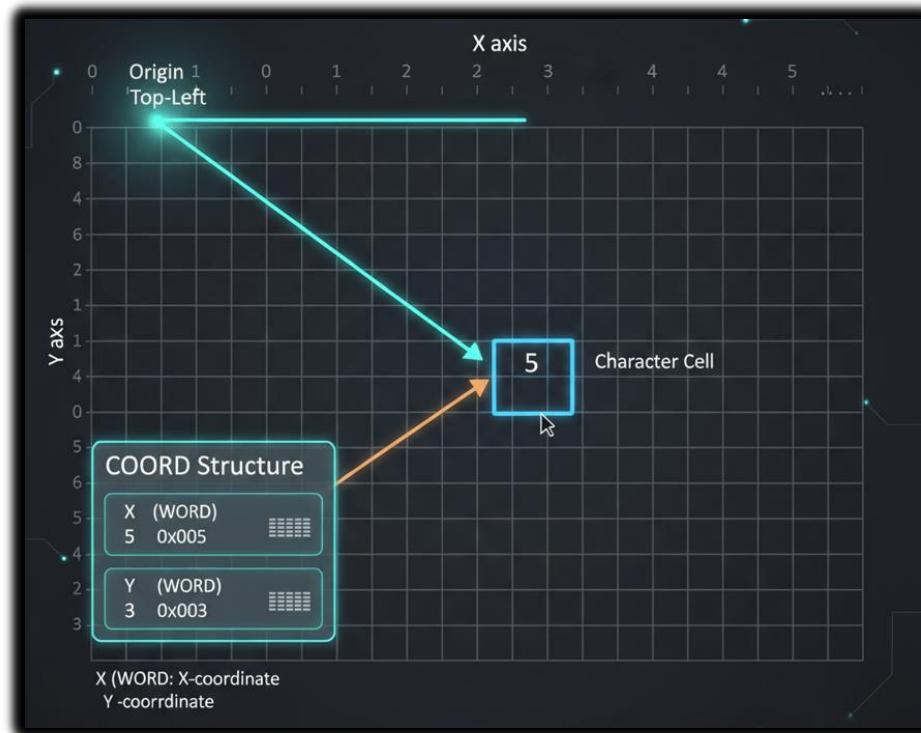
II. COORD Structure

Represents the **coordinates of a character cell** in the console screen buffer.

Origin is at the **top-left** of the console screen.

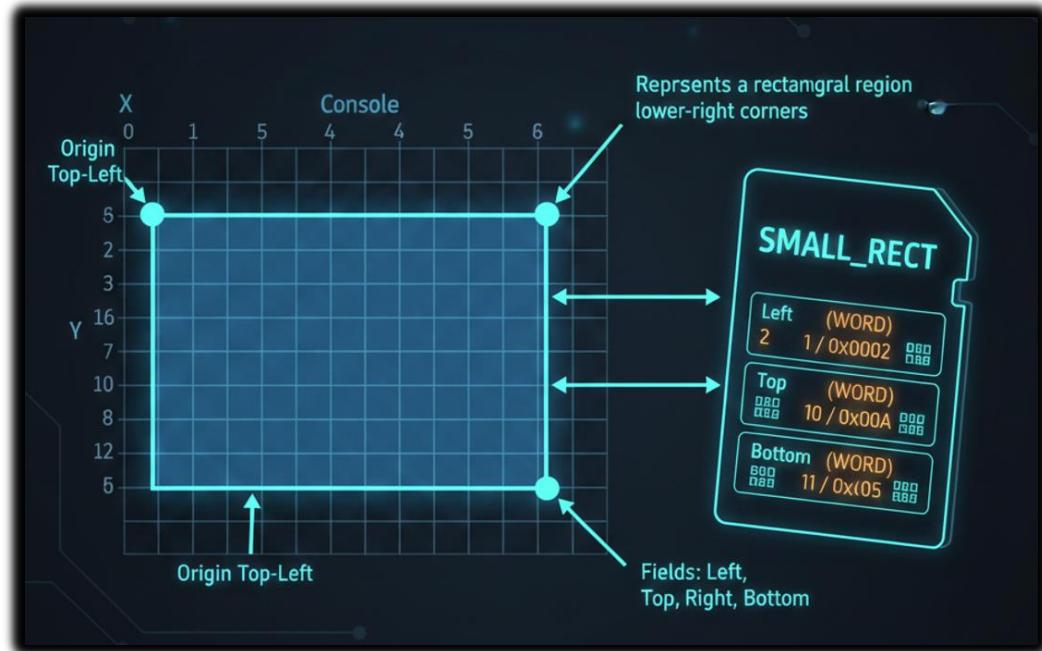
Fields:

- ⊕ X (WORD): X-coordinate.
- ⊕ Y (WORD): Y-coordinate.



III. SMALL_RECT Structure

- Represents a **rectangular region** within the console window.
- Specifies **upper-left and lower-right corners** of the rectangle.
- Useful for defining **character cell regions** in the console.
- **Fields:**
 - ⊕ Left (WORD): Left coordinate.
 - ⊕ Top (WORD): Top coordinate.
 - ⊕ Right (WORD): Right coordinate.
 - ⊕ Bottom (WORD): Bottom coordinate.



These structures are commonly used in **console-related Win32 functions** for managing and manipulating console windows.

`WriteConsole` is especially useful for **directly writing content to the console**.

Example 1: WriteConsole function

```

275 ; Win32 Console Example #1(Console1.asm)
276 ; This program calls the following Win32 Console functions:
277 ; GetStdHandle, ExitProcess, WriteConsole
278 INCLUDE Irvine32.inc
279 .data
280 endl EQU <0dh,0ah>      ; End of line sequence
281 message LABEL BYTE
282     BYTE "This program is a simple demonstration of"
283     BYTE "console mode output, using the GetStdHandle"
284     BYTE "and WriteConsole functions.",endl
285 messageSize DWORD ($ - message)
286 consoleHandle HANDLE 0      ; Handle to standard output device
287 bytesWritten DWORD ?        ; Number of bytes written
288 .code
289 main PROC
290     ; Get the console output handle:
291     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
292     mov consoleHandle,eax
293
294     ; Write a string to the console:
295     INVOKE WriteConsole,
296             consoleHandle,    ; Console output handle
297             ADDR message,    ; String pointer
298             messageSize,    ; String length
299             ADDR bytesWritten, ; Returns number of bytes written
300             0                ; Not used
301
302     ; Exit the program:
303     INVOKE ExitProcess, 0
304 main ENDP
305 END main

```

Console1.asm Program Overview

Demonstrates **GetStdHandle**, **WriteConsole**, and **ExitProcess** in a Win32 console application.

.data Section

- endl: End-of-line sequence (carriage return + line feed).
- message: Multiline text string to write to the console.
- messageSize: Stores the size of the message string.
- consoleHandle: HANDLE variable (initialized to 0) to store standard output handle.
- bytesWritten: DWORD variable to store the number of bytes written by WriteConsole.

.code Section (main procedure)

- INVOKE GetStdHandle, STD_OUTPUT_HANDLE:
 - Retrieves the handle to the standard output (console).
 - Stores the handle in consoleHandle.
- INVOKE WriteConsole:
 - Writes the message string to the console.
 - Parameters:
 - consoleHandle: Console output handle
 - ADDR message: Pointer to the message string
 - messageSize: Length of the message
 - ADDR bytesWritten: Pointer to receive number of bytes written
 - 0: Unused parameter
- INVOKE ExitProcess, 0: Exits the program.

Program Behavior

- Writes the message string to the console.
- Displays multiline text in the console window.
- Exits cleanly after writing the message.

The output will look like this:

This program is a simple demonstration of console mode output, using the GetStdHandle and WriteConsole functions.

This code demonstrates how to use the **Win32 Console functions** to write a message to the console window.

The message is stored in the message variable and is written to the console using the **WriteConsole function**. Finally, the program exits using ExitProcess.

WriteConsoleOutputCharacter function - Writing to the Console Using Win32 Functions

```
311 INCLUDE Irvine32.inc
312
313 .data
314     message BYTE "Hello, World!",0 ; The string to be written
315     coord COORD <5, 5>           ; Starting coordinates in the console
316
317 .code
318 main PROC
319     ; Get the console output handle:
320     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
321     mov edi, eax ; Store the console output handle in EDI
322
323     ; Write the message to the console at the specified coordinates:
324     INVOKE WriteConsoleOutputCharacter, edi, ADDR message, LENGTHOF message - 1, coord, NULL
325
326     ; Exit the program
327     INVOKE ExitProcess, 0
328
329 main ENDP
330
331 END main
```

- **Purpose:** Demonstrates writing messages to the console window in a Win32 program.
- **Message Storage:** The message is stored in a variable (e.g., message).

WriteConsole Function

- Writes a string to the console at the **current cursor position**.
- Handles standard ASCII control characters like **carriage return (CR)** and **line feed (LF)**. Parameters:
 - ⊕ hConsoleOutput: Handle to the console output.
 - ⊕ lpBuffer: Pointer to the string to write.
 - ⊕ nNumberOfCharsToWrite: Number of characters to write.
 - ⊕ lpNumberOfCharsWritten: Pointer to receive the number of characters actually written.
 - ⊕ lpReserved: Not used; set to 0.
- **Use Case:** Simple console output, writing sequentially from the current cursor position.

WriteConsoleOutputCharacter Function

- Writes an array of characters to **specific cells** in the console screen buffer.
- Text can wrap to the next line if it reaches the end of a line.
- **Does not** modify attribute values in the screen buffer.
- **Ignores ASCII control codes** like tab, CR, and LF.
- Parameters:
 - ⊕ hConsoleOutput: Handle to the console output buffer.
 - ⊕ lpCharacter: Pointer to the characters to write.
 - ⊕ nLength: Number of characters to write.
 - ⊕ dwWriteCoord: Coordinates (COORD structure) of the first cell to start writing (X = column, Y = row).
 - ⊕ lpNumberOfCharsWritten: Pointer to receive the number of characters actually written.
- Return value: **Non-zero** if successful. **Zero** if failed.
- **Use Case:** Provides **precise control** over where text appears in the console, unlike WriteConsole.
- **ExitProcess Function** - Exits the program cleanly after writing messages.

FILE HANDLING

CreateFile Function

Creates a new file or opens an existing file.

Return Value:

- Returns a **handle** to the file if successful.
- Returns **INVALID_HANDLE_VALUE** if it fails.

```
335 CreateFile PROTO,
336 1pFilename: PTR BYTE,
337 dwDesiredAccess: DWORD,
338 dwShareMode: DWORD,
339 1pSecurityAttributes: DWORD,
340 dwCreationDisposition: DWORD,
341 dwFlagsAndAttributes: DWORD,
342 hTemplateFile: DWORD
```

Parameters

1. **lpFilename**

- ⊕ Pointer to a null-terminated string with the file path.
- ⊕ Can be full path (C:\folder\file.txt) or just a filename (file.txt).

2. **dwDesiredAccess**

- ⊕ Specifies type of access to the file. Options include:
 - ✓ GENERIC_READ – read access
 - ✓ GENERIC_WRITE – write access
 - ✓ 0 – device query access
- ⊕ Flags can be combined with bitwise OR.

3. **dwShareMode**

- ⊕ Controls how other processes can access the file while it's open:
 - ✓ FILE_SHARE_READ – others can read
 - ✓ FILE_SHARE_WRITE – others can write
 - ✓ 0 – no sharing

4. **lpSecurityAttributes**

- ⊕ Pointer to security structure. Usually NULL if no special security is needed.

5. **dwCreationDisposition**

- ⊕ Specifies action depending on file existence:
 - ✓ CREATE_NEW – create new, fails if exists
 - ✓ CREATE_ALWAYS – create new, overwrite if exists
 - ✓ OPEN_EXISTING – open existing, fails if not found
 - ✓ OPEN_ALWAYS – open if exists, create if not
 - ✓ TRUNCATE_EXISTING – open and truncate to zero, fails if not found

6. **dwFlagsAndAttributes**

- ⊕ File attributes and flags: e.g., FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_ARCHIVE.

7. **hTemplateFile**

- ⊕ Optional handle to a template file. Can supply attributes and extended attributes. Usually set to NULL.

Usage Notes

- dwDesiredAccess and dwShareMode can be combined using **bitwise OR**.
- dwCreationDisposition is **critical** for controlling whether a file is created, overwritten, or opened.
- WriteFile and ReadFile typically follow CreateFile for I/O operations using the returned handle.

Here's an example of how you might use the CreateFile function in assembly language to create or open a text file and get a handle to it:

```
344 .data
345 filePath BYTE "myfile.txt", 0
346 handle HANDLE ?
347
348 .code
349 ; Create or open the file for writing
350 INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
351
352 ; Check if the file handle is valid
353 cmp eax, INVALID_HANDLE_VALUE
354 je fileCreationFailed
355
356 ; Store the file handle
357 mov handle, eax
358
359 ; Now you can write to the file using the handle
360
361 ; Close the file when done
362 INVOKE CloseHandle, handle
363
364 fileCreationFailed:
365 ; Handle the case where file creation/opening failed
366 ; This could include error checking and cleanup
```

This code opens or creates the file "myfile.txt" for writing and checks if the operation was successful.

Combined Code:

Here's a single code example that demonstrates the creation and opening of files using CreateFile, and reading from a file using ReadFile.

It uses the file "mydata.txt" for illustration:

```
370 .data
371 filePath BYTE "mydata.txt", 0
372 handle HANDLE ?
373 bytesRead DWORD ?
374 buffer BYTE 128 DUP(?)
375 .code
376 ; Create or open the file for writing
377 INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
378 ; Check if the file handle is valid
379 cmp eax, INVALID_HANDLE_VALUE
380 je fileCreationFailed
381 ; Store the file handle
382 mov handle, eax
383 ; Write some data to the file (assuming data is in the buffer)
384 ; Close the file handle
385 INVOKE CloseHandle, handle
386 ; Reopen the file for reading
387 INVOKE CreateFile, ADDR filePath, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
388 ; Check if the file handle is valid
389 cmp eax, INVALID_HANDLE_VALUE
390 je fileOpenFailed
391 ; Store the file handle
392 mov handle, eax
393 ; Read data from the file
394 INVOKE ReadFile, handle, ADDR buffer, 128, ADDR bytesRead, 0
395 ; Handle the data read from the file
396 ; Close the file handle
397 INVOKE CloseHandle, handle
398 fileOpenFailed:
399 fileCreationFailed:
400 ; Handle any failures during file creation or opening
```

Things the User Can't Read Directly from the Code and we have to explain:

- **CreateFile behavior**

- Returns a **handle**, not the actual file contents; you need ReadFile or WriteFile to interact with data.
- If it fails, INVALID_HANDLE_VALUE is returned, but **why it failed** isn't obvious until you check GetLastError.

- **Access vs. sharing**

- dwDesiredAccess only requests access; it does **not guarantee** other processes won't access the file. That's determined by dwShareMode.
- Combining GENERIC_READ | GENERIC_WRITE doesn't automatically prevent other processes from writing unless dwShareMode is set to restrict it.

- **Security attributes**
 - ⊕ lpSecurityAttributes controls inheritance and permissions at the system level.
 - ⊕ Even if set to NULL, default security rules apply, which might block access depending on the user or process permissions.
- **Creation disposition subtleties**
 - ⊕ CREATE_NEW ensures you never overwrite existing files.
 - ⊕ OPEN_ALWAYS can **both open or create** a file, which can lead to unexpected behavior if the file already exists.
 - ⊕ TRUNCATE_EXISTING destroys the file's previous content immediately; there's no undo.
- **File attributes (dwFlagsAndAttributes)**
 - ⊕ Attributes like FILE_ATTRIBUTE_HIDDEN or FILE_ATTRIBUTE_READONLY don't prevent access programmatically but affect how the file is treated by the OS and some applications.
- **Template file (hTemplateFile)**
 - ⊕ Rarely used, but if specified, new files can inherit extended attributes (like NTFS alternate data streams) from the template.
- **Closing files**
 - ⊕ CloseHandle isn't just cleanup—it flushes buffers and ensures **data integrity**. Forgetting to close a handle can corrupt files or leak system resources.
- **ReadFile behavior**
 - ⊕ Synchronous vs asynchronous reading: lpOverlapped = NULL makes it synchronous.
 - ⊕ The function fills a buffer and reports exactly how many bytes were read, which may be **less than requested** at the end of the file.
- **Error handling**
 - ⊕ Most failures don't crash the program—they require GetLastError to understand the cause (e.g., access denied, file not found, sharing violation).

WRITEFILE AND SETFILEPOINTER

WriteFile function

The WriteFile function is used to **write data to a file or any output handle** (like a console or screen buffer).

I. How it works:

The function writes data starting at the file's **current position pointer**. After the operation, the pointer automatically moves forward by the number of bytes actually written.

II. Parameters:

- ✚ hFile – handle to the file or output destination
- ✚ lpBuffer – pointer to the data buffer to write
- ✚ nNumberOfBytesToWrite – how many bytes to write
- ✚ lpNumberOfBytesWritten – receives the actual number of bytes written
- ✚ lpOverlapped – set to NULL for synchronous writes; used for asynchronous operations

III. Return value:

- ✚ Non-zero → success
- ✚ Zero → failure (use GetLastError to find out why)

IV. Things to keep in mind:

- ✚ **Partial writes:** Even if you request nNumberOfBytesToWrite, fewer bytes may be written (e.g., disk full). Always check lpNumberOfBytesWritten.
- ✚ **Handles:** Can write not just to files, but also to pipes, consoles, or other output handles.
- ✚ **Synchronous vs. asynchronous:** Synchronous writes block the calling thread; asynchronous writes require OVERLAPPED.

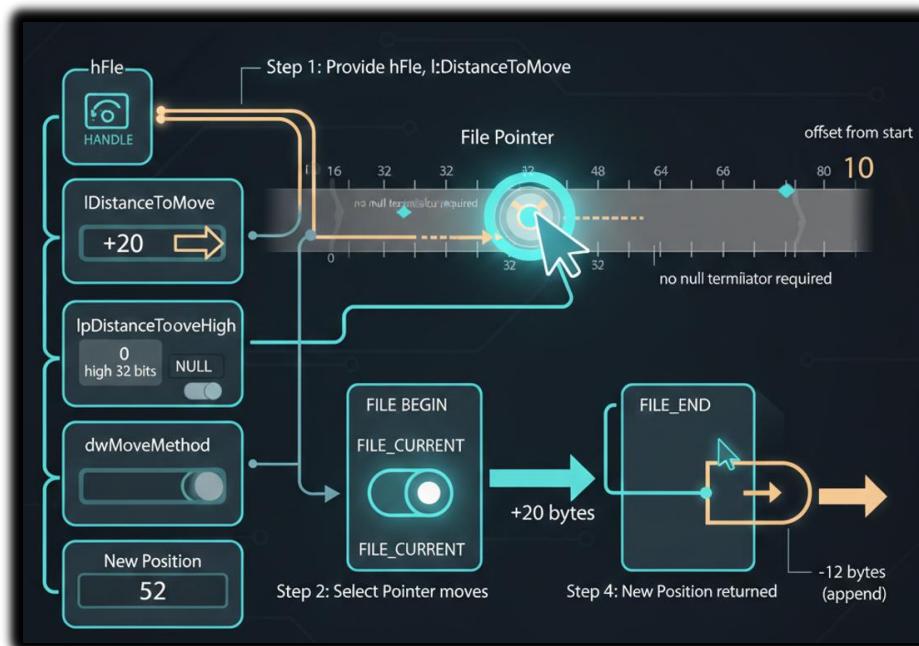
SetFilePointer Function

SetFilePointer moves the **current read/write position** in an open file, which is useful for **random access or appending data**.

```
407 INVOKE SetFilePointer,  
408 fileHandle, ; file handle  
409 0, ; distance low  
410 0, ; distance high  
411 FILE_END ; move method
```

I. Parameters:

- ✚ hFile – handle to the open file
- ✚ lDistanceToMove – number of bytes to move the pointer (can be positive or negative)
- ✚ lpDistanceToMoveHigh – pointer to the high 32 bits of the distance (for files larger than 4 GB; use NULL if not needed)
- ✚ dwMoveMethod – starting point for the move:
 - FILE_BEGIN → absolute position from file start
 - FILE_CURRENT → relative to current pointer
 - FILE_END → relative to file end (easy for appending)



II. Hidden behaviors & tips:

- ↳ Moves the file's **internal pointer**, which affects all subsequent ReadFile and WriteFile calls.
- ↳ Can handle **large files** via lpDistanceToMoveHigh.
- ↳ Supports **negative offsets** to move backward in the file.
- ↳ If the pointer cannot be set (e.g., negative position before file start), the function returns INVALID_SET_FILE_POINTER; call GetLastError to get details.

III. Quick Usage Example

Appending data to a file:

Move to the end of the file:

```
SetFilePointer hFile, 0, NULL, FILE_END
```

Write data:

```
WriteFile hFile, lpBuffer, nBytes, lpBytesWritten, NULL
```

The file pointer automatically advances after the write.

These functions are essential for managing file access and writing data in Windows programming.

They are typically used in sequence:

- **SetFilePointer** positions the file pointer to the desired location.
- **WriteFile** writes data to that specific location.

Proper use of these functions ensures efficient and accurate file manipulation in Windows applications.

```
417 ;-----
418 ; CreateOutputFile PROC
419 ;
420 ; Creates a new file and opens it in output mode.
421 ;
422 ; Receives: EDX points to the filename.
423 ;
424 ; Returns: If the file was created successfully, EAX
425 ; contains a valid file handle. Otherwise, EAX
426 ; equals INVALID_HANDLE_VALUE.
427 ;
428 ;-----
429     INVOKE CreateFile,
430         edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
431             CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
432     ret
433 CreateOutputFile ENDP
434 ;-----
435 ; OpenFile PROC
436 ;
437 ;
438 ; Opens a new text file and opens for input.
439 ;
440 ; Receives: EDX points to the filename.
441 ;
442 ; Returns: If the file was opened successfully, EAX
443 ; contains a valid file handle. Otherwise, EAX equals
444 ; INVALID_HANDLE_VALUE.
445 ;
446 ;-----
447     INVOKE CreateFile,
448         edx, GENERIC_READ, DO_NOT_SHARE, NULL,
449             OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
450     ret
451 OpenFile ENDP
452 ;-----
453 ; WriteToFile PROC
454 ;
455 ;
456 ; Writes a buffer to an output file.
457 ;
458 ; Receives: EAX = file handle, EDX = buffer offset,
459 ; ECX = number of bytes to write
460 ;
461 ; Returns: EAX = number of bytes written to the file.
462 ; If the value returned in EAX is less than the
463 ; argument passed in ECX, an error likely occurred.
464 ;
465 ;-----
466 .data
467 WriteToFile_1 DWORD ?
468 ; number of bytes written
469 .code
470     INVOKE WriteFile,
471         eax,      ; file handle
472         edx,      ; buffer pointer
473         ecx,      ; number of bytes to write
474         ADDR WriteToFile_1, ; number of bytes written
475         0          ; overlapped execution flag
476     mov eax, WriteToFile_1 ; return value
477     ret
478 WriteToFile ENDP
```

```
480 ;-----
481 ; ReadFromFile PROC
482 ;
483 ; Reads an input file into a buffer.
484 ;
485 ; Receives: EAX = file handle, EDX = buffer offset,
486 ; ECX = number of bytes to read
487 ;
488 ; Returns: If CF = 0, EAX = number of bytes read; if
489 ; CF = 1, EAX contains the system error code returned
490 ; by the GetLastError Win32 API function.
491 ;
492 ;-----
493 .data
494 ReadFromFile_1 DWORD ?
495 ; number of bytes read
496 .code
497 INVOKE ReadFile,
498     eax,          ; file handle
499     edx,          ; buffer pointer
500     ecx,          ; max bytes to read
501     ADDR ReadFromFile_1, ; number of bytes read
502     0            ; overlapped execution flag
503 mov eax, ReadFromFile_1
504 ret
505 ReadFromFile ENDP
507 ;-----
508 ; CloseFile PROC
509 ;
510 ; Closes a file using its handle as an identifier.
511 ;
512 ; Receives: EAX = file handle
513 ;
514 ; Returns: EAX = nonzero if the file is successfully closed.
515 ;
516 ;-----
517 INVOKE CloseHandle, eax
518 ret
519 CloseFile ENDP
```

That was the first program to test your knowledge, now let's do the second one:

```
524 ; Creating a File (CreateFile.asm)
525 INCLUDE Irvine32.inc
526 BUFFER_SIZE = 501
527
528 .data
529 buffer BYTE BUFFER_SIZE DUP(?)
530 filename BYTE "output.txt",0
531 fileHandle HANDLE ?
532 stringLength DWORD ?
533 bytesWritten DWORD ?
534 str1 BYTE "Cannot create file",0dh,0ah,0
535 str2 BYTE "Bytes written to file [output.txt]:",0
536 str3 BYTE "Enter up to 500 characters and press [Enter]: ",0dh,0ah,0
537
538 .code
539 main PROC
540     ; Create a new text file.
541     mov edx, OFFSET filename      ; Load the address of the filename.
542     call CreateOutputFile        ; Call the CreateOutputFile procedure.
543     mov fileHandle, eax         ; Store the file handle in fileHandle.
544
545     ; Check for errors.
546     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
547     jne file_ok                 ; If not equal, jump to file_ok.
548
549     ; If there's an error, display the error message and exit.
550     mov edx, OFFSET str1          ; Load the address of the error message.
551     call WriteString             ; Call WriteString to display the error message.
552     jmp quit                     ; Jump to quit to exit.
553
554 file_ok:
555     ; Ask the user to input a string.
556     mov edx, OFFSET str3          ; Load the address of the input prompt.
557     call WriteString             ; Call WriteString to display the input prompt.
558
559     mov ecx, BUFFER_SIZE          ; Load the maximum buffer size.
560
561     ; Input a string.
562     mov edx, OFFSET buffer        ; Load the address of the buffer.
563     call ReadString              ; Call ReadString to get user input.
564     mov stringLength, eax         ; Store the length of the entered string.
565     ; Write the buffer to the output file.
566     mov eax, fileHandle           ; Load the file handle.
567     mov edx, OFFSET buffer        ; Load the address of the buffer.
568     mov ecx, stringLength         ; Load the length of the string.
569     call WriteToFile              ; Call WriteToFile to write to the file.
570     mov bytesWritten, eax         ; Store the number of bytes written.
571     ; Close the file.
572     call CloseFile                ; Call CloseFile to close the file.
573     ; Display the return value.
574     mov edx, OFFSET str2          ; Load the address of the output message.
575     call WriteString              ; Call WriteString to display the message.
576     mov eax, bytesWritten         ; Load the number of bytes written.
577     call WriteDec                 ; Call WriteDec to display the value.
578     call Crlf                      ; Call Crlf to add a new line.
579 quit:
580     exit
581 main ENDP
582 END main
```

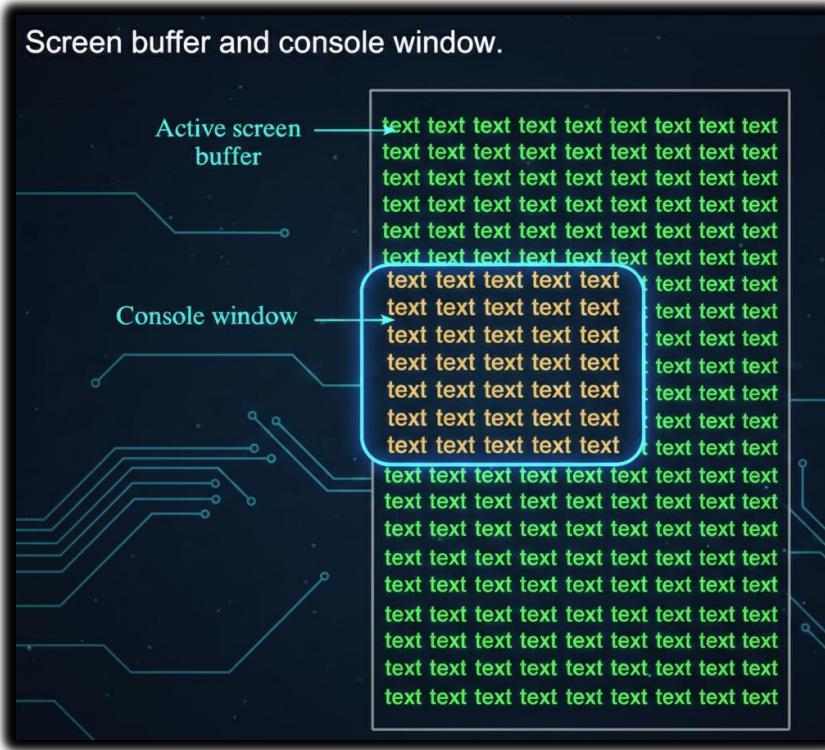
That's the second program.

Let's try another program:

```
587 ; Reading a File (ReadFile.asm)
588 ; Opens, reads, and displays a text file using
589 ; procedures from Irvine32.lib.
590 INCLUDE Irvine32.inc
591 INCLUDE macros.inc
592 BUFFER_SIZE = 5000
593
594 .data
595 buffer BYTE BUFFER_SIZE DUP(?)
596 filename BYTE 80 DUP(0)
597 fileHandle HANDLE ?
598
599 .code
600 main PROC
601     ; Let the user input a filename.
602     mWrite "Enter an input filename: " ; Display the input prompt.
603     mov edx, OFFSET filename ; Load the address of the filename.
604     mov ecx, SIZEOF filename ; Load the size of the filename.
605     call ReadString ; Call ReadString to get user input.
606
607     ; Open the file for input.
608     mov edx, OFFSET filename ; Load the address of the filename.
609     call OpenInputFile ; Call OpenInputFile to open the file.
610     mov fileHandle, eax ; Store the file handle in fileHandle.
611
612     ; Check for errors when opening the file.
613     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
614     jne file_ok ; If not equal, jump to file_ok.
615
616     ; If there's an error, display the error message and exit.
617     mWrite <"Cannot open file", 0dh, 0ah> ; Display the error message.
618     jmp quit ; Jump to quit to exit.
619
620 file_ok:
621     ; Read the file into a buffer.
622     mov edx, OFFSET buffer ; Load the address of the buffer.
623     mov ecx, BUFFER_SIZE ; Load the buffer size.
624     call ReadFromFile ; Call ReadFromFile to read the file.
625
626     jnc check_buffer_size ; If no error, jump to check_buffer_size.
627
628     ; If there's an error, display an error message.
629     mWrite "Error reading file. " ; Display the error message.
630     call WriteWindowsMsg ; Call WriteWindowsMsg to display the Windows error message.
631     jmp close_file ; Jump to close_file to close the file.
632
633 check_buffer_size:
634     cmp eax, BUFFER_SIZE ; Compare the result to BUFFER_SIZE.
635     jb buf_size_ok ; If less, jump to buf_size_ok.
636
637     ; If the buffer is too small for the file, display an error message and exit.
638     mWrite <"Error: Buffer too small for the file", 0dh, 0ah> ; Display the error message.
639     jmp quit ; Jump to quit to exit.
640
641 buf_size_ok:
642     mov buffer[eax], 0 ; Insert a null terminator.
643     mWrite "File size: " ; Display a message about the file size.
644     call WriteDec ; Call WriteDec to display the file size.
645     call Crlf ; Call Crlf to add a new line.
646
647     ; Display the buffer.
648     mWrite <"Buffer:", 0dh, 0ah, 0dh, 0ah> ; Display the buffer message.
649     mov edx, OFFSET buffer ; Load the address of the buffer.
650     call WriteString ; Call WriteString to display the buffer.
651     call Crlf ; Call Crlf to add a new line.
652
653 close_file:
654     mov eax, fileHandle ; Load the file handle.
655     call CloseFile ; Call CloseFile to close the file.
656
657 quit:
658     exit ; Exit the program.
659 main ENDP
660
661 END main
```

That's the 3rd program.

CONSOLE WINDOW MANIPULATION



- **Screen Buffer:** Memory area storing characters and color attributes for the console.
- **Console Window:** The visible window displaying the contents of the screen buffer.

Key Functions

- **WriteConsoleOutput():**
 - ⊕ Writes characters **and color attributes** to a rectangular block in the console buffer.
 - ⊕ Can update a specific area without affecting the rest of the buffer.
- **ReadConsoleOutput():**
 - ⊕ Reads characters **and color attributes** from a rectangular block in the console buffer.
 - ⊕ Useful for saving or inspecting parts of the screen before modifying it.
- **SetConsoleCursorPosition():**
 - ⊕ Moves the cursor to a specific location in the console buffer.
 - ⊕ Affects where the next character will be displayed.

Screen Buffer Character Operations

```
700 ; Get a handle to the console screen buffer.  
701 mov eax, STD_OUTPUT_HANDLE  
702 invoke GetStdHandle  
703 mov ebx, eax  
704  
705 ; Set the cursor position.  
706 mov ecx, 0 ; X coordinate  
707 mov edx, 0 ; Y coordinate  
708 invoke SetConsoleCursorPosition  
709 mov esi, ebx  
710  
711 ; Write the text to the screen buffer.  
712 mov edi, 0 ; X coordinate  
713 mov edi, 0 ; Y coordinate  
714 mov al, 'A'  
715 invoke WriteConsoleOutput  
716  
717 ; Exit the program.  
718 mov eax, 0  
719 invoke ExitProcess
```

- Writing a character (e.g., 'A') to the **top-left corner** updates both the character and its color attribute in the screen buffer.
- The **cursor position is independent**—writing with WriteConsoleOutput doesn't move the console cursor.
- Using **ReadConsoleOutput** allows you to read characters **and their color attributes** from any rectangular block of the screen buffer.
- These functions let you **manipulate the console display directly** without relying on high-level console output functions.
- Reading/writing is **coordinate-based**, using COORD structures for the X (column) and Y (row) positions.

Screen Buffer and Cursor Operations

```
722 ; Get a handle to the console screen buffer.  
723 mov eax, STD_OUTPUT_HANDLE  
724 invoke GetStdHandle  
725 mov ebx, eax  
726  
727 ; Set the cursor position.  
728 mov ecx, 0 ; X coordinate  
729 mov edx, 0 ; Y coordinate  
730 invoke SetConsoleCursorPosition  
731 mov esi, ebx  
732  
733 ; Read a single character from the screen buffer.  
734 mov edi, 0 ; X coordinate  
735 mov edi, 0 ; Y coordinate  
736 mov al, 1 ; Number of characters to read  
737 invoke ReadConsoleOutput  
738  
739 ; Exit the program.  
740 mov eax, 0  
741 invoke ExitProcess
```

- **Reading a character:**
 - ⊕ ReadConsoleOutput can retrieve a character **and its color attributes** from a specific location in the screen buffer.
 - ⊕ Example: reading from the **top-left corner** fetches the first character displayed in the console.
- **Setting the cursor position:**
 - ⊕ SetConsoleCursorPosition moves the cursor to any position in the screen buffer.
 - ⊕ Example: setting it to the **middle of the console window** determines where the next character will appear when using functions like WriteConsole.
- These functions allow **direct manipulation of both content and cursor location**, independent of high-level output functions.
- Coordinates are handled using the **COORD structure** (X = column, Y = row).

SetConsoleCursorPosition – Middle of Console

```
745 ; Get a handle to the console screen buffer.  
746 mov eax, STD_OUTPUT_HANDLE  
747 invoke GetStdHandle  
748 mov ebx, eax  
749  
750 ; Set the cursor position.  
751 mov ecx, 40 ; X coordinate  
752 mov edx, 25 ; Y coordinate  
753 invoke SetConsoleCursorPosition  
754 mov esi, ebx  
755  
756 ; Exit the program.  
757 mov eax, 0  
758 invoke ExitProcess
```

- The function moves the cursor to a **specific location** in the console buffer.
- To place it in the **middle of the console window**, you calculate coordinates as:
 - ⊕ **X (column)** = **window width ÷ 2**
 - ⊕ **Y (row)** = **window height ÷ 2**
- This determines **where the next character output will appear**.
- It does **not write any characters itself**; it only positions the cursor.
- Useful for **centering text** or controlling text layout precisely in console applications.

SetConsoleTitle, GetConsoleScreenBufferInfo, and SetConsoleWindowInfo

I. SetConsoleTitle

- Changes the **title of the console window**.
- You provide a string containing the new title.
- **Use case:** Customize the window title to reflect the program's state or purpose.

```
665 ; SetConsoleTitle function to change the console window's title  
666 .data  
667 titleStr BYTE "New Console Title",0  
668  
669 .code  
670 ; Invoke SetConsoleTitle with the specified title string  
671 INVOKE SetConsoleTitle, ADDR titleStr
```

II. GetConsoleScreenBufferInfo

- Retrieves detailed information about the console window and screen buffer.
- Information is stored in a structure, typically named `consoleInfo`, which includes:
 - ✚ Size of the screen buffer
 - ✚ Current cursor position
 - ✚ Window dimensions
 - ✚ Text attributes
- **Use case:** Determine console layout, cursor location, or buffer size before performing advanced output operations.
- **Code can be inserted here** to call `GetConsoleScreenBufferInfo` and fill the `consoleInfo` structure.

```
675 ; GetConsoleScreenBufferInfo function to retrieve information about the console window
676 .data
677 consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
678 outHandle HANDLE ?
679
680 .code
681 ; Invoke GetConsoleScreenBufferInfo to retrieve information about the console window
682 INVOKE GetConsoleScreenBufferInfo, outHandle, ADDR consoleInfo
```

This code shows how to use the **GetConsoleScreenBufferInfo function** to obtain information about the console window, including screen buffer size, cursor position, and other details. The retrieved information is stored in the `consoleInfo` structure.

```
685 ; SetConsoleWindowInfo function to set the console window's size and position
686 .data
687 windowRect SMALL_RECT <0, 0, 79, 24> ; Example window rectangle
688
689 .code
690 ; Invoke SetConsoleWindowInfo to set the console window's size and position
691 INVOKE SetConsoleWindowInfo, outHandle, TRUE, ADDR windowRect
```

III. SetConsoleWindowInfo

- Sets the **size and position** of the console window relative to its screen buffer.
- You define the new dimensions and location using a structure, typically called `windowRect`.
- **Use case:** Resize or reposition the console window for better text layout or UI alignment.
- **Code can be inserted here** to call `SetConsoleWindowInfo` with the desired window rectangle.

```
INCLUDE Irvine32.inc

.data
windowRect SMALL_RECT <>    ; Structure to define console window dimensions
consoleHandle HANDLE ?        ; Handle to the console output

.code
main PROC
    ; Get handle to standard output (console)
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov consoleHandle, eax

    ; Define new console window rectangle (Left, Top, Right, Bottom)
    mov windowRect.Left, 0
    mov windowRect.Top, 0
    mov windowRect.Right, 79      ; 80 columns (0-based index)
    mov windowRect.Bottom, 24     ; 25 rows (0-based index)

    ; Update console window size and position
    ; TRUE = absolute coordinates relative to the screen buffer
    INVOKE SetConsoleWindowInfo, consoleHandle, TRUE, ADDR windowRect

    exit
main ENDP
END main
```

CONSOLE_SCREEN_BUFFER_INFO structure.

Watch 1

Name	Type
consoleInfo	CONSOLE_SCREEN_BUFFER_INFO
dwSize	COORD
X	unsigned short
Y	unsigned short
dwCursorPosition	COORD
X	unsigned short
Y	unsigned short
wAttributes	unsigned short
srWindow	SMALL_RECT
Left	unsigned short
Top	unsigned short
Right	unsigned short
Bottom	unsigned short
dwMaximumWindowSize	COORD
X	unsigned short
Y	unsigned short

Scroll.asm program:

```

767 INCLUDE Irvine32.inc
768
769 .data
770 message BYTE ": This line of text was written to the screen buffer",0dh,0ah
771 messageSize DWORD ($-message)
772 outHandle HANDLE 0 ; Standard output handle
773 bytesWritten DWORD ?
774 lineNum DWORD 0
775 windowRect SMALL_RECT <0,0,60,11> ; Left, top, right, bottom
776
777 .code
778 main PROC
779     ; Get the standard output handle
780     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
781     mov outHandle, eax
782
783 .REPEAT
784     ; Display the line number
785     mov eax, lineNum
786     call WriteDec
787
788     ; Write the message to the console
789     INVOKE WriteConsole, outHandle, ADDR message, messageSize, ADDR bytesWritten, 0
790
791     ; Increment the line number
792     inc lineNum
793
794 .UNTIL lineNum > 50
795
796 ; Resize and reposition the console window
797 INVOKE SetConsoleWindowInfo, outHandle, TRUE, ADDR windowRect
798
799 ; Wait for a key press
800 call ReadChar
801
802 ; Clear the screen buffer
803 call Clrscr
804
805 ; Wait for a second key press
806 call ReadChar
807
808 ; Exit the program
809 INVOKE ExitProcess, 0
810
811 main ENDP
812
813 END main

```

This code simulates scrolling the console window by writing lines of text to the screen buffer and then resizing and repositioning the console window using SetConsoleWindowInfo.

After running this program, press a key to trigger the scroll, clear the screen, and exit the program.

Another example:

```
819 INCLUDE Irvine32.inc
820
821 .data
822 consoleInfo CONSOLE_CURSOR_INFO <25, 1> ; Default cursor info
823 outHandle HANDLE 0
824 coord COORD <10, 10> ; New cursor position
825
826 .code
827 main PROC
828     ; Get the standard output handle
829     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
830     mov outHandle, eax
831
832     ; Get the current cursor information
833     INVOKE GetConsoleCursorInfo, outHandle, ADDR consoleInfo
834
835     ; Display the current cursor size and visibility
836     mov eax, consoleInfo.dwSize
837     call WriteDec
838     call WriteString, ADDR "- Cursor Size, Visible: "
839     mov eax, consoleInfo.bVisible
840     call WriteDec
841     call Crlf
842
843     ; Set a new cursor size and visibility
844     mov consoleInfo.dwSize, 50
845     mov consoleInfo.bVisible, TRUE
846     INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
847
848     ; Move the cursor to a new position
849     INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
850
851     ; Display a message at the new cursor position
852     call WriteString, ADDR "New Cursor Position"
853
854     ; Wait for a key press
855     call ReadChar
856
857     ; Reset cursor info to the default values
858     mov consoleInfo.dwSize, 25
859     mov consoleInfo.bVisible, TRUE
860     INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
861
862     ; Move the cursor back to the original position
863     mov coord.X, 0
864     mov coord.Y, 0
865     INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
866
867     ; Display a message at the original cursor position
868     call WriteString, ADDR "Original Cursor Position"
869
870     ; Wait for a key press to exit
871     call ReadChar
872
873     INVOKE ExitProcess, 0
874 main ENDP
875
876 END main
```

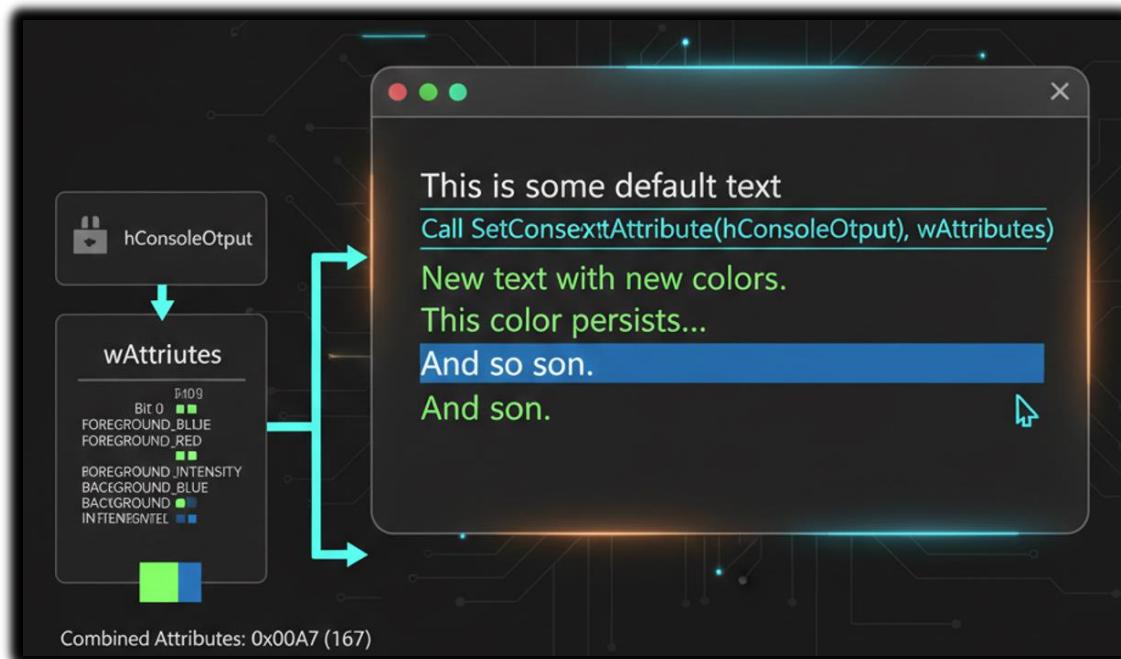
Summary of that program:

- **Retrieve current cursor info:** Uses GetConsoleCursorInfo to store the current size and visibility of the cursor.
- **Change cursor appearance:** Modifies cursor size and visibility using SetConsoleCursorInfo.
- **Move the cursor:** Uses SetConsoleCursorPosition to position the cursor at a new location in the console window.
- **Display message:** Writes text to the console at the new cursor position.
- **Restore original cursor:** Resets the cursor size and visibility to the values retrieved at the start.
- **Wait for key press:** Pauses the program using a key press before exiting, ensuring the user can see the results.

SETTING TEXT COLOR

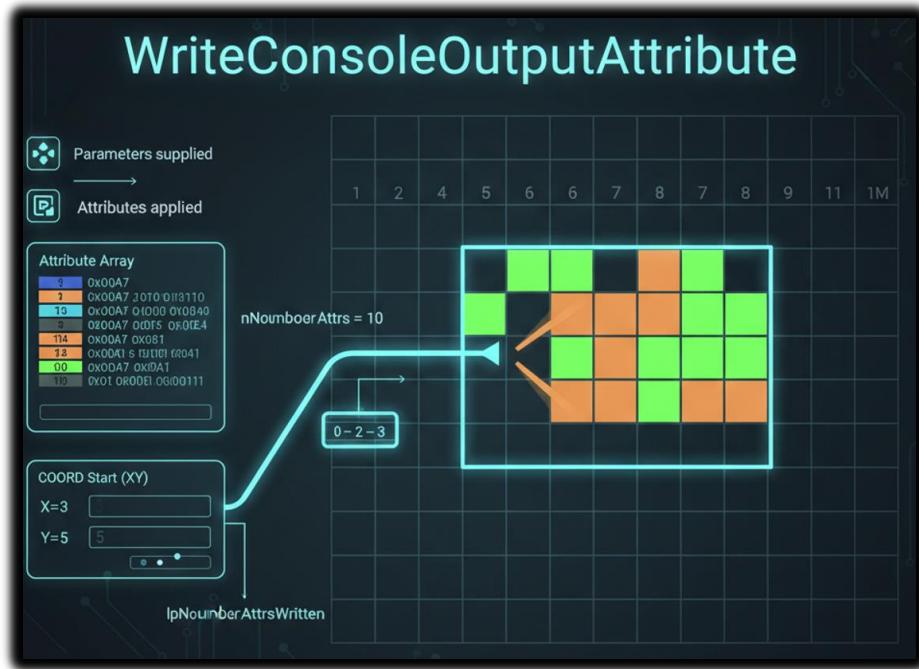
SetConsoleTextAttribute

- Sets the foreground and background colors for all text written afterward.
- Requires the console output handle and a combined color attribute.
- Changes apply to subsequent text output until another color change.



WriteConsoleOutputAttribute

- Sets text attributes (like color) for specific cells in the console screen buffer.
- Requires: an array of attributes, the number of attributes, starting coordinates (COORD), and a variable to receive the count of cells modified.
- Allows fine-grained control over colors of individual characters rather than the whole output.



Use Case / Example Program

- Display different characters with varying colors.
- Can mix foreground and background colors for emphasis or visual effects.
- Useful for creating colored menus, status bars, or highlighting in console apps.

```

883 ; SetTextColors.asm - Demonstrates setting text colors in a console window
884 INCLUDE Irvine32.inc
885
886 .data
887 outHandle HANDLE ?
888 cellsWritten DWORD ?
889 xyPos COORD <10, 2>
890
891 ; Array of character codes
892 buffer BYTE 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
893 BYTE 16,17,18,19,20
894 BufSize DWORD ($-buffer)
895
896 ; Array of attributes (text colors)
897 attributes WORD 0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6
898 WORD 5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h
899
900 .code
901 main PROC
902     ; Get the Console standard output handle
903     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
904     mov outHandle, eax
905
906     ; Set the colors of adjacent cells
907     INVOKE WriteConsoleOutputAttribute, outHandle, ADDR attributes, BufSize, xyPos, ADDR cellsWritten
908
909     ; Write character codes 1 through 20
910     INVOKE WriteConsoleOutputCharacter, outHandle, ADDR buffer, BufSize, xyPos, ADDR cellsWritten
911     INVOKE ExitProcess, 0
912 main ENDP
913 END main

```

Purpose: Display characters 1–20 with different text colors in the console.

Attributes Array: Specifies the color for each character. Each element represents a foreground/background color for a specific character.

WriteConsoleOutputCharacter / WriteConsoleOutputAttribute:

- Characters are written to the console buffer at a specified location.
- Attributes array is applied so each character appears in its defined color.

Result: Console shows colorful text output where each character can have a unique color.

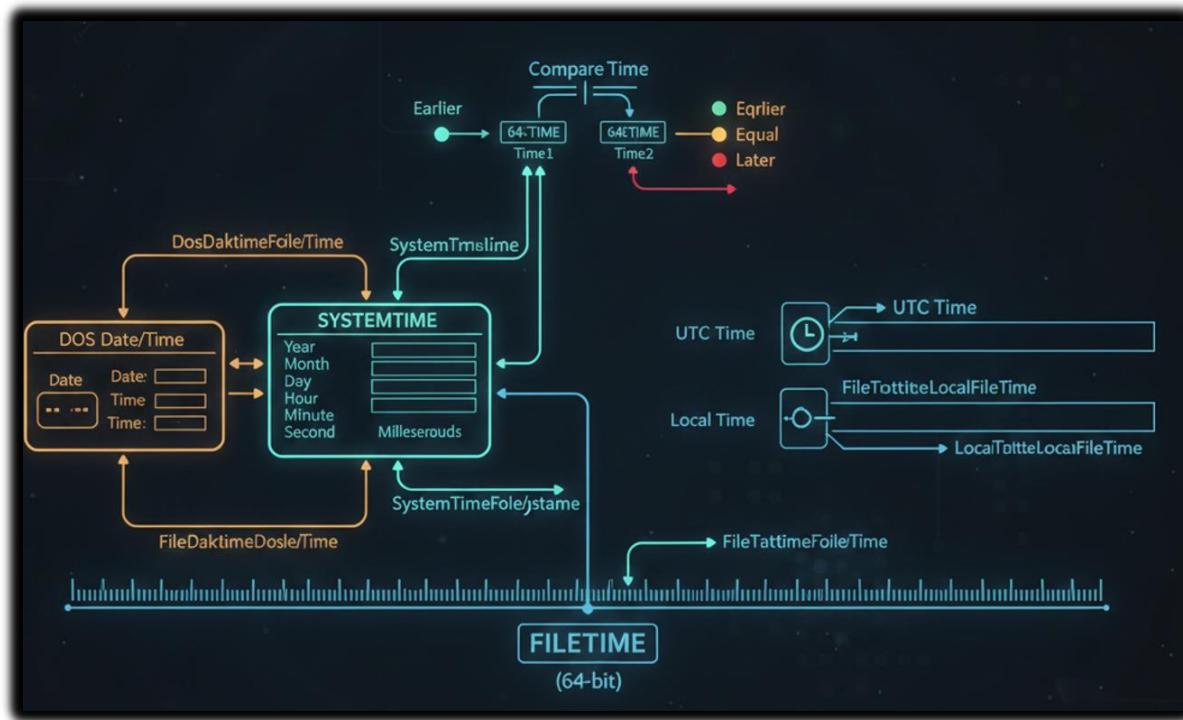
Customization: Modify the attributes array to change foreground/background colors or color patterns.

Use Case: Useful for menus, highlighting, or creating visually distinct console interfaces.

TIME, WINAPI AND ASSEMBLY

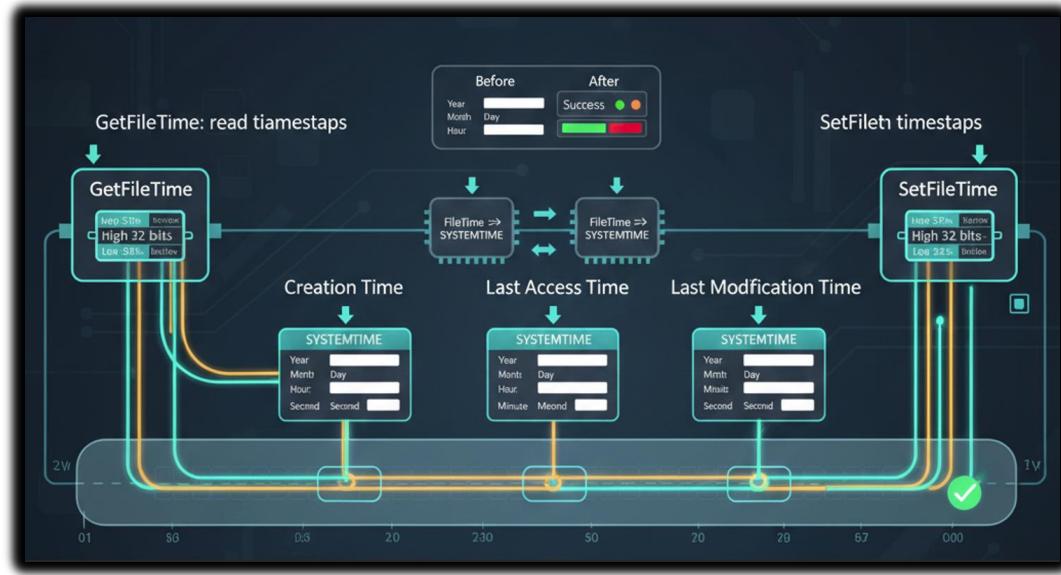
I. File Time Functions

- **CompareFileTime:** Compares two 64-bit file times to determine their chronological order.
- **DosDateTimeToFileTime:** Converts MS-DOS date/time values to 64-bit file time for compatibility with older formats.
- **FileTimeToDosDateTime:** Converts 64-bit file time back to MS-DOS date/time.
- **FileTimeToLocalFileTime:** Converts a UTC file time to local file time.
- **LocalFileTimeToFileTime:** Converts local file time to UTC-based file time.
- **FileTimeToSystemTime:** Converts 64-bit file time to a SYSTEMTIME structure with detailed date and time.
- **SystemTimeToFileTime:** Converts SYSTEMTIME to 64-bit file time.



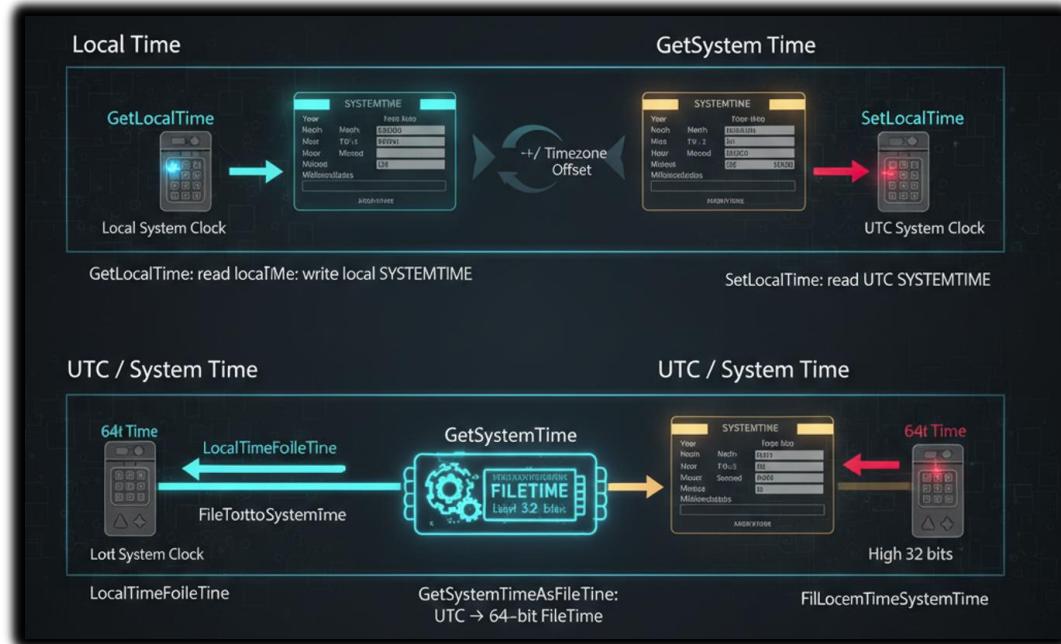
II. File Timestamp Functions

- **GetFileTime:** Retrieves a file's creation, last access, and last modification times.
- **SetFileTime:** Sets a file's creation, last access, or last modification times.



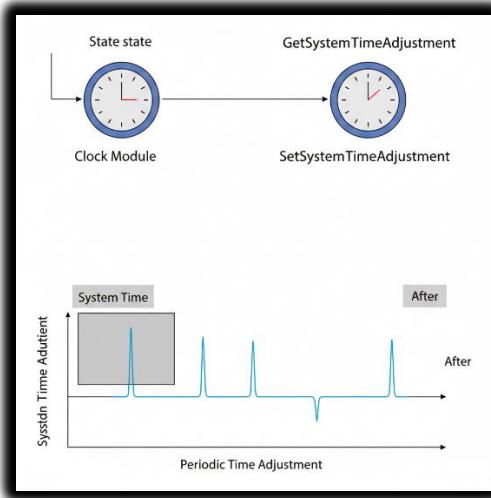
III. System Time Functions

- **GetLocalTime:** Retrieves the current local system date and time.
- **SetLocalTime:** Sets the system's local date and time.
- **GetSystemTime:** Retrieves the current UTC system date and time.
- **SetSystemTime:** Sets the UTC system date and time.
- **GetSystemTimeAsFileTime:** Retrieves current UTC system time as a 64-bit file time.



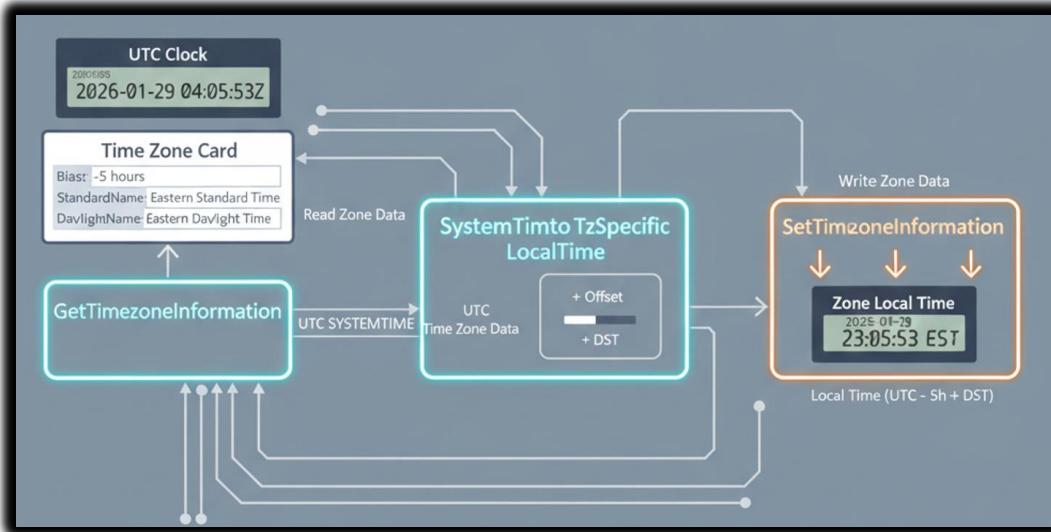
IV. Time Adjustment Functions

- **GetSystemTimeAdjustment:** Checks if the system applies periodic time adjustments (like DST).
- **SetSystemTimeAdjustment:** Enables or disables periodic adjustments to the system clock.



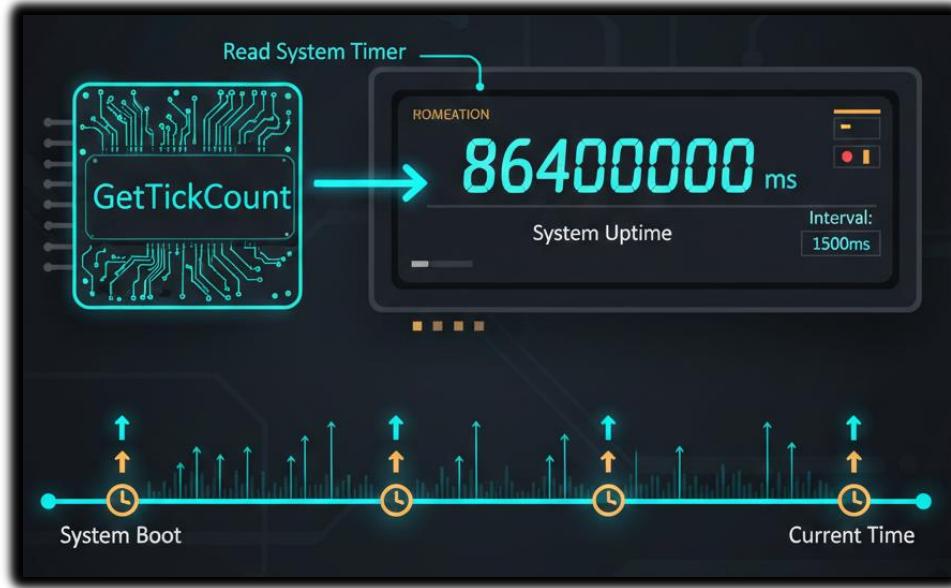
V. Time Zone Functions

- **GetTimeZoneInformation:** Retrieves the system's current time-zone settings.
- **SetTimeZoneInformation:** Sets the system's time-zone parameters.
- **SystemTimeToTzSpecificLocalTime:** Converts UTC time to local time for a specific time zone.



V. Utility / Interval Functions

- **GetTickCount:** Returns milliseconds since system start (system uptime). Useful for timing and measuring intervals.



Summary

These functions together allow you to:

- Read/write file timestamps.
- Convert between local, UTC, and MS-DOS times.
- Retrieve and set system time and local time.
- Handle time zones.
- Implement timers or measure intervals.

```

0917 INCLUDE Irvine32.inc
0918 INCLUDE macros.inc
0919
0920 .data
0921 sysTime SYSTEMTIME <> ; SYSTEMTIME structure
0922 startTime DWORD ? ; Start time for the stopwatch timer
0923
0924 .code
0925 main PROC
0926     ; Get the current local time.
0927     INVOKE GetLocalTime, ADDR sysTime
0928     ; Display the current local time.
0929     call DisplayTime
0930     ; Set the local time to a specific value.
0931     ; For example, you can set it to January 1, 2023, 12:00:00.
0932     mov sysTime.wYear, 2023
0933     mov sysTime.wMonth, 1
0934     mov sysTime.wDay, 1
0935     mov sysTime.wHour, 12
0936     mov sysTime.wMinute, 0
0937     mov sysTime.wSecond, 0
0938     INVOKE SetLocalTime, ADDR sysTime
0939     ; Get the current local time again after setting it.
0940     INVOKE GetLocalTime, ADDR sysTime
0941     ; Display the updated local time.
0942     call DisplayTime
0943     ; Start a stopwatch timer.
0944     INVOKE GetTickCount
0945     mov startTime, eax
0946     ; Perform some calculations to simulate a time-consuming operation.
0947     mov ecx, 10000100h
0948 L1:
0949     imul ebx
0950     imul ebx
0951     imul ebx
0952     loop L1
0953
0954     ; Get the current tick count and calculate elapsed time.
0955     INVOKE GetTickCount
0956     sub eax, startTime
0957
0958     ; Display the elapsed time.
0959     call WriteDec
0960     mWrite <" milliseconds have elapsed", 0dh, 0ah>
0961
0962     exit
0963 main ENDP
0964
0965 DisplayTime PROC
0966     ; Display the current local time.
0967     mWrite "Current Local Time: "
0968     call WriteDec, sysTime.wMonth
0969     mWrite <"/", 0>
0970     call WriteDec, sysTime.wDay
0971     mWrite <"/", 0>
0972     call WriteDec, sysTime.wYear
0973     mWrite < " ", 0>
0974     call WriteDec, sysTime.wHour
0975     mWrite <":", 0>
0976     call WriteDec, sysTime.wMinute
0977     mWrite <":", 0>
0978     call WriteDec, sysTime.wSecond
0979     mWrite <" (Day of Week: ", 0>
0980     call WriteDec, sysTime.wDayOfWeek
0981     mWrite <")", 0dh, 0ah>
0982     ret
0983 DisplayTime ENDP
0984 END main
0985

```

SYSTEMTIME Structure

- Holds detailed date and time information:
 - ✚ Year, Month, Day of Week, Day of Month
 - ✚ Hour, Minute, Second, Millisecond
- Used by Win32 API functions like GetLocalTime and SetLocalTime.

GetLocalTime Function

- Retrieves the current local date and time from the system clock.
- Fills a SYSTEMTIME structure passed as a pointer.
- Used to display or process the current local time.

SetLocalTime Function

- Sets the system's local date and time.
- Takes a SYSTEMTIME structure containing the desired date/time.
- Allows programmatic adjustment of system time.

GetTickCount Function

- Returns the number of milliseconds since the system started.
- Useful for timing operations or measuring elapsed time.
- Does not take parameters; returns value in EAX.

DisplayTime Procedure

- Custom procedure to display components of a SYSTEMTIME structure:
- Year, Month, Day, Hour, Minute, Second, Day of Week
- Uses write functions to output these values to the console.

Program Flow

1. Retrieve and display current local time using GetLocalTime.
2. Set local time to a specific value using SetLocalTime.
3. Retrieve and display the updated local time.
4. Perform a time-consuming operation (loop) and measure duration with GetTickCount.
5. Display the elapsed time.

Sleep Function

- Introduces a pause/delay in program execution.
- Takes a parameter specifying the duration in milliseconds.
- Puts the processor in a low-power wait state until the delay elapses.

GetDateTime Procedure

- Retrieves current date and time in Win32 FILETIME format:
 1. Calls GetLocalTime to populate a SYSTEMTIME structure.
 2. Converts SYSTEMTIME to FILETIME using SystemTimeToFileTime.
 3. Stores the result as a 64-bit quadword, splitting it into two doublewords.
- Returns the current date and time as a 64-bit value representing 100-nanosecond intervals since January 1, 1601.

Summary

- ✚ SYSTEMTIME + GetLocalTime/SetLocalTime → Reading and modifying local system time.
- ✚ GetTickCount → Measuring elapsed time for operations.
- ✚ Sleep → Introducing controlled delays.
- ✚ GetDateTime → Retrieving time in FILETIME format for precise timing or logging.

```

0988 ; Sleep Function
0989 Sleep PROTO,
0990 dwMilliseconds:DWORD
0991
0992 ; GetDateTime Procedure
0993 GetDateTime PROC,
0994 pStartTime:PTR QWORD
0995 LOCAL sysTime:SYSTEMTIME, flTime:FILETIME
0996
0997 ; Get the system local time
0998 INVOKE GetLocalTime,
0999 ADDR sysTime
1000
1001 ; Convert the SYSTEMTIME to FILETIME
1002 INVOKE SystemTimeToFileTime,
1003 ADDR sysTime,
1004 ADDR flTime
1005
1006 ; Copy the FILETIME to a 64-bit integer
1007 mov esi, pStartTime
1008 mov eax, flTime.loDateTime
1009 mov DWORD PTR [esi], eax
1010 mov eax, flTime.hiDateTime
1011 mov DWORD PTR [esi+4], eax
1012 ret
1013 GetDateTime ENDP

```

Sleep Function

The Sleep function lets your program **pause execution for a specified amount of time**, which can be very useful for timing, delays, or controlling program flow.

How it works:

- ⊕ Place the number of milliseconds to pause into the EAX register.
- ⊕ Call the Sleep function.
- ⊕ After the specified time elapses (or if interrupted), execution resumes at the next instruction.

```

1019 ; Sleep for 1 second
1020 mov eax, 1000 ; 1000 milliseconds = 1 second
1021 call sleep
1022
1023 ; Continue execution

```

Important details:

- **Interruptions:** The sleep can be interrupted by events such as a timer interrupt. In that case, your program resumes immediately, even if the full delay hasn't passed.
- **System call:** Sleep is implemented as a system call, so it interacts with the kernel.
- **Blocking:** If another process is holding the kernel lock, your sleep may be delayed until the lock is released.
- **Low-power mode:** The processor may enter a low-power state during sleep. This saves energy but can slightly delay other processes.

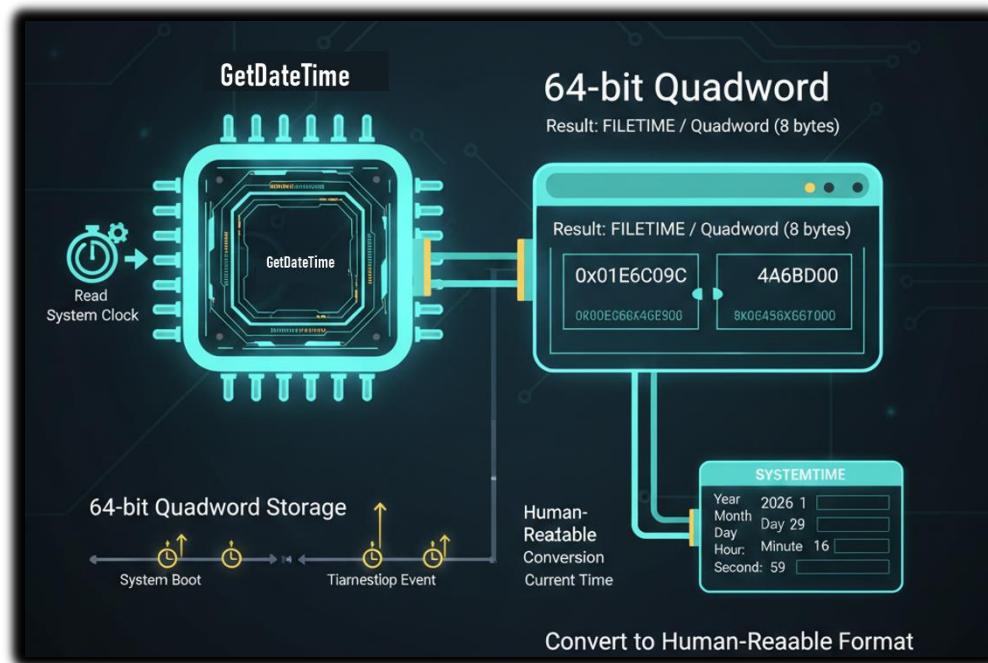
Key takeaway:

Sleep is a simple but powerful tool to **control program timing**, but you need to keep in mind that external events and system conditions can affect the actual sleep duration.

GetDateTime Procedure

The GetDateTime procedure retrieves the **current date and time** and stores it in a **64-bit quadword**.

- You can integrate it into your MASM programs whenever you need to **log time, timestamp events, or calculate durations**.
- The value returned can be processed or converted into a human-readable format as needed.



Summary:

- Use Sleep to pause program execution for a set duration.
- Use GetDateTime to retrieve the current date and time in a format you can store or manipulate.
- Both functions are simple to use but can be influenced by interrupts, kernel activity, and system timing behavior.

CALLING 64-BIT WINAPI FUNCTION IN MASM

Steps to Call a 64-bit Windows API Function in MASM

Reserve Shadow Space

- + Subtract at least 32 bytes from RSP (stack pointer) for the function's shadow space.

Align Stack

- + Ensure RSP is aligned to a 16-byte boundary before the call.

Pass Arguments in Registers

- + First four arguments go in registers (in order):
 - RCX → 1st argument
 - RDX → 2nd argument
 - R8 → 3rd argument
 - R9 → 4th argument

Push Extra Arguments on Stack

- + If there are more than four arguments, push the remaining ones onto the stack.

Call the Function

- + Use the call instruction with the function name.

Restore Stack

- + Add back the value subtracted from RSP to restore the original stack pointer.

Return Value

- + The function returns a 64-bit integer in RAX.

Calling WriteConsoleA Function (64-bit Windows API)

```
1027 .data
1028     STD_OUTPUT_HANDLE EQU -11
1029     consoleOutHandle QWORD ?
1030
1031 .code
1032     sub rsp, 40 ; reserve shadow space & align RSP
1033     mov rcx, STD_OUTPUT_HANDLE
1034     mov rdx, message ; pointer to the string
1035     mov r8, message_length ; length of the string
1036     lea r9, bytesWritten
1037     mov qword ptr [rsp + 4 * SIZEOF QWORD], 0 ; (always zero)
1038     call WriteConsoleA
1039     add rsp, 40 ; restore RSP
```

- **Arguments:**

1. Console handle (hConsoleOutput)
2. Pointer to the string to write (lpBuffer)
3. Length of the string to write (nNumberOfCharsToWrite)
4. Pointer to a variable to store number of bytes written
(lpNumberOfCharsWritten)
5. Dummy parameter, typically set to 0 (lpReserved)

- **Important Notes:**

- ✚ The first **four arguments** are passed in **RCX, RDX, R8, R9** registers.
- ✚ Shadow space of at least **32 bytes** must be reserved on the stack, even if unused.
- ✚ Return value is in **RAX**; non-zero indicates success.
- ✚ **bytesWritten** variable stores the number of bytes actually written and can be checked after the call.

Creating a Graphical Windows Application

1. Include Required Libraries and Headers

- ⊕ Libraries: kernel32.lib, user32.lib
- ⊕ Headers: contain structures, constants, and function prototypes (e.g., Windows.h)

2. Create Main Window

- ⊕ Use CreateWindowEx() to create the window.

3. Display the Window

- ⊕ Use ShowWindow() to show the main window on the screen.

4. Handle Mouse Events

- ⊕ Respond to messages like WM_MOUSEMOVE and WM_LBUTTONDOWN in the window procedure.

5. Display Message Boxes

- ⊕ Use MessageBox() to show alerts, notifications, or dialogs.

Here is a simple example of a graphical Windows application in assembly language:



Read the code file - **GraphicalWindow.asm**

Ignore this program, it's just a trial program:

```
1085 RECT STRUCT
1086     left DWORD ?
1087     top DWORD ?
1088     right DWORD ?
1089     bottom DWORD ?
1090 RECT ENDS
1091 .data
1092     rect1 RECT <10, 20, 100, 150> ; Define a RECT structure with specific coordinates
1093 .code
1094 main PROC
1095     mov eax, rect1.left      ; Access the left coordinate
1096     mov ebx, rect1.top       ; Access the top coordinate
1097     mov ecx, rect1.right     ; Access the right coordinate
1098     mov edx, rect1.bottom    ; Access the bottom coordinate
1099     ; Now you can use these values for various tasks
1100     ; For example, you can calculate the width and height of the rectangle
1101     sub ecx, eax            ; Width = right - left
1102     sub edx, ebx             ; Height = bottom - top
1103     ; Display the width and height
1104     call DisplayWidthAndHeight
1105     ; You can also modify the coordinates or dimensions as needed
1106     add rect1.left, 5        ; Move the left side 5 units to the right
1107     sub rect1.right, 10       ; Shrink the width by 10 units
1108     ; Now the rect1 structure has been updated
1109     exit
1110 main ENDP
1111 DisplayWidthAndHeight PROC
1112     ; Display the width and height
1113     ; You can implement this function as needed
1114     ret
1115 DisplayWidthAndHeight ENDP
```

Read **GraphicalWindow2.asm**

And then finish off with the **GraphicalWindow3.asm**

WinMain - Entry point; initializes program, registers window class, creates main window, shows greeting, and starts the message loop.

WinProc

1. Handles messages/events like:

- ⊕ WM_CREATE → Display “application loaded” popup
- ⊕ WM_LBUTTONDOWN → Display “clicked mouse” popup
- ⊕ WM_CLOSE → Display “closing” popup & quit

2. Other messages → forwarded to DefWindowProc.

ErrorHandler

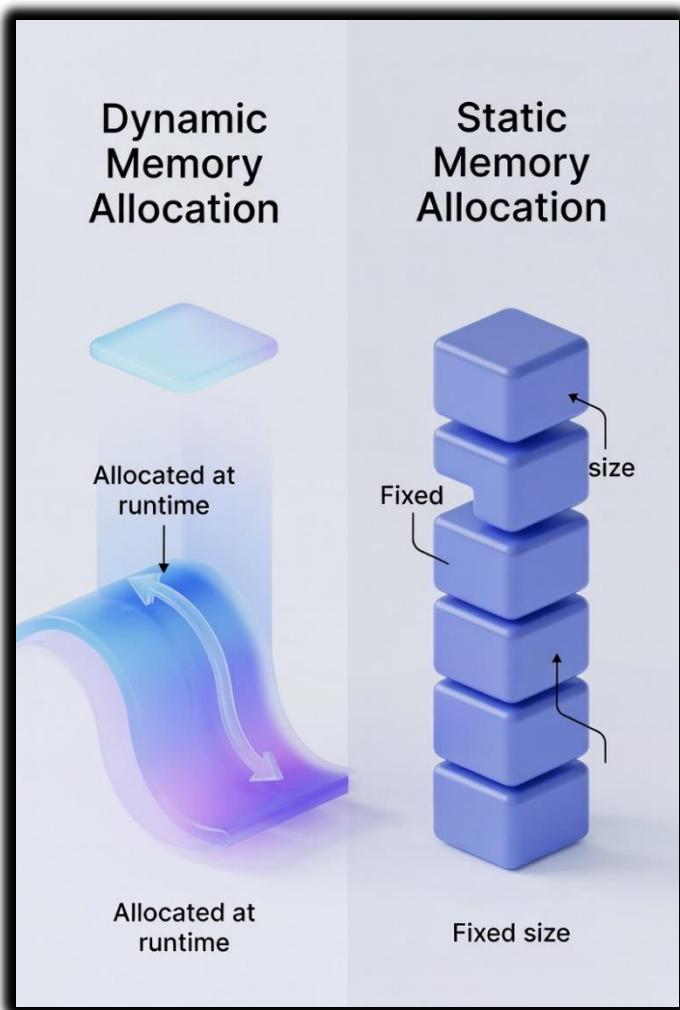
- ⊕ Called if registration or window creation fails.
- ⊕ Retrieves system error message and shows it in a MessageBox.
- ⊕ Frees memory allocated for error message.

3. **MessageBox Demos** - Shows how to display popups for various events, similar to the C version provided.
4. **Supports Multiple Features Together** - Combines greeting popup, left-click message, close message, and error handling into one MASM program.

DYNAMIC MEMORY

Dynamic memory allocation is the process of allocating memory **at runtime** during program execution.

This contrasts with **static memory allocation**, where memory is allocated at **compile time**.



Methods of Dynamic Memory Allocation

1. Using System Calls

- Make calls to the operating system to allocate and free memory.
- Example: Windows API functions like HeapAlloc, HeapFree, GlobalAlloc, and VirtualAlloc.

2. Implementing a Heap Manager

- Create your own data structures and algorithms to manage memory.
- More flexible but requires careful design to avoid fragmentation or leaks.

Steps for Dynamic Memory Allocation Using System Calls

1. Allocate memory

- Call a Windows API function to allocate a memory block.
- The function returns a pointer to the allocated memory.

2. Use the memory - Access or modify the memory as needed for program operations.

3. Deallocate memory

- Call a system function to free the allocated memory.
- Ensures that memory is returned to the system and prevents leaks.

Common Win32 API Functions for Dynamic Memory Allocation

FUNCTION	DESCRIPTION	SUCCESS (EAX)	FAILURE (EAX)
<code>GetProcessHeap</code>	Obtains a handle to the default heap of the calling process.	<code>Heap Handle</code>	<code>NULL (0)</code>
<code>HeapCreate</code>	Creates a private heap object for the application.	<code>Heap Handle</code>	<code>NULL (0)</code>
<code>HeapAlloc</code>	Allocates a block of memory from a specific heap.	<code>Memory Address</code>	<code>NULL (0)</code>
<code>HeapReAlloc</code>	Resizes an existing memory block from a heap.	<code>New Address</code>	<code>NULL (0)</code>
<code>HeapFree</code>	Frees a memory block allocated from a heap.	<code>Non-zero</code>	<code>0</code>
<code>HeapDestroy</code>	Destroys a private heap and all its allocated blocks.	<code>Non-zero</code>	<code>0</code>
<code>HeapSize</code>	Retrieves the size of a block allocated from a heap.	<code>Size (Bytes)</code>	<code>SIZE_T -1</code>

- Always **pair every allocation with a deallocation** to avoid memory leaks.
- System calls handle most of the memory management for you, unlike custom heap managers.
- Dynamic allocation is especially useful when the program **doesn't know the exact memory size** it will need at compile time.

Win32 Heap Functions

- **GetProcessHeap()**
 - Returns a handle to the **current process's default heap**.
 - Use when the default heap is sufficient for your program.
- **HeapCreate()**
 - Creates a **new private heap** for the process.
 - Use when you need **more control** over memory management.
- **HeapDestroy()**
 - Destroys an existing private heap.
 - Use when the heap is no longer needed to free resources.
- **HeapAlloc()**
 - Allocates a block of memory from a specified heap.
 - Always pair with **HeapFree()** to avoid leaks.
- **HeapFree()** - Frees a memory block previously allocated from a heap.

When to Use Each Function

- Use **GetProcessHeap()** if the **default heap** is sufficient.
- Use **HeapCreate()** if you need a **private heap** for more controlled memory management.
- Use **HeapDestroy()** to free a private heap when finished.
- Use **HeapAlloc()** to **allocate memory** from a heap.
- Use **HeapFree()** to **free memory** allocated from a heap.

Example: Allocating and Freeing Memory from a Heap (MASM)

```
.data
hHeap    HANDLE ?
pBlock   DWORD ?

.code
start:
    ; Get handle to the process's default heap
    invoke GetProcessHeap
    mov hHeap, eax

    ; Allocate 256 bytes from the heap
    invoke HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 256
    mov pBlock, eax
    .IF eax == 0
        ; Allocation failed
        invoke MessageBox, NULL, "Heap allocation failed!", "Error", MB_OK
        jmp exit
    .ENDIF

    ; -----
    ; Memory block can be used here
    ; For example, store a value:
    mov dword ptr [pBlock], 12345

    ; Free the memory block
    invoke HeapFree, hHeap, 0, pBlock

exit:
    invoke ExitProcess, 0
```

Or

```
1225 ; Create a new private heap.
1226 INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
1227
1228 ; Allocate a block of memory from the heap.
1229 INVOKE HeapAlloc, hHeap, 0, 1000
1230
1231 ; Use the allocated memory block.
1232 ; ...
1233
1234 ; Free the allocated memory block.
1235 INVOKE HeapFree, hHeap, 0, pArray
1236
1237 ; Destroy the private heap.
1238 INVOKE HeapDestroy, hHeap
```

It is important to note that dynamic memory allocation should be used carefully to avoid memory leaks.

A **memory leak** occurs when a program **allocates memory** but **does not free it** when it is finished using it.

Memory leaks can lead to performance problems and eventually cause the program to crash.

Read **HeapTest1.asm** and **HeapTest2.asm**

Key Highlights from HeapTest2:

- **Custom Heap Management:** Unlike the default process heap, this creates a specific sandbox with a **400MB limit**.
- **Error Handling:** It uses the Windows API WriteWindowsMsg to translate numeric error codes into human-readable text.
- **Visual Feedback:** By printing a . for every \$0.5\text{text}\\$ allocated, you can actually see the memory "filling up" in the console.
- **Status Flags:** The allocate_block procedure uses the Carry Flag as a clean way to communicate status back to the main loop without cluttering registers.

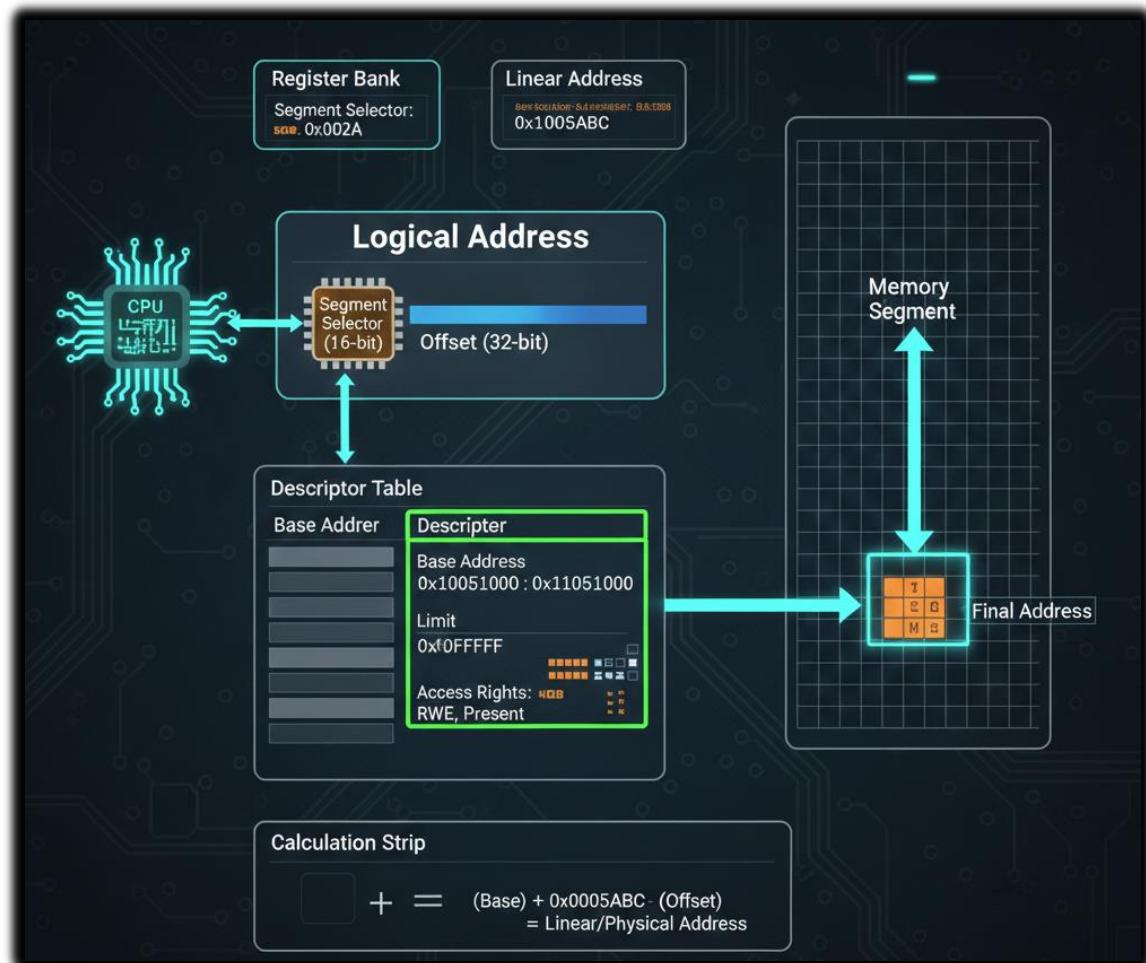
x86 MEMORY MANAGEMENT

In x86 processors, memory can be addressed in two ways: **logical addresses** and **linear addresses**.

I. Logical Address:

Think of this as a “map” to a memory location. It’s made up of two parts:

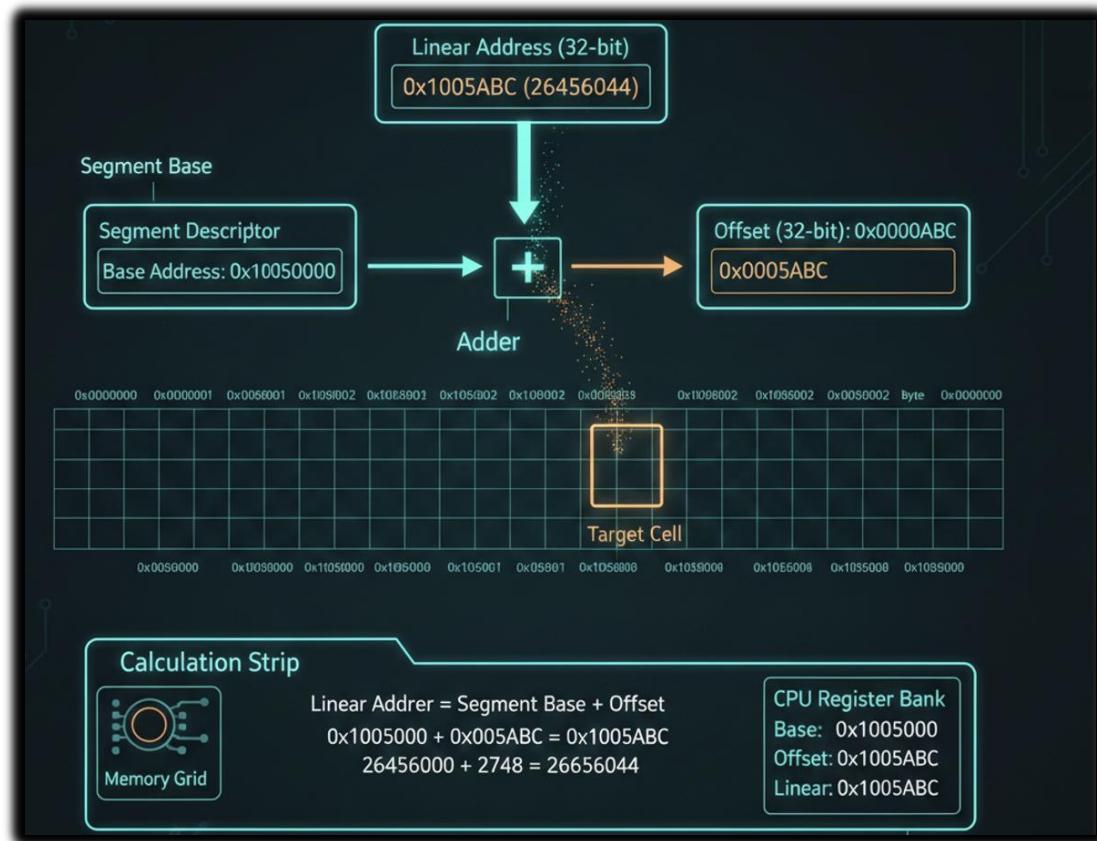
- **Segment selector** – a 16-bit value that points to a segment descriptor. The descriptor tells the CPU all about that segment of memory.
- **Offset** – a 32-bit number that tells you the exact spot inside that segment.



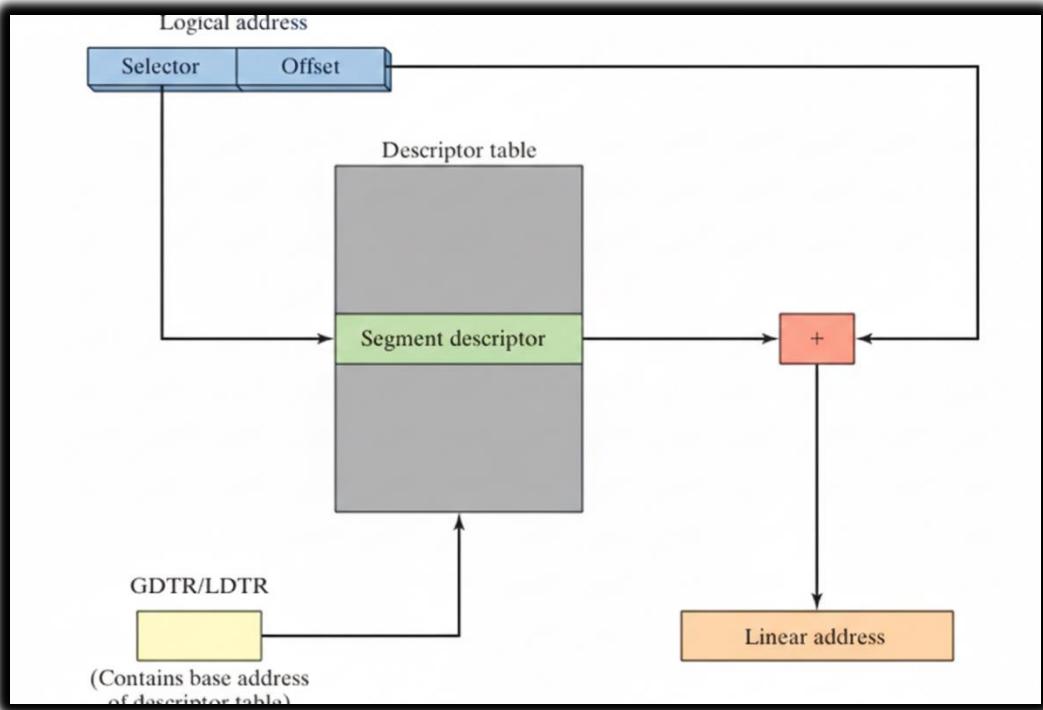
II. Linear Address:

This is the “real” memory location. It’s a 32-bit value that directly identifies a spot in memory.

You get it by adding the **segment’s base address** to the **offset**.



So, when the CPU goes from a logical address to a linear address, it’s essentially following a two-step translation process.



The **segment selector** acts like an index into a table of segment descriptors—either the **GDT** (Global Descriptor Table) or **LDT** (Local Descriptor Table). Once the CPU finds the right segment descriptor, it grabs the **segment's base address**.

Next, it simply adds the **offset** from the logical address to this base address, giving the **linear address**—the CPU's actual pointer to memory.

You can imagine it like looking up a street name in a city directory (the segment descriptor table) and then walking a certain number of steps down the street (the offset) to reach the exact house (linear address).

```

1442 Logical address = segment selector + offset
1443 Segment base address = segment descriptor table[segment selector]
1444 Linear address = segment base address + offset

```

Once the **linear address** is calculated, the CPU can go straight to memory and access the data it needs.

Example:

Imagine we have a program with a variable stored **200h** bytes into a segment.

The segment selector for this segment is **0x1000**, and the segment descriptor table shows that this segment's **base address** is **0x100000**.

To find the variable in memory, the processor just adds the offset to the segment base address:

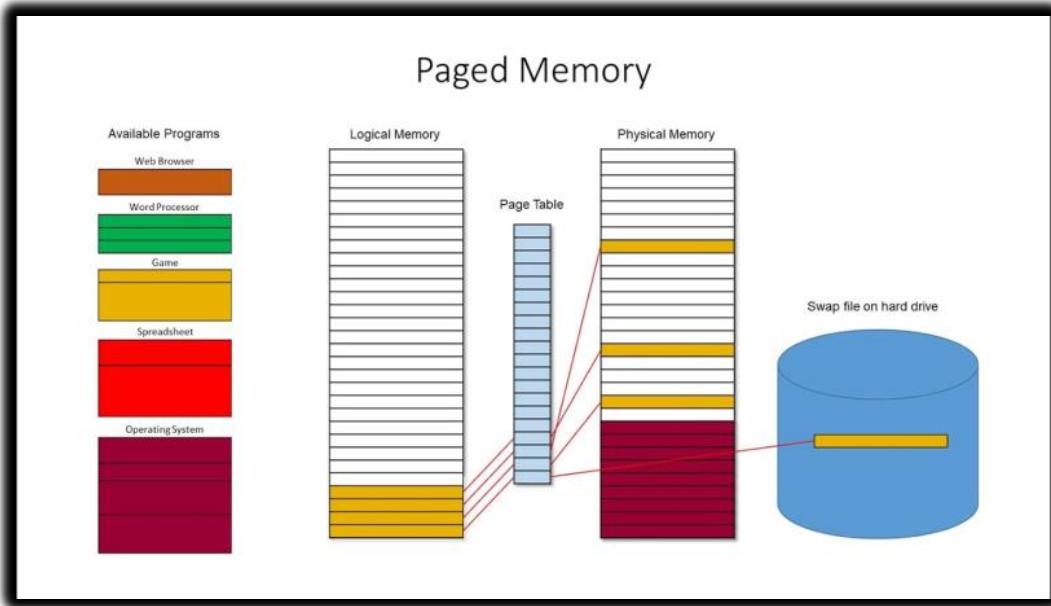
```
1448 Linear address = segment base address + offset  
1449 Linear address = 0x100000 + 200h  
1450 Linear address = 0x100200
```

So, the CPU can now directly access the variable at **0x100200** in memory.

Paging

Paging is a memory management trick that helps the operating system organize memory more efficiently. It breaks **physical memory** into small, fixed-size chunks called **pages**.

Think of it like cutting a big cake into neat slices—each slice (page) is easier to manage and track than the whole cake at once.



Pages are usually **4KB** in size, but they can also be bigger—like **2MB** or more—depending on the system.

When a program wants to access memory, the **operating system** steps in to translate the program's **linear address** into a **physical address**. It does this using a **page table**.

The **page table** is basically a map that tells the CPU exactly which chunk of physical memory corresponds to each linear address.

You can picture it like a GPS: the linear address is your “destination name,” the page table is the “map,” and the physical address is the exact street location you need to reach.

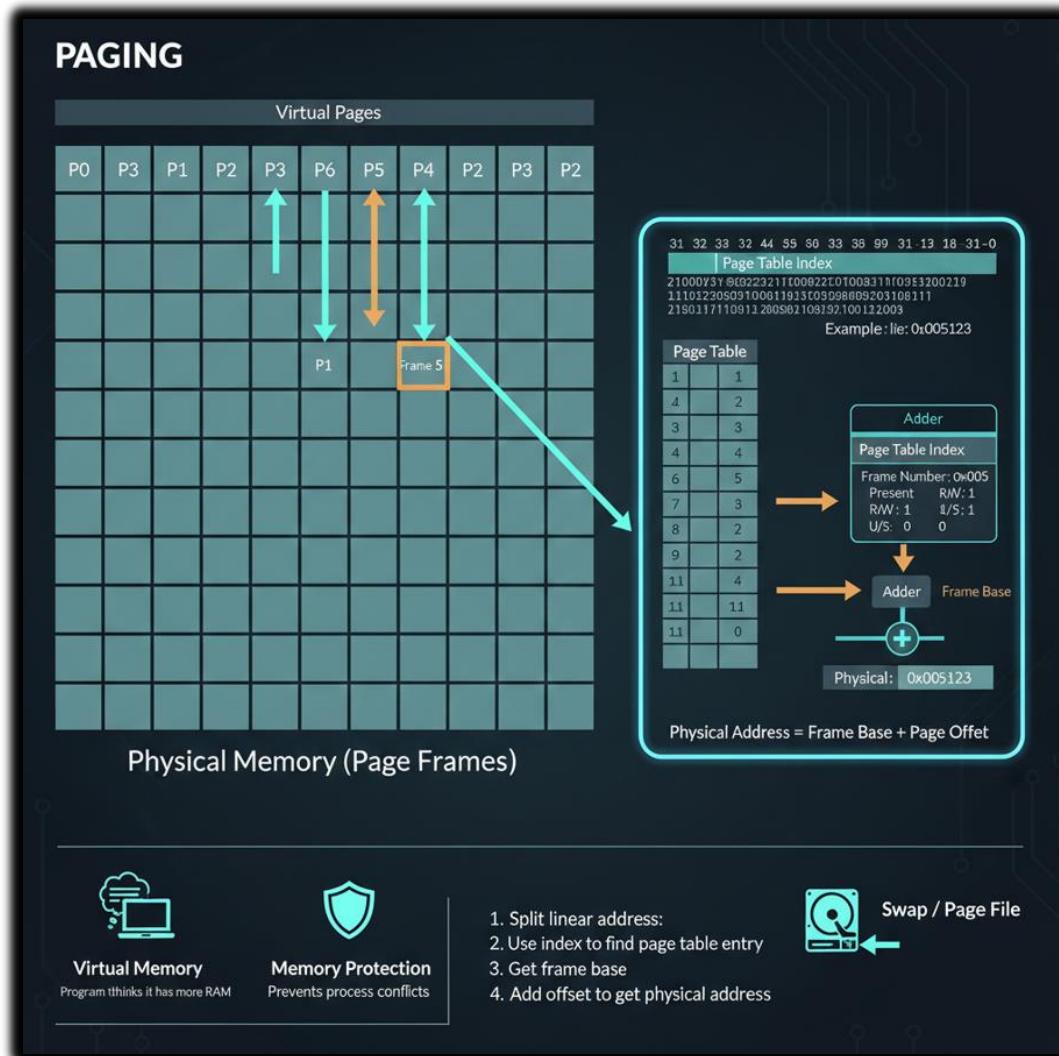
1454 Linear address = page table index + page offset

1455 Physical address = page table[page table index] + page offset

In paging, the **linear address** is split into two parts:

- **Page table index** – the upper 20 bits, which point to the right entry in the page table.
- **Page offset** – the lower 12 bits, which tell you the exact spot within that page.

Paging gives the operating system some powerful tools, like **virtual memory** (making programs think they have more memory than physically exists) and **memory protection** (keeping programs from accidentally messing with each other's memory).



Conclusion

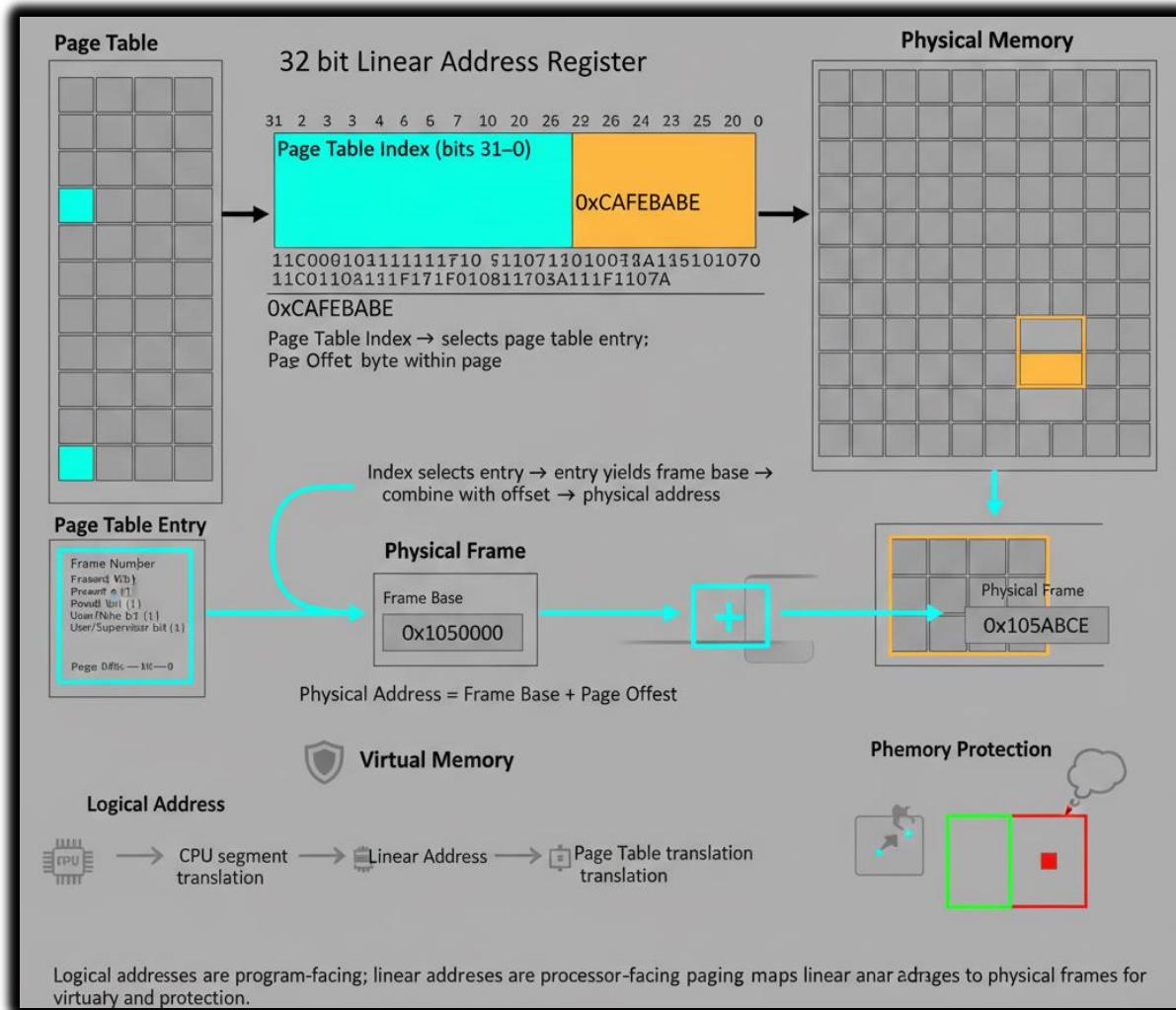
Memory in an x86 processor can be addressed in two ways:

- **Logical addresses** – used by programs to refer to memory.
- **Linear addresses** – used by the processor to pinpoint locations in memory.

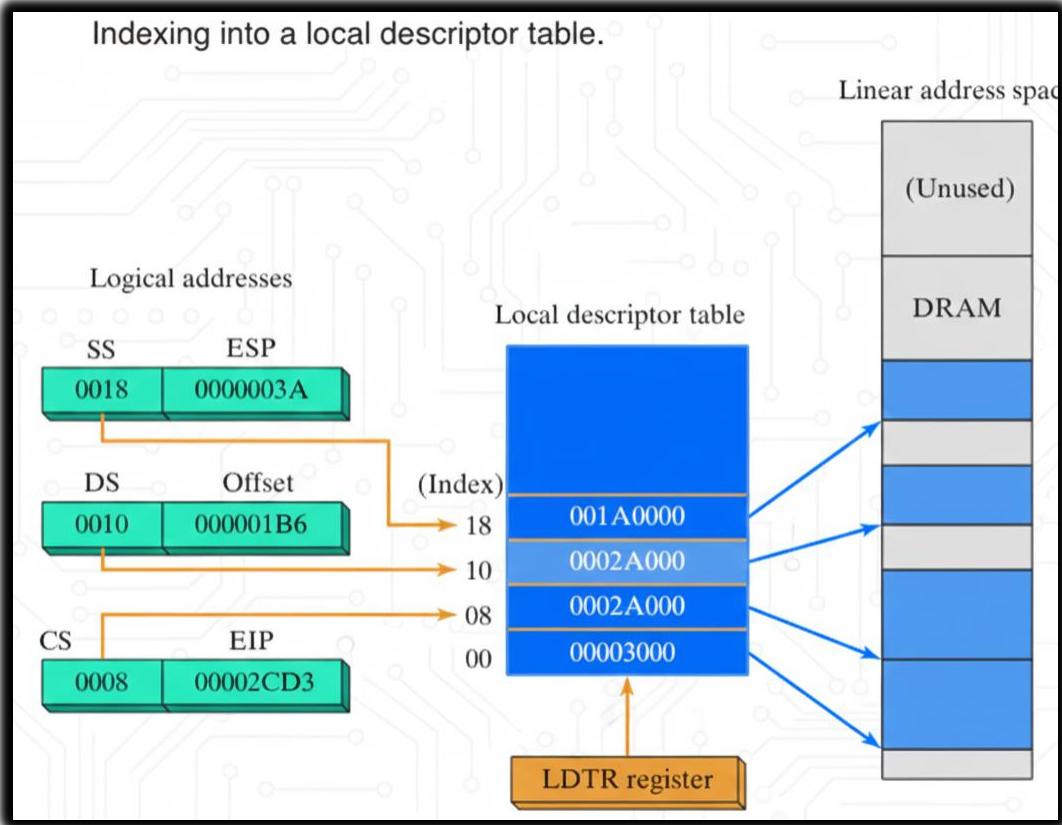
Here's how it all comes together:

1. The CPU translates **logical addresses** into **linear addresses** using **segment descriptors**.
2. Linear addresses can then be translated into **physical addresses** using a **page table**.

It's like a two-step navigation system: first the CPU finds the right "street" (linear address), then it finds the exact "house" (physical address).



Descriptor tables



Descriptor tables are like guides that tell the CPU everything it needs to know about memory segments. A **segment** is just a chunk of memory where a program can store code or data.

There are two main types of descriptor tables:

- **Global Descriptor Table (GDT):** Contains descriptors for all the memory segments used by the system.
- **Local Descriptor Table (LDT):** Each process or task has its own LDT, containing descriptors for the segments it uses.

Each **segment descriptor** holds key information about a segment, such as its **base address**, **size**, and **access rights**. The CPU uses this info to translate **logical addresses** into **linear addresses**.

A **logical address** has two parts:

1. **Segment selector** – a 16-bit value that points to a segment descriptor in the GDT or LDT.
2. **Offset** – a 32-bit value that tells the CPU the exact location inside that segment.

To get the **linear address**, the CPU simply adds the segment's **base address** to the **offset**. The linear address is then ready to directly access memory.

Example:

Imagine a program with a variable located at **offset 200h** in a segment with **segment selector 0x1000**. The GDT shows that this segment has a **base address of 0x100000**.

To access the variable, the CPU calculates the linear address:

```
1459 Linear address = segment base address + offset  
1460 Linear address = 0x100000 + 200h  
1461 Linear address = 0x100200
```

Once that's done, the CPU can directly access the variable at **memory location 0x100200**.

Segment Descriptor Details

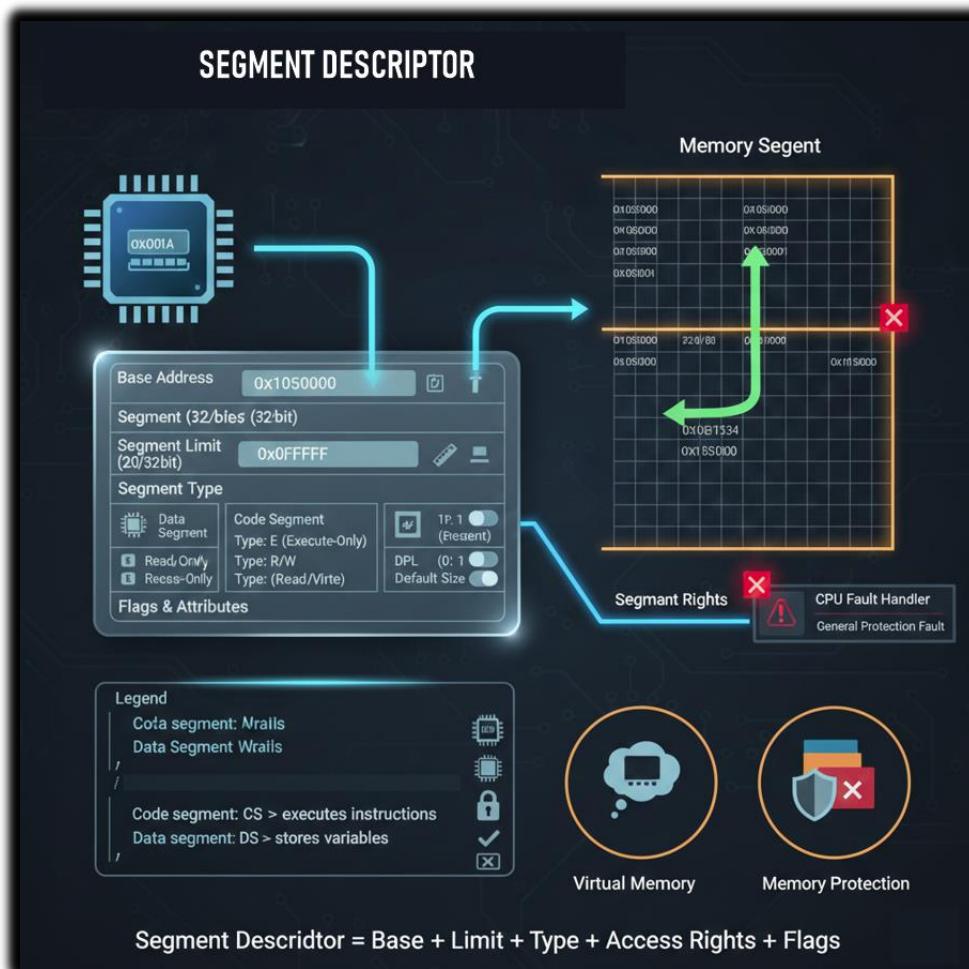
A **segment descriptor** is more than just a base address—it's like a detailed profile for a segment that tells the CPU how to use it safely and correctly. Here's what it contains:

- **Segment Limit:** This sets the maximum size of the segment. If a program tries to go beyond this limit, the CPU raises a **fault** to prevent memory errors.
- **Segment Type:** This describes what kind of data the segment holds. For example:
 - **Code segment** – stores instructions
 - **Data segment** – stores variables or other data
- **Access Rights:** These control what operations are allowed on the segment. For instance:
 - **Read-only** segments can only be read
 - **Write-only** segments can only be written toThe CPU uses these rules to prevent unauthorized memory access.

Other important fields in a segment descriptor include:

- **Base Address:** Where the segment starts in the linear address space
- **Privilege Level:** Indicates who can access the segment, ranging from **0 (most privileged)** to **3 (least privileged)**
- **Segment Present Flag:** Shows whether the segment is currently in memory
- **Granularity Flag:** Determines whether the segment limit is measured in **bytes** or **4KB blocks**
- **Segment Limit:** Maximum size of the segment
- **Protection Level:** Helps protect operating system data from access by applications

The CPU enforces these rules: if a program with a higher privilege level tries to access a segment with a lower privilege level (like a user program trying to touch OS data), it triggers a **processor fault**. This keeps critical system data safe from mistakes or malicious activity.



So, here's how those fields actually work together:

- **Segment Type Field:**

This field defines *both* what kind of data the segment holds and what you're allowed to do with it. For example, a **code segment** can be executed, while a **data segment** can be read from or written to.

- **Segment Present Flag:**

This flag tells the processor whether the segment is currently loaded in memory.

- ⊕ If the flag is **set**, the segment is present and can be accessed.
- ⊕ If it's **not set**, the segment isn't in memory, and any attempt to use it will fail.

- **Granularity Flag:**

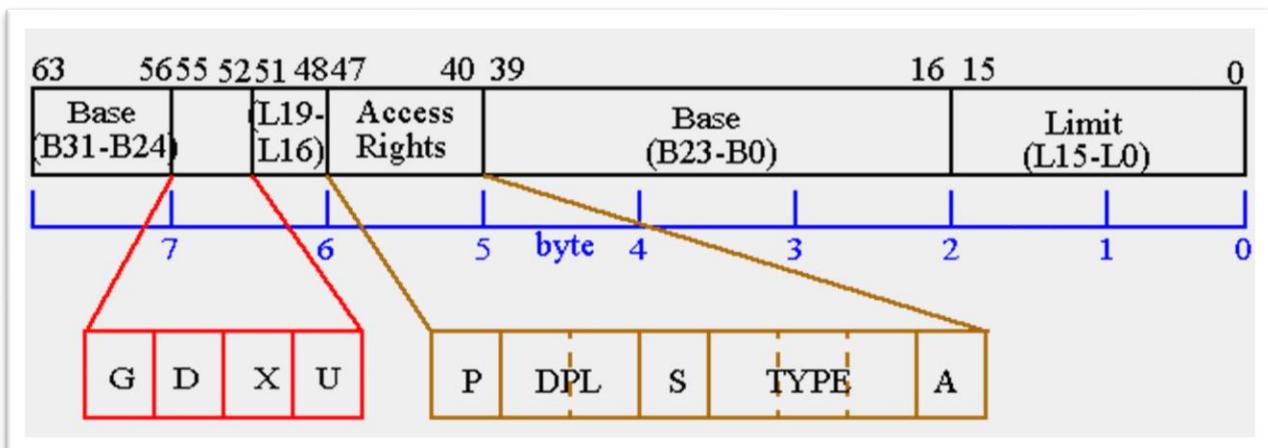
This flag controls how the **segment limit** is measured.

- ⊕ If the flag is **set**, the limit is interpreted in **4096-byte (4KB) units**.
- ⊕ If it's **not set**, the limit is interpreted in **bytes**.

- **Segment Limit Field:**

This field defines the maximum size of the segment. If a program tries to access memory beyond this limit, the processor immediately raises a **fault** to stop the access.

Overall, **segment descriptors** play a critical role in x86 memory management. They allow the processor to translate **logical addresses** into **linear addresses**, while also enforcing protection rules so programs can't access memory they're not supposed to.



Described segment descriptor:

Segment Descriptor Fields		
FIELD	SIZE (BITS)	DESCRIPTION
Base address	32	The starting address of the segment in the linear address space.
Privilege level	2	The privilege level required to access the segment.
Segment type	2	The type of segment, such as code, data, or stack.
Segment present flag	1	Indicates whether the segment is present in memory.
Granularity flag	1	Determines whether the segment limit is interpreted in bytes or 4096-byte units.
Segment limit	20	The maximum size of the segment.

The **segment descriptor diagram** also highlights a few important details:

- The **Global Descriptor Table (GDT)** is located at **address 0x00000000**.
- The **segment selector** is a **16-bit value** that points to a specific segment descriptor inside the GDT.
- The **linear address** is a **32-bit value** that identifies an exact location in the linear address space.

Here's how the processor puts it all together:

The CPU uses the **segment selector** to index into the **GDT** and fetch the correct **segment descriptor**. It then uses the information in that descriptor—especially the **base address**—to calculate the **linear address** of the requested memory location.

In short, the diagram shows how the processor moves step by step from a logical reference to a real, usable memory address.

NB(FORMER SUBTOPIC):

Logical to Linear Address Translation

This process converts a logical address (Selector + Offset) into a linear address. The CPU uses the **GDTR/LDTR** to find the **Descriptor Table**, indexes it via the **Selector**, and adds the resulting **Base Address** to the **Offset**.

Page translation

Page translation is the process of converting a linear address to a physical address in an x86 processor when paging is enabled.

A linear address is a 32-bit value that uniquely identifies a location in memory. A physical address is also a 32-bit value, but it identifies a location in physical memory.

Paging allows the operating system to divide physical memory into pages, which are typically 4KB in size. The operating system then uses a page table to map virtual addresses to physical addresses.

The page table is a data structure that contains one entry for each page in the virtual address space.

Each entry contains the physical address of the page and its access rights.

When the processor needs to access a memory location, it first translates the linear address to a physical address using the page table.

The processor does this by looking up the page table entry for the linear address. The page table entry contains the physical address of the page and its access rights.

If the page table entry is valid, the processor uses the physical address to access the memory location.

If the page table entry is not valid, the processor generates a page fault.

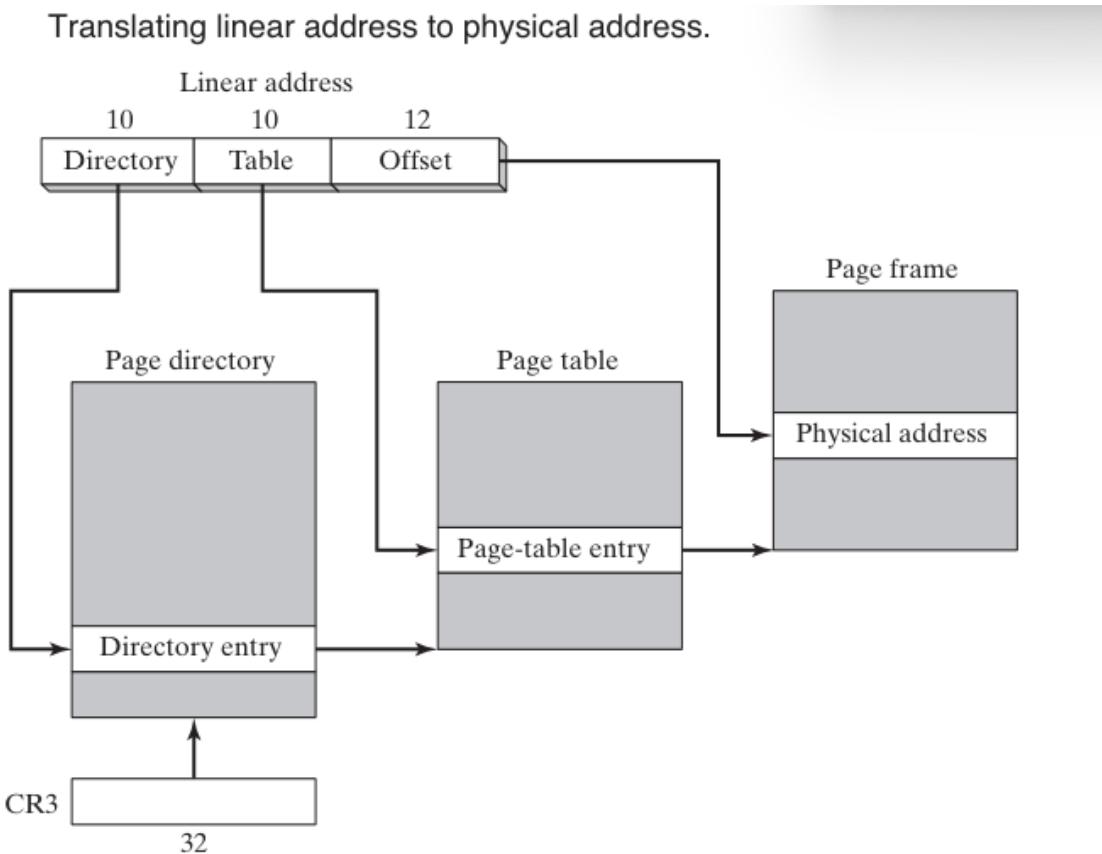
Steps in page translation

Let's describe the page table image first:

The linear address references a location in the linear address space. The 10-bit directory field in the linear address is an index to a page-directory entry. The page-directory entry contains the base address of a page table.

The 10-bit table field in the linear address is an index into the page table identified by the page-directory entry. The page-table entry at that position contains the base location of a page in physical memory.

The 12-bit offset field in the linear address is added to the base address of the page, generating the exact physical address of the operand.



The following steps are carried out by the processor when translating a linear address to a physical address:

The processor splits the linear address into three fields:

Directory field: The directory field is the upper 10 bits of the linear address.

Table field: The table field is the middle 10 bits of the linear address. **Offset field:** The offset field is the lower 12 bits of the linear address.

The processor uses the directory field to index the page directory. The page directory is a table of 1024 4-byte entries.

Each entry in the page directory points to a page table. The processor uses the table field to index the page table pointed to by the directory entry.

The page table is also a table of 1024 4-byte entries. Each entry in the page table points to a physical page frame.

The processor adds the offset field to the physical address of the page frame pointed to by the page table entry. This results in the physical address of the memory location. Example:

Suppose we have a linear address of 0x12345678. The **directory field** would be 0x1234, the **table field** would be 0x5678, and the **offset field** would be 0x123456.

The processor would first use the directory field to index the page directory. The page directory entry at index 0x1234 would contain the address of the page table.

The processor would then use the table field to index the page table. The page table entry at index 0x5678 would contain the physical address of the page frame.

Finally, the processor would add the offset field to the physical address of the page frame. This results in the physical address of the memory location, which is 0x12345678.

Conclusion

The operating system has the option of using a single page directory for all running programs and tasks, or one page directory per task, or a combination of the two.

Page translation is an important part of memory management in x86 processors. It allows the operating system to divide physical memory into pages and to map virtual addresses to physical addresses.

This allows the operating system to implement virtual memory and to protect memory from unauthorized access.

=====

Windows Virtual Machine Manager

=====

Windows Virtual Machine Manager (VMM) is the 32-bit protected mode operating system at the core of Windows. It is responsible for creating, running, monitoring, and terminating virtual machines. It also manages memory, processes, interrupts, and exceptions.



Microsoft®
System Center
Virtual Machine Manager

VMM uses a single 32-bit flat model address space at privilege level 0. This means that all of the virtual machines, the VMM itself, and any virtual devices all share the same address space.

The VMM creates two global descriptor table (GDT) entries for each virtual machine, one for code and one for data. These segments are fixed at linear address 0.

VMM provides multithreaded, preemptive multitasking. This means that it can run multiple applications simultaneously by sharing CPU time between the virtual machines in which the applications run.

How VMM handles memory management

VMM uses a technique called paging to manage memory. Paging divides physical memory into pages, which are typically 4KB in size. VMM then uses a page table to map virtual addresses to physical addresses.

Each virtual machine has its own page table. The page table tells the processor which physical page contains a particular virtual address.

When a virtual machine tries to access a memory location, the processor first looks up the page table entry for that virtual address.

If the page table entry is valid, the processor uses it to access the physical memory location. If the page table entry is not valid, the processor generates a page fault.

Page faults are handled by the VMM. When a page fault occurs, the VMM checks to see if the virtual address is valid. If it is, the VMM loads the corresponding physical page into memory and updates the page table entry.

If the virtual address is not valid, the VMM generates an exception.

Benefits of using VMM

VMM offers a number of benefits, including:

Isolation: VMM isolates virtual machines from each other, so that a failure in one virtual machine does not affect other virtual machines.



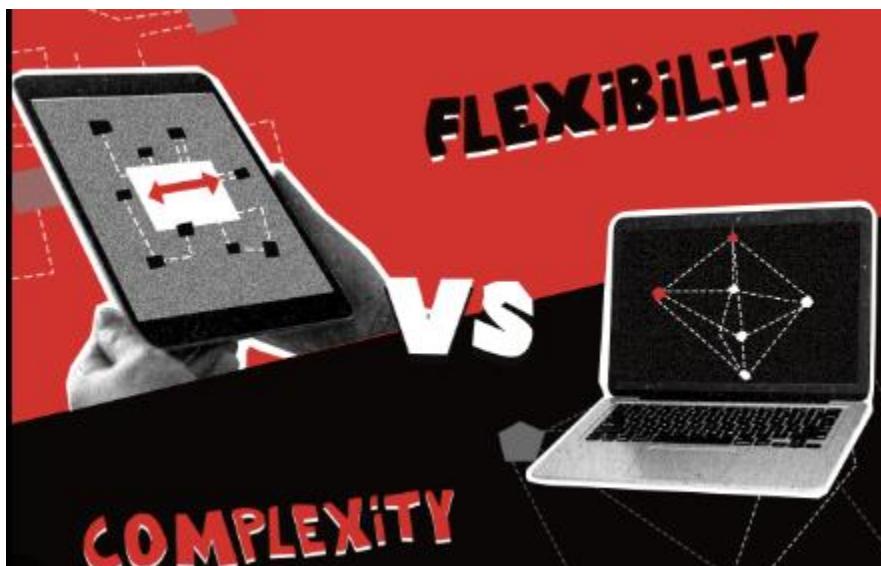
Security: VMM can be used to implement security features such as access control and encryption.



Performance: VMM can improve the performance of applications by running them in separate virtual machines.



Flexibility: VMM can be used to create and manage different types of virtual machines, such as servers, desktops, and test environments.



Windows Virtual Machine Manager is a powerful tool that can be used to create, run, and manage virtual machines. VMM offers a number of benefits, including isolation, security, performance, and flexibility.