WinAPI in ASM

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:

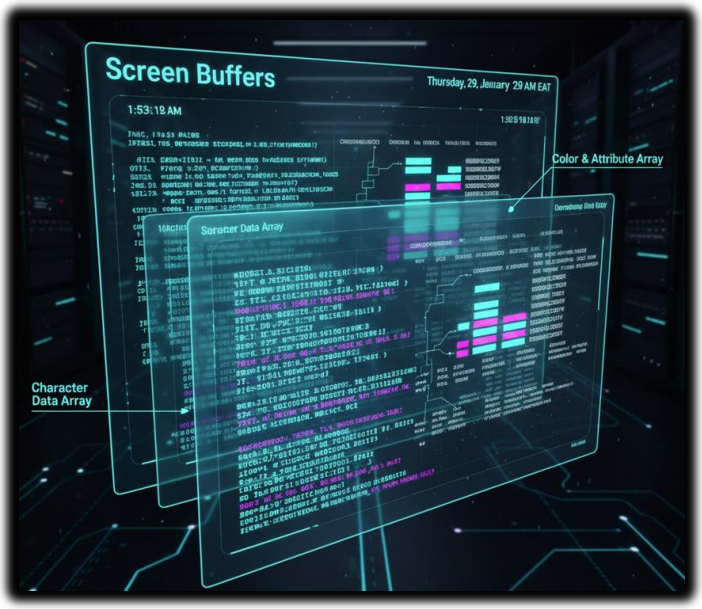
One input buffer, which stores input records. These records describe things like:

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

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

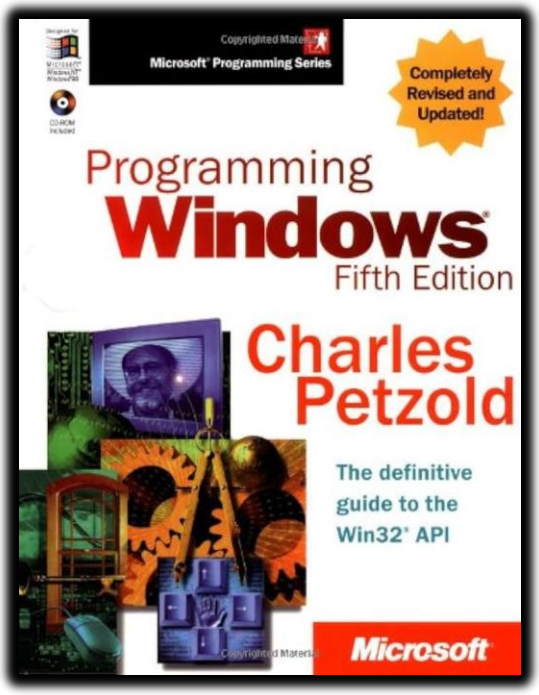
- Character data
- Color and attribute information

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.lib
    These 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:

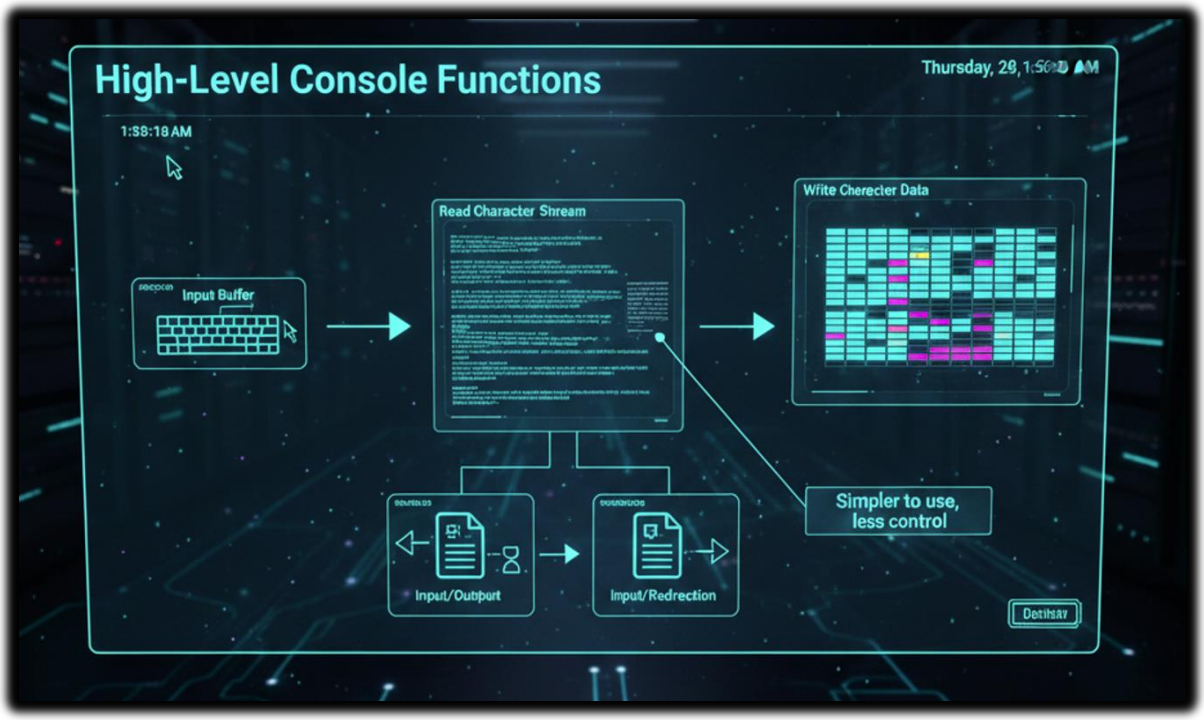
```
WriteConsole EQU <WriteConsoleA>
```

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:

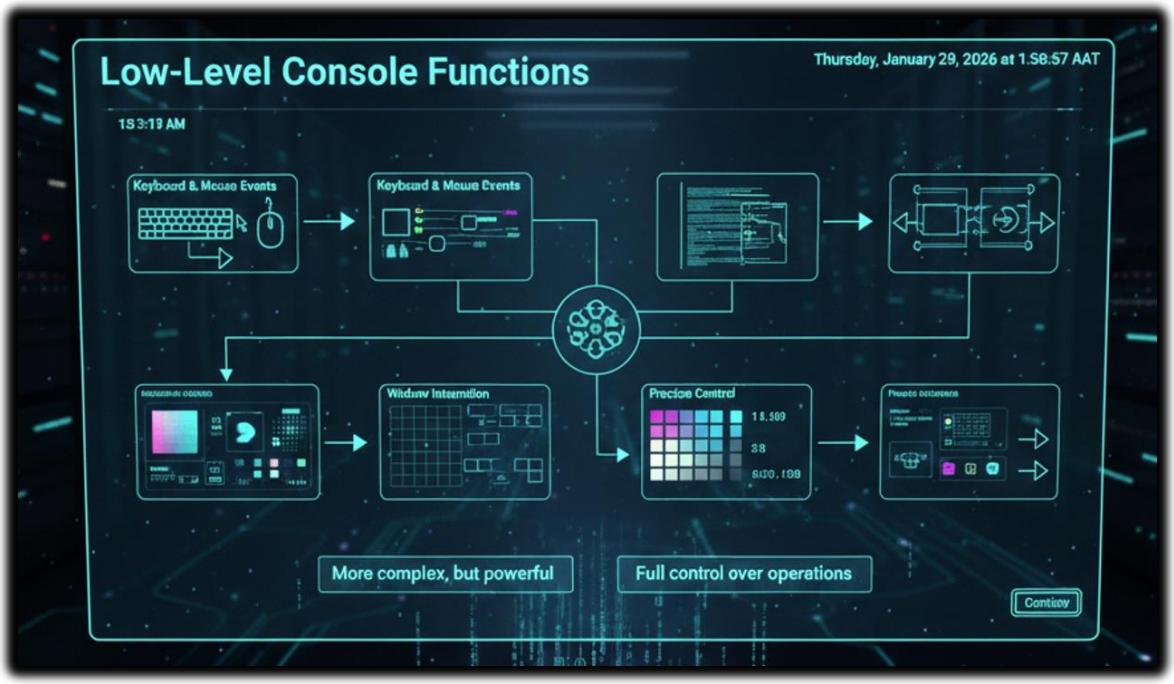
High-Level Console Functions

- 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:

```
handleVariable: DWORD

; Get a handle to the console window.
invoke GetConsoleWindow, handleVariable

; Use the handle to write a message to the console.
invoke WriteConsole, handleVariable, addr message, length message, bytesWritten, NULL

; Close the handle to the console window.
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:

```
; Get a handle to the standard input handle.
invoke GetStdHandle, STD_INPUT_HANDLE, handleVariable

; Get a handle to the standard output handle.
invoke GetStdHandle, STD_OUTPUT_HANDLE, handleVariable

; Get a handle to the standard error handle.
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.

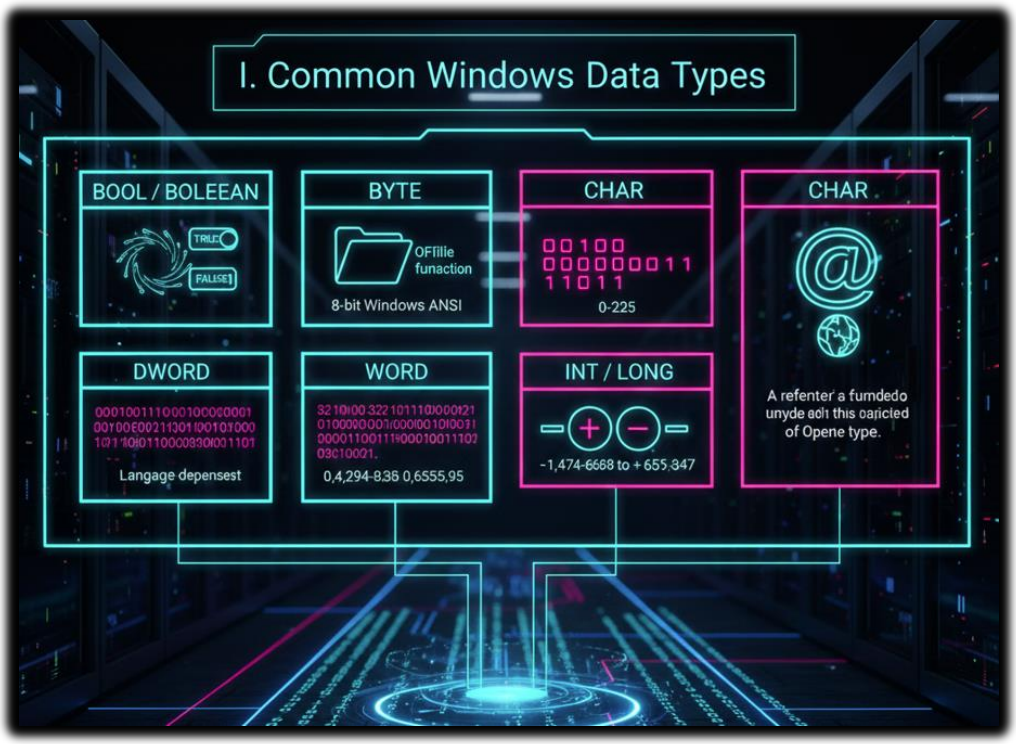
It's automatically included when you use Irvine32.inc, so in most MASM projects, you already have access to it.

I. Common Windows Data Types

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

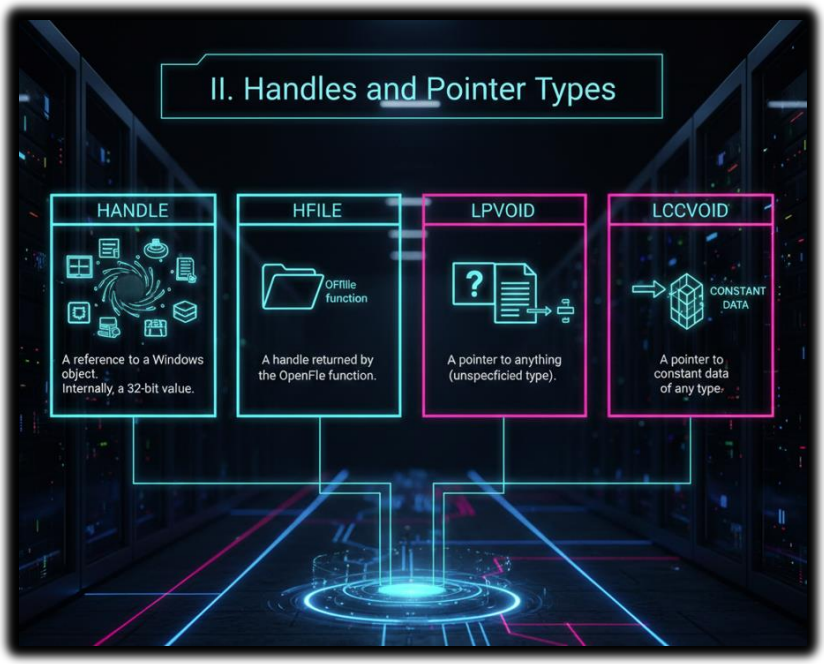
- **BOOL / BOOLEAN**
  A simple true-or-false value.
- **BYTE**
  An 8-bit unsigned integer (0–255).
- **CHAR**
  An 8-bit Windows ANSI character.
  This is what older (non-Unicode) Windows programs use for text. The exact character set depends on language and region.
- **DWORD**
  A 32-bit unsigned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

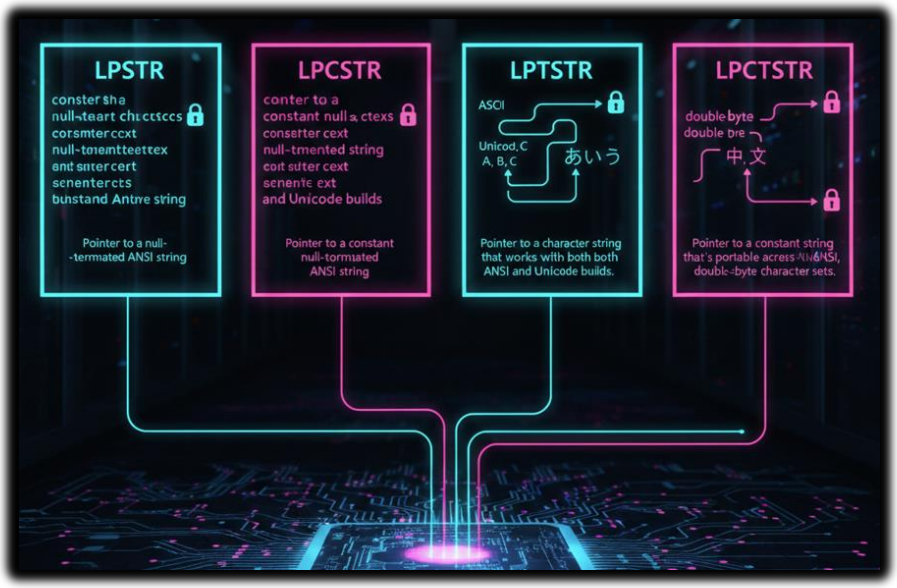
These show up everywhere in the Windows API:

LPSTR - Pointer to a null-terminated ANSI string.

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

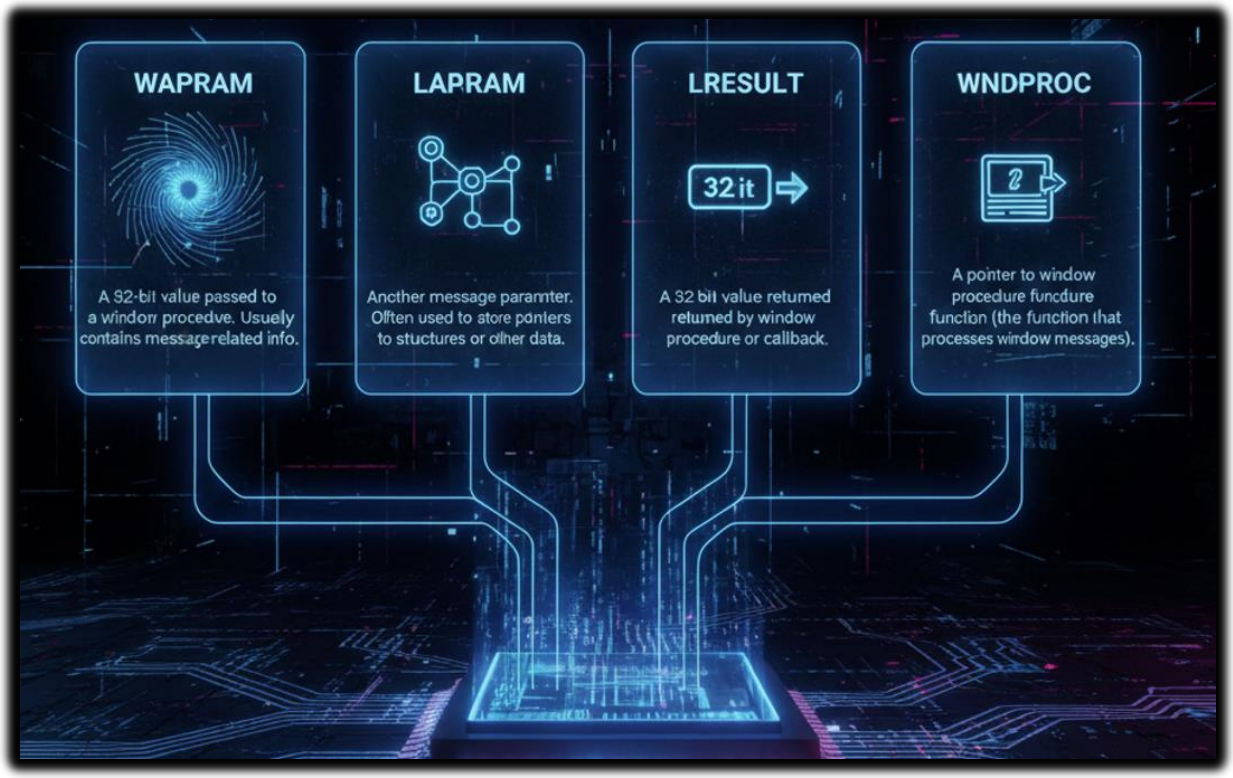
LPTSTR - Pointer to a character string that works with both ANSI and Unicode builds.

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).



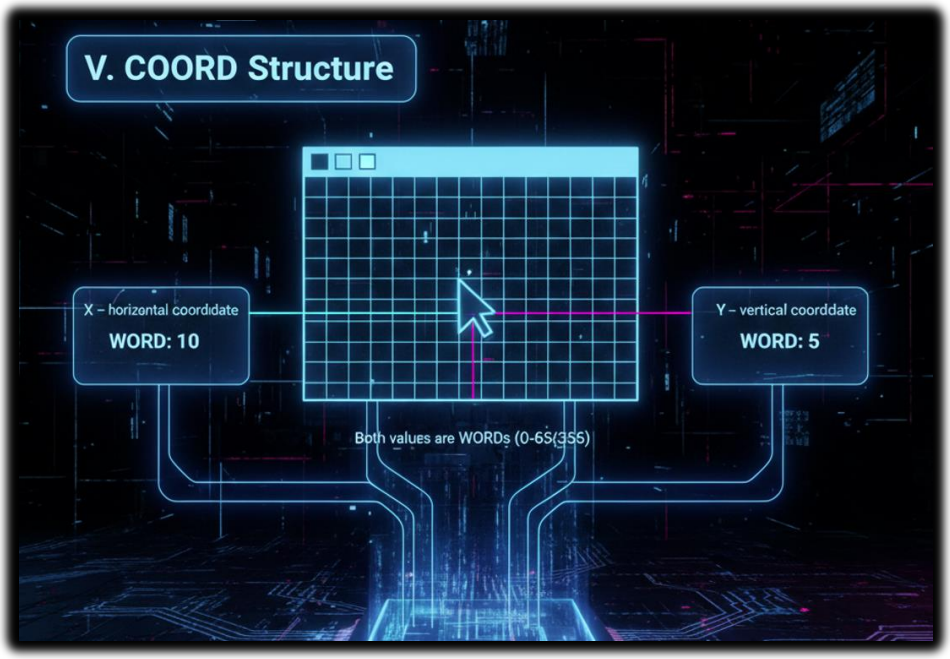
Special Structures

I. COORD Structure

Used to store a position in the console window.

- X – horizontal coordinate
- Y – vertical coordinate

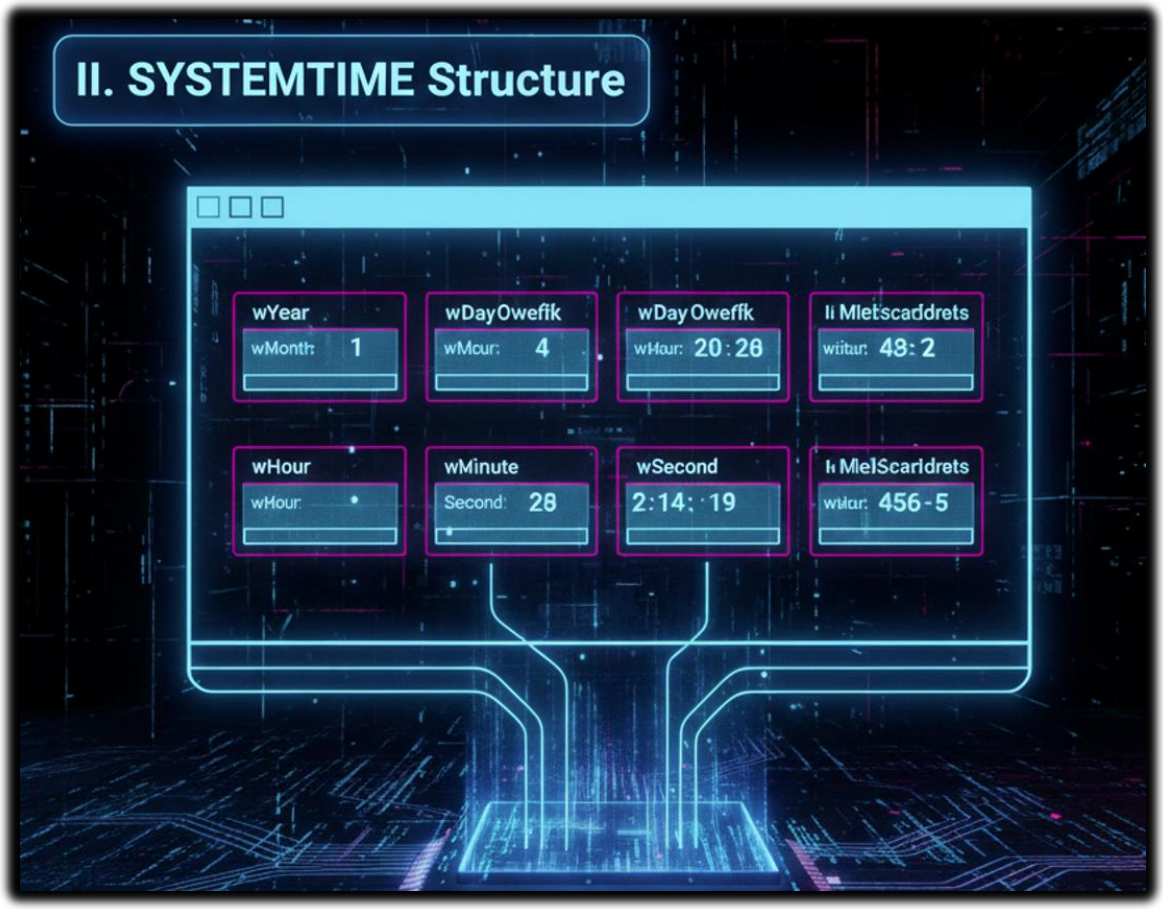
Both values are WORDs.



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



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

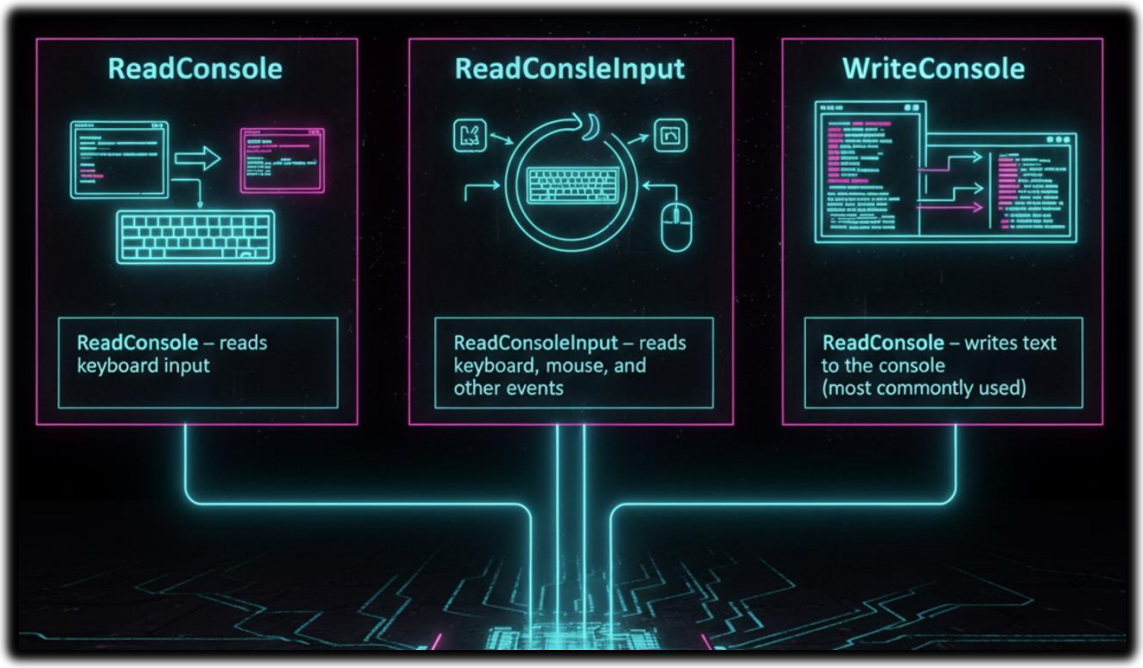
You retrieve these using: **GetStdHandle**

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)

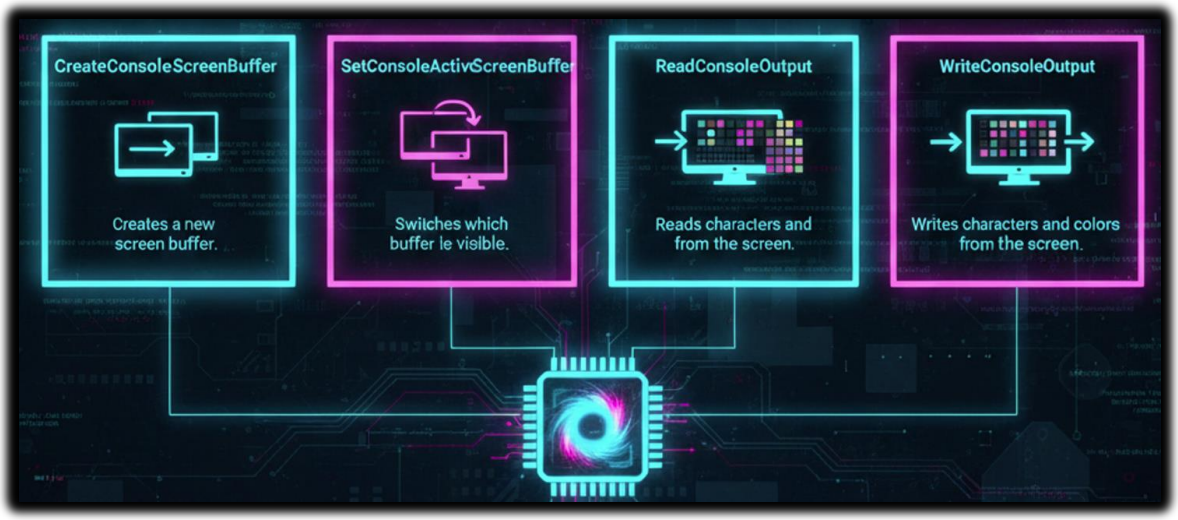


II. Screen Buffer Access

- **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

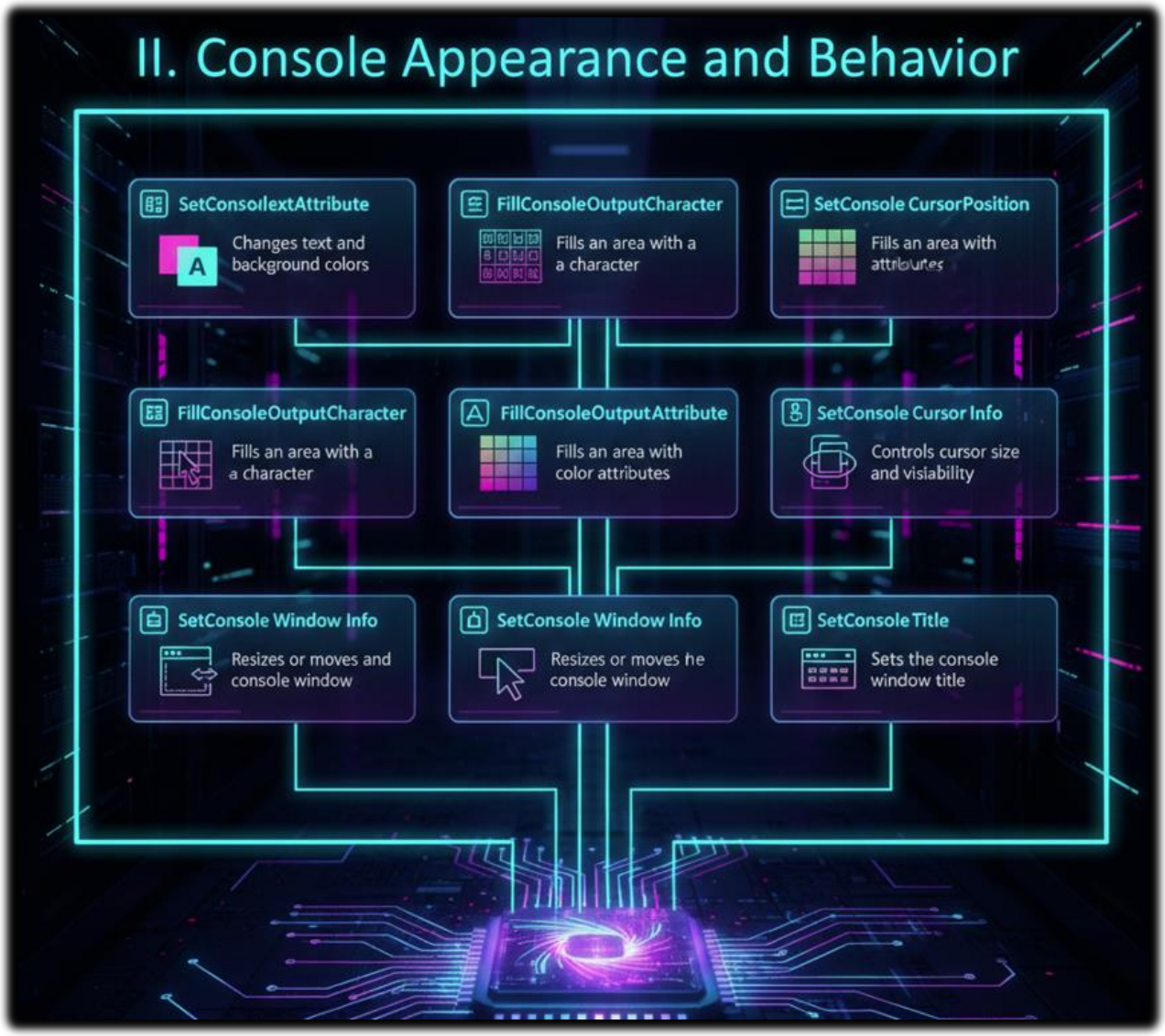
Windows Console Buffer Management

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



III. Console Appearance and Behavior

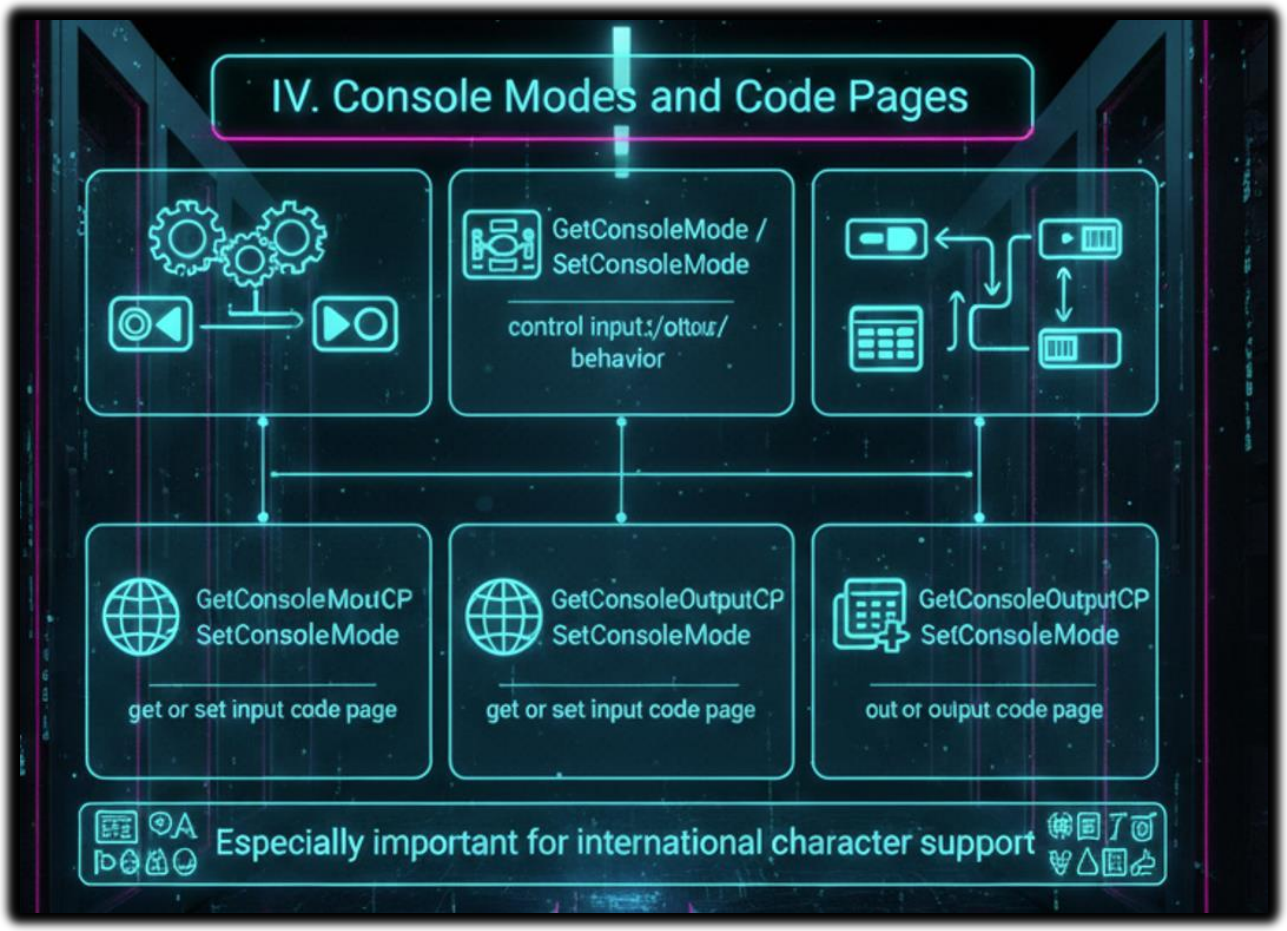
- **SetConsoleTextAttribute** – changes text and background colors
- **FillConsoleOutputCharacter** – fills an area with a character
- **FillConsoleOutputAttribute** – fills an area with color attributes
- **SetConsoleCursorPosition** – moves the cursor
- **SetConsoleCursorInfo** – controls cursor size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



IV. Console Modes and Code Pages

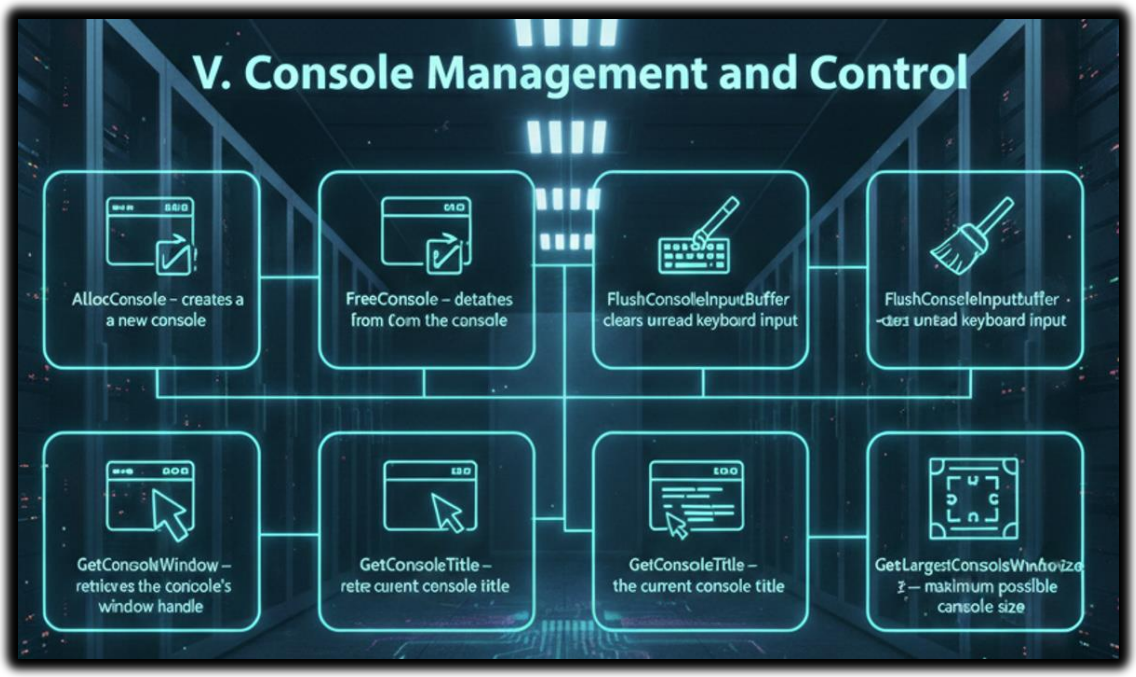
- **GetConsoleMode / SetConsoleMode** – control input/output behavior
- **GetConsoleCP / SetConsoleCP** – get or set input code page
- **GetConsoleOutputCP / SetConsoleOutputCP** – get or set output code page

These are especially important for international character support.



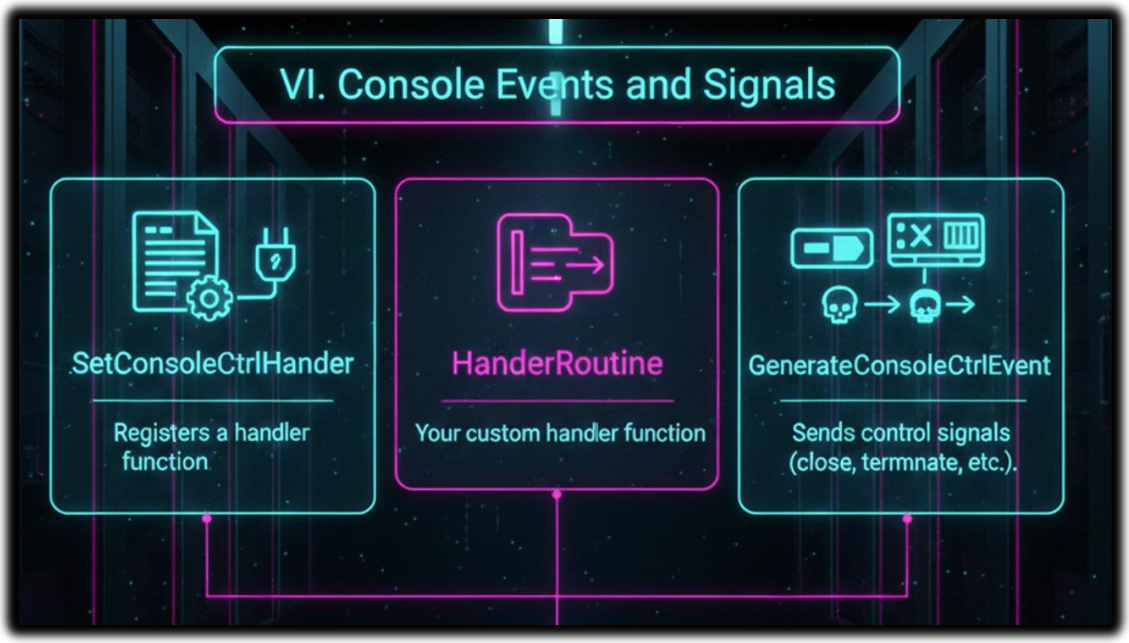
V. Console Management and Control

- **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



VI. Console Events and Signals

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



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.

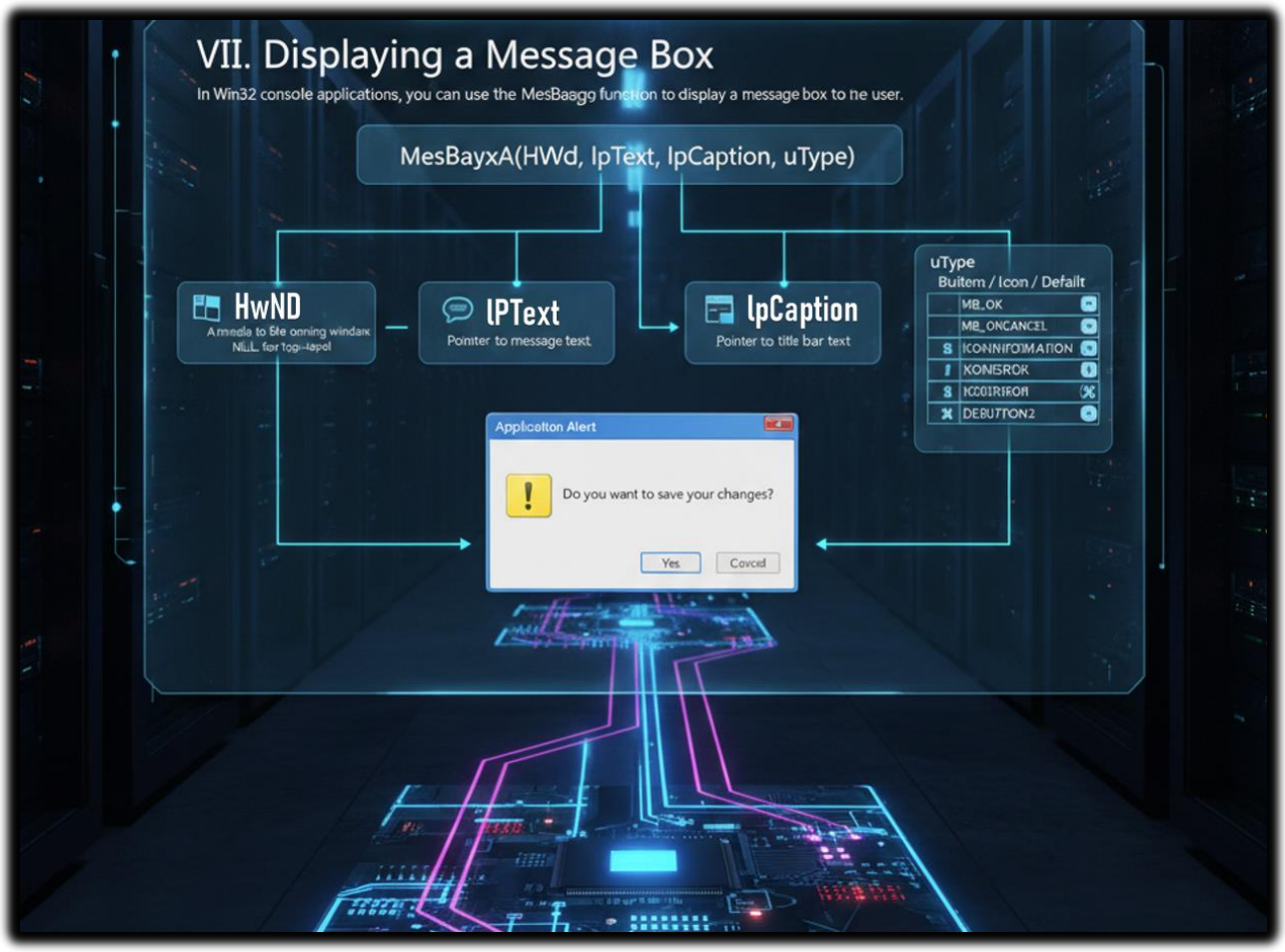
lpCaption

- 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

```
#include <windows.h>

int main() {
    ;Display a message box with an OK button
    MessageBoxW(NULL, L"This is an example", L"MessageBox", MB_OK);

    ;Display a message box with Yes and No buttons
    int result = MessageBoxW(NULL, L"Question?", L"MessageBox", MB_YESNO);

    ;Check the user's response
    if (result == IDYES) {
        ;User clicked Yes
    } else if (result == IDNO) {
        ;User clicked No
    }

    ;Display a message box with an exclamation-point icon
    MessageBoxW(NULL, L"Warning!", L"MessageBox", MB_ICONEXCLAMATION);

    return 0;
}
```

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

```c
#include <windows.h>

int main() {
    ;Display a message box with an OK button
    MessageBoxW(NULL, L"This is an example", L"MessageBox", 0);

    ;Display a message box with Yes and No buttons
    int result = MessageBoxW(NULL, L"Question?", L"MessageBox", 4);

    ;Check the user's response
    if (result == 6) {
        ;User clicked Yes
    } else if (result == 7) {
        ;User clicked No
    }

    ;Display a message box with an exclamation-point icon
    MessageBoxW(NULL, L"Warning!", L"MessageBox", 48);

    return 0;
}
```

- 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

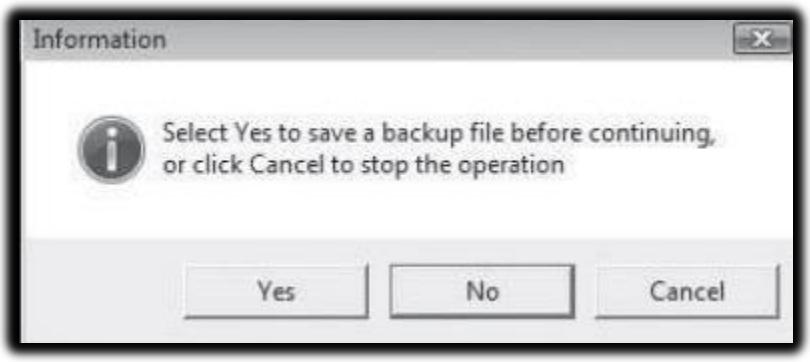
Program that demonstrates some capabilities of the MessageBoxA function.

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:

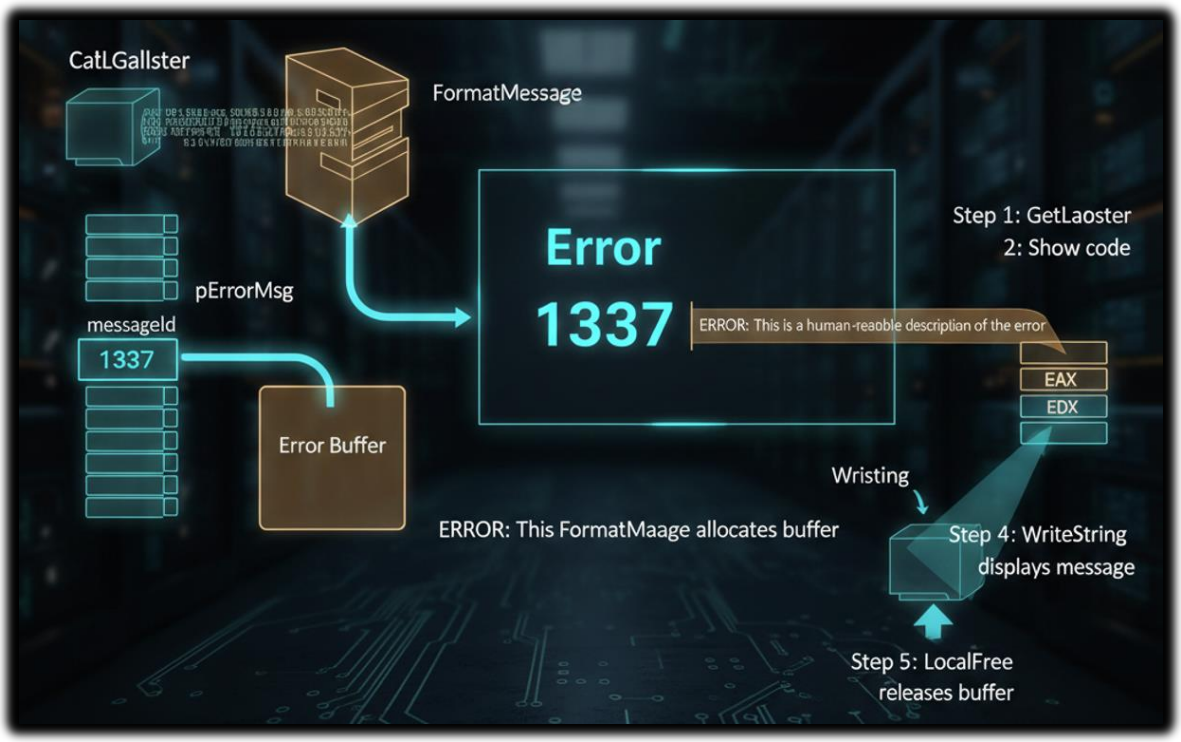
- Defines **buffer size** (BufSize) and declares necessary data variables:
  - Buffer for storing input
  - Handle for standard input (stdInHandle)
  - Variable for number of bytes read (bytesRead)
- Retrieves **standard input handle** using GetStdHandle.
- Calls **ReadConsole** to read input from the 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😂

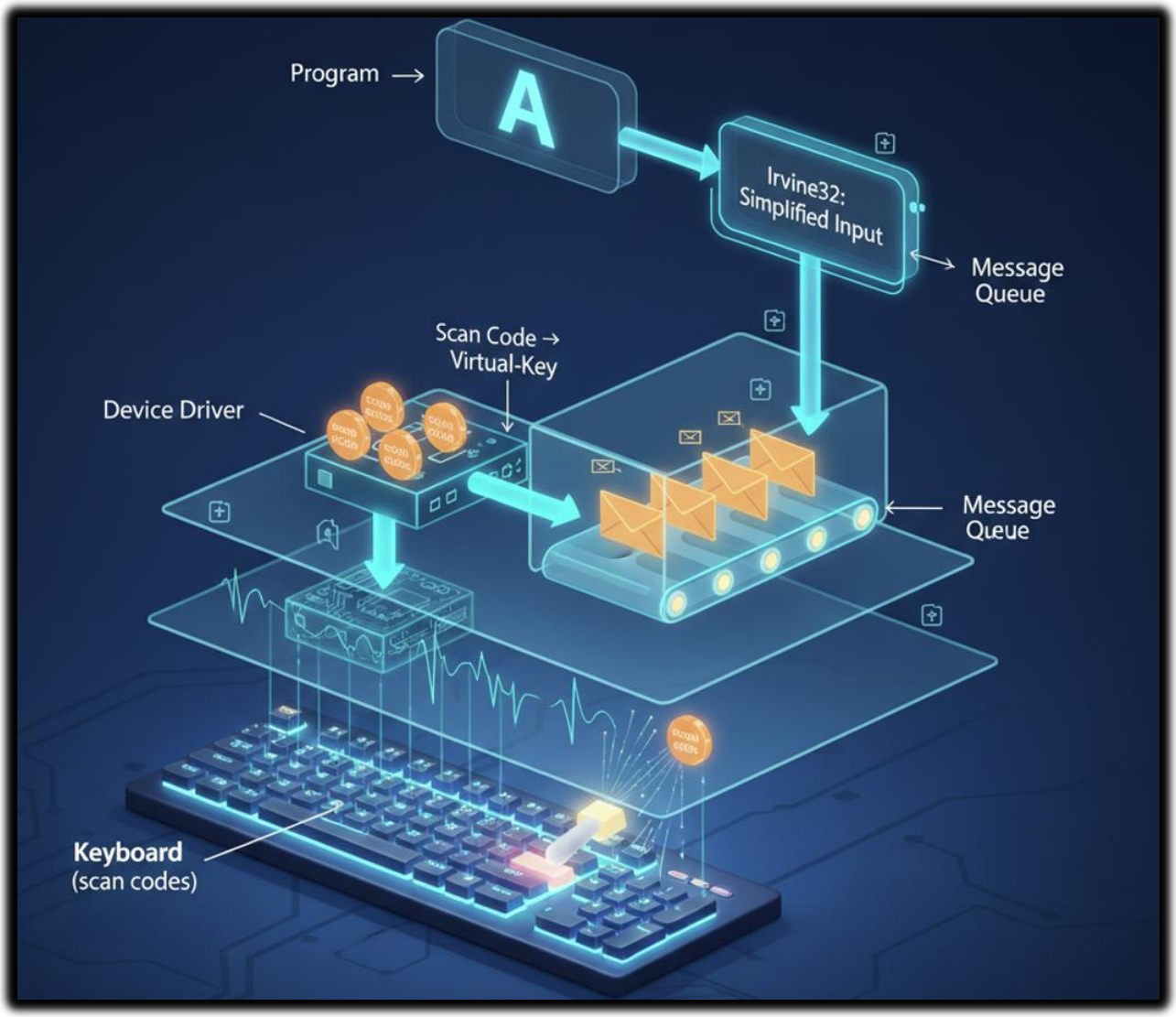


SINGLE CHARACTER INPUT

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

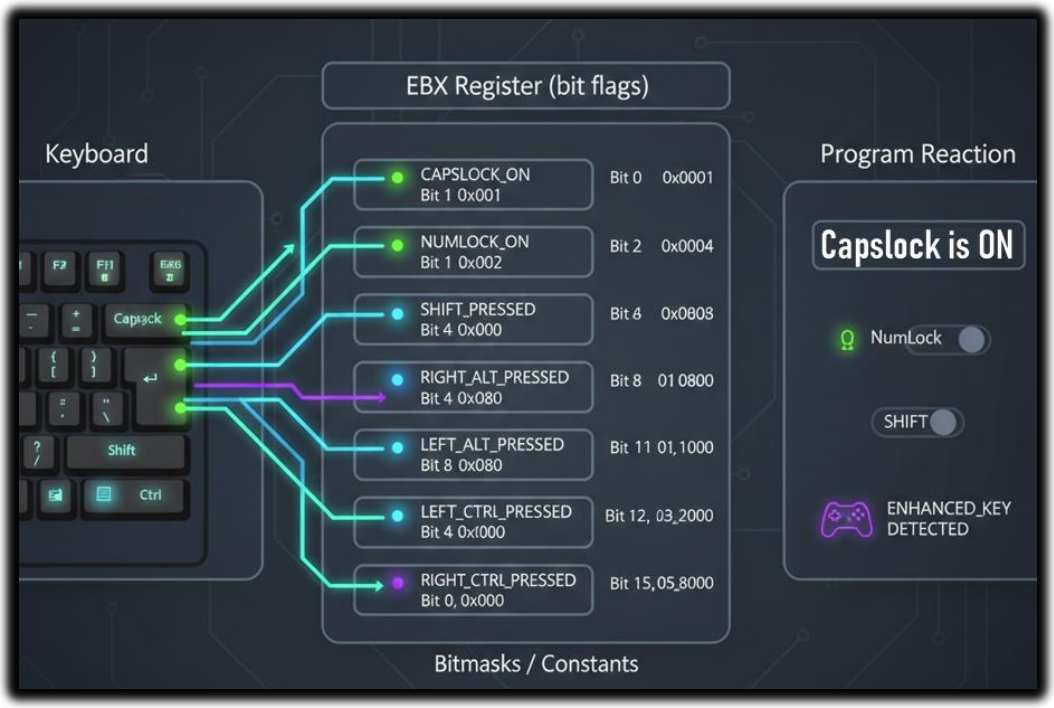
- If AL is zero, the key is a **special key** (like a function key or arrow key).
- AH contains the **scan code** for the key.
- DX contains the **virtual-key code**.
- EBX contains **control key state information**.

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.



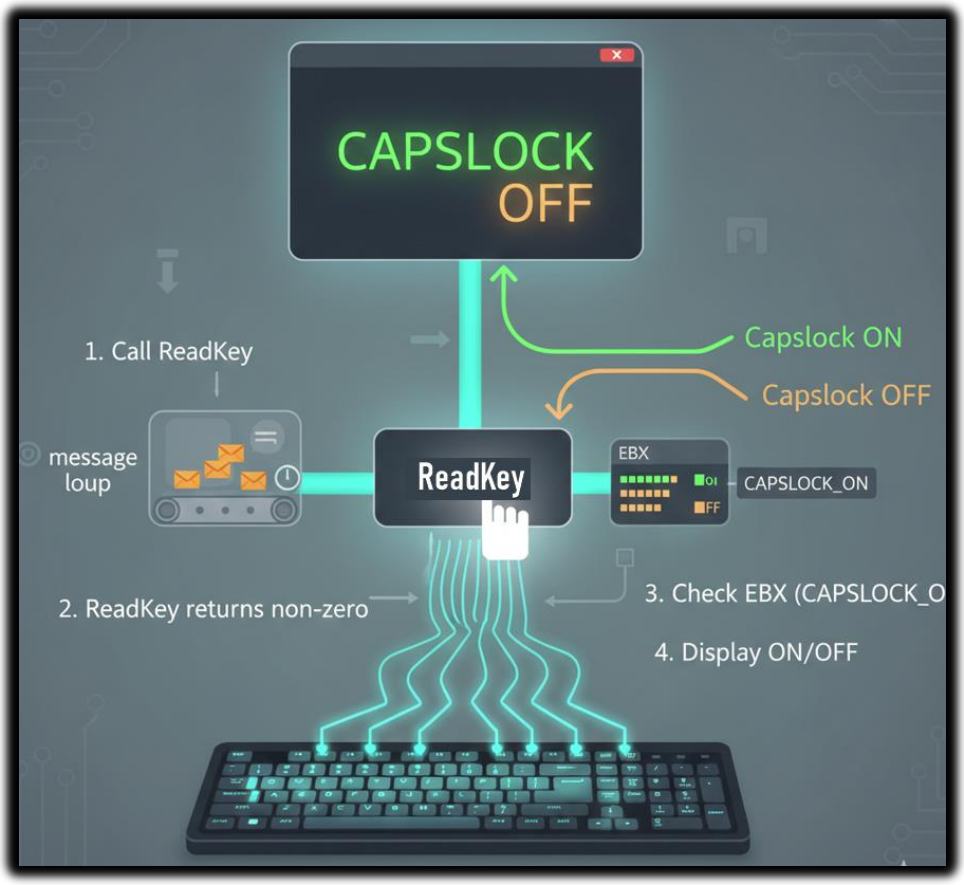
Testing Keyboard Input

Using ReadKey

You can write a simple test program:

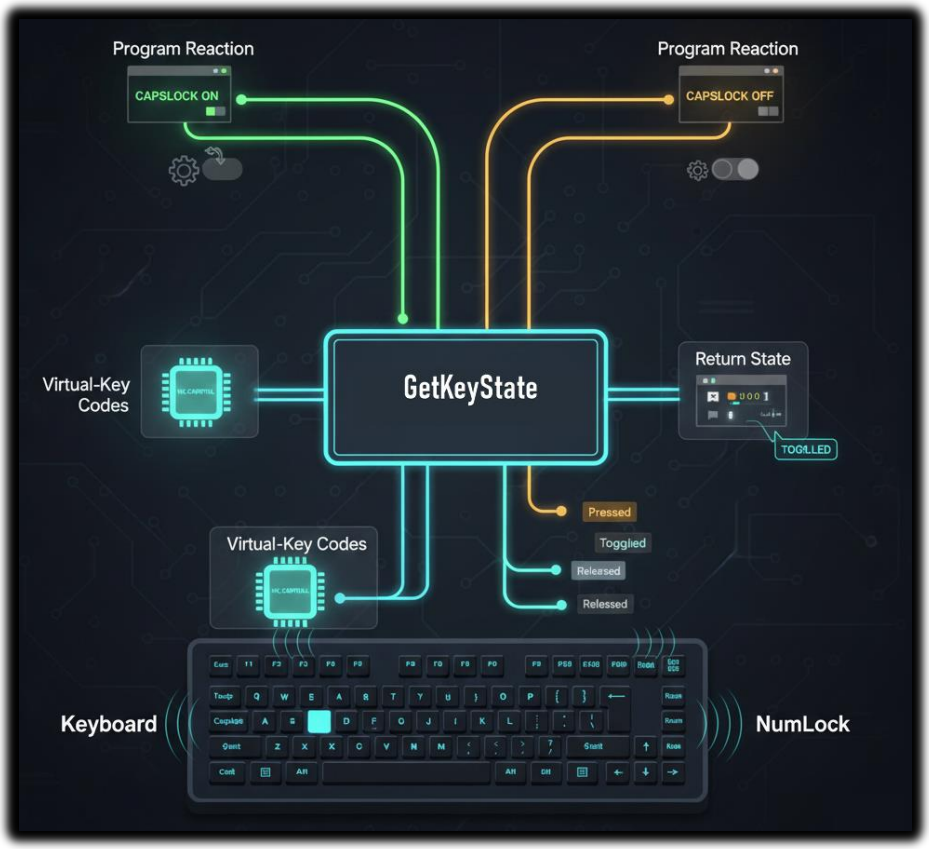
1. Call ReadKey (optionally with a delay to allow Windows to process its message loop).
2. If a key is pressed (ReadKey returns non-zero):
   - Check EBX using constants like CAPSLOCK_ON
   - Display whether CapsLock (or other keys) is ON or OFF

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	VK_NUMLOCK	0
Scroll Lock	VK_SCROLL	0
Left Shift	VK_LSHIFT	15
Right Shift	VK_RSHIFT	15
Left Ctrl	VK_LCONTROL	15
Right Ctrl	VK_RCONTROL	15
Left Menu (Alt)	VK_LMENU	15
Right Menu (Alt)	VK_RMENU	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.

```
;Testing Keyboard Input with ReadKey (TestReadkey.asm)
INCLUDE Irvine32.inc
INCLUDE Macros.inc

.code
main PROC
L1:
    mov eax, 10     ; Delay for message processing
    call Delay
    call ReadKey    ; Wait for a keypress
    jz L1

    test ebx, CAPSLOCK_ON
    jz L2
    mWrite <"CapsLock is ON", 0dh, 0ah>
    jmp L3
L2:
    mWrite <"CapsLock is OFF", 0dh, 0ah>
L3:
    exit
main ENDP
END main
```

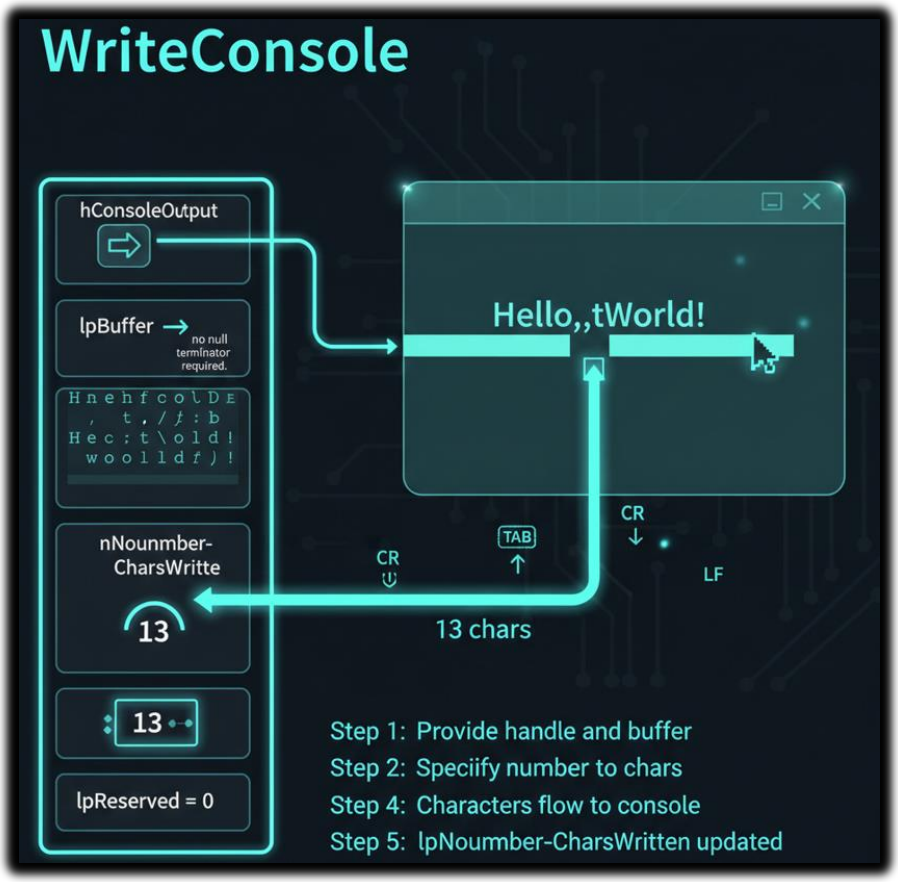
Program 2:

```
;GetKeyState
INCLUDE Irvine32.inc
INCLUDE Macros.inc

.code
main PROC
    INVOKE GetKeyState, VK_NUMLOCK
    test al, 1
    .IF !Zero?
        mWrite <"The NumLock key is ON", 0dh, 0ah>
    .ENDIF

    INVOKE GetKeyState, VK_LSHIFT
    test eax, 80000000h
    .IF !Zero?
        mWrite <"The Left Shift key is currently DOWN", 0dh, 0ah>
    .ENDIF

    exit
main ENDP
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**.



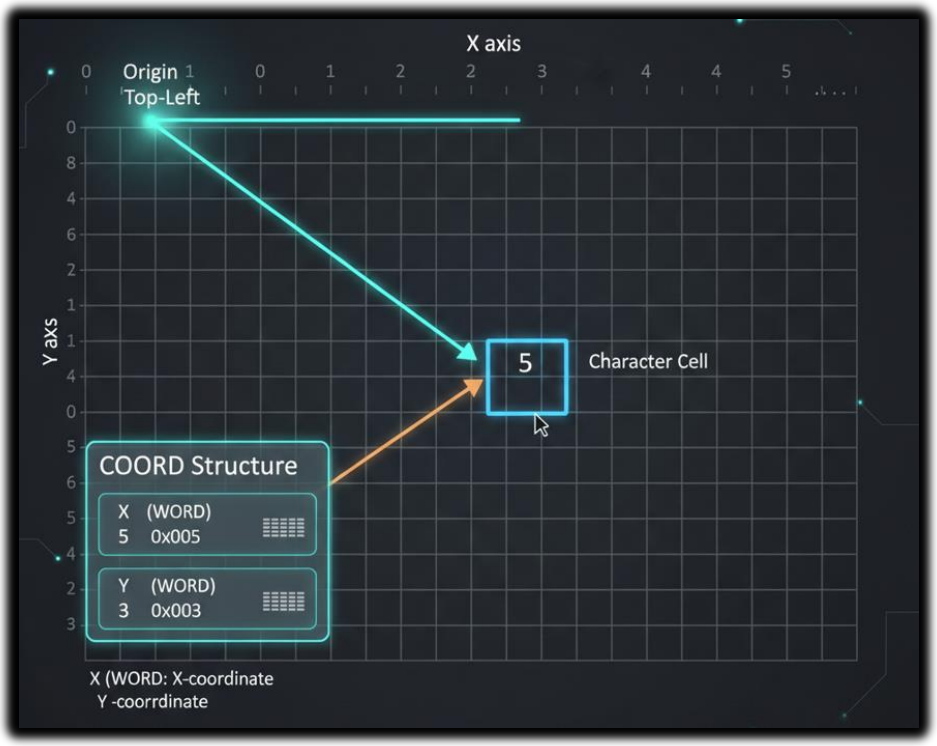
II. COORD Structure

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

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

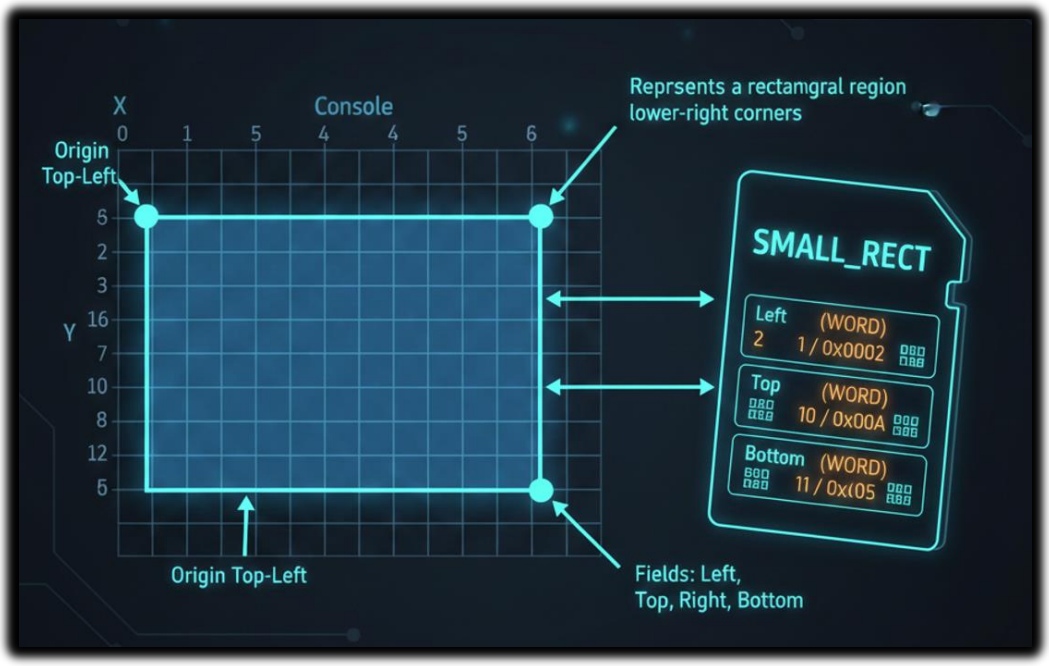
Fields:

- 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

```
; Win32 Console Example #1(Console1.asm)
; This program calls the following Win32 Console functions:
; GetStdHandle, ExitProcess, WriteConsole
INCLUDE Irvine32.inc
.data
endl EQU <0dh,0ah>       ; End of line sequence
message LABEL BYTE
    BYTE "This program is a simple demonstration of"
    BYTE "console mode output, using the GetStdHandle"
    BYTE "and WriteConsole functions.",endl
messageSize DWORD ($ - message)
consoleHandle HANDLE 0    ; Handle to standard output device
bytesWritten DWORD ?      ; Number of bytes written
.code
main PROC
    ; Get the console output handle:
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov consoleHandle,eax

    ; Write a string to the console:
    INVOKE WriteConsole,
        consoleHandle,   ; Console output handle
        ADDR message,    ; String pointer
        messageSize,     ; String length
        ADDR bytesWritten, ; Returns number of bytes written
        0                ; Not used

    ; Exit the program:
    INVOKE ExitProcess, 0
main ENDP
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 lpFilename: PTR BYTE,
337 dwDesiredAccess: DWORD,
338 dwShareMode: DWORD,
339 lpSecurityAttributes: 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:

```
.data
  filePath BYTE "myfile.txt", 0
  handle HANDLE ?

.code
  ; Create or open the file for writing
  INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0

  ; Check if the file handle is valid
  cmp eax, INVALID_HANDLE_VALUE
  je fileCreationFailed

  ; Store the file handle
  mov handle, eax

  ; Now you can write to the file using the handle

  ; Close the file when done
  INVOKE CloseHandle, handle

fileCreationFailed:
  ; Handle the case where file creation/opening failed
  ; 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:

```
.data
  filePath BYTE "mydata.txt", 0
  handle HANDLE ?
  bytesRead DWORD ?
  buffer BYTE 128 DUP(?)
.code
  ; Create or open the file for writing
  INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
  ; Check if the file handle is valid
  cmp eax, INVALID_HANDLE_VALUE
  je fileCreationFailed
  ; Store the file handle
  mov handle, eax
  ; Write some data to the file (assuming data is in the buffer)
  ; Close the file handle
  INVOKE CloseHandle, handle
  ; Reopen the file for reading
  INVOKE CreateFile, ADDR filePath, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
  ; Check if the file handle is valid
  cmp eax, INVALID_HANDLE_VALUE
  je fileOpenFailed
  ; Store the file handle
  mov handle, eax
  ; Read data from the file
  INVOKE ReadFile, handle, ADDR buffer, 128, ADDR bytesRead, 0
  ; Handle the data read from the file
  ; Close the file handle
  INVOKE CloseHandle, handle
fileOpenFailed:
fileCreationFailed:
  ; 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.

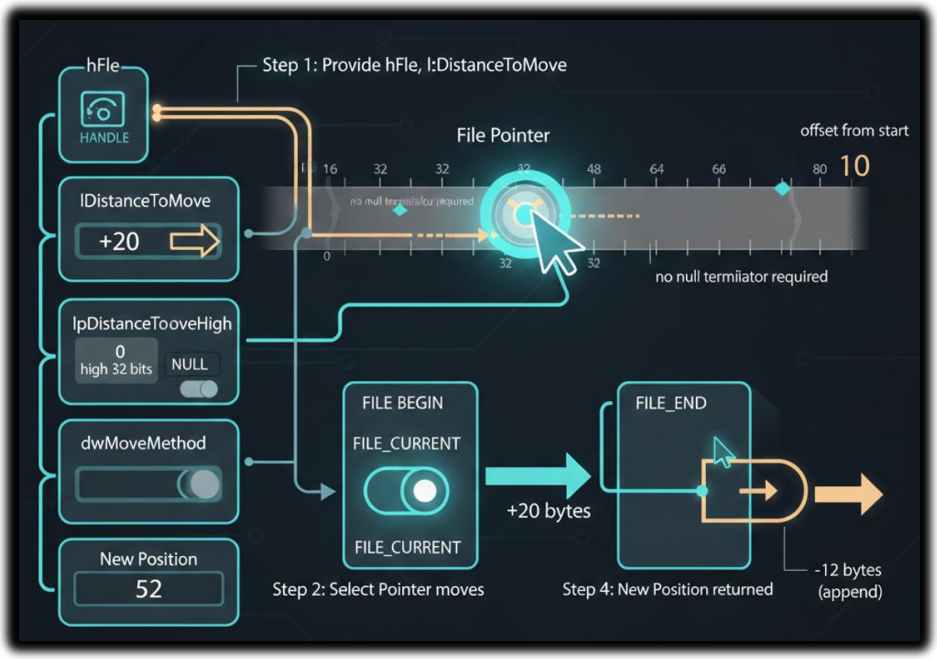
SetFilePointer Function

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.

```
;---------------------
; CreateOutputFile PROC
;
; Creates a new file and opens it in output mode.
;
; Receives: EDX points to the filename.
;
; Returns: If the file was created successfully, EAX
; contains a valid file handle. Otherwise, EAX
; equals INVALID_HANDLE_VALUE.
;
;---------------------
INVOKE CreateFile,
    edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
ret
CreateOutputFile ENDP

;---------------------
; OpenFile PROC
;
; Opens a new text file and opens for input.
;
; Receives: EDX points to the filename.
;
; Returns: If the file was opened successfully, EAX
; contains a valid file handle. Otherwise, EAX equals
; INVALID_HANDLE_VALUE.
;
;---------------------
INVOKE CreateFile,
    edx, GENERIC_READ, DO_NOT_SHARE, NULL,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
ret
OpenFile ENDP

;---------------------
; WriteToFile PROC
;
; Writes a buffer to an output file.
;
; Receives: EAX = file handle, EDX = buffer offset,
; ECX = number of bytes to write
;
; Returns: EAX = number of bytes written to the file.
; If the value returned in EAX is less than the
; argument passed in ECX, an error likely occurred.
;
;---------------------
.data
WriteToFile_1 DWORD ?
; number of bytes written
.code
INVOKE WriteFile,
    eax,        ; file handle
    edx,        ; buffer pointer
    ecx,        ; number of bytes to write
    ADDR WriteToFile_1, ; number of bytes written
    0           ; overlapped execution flag
mov eax, WriteToFile_1 ; return value
ret
WriteToFile ENDP
```

```
;---------------------
; ReadFromFile PROC
;
; Reads an input file into a buffer.
;
; Receives: EAX = file handle, EDX = buffer offset,
; ECX = number of bytes to read
;
; Returns: If CF = 0, EAX = number of bytes read; if
; CF = 1, EAX contains the system error code returned
; by the GetLastError Win32 API function.
;
;---------------------
.data
ReadFromFile_1 DWORD ?
; number of bytes read
.code
INVOKE ReadFile,
    eax,        ; file handle
    edx,        ; buffer pointer
    ecx,        ; max bytes to read
    ADDR ReadFromFile_1, ; number of bytes read
    0           ; overlapped execution flag
mov eax, ReadFromFile_1
ret
ReadFromFile ENDP
```

```
;---------------------
; CloseFile PROC
;
; Closes a file using its handle as an identifier.
;
; Receives: EAX = file handle
;
; Returns: EAX = nonzero if the file is successfully closed.
;
;---------------------
INVOKE CloseHandle, eax
ret
CloseFile ENDP
```

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

```
; Creating a File (CreateFile.asm)
INCLUDE Irvine32.inc
BUFFER_SIZE = 501

.data
buffer BYTE BUFFER_SIZE DUP(?)
filename BYTE "output.txt",0
fileHandle HANDLE ?
stringLength DWORD ?
bytesWritten DWORD ?
str1 BYTE "Cannot create file",0dh,0ah,0
str2 BYTE "Bytes written to file [output.txt]:",0
str3 BYTE "Enter up to 500 characters and press [Enter]: ",0dh,0ah,0

.code
main PROC
    ; Create a new text file.
    mov edx, OFFSET filename    ; Load the address of the filename.
    call CreateOutputFile       ; Call the CreateOutputFile procedure.
    mov fileHandle, eax         ; Store the file handle in fileHandle.

    ; Check for errors.
    cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
    jne file_ok                 ; If not equal, jump to file_ok.

    ; If there's an error, display the error message and exit.
    mov edx, OFFSET str1        ; Load the address of the error message.
    call WriteString            ; Call WriteString to display the error message.
    jmp quit                    ; Jump to quit to exit.
file_ok:
    ; Ask the user to input a string.
    mov edx, OFFSET str3        ; Load the address of the input prompt.
    call WriteString            ; Call WriteString to display the input prompt.

    mov ecx, BUFFER_SIZE        ; Load the maximum buffer size.

    ; Input a string.
    mov edx, OFFSET buffer      ; Load the address of the buffer.
    call ReadString             ; Call ReadString to get user input.
    mov stringLength, eax       ; Store the length of the entered string.
    ; Write the buffer to the output file.
    mov eax, fileHandle         ; Load the file handle.
    mov edx, OFFSET buffer      ; Load the address of the buffer.
    mov ecx, stringLength      ; Load the length of the string.
    call WriteToFile            ; Call WriteToFile to write to the file.
    mov bytesWritten, eax       ; Store the number of bytes written.
    ; Close the file.
    call CloseFile              ; Call CloseFile to close the file.
    ; Display the return value.
    mov edx, OFFSET str2        ; Load the address of the output message.
    call WriteString            ; Call WriteString to display the message.
    mov eax, bytesWritten       ; Load the number of bytes written.
    call WriteDec               ; Call WriteDec to display the value.
    call Crlf                   ; Call Crlf to add a new line.
quit:
    exit
main ENDP
END main
```

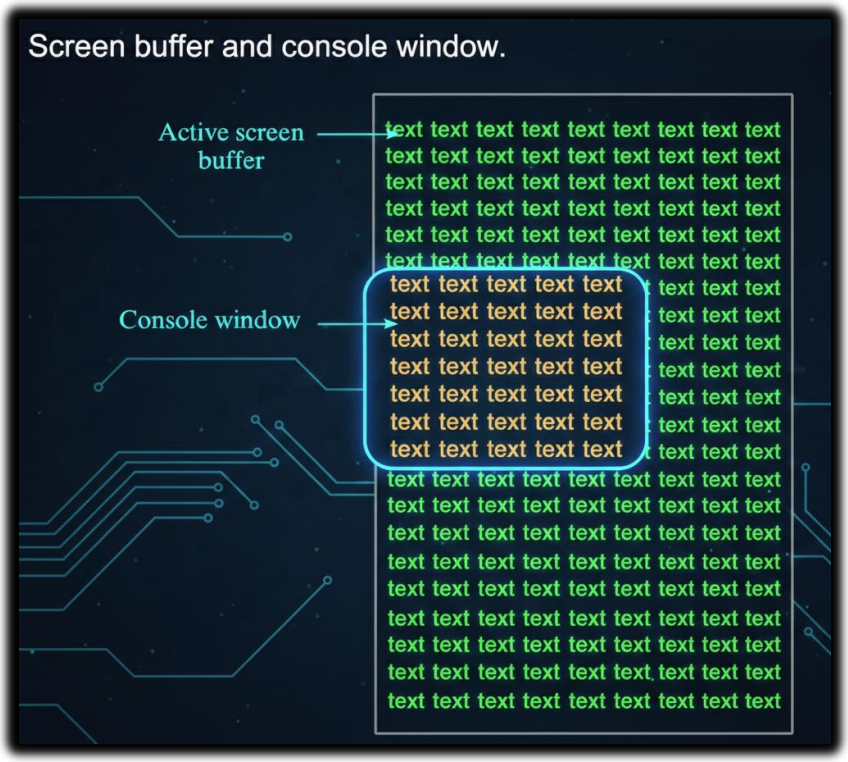
That's the second program.

Let's try another program:

```
; Reading a File (ReadFile.asm)
; Opens, reads, and displays a text file using
; procedures from Irvine32.lib.
INCLUDE Irvine32.inc
INCLUDE macros.inc
BUFFER_SIZE = 5000

.data
buffer BYTE BUFFER_SIZE DUP(?)
filename BYTE 80 DUP(0)
fileHandle HANDLE ?

.code
main PROC
    ; Let the user input a filename.
    mWrite "Enter an input filename: "  ; Display the input prompt.
    mov edx, OFFSET filename  ; Load the address of the filename.
    mov ecx, SIZEOF filename  ; Load the size of the filename.
    call ReadString  ; Call ReadString to get user input.

    ; Open the file for input.
    mov edx, OFFSET filename  ; Load the address of the filename.
    call OpenInputFile  ; Call OpenInputFile to open the file.
    mov fileHandle, eax  ; Store the file handle in fileHandle.

    ; Check for errors when opening the file.
    cmp eax, INVALID_HANDLE_VALUE  ; Compare the result to INVALID_HANDLE_VALUE.
    jne file_ok  ; If not equal, jump to file_ok.
    ; If there's an error, display the error message and exit.
    mWrite <"Cannot open file", 0dh, 0ah>  ; Display the error message.
    jmp quit  ; Jump to quit to exit.

file_ok:
    ; Read the file into a buffer.
    mov edx, OFFSET buffer  ; Load the address of the buffer.
    mov ecx, BUFFER_SIZE  ; Load the buffer size.
    call ReadFromFile  ; Call ReadFromFile to read the file.

    jnc check_buffer_size  ; If no error, jump to check_buffer_size.

    ; If there's an error, display an error message.
    mWrite "Error reading file. "  ; Display the error message.
    call WriteWindowsMsg  ; Call WriteWindowsMsg to display the Windows error message.
    jmp close_file  ; Jump to close_file to close the file.

check_buffer_size:
    cmp eax, BUFFER_SIZE  ; Compare the result to BUFFER_SIZE.
    jb buf_size_ok  ; If less, jump to buf_size_ok.

    ; If the buffer is too small for the file, display an error message and exit.
    mWrite <"Error: Buffer too small for the file", 0dh, 0ah>  ; Display the error message.
    jmp quit  ; Jump to quit to exit.

buf_size_ok:
    mov buffer[eax], 0  ; Insert a null terminator.
    mWrite "File size: "  ; Display a message about the file size.
    call WriteDec  ; Call WriteDec to display the file size.
    call Crlf  ; Call Crlf to add a new line.

    ; Display the buffer.
    mWrite <"Buffer:", 0dh, 0ah, 0dh, 0ah>  ; Display the buffer message.
    mov edx, OFFSET buffer  ; Load the address of the buffer.
    call WriteString  ; Call WriteString to display the buffer.
    call Crlf  ; Call Crlf to add a new line.

close_file:
    mov eax, fileHandle  ; Load the file handle.
    call CloseFile  ; Call CloseFile to close the file.

quit:
    exit  ; Exit the program.
main ENDP

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
```

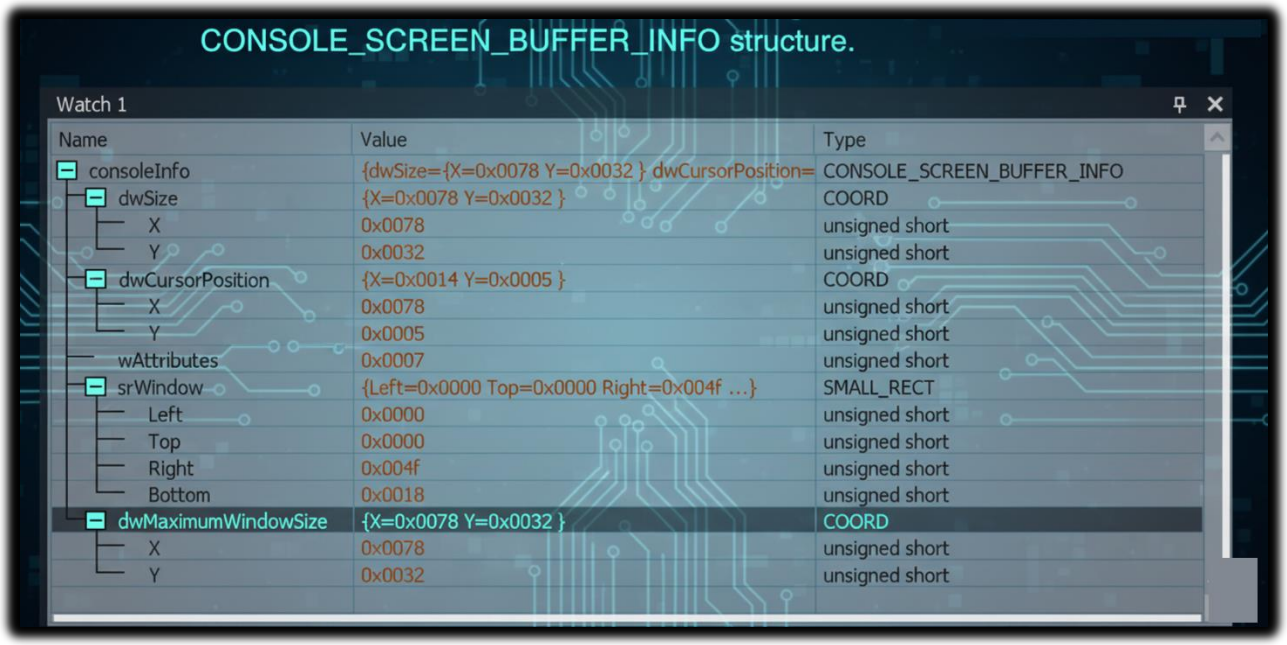
This code shows how to use the **GetConsoleScreenBufferInfo function** to obtain information about the console window, including screen buffer size, cursor position, and other details. The retrieved information is stored in the consoleInfo structure.

```
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
```



Scroll.asm program:

```
INCLUDE Irvine32.inc

.data
message BYTE ": This line of text was written to the screen buffer",0dh,0ah
messageSize DWORD ($-message)
outHandle HANDLE 0 ; Standard output handle
bytesWritten DWORD ?
lineNum DWORD 0
windowRect SMALL_RECT <0,0,60,11> ; Left, top, right, bottom

.code
main PROC
    ; Get the standard output handle
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov outHandle, eax

    .REPEAT
        ; Display the line number
        mov eax, lineNum
        call WriteDec

        ; Write the message to the console
        INVOKE WriteConsole, outHandle, ADDR message, messageSize, ADDR bytesWritten, 0

        ; Increment the line number
        inc lineNum

    .UNTIL lineNum > 50

    ; Resize and reposition the console window
    INVOKE SetConsoleWindowInfo, outHandle, TRUE, ADDR windowRect

    ; Wait for a key press
    call ReadChar

    ; Clear the screen buffer
    call Clrscr

    ; Wait for a second key press
    call ReadChar

    ; Exit the program
    INVOKE ExitProcess, 0

main ENDP

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:

```
INCLUDE Irvine32.inc

.data
consoleInfo CONSOLE_CURSOR_INFO <25, 1> ; Default cursor info
outHandle HANDLE 0
coord COORD <10, 10> ; New cursor position

.code
main PROC
    ; Get the standard output handle
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov outHandle, eax

    ; Get the current cursor information
    INVOKE GetConsoleCursorInfo, outHandle, ADDR consoleInfo

    ; Display the current cursor size and visibility
    mov eax, consoleInfo.dwSize
    call WriteDec
    call WriteString, ADDR " - Cursor Size, Visible: "
    mov eax, consoleInfo.bVisible
    call WriteDec
    call Crlf

    ; Set a new cursor size and visibility
    mov consoleInfo.dwSize, 50
    mov consoleInfo.bVisible, TRUE
    INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo

    ; Move the cursor to a new position
    INVOKE SetConsoleCursorPosition, outHandle, ADDR coord

    ; Display a message at the new cursor position
    call WriteString, ADDR "New Cursor Position"

    ; Wait for a key press
    call ReadChar

    ; Reset cursor info to the default values
    mov consoleInfo.dwSize, 25
    mov consoleInfo.bVisible, TRUE
    INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo

    ; Move the cursor back to the original position
    mov coord.X, 0
    mov coord.Y, 0
    INVOKE SetConsoleCursorPosition, outHandle, ADDR coord

    ; Display a message at the original cursor position
    call WriteString, ADDR "Original Cursor Position"

    ; Wait for a key press to exit
    call ReadChar

    INVOKE ExitProcess, 0
main ENDP

END main
```

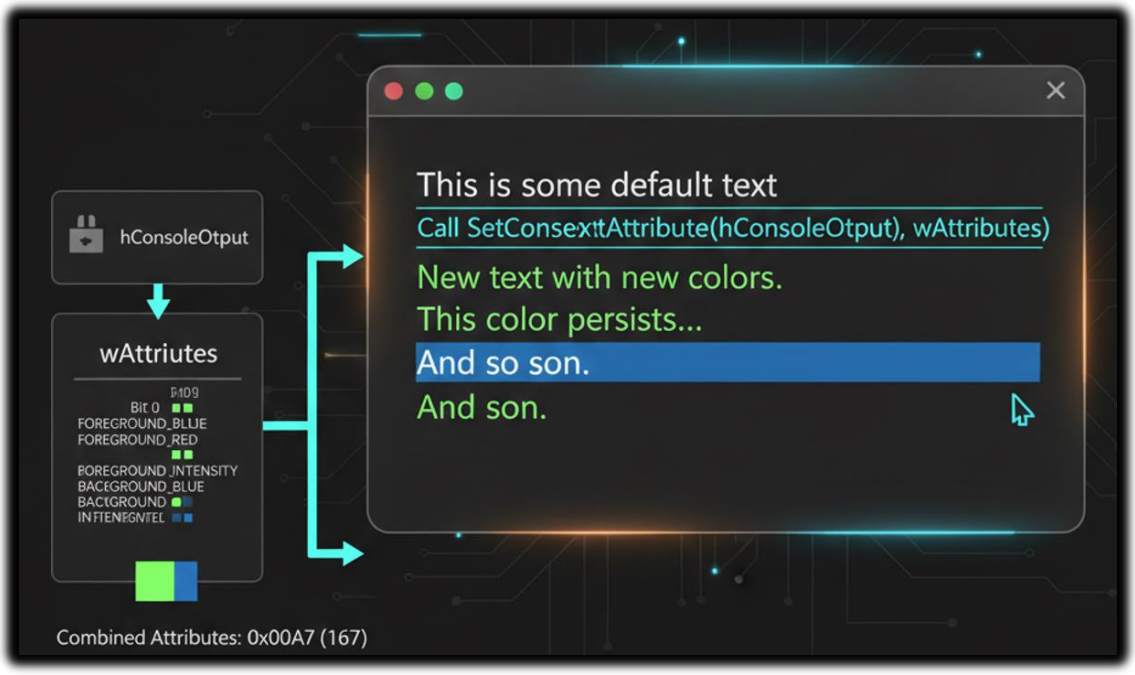
Summary of that program:

- **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

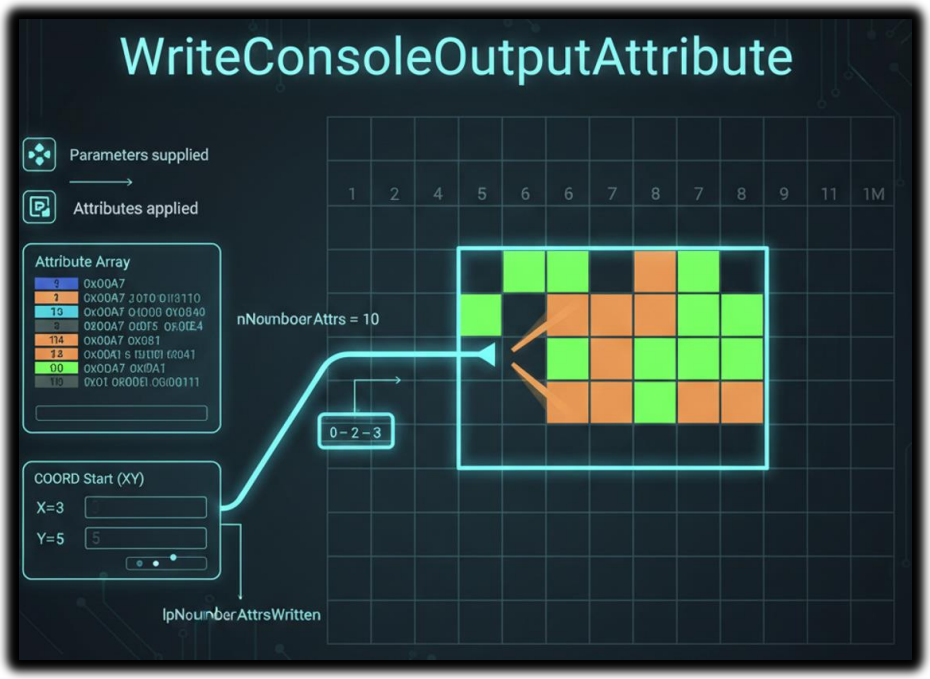
SetConsoleTextAttribute

- 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.

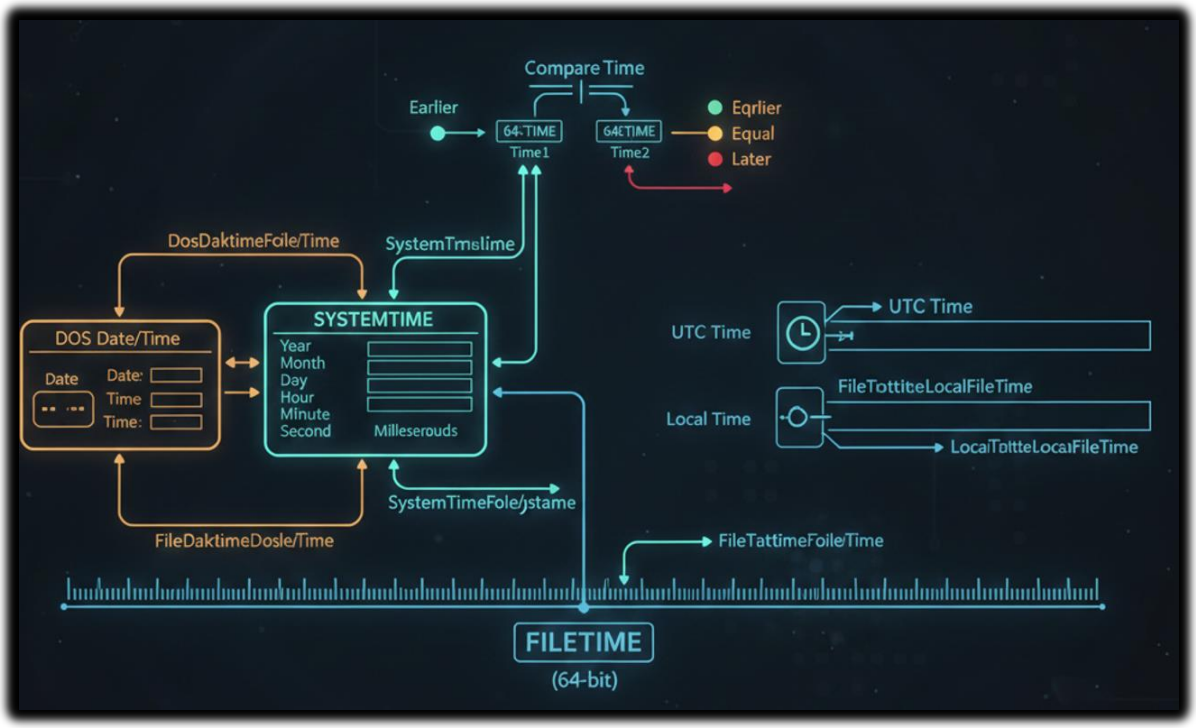
Customization: Modify the attributes array to change foreground/background colors or color patterns.

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

TIME, WINAPI AND ASSEMBLY

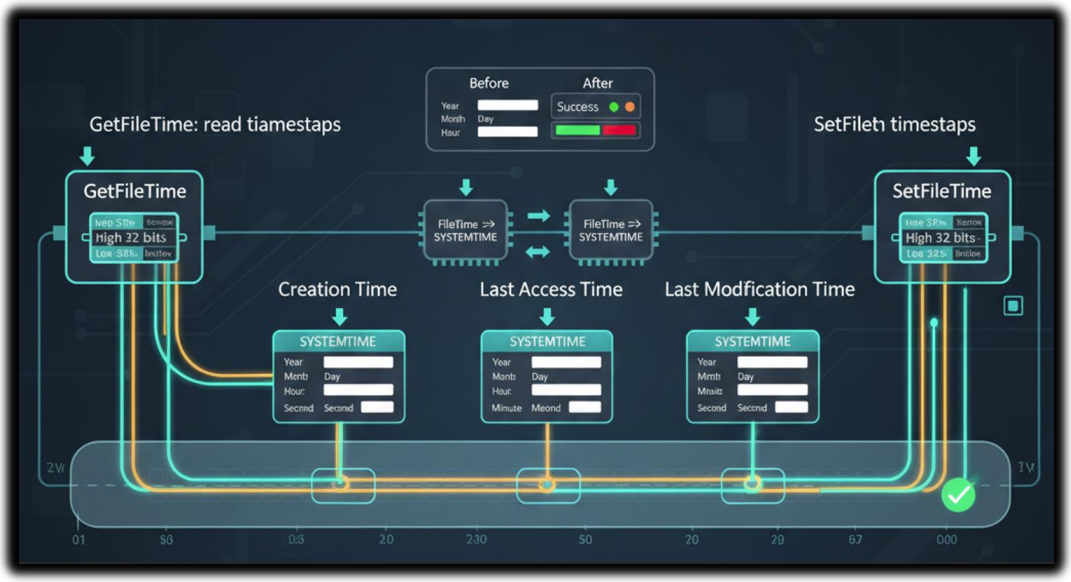
I. File Time Functions

- **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.



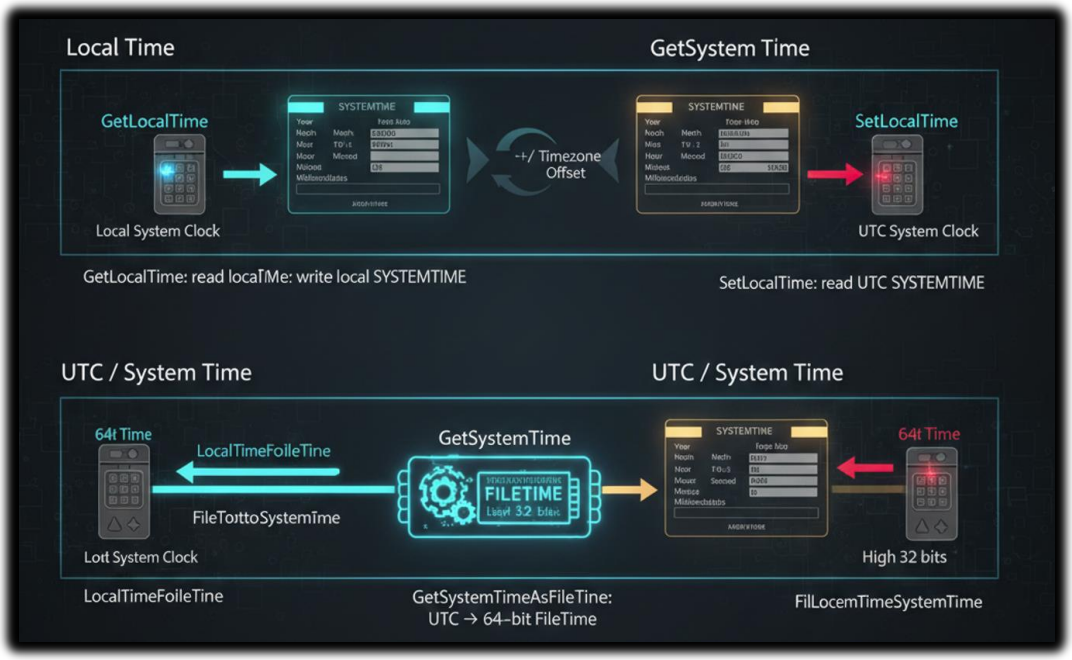
II. File Timestamp Functions

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



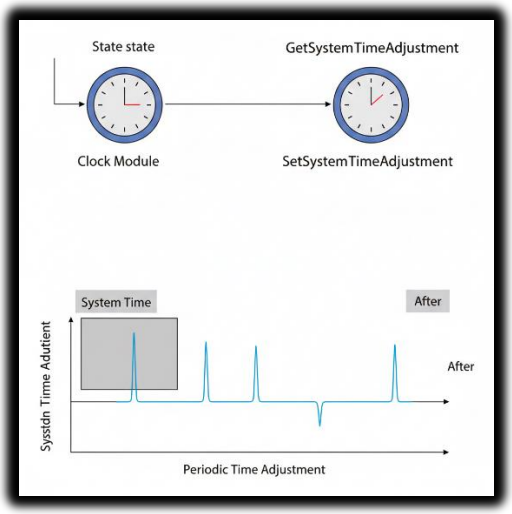
III. System Time Functions

- **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.



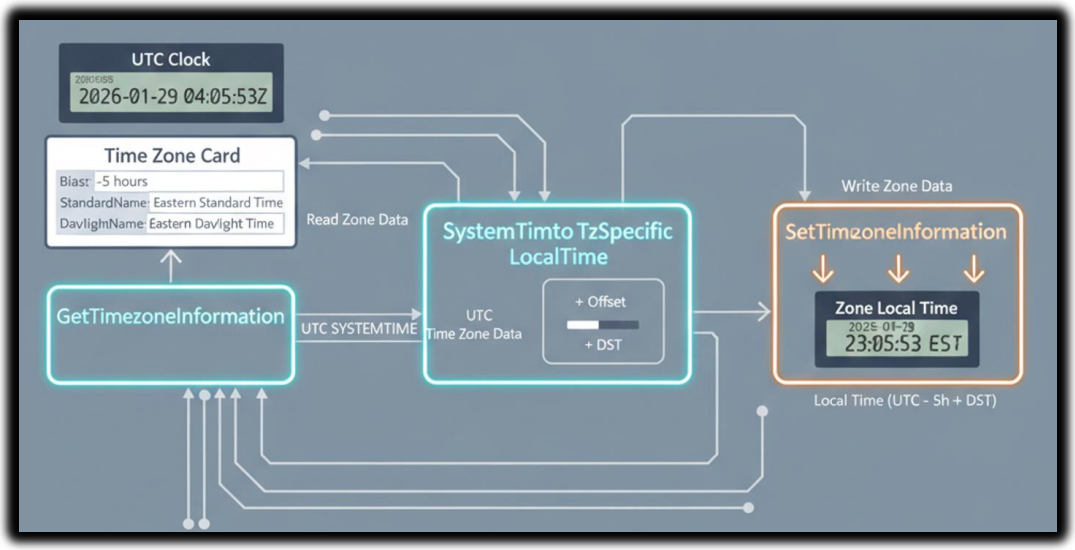
IV. Time Adjustment Functions

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



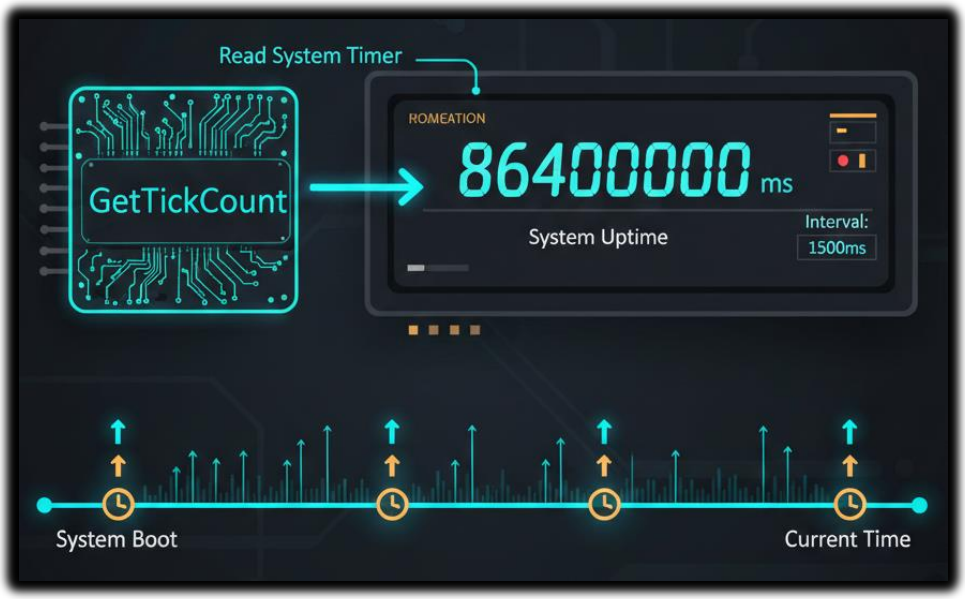
V. Time Zone Functions

- **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
```

Here's an explanation of the program in paragraph format:

The program starts by defining a structure called SYSTEMTIME, which is used to hold information about date and time. It includes fields like year, month, day of the week, day of the month, hours, minutes, seconds, and milliseconds. This structure is essential for working with date and time-related functions in the Windows API.

The program utilizes the GetLocalTime function, a Windows API function that retrieves the current local date and time according to the system's clock. It takes a single parameter, a pointer to a SYSTEMTIME structure, where it stores the current date and time values. This function is essential for obtaining the current time for further processing.

On the other hand, the SetLocalTime function is another Windows API function used to set the system's local date and time. It also takes a SYSTEMTIME structure as a parameter, but this time, it contains the desired date and time values. By calling this function, you can modify the system's date and time settings programmatically.

The program also incorporates the GetTickCount function, which is used to measure the number of milliseconds that have passed since the system started. This function doesn't require any parameters and returns the elapsed time in the EAX register. It's particularly useful for timing operations and determining the duration of processes.

There's a custom procedure in the program called DisplayTime, which serves to display the various components of a SYSTEMTIME structure, such as the year, month, day, hour, minute, second, and day of the week. This procedure uses different write functions to display these components on the console.

The program's main procedure is the entry point. It first calls GetLocalTime to retrieve and display the current local time. After that, it sets the local time to a specific value, allowing you to modify the date and time as needed. The program then calls GetLocalTime again to retrieve and display the updated local time. To simulate a time-consuming operation, the program performs calculations in a loop. Before and after this loop, it uses GetTickCount to measure the elapsed time and displays it.

In summary, this program showcases how to work with date and time in a Windows environment using the SYSTEMTIME structure and relevant Windows API functions. It demonstrates retrieving and displaying the current local time, setting the local time, and measuring elapsed time using GetTickCount. You can adjust the date and time values as necessary for your specific requirements.

==========

The **Sleep function** is a part of the Win32 API that allows programs to introduce pauses or delays. This can be useful for controlling the timing of various operations in a program.

The function takes a parameter that specifies the length of time to sleep, and then it puts the processor into a low-power state until the specified time has elapsed.

The **GetDateTime procedure** is a convenient utility to retrieve date and time information. It returns the number of 100-nanosecond intervals that have elapsed since January 1, 1601.

The procedure generally follows these steps:

It calls a function like GetLocalTime, which populates a SYSTEMTIME structure with the current date and time information.

It converts this SYSTEMTIME structure to a FILETIME structure using the SystemTimeToFileTime function. Then, it copies the resulting FILETIME structure to a 64-bit quadword.

The FILETIME structure is used to divide a 64-bit quadword into two doublewords.

The GetDateTime procedure, which receives a pointer to a 64-bit quadword variable as an argument, is responsible for storing the current date and time in the specified variable in the FILETIME format used by Win32.

In simpler terms, the **Sleep functio**n allows programs to pause for a specified period of time, while the **GetDateTime procedure** allows programs to retrieve the current date and time.

Both functions are useful for controlling the timing of various operations in Win32 applications.

```
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
```

The Sleep function allows you to introduce time delays, and the GetDateTime procedure retrieves the current date and time and stores it in a 64-bit quadword. This code can be integrated into your assembly programs as needed.

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

The eax register is used to specify the length of time to sleep. The call sleep instruction then calls the sleep function. Once the sleep function has returned, the program will continue execution.

It is important to note that the sleep function can be interrupted by certain events, such as a timer interrupt. If this happens, the program will resume execution immediately, even if the specified sleep time has not yet elapsed.

Here are some additional things to keep in mind when using the sleep function in MASM:

The sleep function is typically implemented as a system call.

This means that it must be executed in a privileged mode, such as kernel mode. The sleep function can be blocked by other processes.

This means that if another process is holding the kernel lock, the sleep function will not be able to execute until the other process releases the lock.

The sleep function can cause the processor to enter a low-power state. This can save power, but it can also delay the execution of other programs.

Overall, the sleep function is a powerful tool that can be used to control the execution of a program. However, it is important to be aware of the limitations of the function and to use it carefully.

CALLING 64-BIT WINAPI FUNCTION IN MASM

To call a 64-bit Windows API function in MASM, you must follow these steps:

Reserve at least 32 bytes of shadow space by subtracting 32 from the stack pointer (RSP) register.

Make sure RSP is aligned on a 16-byte address boundary.

Place the first four arguments in the following registers, from left to right:

RCX, RDX, R8, and R9.

Push additional arguments on the runtime stack.

Call the function using the call instruction.

Restore RSP to its original value by adding the same value to it that was subtracted before the function call.

The system function will return a 64-bit integer value in RAX. Here is an example of how to call the 64-bit WriteConsoleA function:

```
.data
    STD_OUTPUT_HANDLE EQU -11
    consoleOutHandle QWORD ?

.code
    sub rsp, 40 ; reserve shadow space & align RSP
    mov rcx, STD_OUTPUT_HANDLE
    mov rdx, message ; pointer to the string
    mov r8, message_length ; length of the string
    lea r9, bytesWritten
    mov qword ptr [rsp + 4 * SIZEOF QWORD], 0 ; (always zero)
    call WriteConsoleA
    add rsp, 40 ; restore RSP
```

The WriteConsoleA function takes five arguments:

- The console handle.
- A pointer to the string to write.
- The length of the string to write.
- A pointer to the variable that will store the number of bytes written.
- A dummy zero parameter.
- The bytesWritten variable is used to store the number of bytes that were actually written.

Once you have called the WriteConsoleA function, you can check the value of the bytesWritten variable to see how many bytes were written.

To write a graphical Windows application, you need to:

Include the necessary libraries and header files. This includes the kernel32.lib and user32.lib libraries, as well as a header file that contains structures, constants, and function prototypes used by the program.

Create a main window. This is done using the CreateWindowEx() function. Display the main window. This is done using the ShowWindow() function. Respond to mouse events. This is done by handling the WM_MOUSEMOVE and WM_LBUTTONDOWN messages. Display message boxes. This is done using the MessageBox() function.

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


```
include "graphwin.inc"

.data
className db "WinApp", 0
instance HANDLE
window HANDLE

.code
start:
  ; Register the window class
  invoke RegClassEx, addr className

  ; Create the main window
  invoke CreateWindowEx, 0, addr className, addr className, WS_OVERLAPPEDWINDOW,
0, 0, CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, 0, instance, 0
  mov window, eax

  ; Show the main window
  invoke ShowWindow, window, SW_SHOW

  ; Message loop
messageLoop:
  invoke GetMessage, addr msg, 0, 0, 0
  cmp eax, -1
  je end

  ; Translate and dispatch the message
  invoke TranslateMessage, addr msg
```

```
  invoke DispatchMessage, addr msg

  jmp messageLoop

end:
  invoke ExitProcess, 0
```

This program creates a simple window with the title "WinApp". The window fills the screen and is centered on the desktop.

The program also handles mouse events and displays a message box when the user clicks the left mouse button.

To build and run the program, you can use the following steps:

Create a new assembly language project in Visual Studio. Add the following files to the project: WinApp.asm GraphWin.inc Add the kernel32.lib and user32.lib libraries to the project. Set the subsystem to Windows (/SUBSYSTEM:WINDOWS).

Build and run the program. When you run the program, you will see a simple window with the title "WinApp". If you click the left mouse button, the program will display a message box.

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

```
RECT STRUCT
    left DWORD ?
    top DWORD ?
    right DWORD ?
    bottom DWORD ?
RECT ENDS
.data
    rect1 RECT <10, 20, 100, 150> ; Define a RECT structure with specific coordinates
.code
main PROC
    mov eax, rect1.left     ; Access the left coordinate
    mov ebx, rect1.top      ; Access the top coordinate
    mov ecx, rect1.right    ; Access the right coordinate
    mov edx, rect1.bottom   ; Access the bottom coordinate
    ; Now you can use these values for various tasks
    ; For example, you can calculate the width and height of the rectangle
    sub ecx, eax            ; Width = right - left
    sub edx, ebx            ; Height = bottom - top
    ; Display the width and height
    call DisplayWidthAndHeight
    ; You can also modify the coordinates or dimensions as needed
    add rect1.left, 5      ; Move the left side 5 units to the right
    sub rect1.right, 10    ; Shrink the width by 10 units
    ; Now the rect1 structure has been updated
    exit
main ENDP
DisplayWidthAndHeight PROC
    ; Display the width and height
    ; You can implement this function as needed
    ret
DisplayWidthAndHeight ENDP
```

Rectangle struct: The RECT structure is used to define the boundaries of a rectangle. It includes four members that determine the position and size of the rectangle. The "left" member holds the X-coordinate of the left side of the rectangle, while the "top" member stores the Y-coordinate of the top side.

Similarly, the "right" and "bottom" members hold values for the right and bottom sides of the rectangle, respectively. Together, these members specify the dimensions and position of the rectangle on the screen.

```
RECT STRUCT
    left DWORD ?
    top DWORD ?
    right DWORD ?
    bottom DWORD ?
RECT ENDS
```

The **MSGStruct structure** defines the data needed for an MS-Windows message:

```
1122 MSGStruct STRUCT
1123     msgWnd      DWORD ?
1124     msgMessage  DWORD ?
1125     msgWparam   DWORD ?
1126     msgLparam   DWORD ?
1127     msgTime     DWORD ?
1128     msgPt       POINT <>
1129 MSGStruct ENDS
```

The **WNDCLASS structure** is used to define a window class in a Windows application. Every window within a program is associated with a specific class, and the program must register this class with the operating system before the main window can be displayed. Here is the WNDCLASS structure:

```
1144 WNDCLASS STRUC
1145     style         DWORD ?    ; Window style options
1146     lpfnWndProc   DWORD ?    ; Pointer to the Window Procedure function
1147     cbClsExtra    DWORD ?    ; Extra shared memory
1148     cbWndExtra    DWORD ?    ; Number of extra bytes
1149     hInstance     DWORD ?    ; Handle to the current program
1150     hIcon         DWORD ?    ; Handle to the icon
1151     hCursor       DWORD ?    ; Handle to the cursor
1152     hbrBackground DWORD ?    ; Handle to the background brush
1153     lpszMenuName  DWORD ?    ; Pointer to the menu name
1154     lpszClassName DWORD ?    ; Pointer to the window class name
1155 WNDCLASS ENDS
```

This structure holds various parameters and settings for a window class, including its appearance, behavior, and how it interacts with the operating system. Registering a window class allows the program to create and manage windows of that class.

Here's a concise summary of the parameters within the WNDCLASS structure:

style: A combination of style options, such as WS_CAPTION and WS_BORDER, that determine the window's appearance and behavior.

lpfnWndProc: A function pointer that specifies the program's function for processing event messages triggered by the user.

cbClsExtra: Refers to shared memory used by all windows belonging to the class, and it can be set to null if not needed.

cbWndExtra: Specifies the number of extra bytes to allocate following the window instance.

hInstance: Holds a handle to the current program instance, allowing the class to be associated with this instance of the program.

hIcon and hCursor: Hold handles to icon and cursor resources for the current program, influencing the visual elements used in the window.

hbrBackground: Holds a handle to a background brush, which determines the window's background color.

lpszMenuName: Points to a menu name, defining the menu associated with the window.

lpszClassName: Points to a null-terminated string containing the window's class name, allowing the program to identify and manage windows of this class effectively.

The MessageBox Function

The MessageBox function is the easiest way to display text in a Windows application. It displays a simple message box with a text message, a caption, and one or more buttons. The buttons can be used to get the user's response to the message.

The WinMain Procedure

The WinMain procedure is the startup procedure for every Windows application. It is responsible for the following tasks:

- Getting a handle to the current program.
- Loading the program's icon and mouse cursor.
- Registering the program's main window class and identifying the procedure that will process event messages for the window.
- Creating the main window.
- Showing and updating the main window.
- Beginning a loop that receives and dispatches messages.
- The loop continues until the user closes the application window.

The WinProc Procedure

The WinProc procedure receives and processes all event messages relating to a window.

Most events are initiated by the user by clicking and dragging the mouse, pressing keyboard keys, and so on.

The WinProc procedure's job is to decode each message, and if the message is recognized, to carry out application-oriented tasks relating to the message.

The following example code shows a simple Windows application that uses the MessageBox function to display a message to the user when the user clicks the left mouse button.

```
#include <windows.h>

LRESULT CALLBACK WinProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
   switch (uMsg) {
   case WM_LBUTTONDOWN:
      MessageBox(hWnd, "You clicked the left mouse button!", "Message Box Example",
MB_OK);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   default:
      return DefWindowProc(hWnd, uMsg, wParam, lParam);
   }
   return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
   WNDCLASSEX wc;
   HWND hWnd;

   // Register the window class.
   wc.cbSize = sizeof(WNDCLASSEX);
   wc.style = 0;
   wc.lpfnWndProc = WinProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName = NULL;
   wc.lpszClassName = "MyWindowClass";

   if (!RegisterClassEx(&wc)) {
      return 0;
   }
```

```
    // Create a window.
    hWnd = CreateWindowEx(0, "MyWindowClass", "Message Box Example",
WS_OVERLAPPEDWINDOW, 100, 100, 300, 200, NULL, NULL, hInstance, NULL);
    if (!hWnd) {
        return 0;
    }

    // Show the window.
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // Wait for a key press.
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    getchar(); // Wait for a key press
    return msg.wParam;
}
```

If you compile and run this code, you should see a window with the title "Message Box Example". Click the left mouse button in the window, and a message box should appear with the text "You clicked the left mouse button!".

It seems like you've provided a portion of code and information related to writing a graphical Windows application. This code appears to be written in assembly language, specifically designed for Windows programming. Here's a breakdown of the code and related information:

ErrorHandler Procedure: This procedure is called when an error occurs during the registration and creation of the program's main window. It performs several tasks, including retrieving the system error number, formatting the system error message, displaying it in a popup message box, and freeing the memory used by the error message string.

Program Listing: This part of the code defines various data structures and constants for the Windows application, such as window titles, messages, and class names. These are used throughout the application for display and interaction.

MainWin WNDCLASS Structure: It defines the window class structure for the application. It includes settings like window procedure, icon, cursor, and other attributes.

WinMain Procedure: This is the entry point of the application. It initializes various components, including registering the window class, creating the main window, displaying messages, and entering a message-handling loop.

Message Handling Loop: The code enters a continuous message-handling loop using GetMessage, processes messages with DispatchMessage, and continues until there are no more messages. When there are no more messages, it exits the program using ExitProcess.

This code is a part of a Windows application written in assembly language, which creates a main window, displays messages, and handles messages in a loop. If you have any specific questions or need further details about this code, please let me know, and I'll address them accordingly.

```
; Windows Application (WinApp.asm)
; This program displays a resizable application window and
; several popup message boxes. Special thanks to Tom Joyce
; for the first version of this program.
.386
.model flat,STDCALL
INCLUDE GraphWin.inc

; ==================== DATA =======================
.data
AppLoadMsgTitle BYTE "Application Loaded",0
AppLoadMsgText BYTE "This window displays when the WM_CREATE "
            BYTE "message is received",0
PopupTitle BYTE "Popup Window",0
PopupText BYTE "This window was activated by a "
         BYTE "WM_LBUTTONDOWN message",0
GreetTitle BYTE "Main Window Active",0
GreetText BYTE "This window is shown immediately after "
        BYTE "CreateWindow and UpdateWindow are called.",0
CloseMsg BYTE "WM_CLOSE message received",0
ErrorTitle BYTE "Error",0
WindowName BYTE "ASM Windows App",0
className BYTE "ASMWin",0

; Define the Application's Window class structure.
MainWin WNDCLASS <NULL,WinProc,NULL,NULL,NULL,NULL,NULL, \
            COLOR_WINDOW,NULL,className>

msg MSGStruct <>
winRect RECT <>
hMainWnd DWORD ?
```

```
hInstance DWORD ?

; ==================== CODE ========================
.code
WinMain PROC
  ; Get a handle to the current process.
  INVOKE GetModuleHandle, NULL
  mov hInstance, eax
  mov MainWin.hInstance, eax

  ; Load the program's icon and cursor.
  INVOKE LoadIcon, NULL, IDI_APPLICATION
  mov MainWin.hIcon, eax
  INVOKE LoadCursor, NULL, IDC_ARROW
  mov MainWin.hCursor, eax

  ; Register the window class.
  INVOKE RegisterClass, ADDR MainWin
  .IF eax == 0
    call ErrorHandler
    jmp Exit_Program
  .ENDIF

  ; Create the application's main window.
  INVOKE CreateWindowEx, 0, ADDR className,
        ADDR WindowName,MAIN_WINDOW_STYLE,
        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
        CW_USEDEFAULT,NULL,NULL,hInstance,NULL
  ; If CreateWindowEx failed, display a message and exit.
  .IF eax == 0
    call ErrorHandler
    jmp Exit_Program
  .ENDIF

  ; Save the window handle, show and draw the window.
  mov hMainWnd,eax
  INVOKE ShowWindow, hMainWnd, SW_SHOW
  INVOKE UpdateWindow, hMainWnd

  ; Display a greeting message.
  INVOKE MessageBox, hMainWnd, ADDR GreetText,
              ADDR GreetTitle, MB_OK
```

```
  ; Begin the program's continuous message-handling loop.
Message_Loop:
  ; Get next message from the queue.
  INVOKE GetMessage, ADDR msg, NULL,NULL,NULL
  ; Quit if no more messages.
  .IF eax == 0
    jmp Exit_Program
  .ENDIF
  ; Relay the message to the program's WinProc.
  INVOKE DispatchMessage, ADDR msg
  jmp Message_Loop

Exit_Program:
  INVOKE ExitProcess,0

WinMain ENDP

; The ErrorHandler Procedure
; This procedure handles errors during window registration and creation.
ErrorHandler PROC
  ; Call GetLastError to retrieve the system error number.
  INVOKE GetLastError
  ; Call FormatMessage to retrieve the appropriate system-formatted error message string.
  INVOKE FormatMessage, FORMAT_MESSAGE_FROM_SYSTEM, NULL, eax, \
        0, ADDR ErrorTitle, 256, 0
  ; Call MessageBox to display a popup message box containing the error message string.
  INVOKE MessageBox, NULL, eax, ADDR ErrorTitle, MB_OK
  ; Call LocalFree to free the memory used by the error message string.
  INVOKE LocalFree, eax
  ret
ErrorHandler ENDP
```

This combined code includes the ErrorHandler procedure and the WinMain procedure along with the relevant data and constants. It's ready to be used in a Windows application written in assembly language.

WinMain Procedure:

WinMain is the entry point of the application, where the program execution begins.

It starts by getting a handle to the current process using GetModuleHandle and stores it in hInstance.

It loads the program's icon and cursor using LoadIcon and LoadCursor functions and assigns them to the MainWin structure, which defines the window class.

The window class is registered using RegisterClass.

If the registration fails (indicated by eax == 0), the ErrorHandler procedure is called, and the program exits.

If the registration is successful, the application's main window is created using CreateWindowEx.

If this fails, it also calls the ErrorHandler procedure and exits.

After creating the main window, it's displayed and updated with ShowWindow and UpdateWindow functions.

A greeting message is displayed in a message box.

The program enters a message-handling loop using GetMessage, processes the messages with DispatchMessage, and continues until there are no more messages.

Exit_Program Label:

The Exit_Program label is used to handle the program's exit. It's reached when there are no more messages in the message loop, and it invokes ExitProcess to terminate the program.

This code sets up the application's main window, registers its class, and enters the message-handling loop.

It handles basic application initialization, including window creation and message processing.

The Exit_Program label is used for a clean program exit when there are no more messages to process.

==========

```
;-----------------------------------------------------
WinProc PROC,
hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD
;
; The application's message handler, which handles
; application-specific messages. All other messages
; are forwarded to the default Windows message
; handler.
;-----------------------------------------------------
   mov eax, localMsg
   .IF eax == WM_LBUTTONDOWN
      ; Mouse button?
```

```
      INVOKE MessageBox, hWnd, ADDR PopupText,
        ADDR PopupTitle, MB_OK
      jmp WinProcExit
    .ELSEIF eax == WM_CREATE
      ; Create window?
      INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
        ADDR AppLoadMsgTitle, MB_OK
      jmp WinProcExit
    .ELSEIF eax == WM_CLOSE
      ; Close window?
      INVOKE MessageBox, hWnd, ADDR CloseMsg,
        ADDR WindowName, MB_OK
      INVOKE PostQuitMessage,0
      jmp WinProcExit
    .ELSE
      ; Other message?
      INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
      jmp WinProcExit
    .ENDIF
WinProcExit:
    ret
WinProc ENDP

;-----------------------------------------------------
ErrorHandler PROC
; Display the appropriate system error message.
;-----------------------------------------------------
    .data
    pErrorMsg DWORD ?
    ; Pointer to error message
    messageID DWORD ?

    .code
    INVOKE GetLastError
    ; Returns message ID in EAX
    mov messageID, eax

    ; Get the corresponding message string.
    INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
      FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
      ADDR pErrorMsg, NULL, NULL
```

```
    ; Display the error message.
    INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
      MB_ICONERROR + MB_OK

    ; Free the error message string.
    INVOKE LocalFree, pErrorMsg
    ret
ErrorHandler ENDP

END WinMain
```

This code combines the WinProc and ErrorHandler procedures with your existing code and includes appropriate comments for clarity. It's ready for use in a Windows application written in assembly language.

WinProc Procedure:

WinProc is a procedure that serves as the message handler for the Windows application. It takes four parameters: hWnd (a handle to the window), localMsg (the message ID), wParam, and lParam (message-specific data).

The purpose of WinProc is to handle application-specific messages. It checks the localMsg parameter to determine the type of message received.

If localMsg is equal to WM_LBUTTONDOWN, it displays a message box indicating that the left mouse button was clicked.

If localMsg is equal to WM_CREATE, it displays a message box indicating that the window was created.

If localMsg is equal to WM_CLOSE, it displays a message box indicating that the window is about to close and triggers the application to quit.

If the message is none of the above, it forwards the message to the default Windows message handler using DefWindowProc.

ErrorHandler Procedure:

ErrorHandler is a procedure designed to handle errors during window registration and creation.

It first declares data and code sections for its implementation.

Inside, it uses the GetLastError function to retrieve the system error number and stores it in messageID.

It then calls FormatMessage to retrieve the corresponding system-formatted error message string, which is allocated dynamically and stored in pErrorMsg.

Next, it displays the error message in a message box with the title "Error."

Finally, it frees the memory used by the error message string using LocalFree.

The provided code integrates these two procedures with your existing code to handle messages and errors in your Windows application written in assembly language.

It adds comments to explain each part of the code for better understanding and maintainability.

This code is now ready to be used in your application.

DYNAMIC MEMORY

Dynamic memory allocation is the process of allocating memory during the execution of a program.

This is in contrast to **static memory allocation**, where memory is allocated at compile time.

There are two main ways to perform dynamic memory allocation in assembly language:

Using system calls: This involves making calls to the operating system to allocate and deallocate memory.

Implementing a heap manager: This involves implementing your own data structure and algorithms to manage memory allocation and deallocation.

The example program in the section you provided uses the first method. It makes system calls to the Windows operating system to allocate and deallocate memory.

Here is a summary of the steps involved in dynamic memory allocation using system calls:

Make a system call to allocate memory.

This will return a pointer to the allocated memory block. Use the allocated memory block.

Make a system call to deallocate the memory block when you are finished using it. The following table lists some of the Win32 API functions that can be used for dynamic memory allocation:

Function	Description
GetProcessHeap	Returns a 32-bit integer handle to the program's existing heap area in EAX. If the function succeeds, it returns a handle to the heap in EAX. If it fails, the return value in EAX is NULL.
HeapAlloc	Allocates a block of memory from a heap. If it succeeds, the return value in EAX contains the address of the memory block. If it fails, the returned value in EAX is NULL.
HeapCreate	Creates a new heap and makes it available to the calling program. If the function succeeds, it returns a handle to the newly created heap in EAX. If it fails, the return value in EAX is NULL.
HeapDestroy	Destroys the specified heap object and invalidates its handle. If the function succeeds, the return value in EAX is nonzero.
HeapFree	Frees a block of memory previously allocated from a heap, identified by its address and heap handle. If the block is freed successfully, the return value is nonzero.
HeapReAlloc	Reallocates and resizes a block of memory from a heap. If the function succeeds, the return value is a pointer to the reallocated memory block. If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.
HeapSize	Returns the size of a memory block previously allocated by a call to HeapAlloc or HeapReAlloc. If the function succeeds, EAX contains the size of the allocated memory block, in bytes. If the function fails, the return value is SIZE_T – 1. (SIZE_T equals the maximum number of bytes to which a pointer can point.)

Here is a summary of the heap functions you provided:

GetProcessHeap() returns a handle to the current process's default heap.

HeapCreate() creates a new private heap for the current process.

HeapDestroy() destroys an existing private heap.

HeapAlloc() allocates a block of memory from a heap.

HeapFree() frees a block of memory previously allocated from a heap.

When to use which function:

Use GetProcessHeap() if you are content to use the default heap owned by the current program.

Use HeapCreate() to create a new private heap if you need more control over memory management.

Use HeapDestroy() to destroy a private heap when you are finished using it. Use HeapAlloc() to allocate memory from a heap.

Use HeapFree() to free memory that was allocated from a heap.

Here is an example of how to use the HeapAlloc() and HeapFree() functions to allocate and free a block of memory from a heap:

```
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.

Here's the complete program:

```
1245 ; Heap Test #1 (Heaptest1.asm)
1246 INCLUDE Irvine32.inc
1247 ; This program uses dynamic memory allocation to allocate and
1248 ; fill an array of bytes.
1249
1250 .data
1251 ARRAY_SIZE = 1000
1252 FILL_VAL EQU 0FFh
1253 hHeap HANDLE ?
1254 ; handle to the process heap
1255 pArray DWORD ?
1256 ; pointer to block of memory
1257
1258 .code
1259 main PROC
1260 INVOKE GetProcessHeap
1261 ; get handle to the program heap
1262 .IF eax == NULL
1263 ; if failed, display message
1264     call WriteWindowsMsg
1265     jmp quit
1266 .ELSE
1267     mov hHeap,eax
1268     ; success
1269 .ENDIF
1270
```

```
1271 call allocate_array
1272 jnc arrayOk
1273 ; failed (CF = 1)?
1274 call WriteWindowsMsg
1275 call Crlf
1276 jmp quit
1277
1278 arrayOk:
1279 ; ok to fill the array
1280 call fill_array
1281 call display_array
1282 call Crlf
1283 ; free the array
1284 INVOKE HeapFree, hHeap, 0, pArray
1285
1286 quit:
1287     exit
1288
1289 main ENDP
```

```
1291 ;-----------------------------------------------------
1292 allocate_array PROC USES eax
1293 ;
1294 ; Dynamically allocates space for the array.
1295 ; Receives: EAX = handle to the program heap
1296 ; Returns: CF = 0 if the memory allocation succeeds.
1297 ;-----------------------------------------------------
1298 INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, ARRAY_SIZE
1299 .IF eax == NULL
1300     stc
1301 ; return with CF = 1
1302 .ELSE
1303     mov pArray,eax
1304 ; save the pointer
1305     clc
1306 ; return with CF = 0
1307 .ENDIF
1308     ret
1309
1310 allocate_array ENDP
1311
```

```
1312 ;-----------------------------------------------------
1313 fill_array PROC USES ecx edx esi
1314 ;
1315 ; Fills all array positions with a single character.
1316 ; Receives: nothing
1317 ; Returns: nothing
1318 ;-----------------------------------------------------
1319 mov ecx,ARRAY_SIZE
1320 ; loop counter
1321 mov esi,pArray
1322 ; point to the array
1323 L1:
1324 mov BYTE PTR [esi],FILL_VAL
1325 ; fill each byte
1326 inc esi
1327 ; next location
1328 loop L1
1329 ret
1330
1331 fill_array ENDP
1332
```

```
1333 ;-----------------------------------------------------
1334 display_array PROC USES eax ebx ecx esi
1335 ;
1336 ; Displays the array
1337 ; Receives: nothing
1338 ; Returns: nothing
1339 ;-----------------------------------------------------
1340 mov ecx,ARRAY_SIZE
1341 ; loop counter
1342 mov esi,pArray
1343 ; point to the array
1344 L1:
1345 mov al,[esi]
1346 ; get a byte
1347 mov ebx,TYPE BYTE
1348 call WriteHexB
1349 ; display it
1350 inc esi
1351 ; next location
1352 loop L1
1353 ret
1354
1355 display_array ENDP
1356
1357 END main
```

The HeapTest1.asm program is an assembly language example that showcases dynamic memory allocation and manipulation in the Windows environment.

The code demonstrates how to allocate memory from the heap, fill that memory with specific values, and display the allocated memory's contents.

The program starts with the .data section, where constants and variables are defined. It specifies the size of the array to be allocated, which is set to 1000 bytes, and the value used to fill the array, which is 0FFh.

The .code section begins with the main procedure. In this procedure, the program performs the following tasks:

It calls the GetProcessHeap function to obtain a handle to the default heap owned by the current process.

This is where memory allocations will be made. If obtaining the heap handle fails (resulting in a NULL handle), the program calls the WriteWindowsMsg function to display an error message and then exits.

If the GetProcessHeap call is successful, the obtained heap handle is stored in the hHeap variable for later use.

The program then proceeds to allocate memory for an array by calling the allocate_array procedure.

If memory allocation fails (indicated by the Carry Flag being set), it calls the WriteWindowsMsg function to display an error message and exits.

If allocation is successful, the pointer to the allocated memory is saved in the pArray variable.

After successful allocation, the program calls the fill_array procedure, which fills the allocated memory with a specified value (0FFh in this case).

Following the memory filling, the program calls the display_array procedure to display the contents of the allocated memory in hexadecimal format.

After displaying the memory contents, the program frees the allocated memory by invoking the HeapFree function.

The program then proceeds to the quit label, where it invokes the Exit system call to terminate the program.

In summary, HeapTest1.asm demonstrates the process of dynamic memory allocation in assembly language within the Windows environment.

It allocates memory from the default process heap, fills that memory with specific values, displays the memory's contents, and finally releases the allocated memory.

The program uses the GetProcessHeap function to obtain the default heap handle and the HeapAlloc and HeapFree functions for memory allocation and deallocation, respectively.

Let's move on to heaptest2.asm:

```
1360 ; Heap Test #2 (Heaptest2.asm)
1361 INCLUDE Irvine32.inc
1362
1363 .data
1364 HEAP_START = 2000000   ; 2 MByte
1365 HEAP_MAX = 400000000   ; 400 MByte
1366 BLOCK_SIZE = 500000    ; 0.5 MByte
1367 hHeap HANDLE ?
1368 pData DWORD ?
1369 str1 BYTE 0dh, 0ah, "Memory allocation failed", 0dh, 0ah, 0
1370
1371 .code
1372 main PROC
1373     ; Create a new heap with specified size limits
1374     INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
1375     .IF eax == NULL
1376         ; Failed to create heap
1377         call WriteWindowsMsg
1378         call Crlf
1379         jmp quit
1380     .ELSE
1381         mov hHeap, eax
1382         ; Success: store the heap handle
1383     .ENDIF
1384
1385     mov ecx, 2000   ; Loop counter
1386
```

```
1387 L1:
1388     call allocate_block
1389     ; Allocate a block
1390
1391     .IF Carry?
1392         ; Allocation failed
1393         mov edx, OFFSET str1
1394         ; Display error message
1395         call WriteString
1396         jmp quit
1397     .ELSE
1398         ; Allocation successful
1399         mov al, '.'
1400         ; Show progress with a dot
1401         call WriteChar
1402     .ENDIF
1403
1404     loop L1
1405
1406 quit:
1407     ; Destroy the heap
1408     INVOKE HeapDestroy, hHeap
1409     .IF eax == NULL
1410         ; Failed to destroy heap
1411         call WriteWindowsMsg
1412         call Crlf
1413     .ENDIF
1414
1415     exit
1416 main ENDP
1417
```

```
1417
1418 allocate_block PROC USES ecx
1419     ; Allocate a block and fill it with all zeros
1420     INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, BLOCK_SIZE
1421     .IF eax == NULL
1422         stc
1423         ; Return with CF = 1 (allocation failed)
1424     .ELSE
1425         mov pData, eax
1426         ; Save the pointer to the allocated memory
1427         clc
1428         ; Return with CF = 0 (allocation succeeded)
1429     .ENDIF
1430     ret
1431 allocate_block ENDP
1432
1433 free_block PROC USES ecx
1434     ; Free a previously allocated block
1435     INVOKE HeapFree, hHeap, 0, pData
1436     ret
1437 free_block ENDP
1438
1439 END main
```

HeapTest2.asm is an assembly program that demonstrates dynamic memory allocation and usage of custom heap management.

It aims to allocate large blocks of memory repeatedly until the specified heap size limit is reached. The code is divided into sections for clarity.

The data section, defined using the .data directive, starts by declaring constants and variables.

HEAP_START is set to 2 megabytes (2MB), representing the initial heap size.

HEAP_MAX is set to 400 megabytes (400MB), indicating the maximum heap size.

BLOCK_SIZE is set to 0.5 megabytes (0.5MB), representing the size of memory blocks to be allocated.

The program uses hHeap to store the handle to the custom heap and pData to hold the pointer to the allocated memory. str1 is a string that will be used to display an error message in case of allocation failure.

The .code section contains the main procedure, labeled main PROC. It begins by invoking the HeapCreate function to create a new heap with specified initial and maximum sizes.

If the creation of the heap fails (resulting in a NULL heap handle), the program calls the WriteWindowsMsg function to display an error message and then jumps to the quit label to exit.

In case of a successful heap creation, the handle to the custom heap is stored in the hHeap variable for later use.

A loop is initiated using ecx as a loop counter, set to 2000 iterations. The purpose of this loop is to repeatedly allocate memory blocks.

Within the loop, the program calls the allocate_block procedure. This procedure uses the HeapAlloc function to allocate memory from the custom heap.

If memory allocation fails (indicated by the Carry Flag being set), the program displays an error message using str1, calls WriteString to print the message, and jumps to the quit label to exit.

If memory allocation is successful, a dot ('.') is displayed on the screen as a progress indicator, indicating a successful memory allocation.

The program continues the loop until all 2000 iterations are completed, each time allocating a memory block.

After the loop finishes, the program reaches the quit label, where it invokes HeapDestroy to destroy the custom heap.

If HeapDestroy fails (returns NULL), an error message is displayed using WriteWindowsMsg, and the program exits using the exit system call.

In summary, HeapTest2.asm showcases dynamic memory allocation using custom heap management. It repeatedly allocates memory blocks until a specified heap size limit is reached.

The program uses functions like HeapCreate, HeapAlloc, and HeapDestroy to manage custom heaps and memory allocation.

Progress is indicated by displaying dots for successful allocations, and any errors are communicated using appropriate error messages.

The program demonstrates the flexibility of heap management in assembly language within the Windows environment.

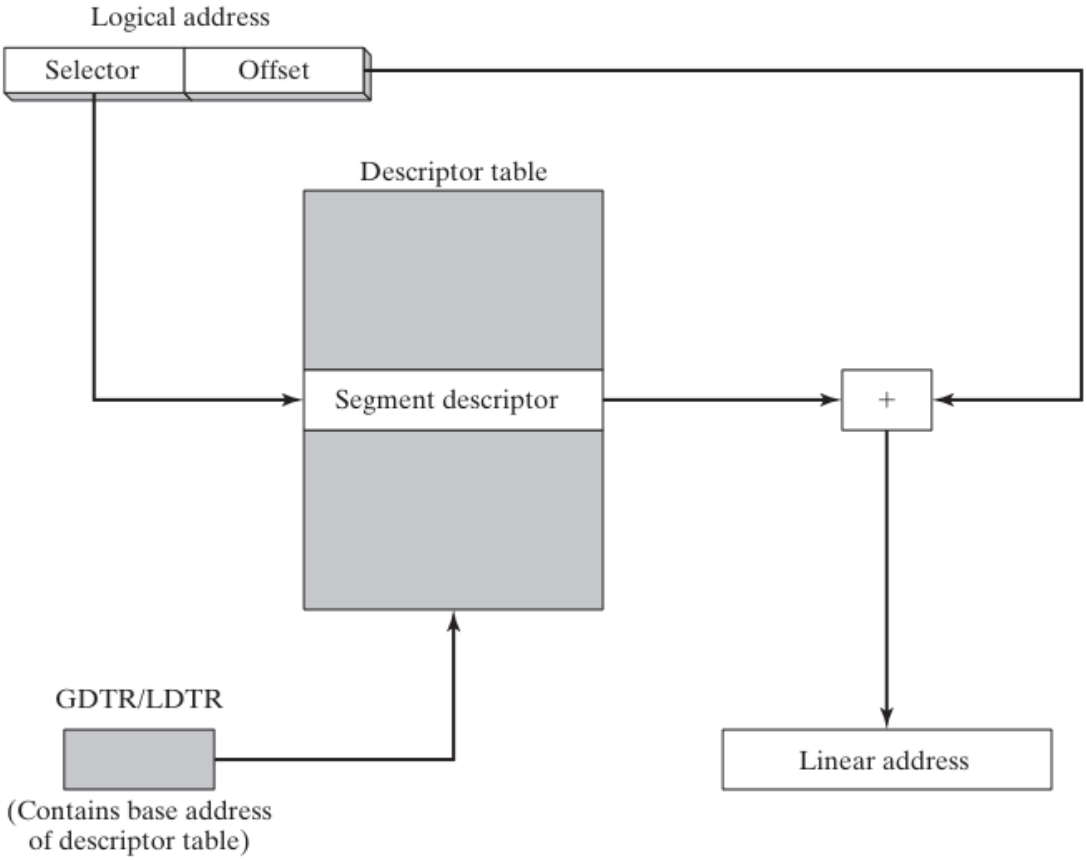
x86 MEMORY MANAGEMENT

Logical addresses and linear addresses are two different ways of addressing memory in an x86 processor.

A **logical address** is a combination of a segment selector and a 32-bit offset. The segment selector is a 16-bit value that identifies a segment descriptor, which in turn contains information about a memory segment. The offset is a 32-bit value that identifies a location within the segment.

A **linear address** is a 32-bit value that uniquely identifies a location in memory. It is calculated by adding the segment base address to the offset.

The x86 processor uses a two-step process to translate logical addresses to linear addresses:



The **segment selector** is used to index the segment descriptor table (GDT or LDT) to obtain the segment descriptor.

The **segment base address** is added to the offset to produce the linear address. The following diagram shows the process of translating logical addresses to linear addresses:

```
Logical address = segment selector + offset
Segment base address = segment descriptor table[segment selector]
Linear address = segment base address + offset
```

Once the linear address has been calculated, the processor can use it to access memory directly.

Example

Suppose we have a program that has a variable at offset 200h in a segment with the segment selector value 0x1000. The segment descriptor table contains a segment descriptor for this segment with a base address of 0x100000.

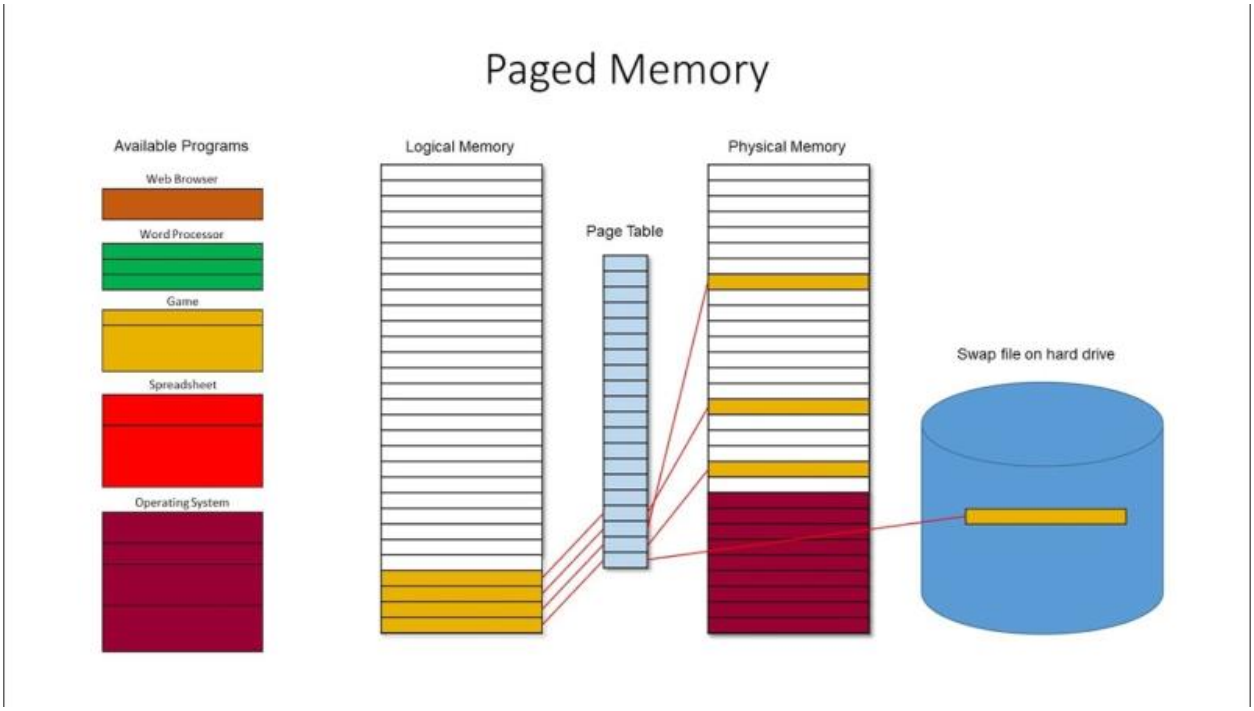
To access this variable, the processor would first calculate the linear address:

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

Once the linear address has been calculated, the processor can use it to access the variable at memory location 0x100200.

Paging

Paging is a memory management technique that allows the operating system to divide physical memory into pages.



Pages are typically 4KB in size, but can also be 2MB or larger.

When a program needs to access memory, the operating system converts the program's linear address to a physical address using a page table.

The **page table** is a data structure that maps linear addresses to physical addresses.

The following diagram shows the process of translating linear addresses to physical addresses using a page table:

```
1454 Linear address = page table index + page offset
1455 Physical address = page table[page table index] + page offset
```

The page table index is the upper 20 bits of the linear address. The page offset is the lower 12 bits of the linear address.

The operating system can use paging to implement a number of features, such as virtual memory and memory protection.

Conclusion

Logical addresses and linear addresses are two different ways of addressing memory in an x86 processor. Logical addresses are used by programs, while linear addresses are used by the processor.

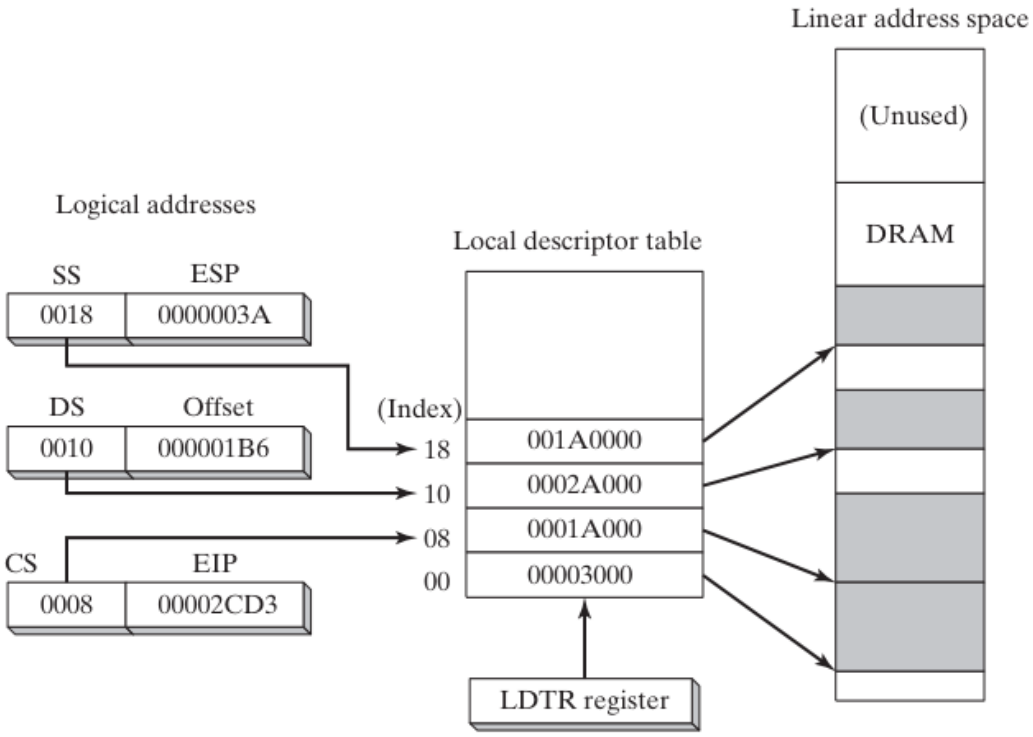
The processor translates logical addresses to linear addresses using segment descriptors. Linear addresses can then be translated to physical addresses using a page table.

==========

Descriptor tables

==========

Indexing into a local descriptor table.



Descriptor tables are data structures that contain information about memory segments. A segment is a variable-sized area of memory that is used by a program to store code or data.

There are two types of descriptor tables:

Global Descriptor Table (GDT): The GDT contains segment descriptors for all of the segments that are used by the system.

Local Descriptor Table (LDT): Each task or process has its own LDT, which contains segment descriptors for the segments that are used by that task or process.

Segment descriptors contain information about a segment, such as its base address, size, and access rights. The processor uses this information to translate logical addresses to linear addresses.

A logical address is a combination of a segment selector and a 32-bit offset. The segment selector is a 16-bit value that identifies a segment descriptor in the GDT or LDT. The offset is a 32-bit value that identifies a location within the segment.

The processor calculates the linear address by adding the segment base address to the offset. The linear address is then used to access memory directly.

Suppose we have a program that has a variable at offset 200h in a segment with the segment selector value 0x1000.

The GDT contains a segment descriptor for this segment with a base address of 0x100000.

To access this variable, the processor would first calculate the linear address:

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

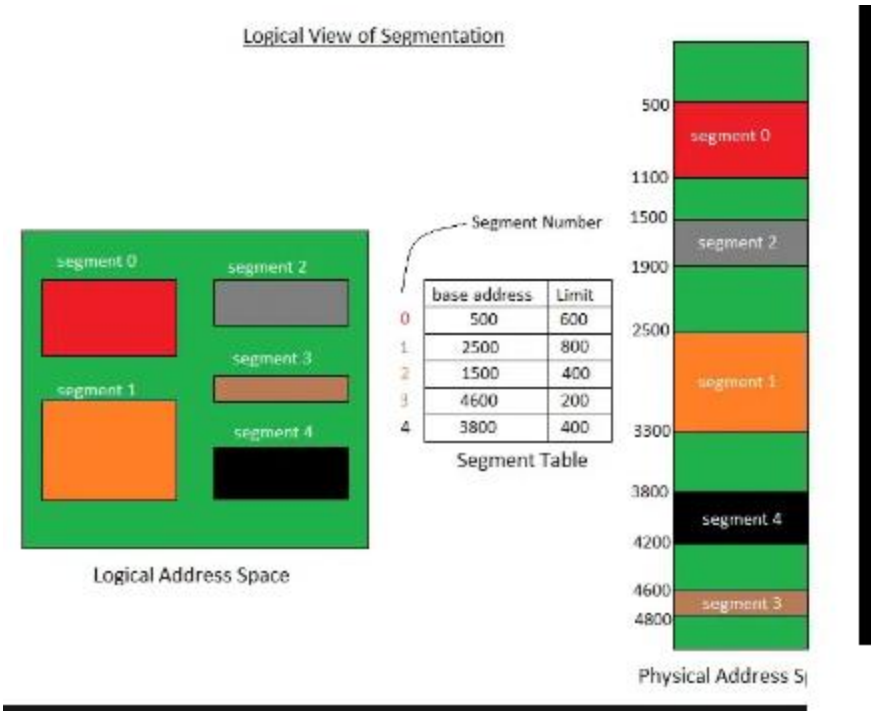
Once the linear address has been calculated, the processor can use it to access the variable at memory location 0x100200.

Segment descriptor details

In addition to the segment's base address, the segment descriptor contains the following information:

Segment limit: The segment limit specifies the maximum size of the segment. If a program tries to access a memory location outside of the segment limit, a processor fault is generated.

Segment type: The segment type specifies the type of data that is stored in the segment. For example, a code segment contains code, and a data segment contains data.



Access rights: The access rights specify which operations are allowed on the segment. For example, a read-only segment can only be read, and a write-only segment can only be written to. The processor uses this information to ensure that programs do not access memory in an unauthorized way.

Segment descriptors in x86 processors contain a number of fields that control how the segment is used, including:

Base address: The starting address of the segment in the linear address space.

Privilege level: The privilege level required to access the segment.

Segment type: The type of segment, such as code, data, or stack.

Segment present flag: Indicates whether the segment is present in memory.

Granularity flag: Determines whether the segment limit is interpreted in bytes or 4096-byte units.

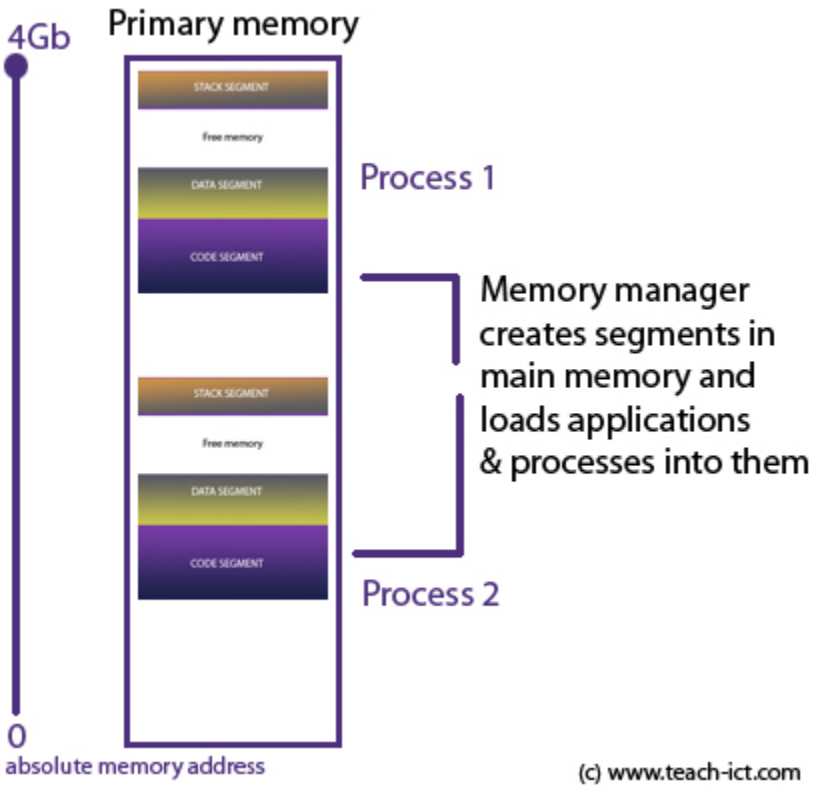
Segment limit: The maximum size of the segment.

The **protection level field** is used to protect operating system data from access by application programs.

Each segment can be assigned a **privilege level between 0 and 3**, where 0 is the most privileged and 3 is the least privileged.

If a program with a **higher privilege level** tries to access a segment with a lower privilege level, a processor fault is generated.

This prevents application programs from **accidentally or maliciously** modifying operating system data.

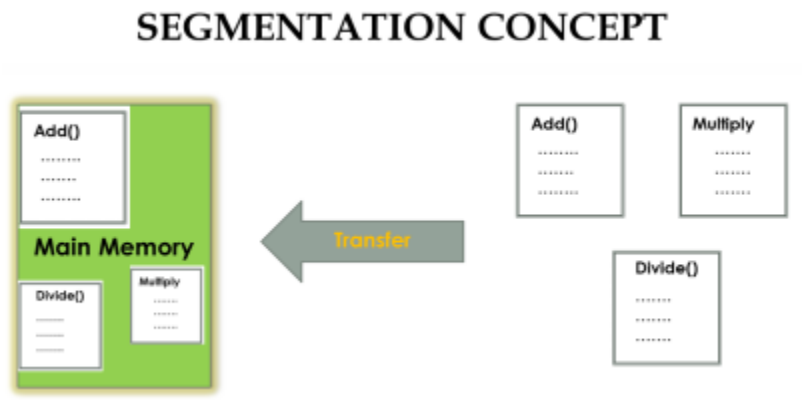


So,

The **segment type field** is used to specify the type of data that is stored in the segment and the type of access that is allowed. For example, a code segment can only be executed, and a data segment can only be read or written to.

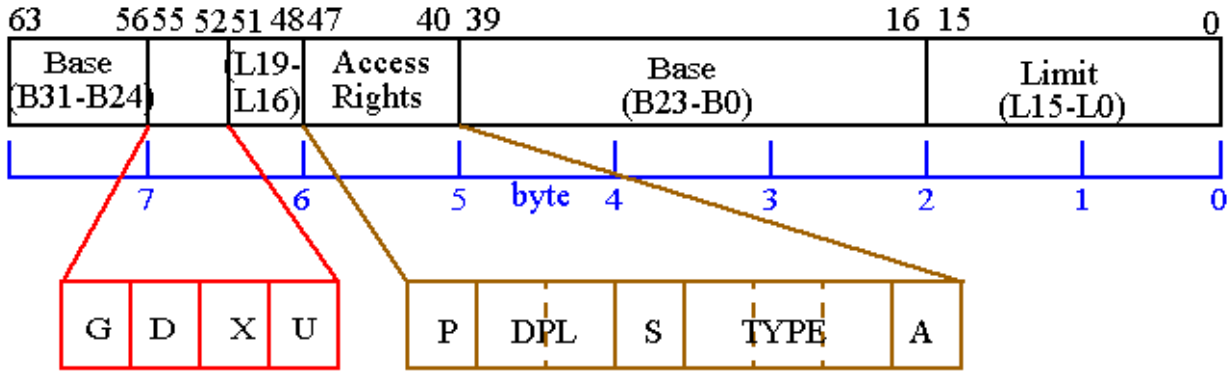
The **segment present flag** is used to indicate whether the segment is currently present in memory. If the flag is set, the segment is present in memory and can be accessed. If the flag is not set, the segment is not present in memory and cannot be accessed.

The **granularity flag** is used to determine how the segment limit field is interpreted. If the flag is set, the segment limit is interpreted in 4096-byte units. If the flag is not set, the segment limit is interpreted in bytes.



The **segment limit field** specifies the maximum size of the segment. If a program tries to access a memory location outside of the segment limit, a processor fault is generated.

Segment descriptors are an important part of memory management in x86 processors. They allow the processor to translate logical addresses to linear addresses and to ensure that programs do not access memory in an unauthorized way.



Described segment descriptor:

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 image also shows the following:

The segment descriptor table (GDT) is located at address 0x00000000. The segment selector is a 16-bit value that identifies a segment descriptor in the GDT.

The linear address is a 32-bit value that identifies a memory location in the linear address space.

The processor uses the segment selector to index the GDT to obtain the segment descriptor.

The segment descriptor is then used to calculate the linear address of the memory location.

==========

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.

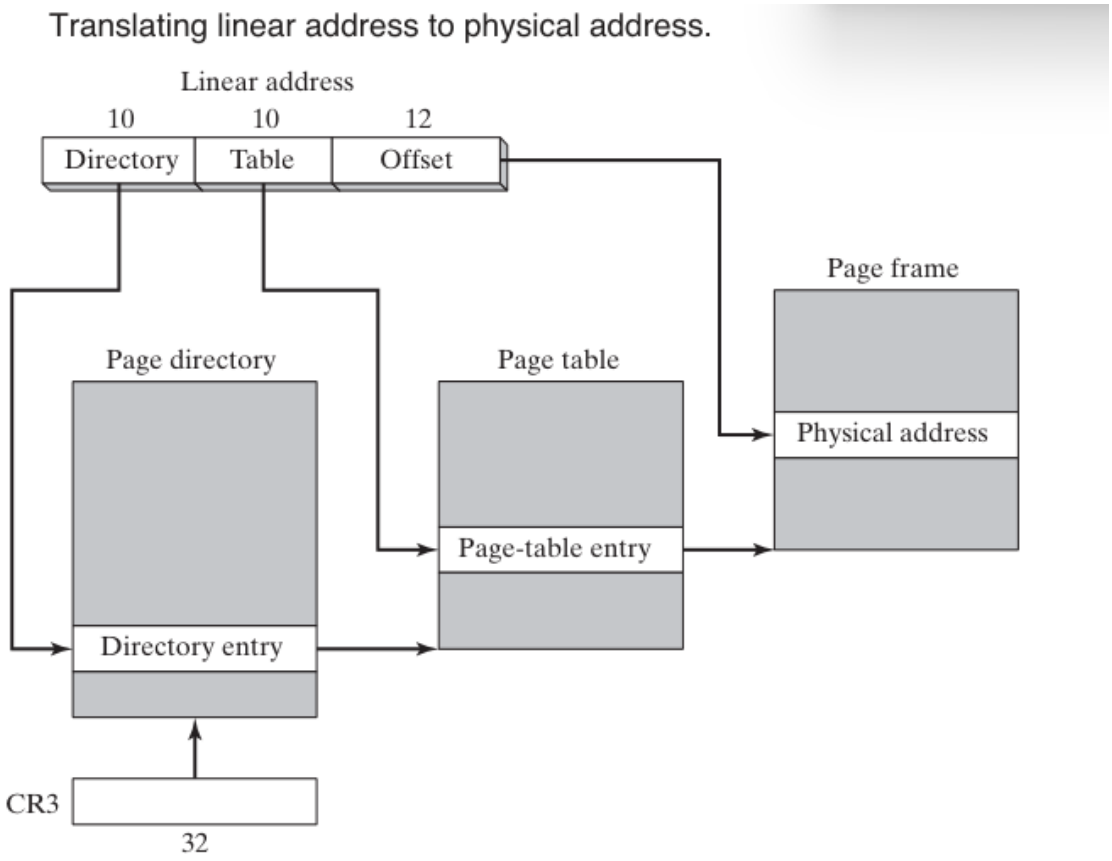
Steps in page translation

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.

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

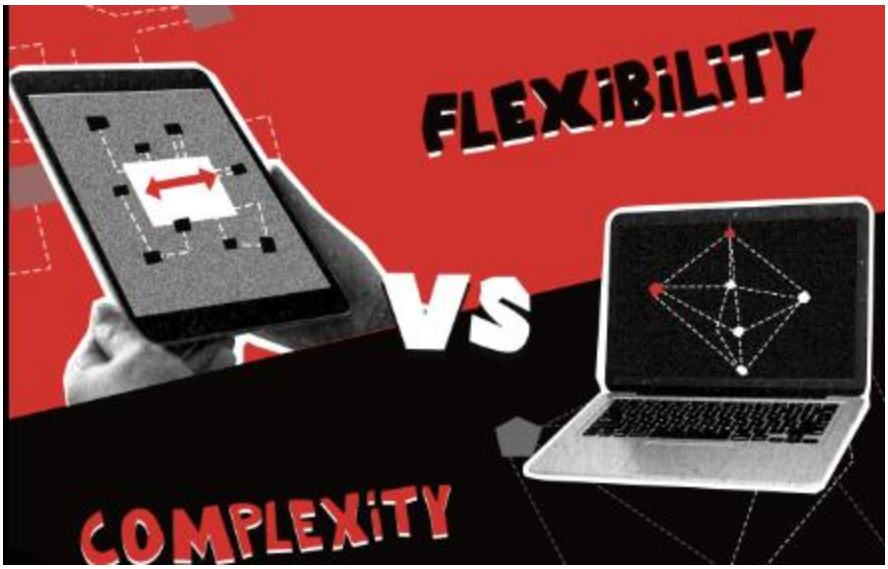
VMM offers a number of benefits, including:

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.