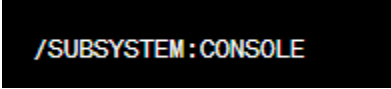


Contents

WINAPI IN ASSEMBLY INTRODUCTION.....	2
DISPLAYING A MESSAGEBOX	17
CONSOLE INPUT.....	24
CHECKING FOR ERRORS	25
SINGLE CHARACTER INPUT	29
CONSOLE OUTPUT	33
FILE HANDLING.....	38
WRITEFILE	43
CONSOLE WINDOW MANIPULATION.....	52
SETTING TEXT COLOR.....	60
TIME, WINAPI AND ASSEMBLY.....	61
CALLING 64-BIT WINAPI FUNCTION IN MASM.....	68
DYNAMIC MEMORY	83
x86 MEMORY MANAGEMENT.....	95

WINAPI IN ASSEMBLY INTRODUCTION

When a Windows application launches, it can create either a console window or a graphical window. In our project files, we've used the following option with the LINK command to specify a console-based application:



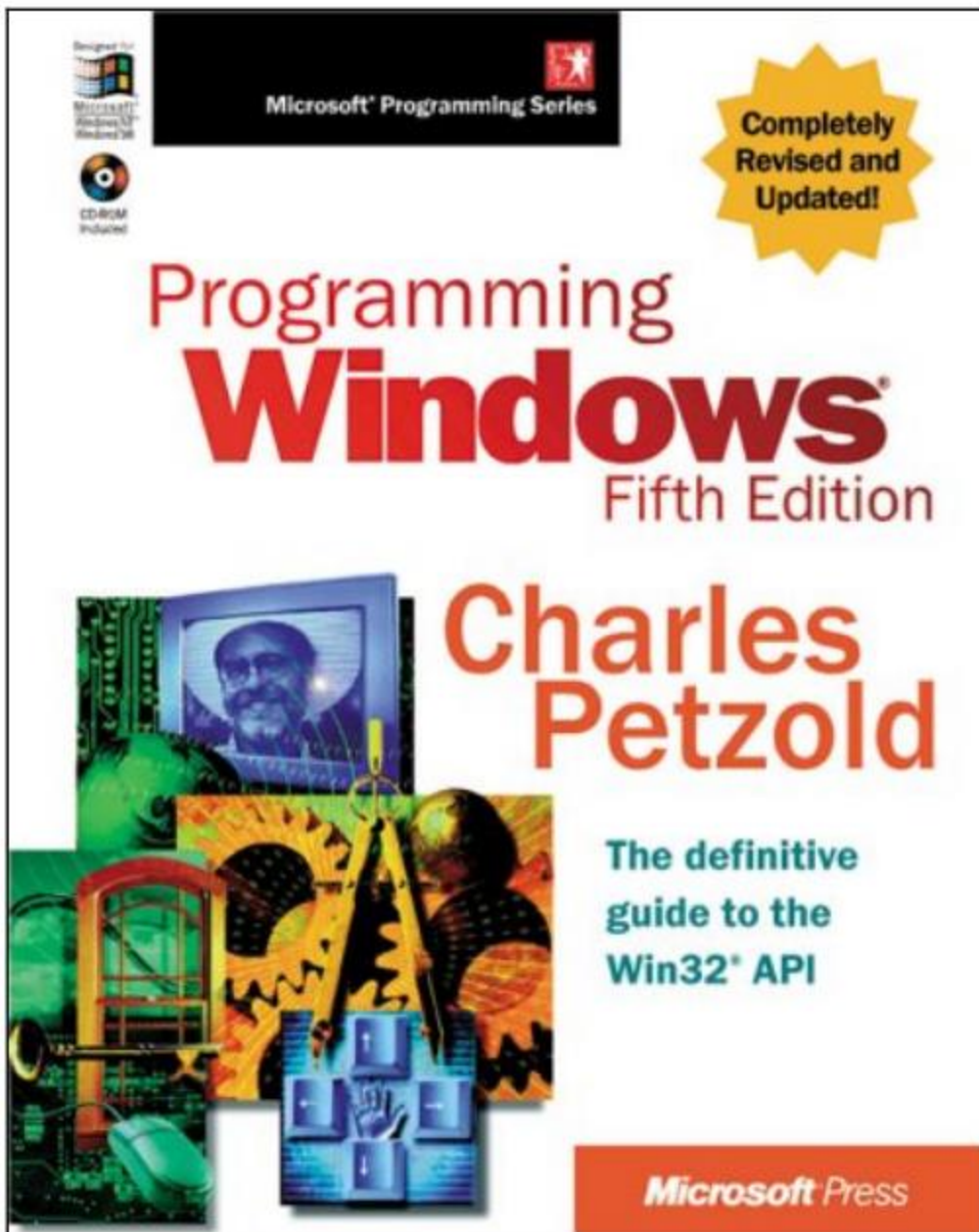
```
/SUBSYSTEM:CONSOLE
```

A console program resembles an MS-DOS window but with additional features, as we'll explore shortly.

It includes a single input buffer for queuing input records, which contain data about input events such as keyboard input, mouse clicks, and user actions like resizing the console window.

Additionally, it features one or more screen buffers, which are two-dimensional arrays containing character and color data that affect the appearance of text in the console window.

Win32 API Reference Information



Copyright© 2002 by The A-Team – Version 0.0.2

Here is a summary of the key points:

- This section introduces a subset of Win32 API functions with simple examples, but does not cover every detail due to space constraints.
- The Microsoft MSDN website contains full documentation on the Win32 APIs. Make sure to filter for "Platform SDK" when searching.



- The sample programs include lists of function names in kernel32.lib and user32.lib libraries for reference.
- Win32 API functions often use named constants like TIME_ZONE_ID_UNKNOWN.

```
#include <Windows.h>

// What is the
// Windows API?
```

- Some constants are defined in SmallWin.inc, others can be found by referring to Windows header files like WinNT.h on the book's website.
- The header files define groups of related constants used by the Win32 functions.

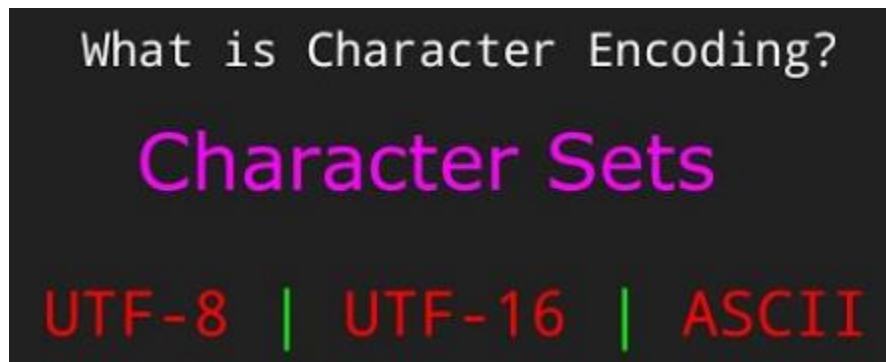


Windows API

- This overview provides a starting point on using Win32 APIs in assembly, but full details can be found in the Microsoft documentation and header files.
- The example code illustrates simple usage of some key functions.

Character Sets and Windows API Functions

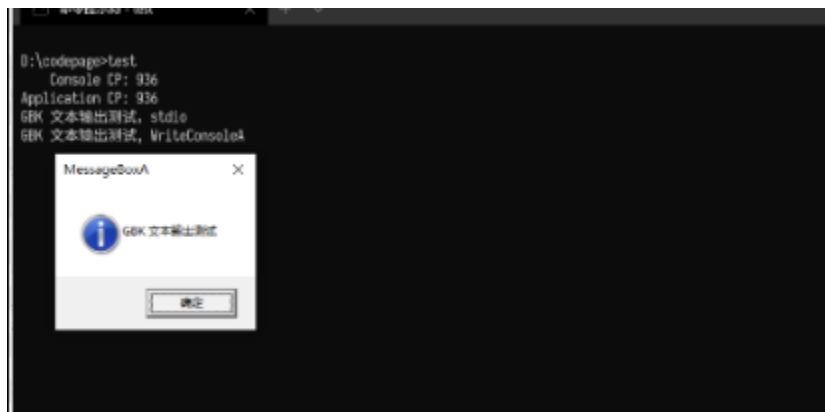
When calling functions in the Win32 API, two character sets are commonly used: the 8-bit ASCII/ANSI character set and the 16-bit Unicode set, which is available in recent Windows versions.



Win32 functions related to text come in two versions: one ending with 'A' (for 8-bit ANSI characters) and the other ending with 'W' (for wide character sets, including Unicode). For example, there are two versions of the WriteConsole function:

- • **WriteConsoleA**
- **WriteConsoleW**

It's important to note that function names ending with 'W' are not supported in Windows 95 or 98.



In modern Windows versions, Unicode is the native character set. If you call a function like WriteConsoleA, the operating system performs character conversion from ANSI to Unicode and then calls WriteConsoleW.



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

In the program's include files provided with this book, function names like `WriteConsoleA` are redefined as follows:

```
WriteConsole EQU <WriteConsoleA>
```

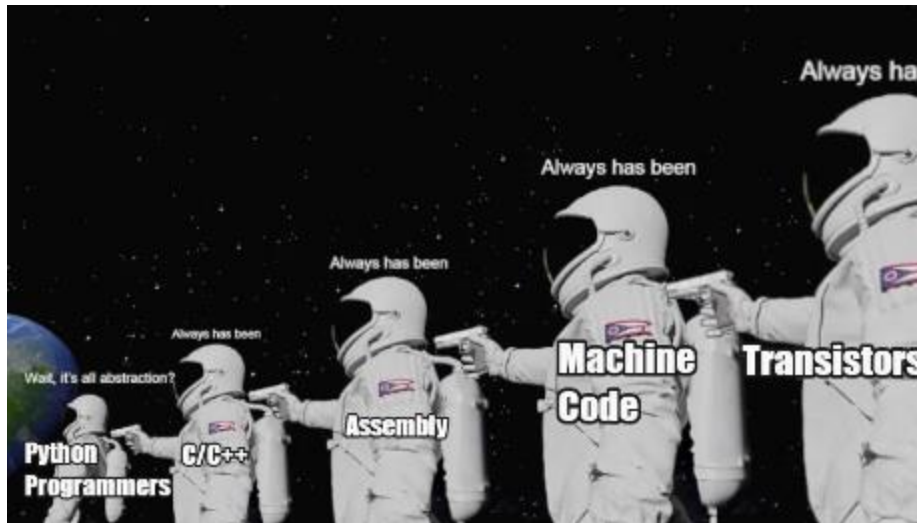
This definition allows you to call `WriteConsole` using the generic name.

High-Level and Low-Level Console Access

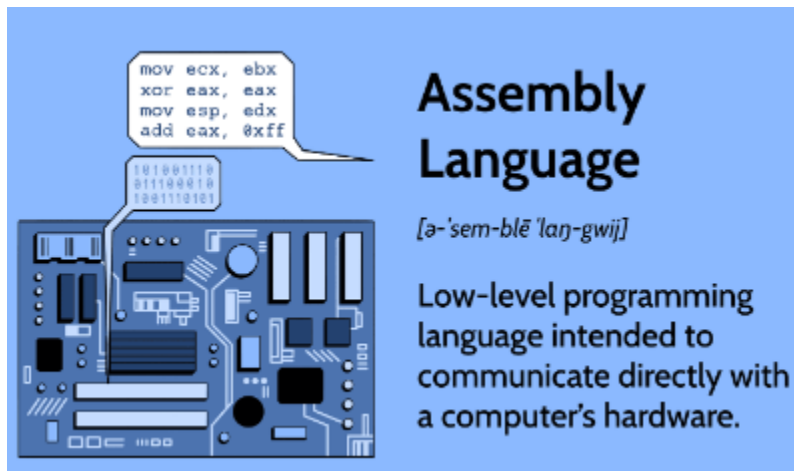
There are two levels of access to the console, each offering a trade-off between simplicity and complete control:



High-Level Console Functions: These functions read a stream of characters from the console's input buffer and write character data to the console's screen buffer. Both input and output can be redirected to read from or write to text files.



Low-Level Console Functions: These functions provide detailed information about keyboard and mouse events, as well as user interactions with the console window (e.g., dragging, resizing). They also enable precise control over the window's size, position, and text colors.



This summary should provide you with a clear understanding of character sets and the distinctions between high-level and low-level console access in Windows API programming. If you have any further questions or need more information, please feel free to ask.

Windows Data Types

The MASM translations of the MS-Windows data types in Table 11-1 are as follows:

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, the following MASM types are equivalent to the corresponding MS-Windows types:

```

01 DWORD = BOOL = BOOLEAN
02 BYTE = CHAR

```

It is important to note that the **HANDLE type in MS-Windows is also a DWORD**. This means that a HANDLE variable can be used to store a handle to any type of object, such as a window, a file, or a memory region.

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

```

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

```

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

It is automatically included in programs by Irvine32.inc. The file contains definitions for several Win32 data types, including the HANDLE type.

Here are some examples of how to use the SmallWin.inc include file:

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

The SmallWin.inc include file can be used to simplify the development of Win32 API programs in MASM.

Here is a clear and concise explanation of the MS-Windows data types listed in your notes:

BOOL, BOOLEAN: A boolean value, either TRUE or FALSE.

BYTE: An 8-bit unsigned integer, meaning that it can store values from 0 to 255.

CHAR: An 8-bit Windows ANSI character. ANSI characters are used in older Windows applications and are encoded using a variety of different character sets, depending on the language and region.

COORD: A structure that contains two WORD values, X and Y, which represent the coordinates of a point on the screen.

SYSTEMTIME: A structure that contains information about a date and time, including the year, month, day, hour, minute, second, and millisecond.

COLORREF: A 32-bit value used to represent a color.

DWORD: A 32-bit unsigned integer, meaning that it can store values from 0 to 4,294,967,295.

HANDLE: A handle is a reference to an object, such as a window, file, or memory region.

HFILE: A handle to a file opened by the OpenFile function.

INT: A 32-bit signed integer, meaning that it can store values from -2,147,483,648 to 2,147,483,647.

LONG: A 32-bit signed integer, the same as INT.

LPARAM: A message parameter used by window procedures and callback functions. LPARAM can be used to store any type of data, but it is typically used to store pointers to structures or other data structures.

LPCSTR: A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters.

LPCVOID: A pointer to a constant of any type.

LPSTR: A pointer to a null-terminated string of 8-bit Windows (ANSI) characters.

LPCTSTR: A pointer to a constant character string that is portable for Unicode and double-byte character sets. Unicode is a modern character encoding that can represent characters from all over the world. Double-byte character sets are used to represent characters in languages such as Chinese and Japanese.

LPTSTR: A pointer to a character string that is portable for Unicode and double-byte character sets.

LPVOID: A pointer to an unspecified type.

LRESULT: A 32-bit value returned from a window procedure or callback function.

SIZE_T: The maximum number of bytes to which a pointer can point.

UINT: A 32-bit unsigned integer, the same as DWORD.

WNDPROC: A pointer to a window procedure. A window procedure is a function that is responsible for handling messages sent to a window.

WORD: A 16-bit unsigned integer, meaning that it can store values from 0 to 65,535.

WPARAM: A 32-bit value passed as a parameter to a window procedure or callback function. WPARAM can be used to store any type of data, but it is typically used to store the message ID or other information about the message.

The SmallWin.inc include file contains structure definitions, data type definitions, and function prototypes for Win32 API programming. It is automatically included in MASM programs by the Irvine32.inc include file.

Structures Explained:

The **COORD structure** is used to store the coordinates of a point on the screen. It contains two WORD members, X and Y, which represent the horizontal and vertical coordinates of the point, respectively.

The **SYSTEMTIME structure** is used to store information about a date and time. It contains the following members:

- • **wYear:** The year.
- • **wMonth:** The month.
- • **wDayOfWeek:** The day of the week.

- • • **wDay:** The day of the month.
- • • **wHour:** The hour.
- • • **wMinute:** The minute.
- • • **wSecond:** The second.
- • • **wMilliseconds:** The millisecond.

Console handles

Console handles are 32-bit unsigned integers that uniquely identify console devices, such as the keyboard, display, and printer. They are used by Win32 console functions to perform input and output operations.

The three standard console handles are:

STD_INPUT_HANDLE: The standard input handle is used to read keyboard input.

STD_OUTPUT_HANDLE: The standard output handle is used to write to the console display.

STD_ERROR_HANDLE: The standard error handle is used to write error messages to the console display. To get a handle to a console device, you can use the **GetStdHandle function**. This function takes a console handle type as a parameter and returns a handle to the corresponding console device.

Once you have a handle to a console device, you can use it to perform input and output operations. For example, to read a character from the keyboard, you can use the `ReadConsole` function. This function takes a console input handle and a buffer as parameters and reads a specified number of characters from the console input buffer into the buffer.

To write a character to the console display, you can use the `WriteConsole` function. This function takes a console output handle, a buffer, and a number of characters to write as parameters and writes the specified number of characters from the buffer to the console display.

You can also use console handles to control the appearance and behavior of the console window. For example, to set the title of the console window, you can use the `SetConsoleTitle` function. This function takes a console window handle and a title string as parameters and sets the title of the console window to the specified string.

Console handles are an essential part of Win32 console programming. By understanding how to use console handles, you can develop powerful and efficient console-based applications.

The handles are:

AllocConsole

This function allocates a new console for the calling process. This is useful for applications that need to create their own console, such as console-based games or debugging tools.

CreateConsoleScreenBuffer

This function creates a new console screen buffer. A console screen buffer is a memory area that stores the text and color attributes for the console display.

ExitProcess

This function ends a process and all its threads. It is typically used to terminate an application when it is finished running or when an error occurs.

FillConsoleOutputAttribute

This function sets the text and background color attributes for a specified number of character cells. This can be used to change the appearance of text on the console display.

FillConsoleOutputCharacter

This function writes a character to the screen buffer a specified number of times. This can be used to fill a rectangular area of the console display with a single character.

FlushConsoleInputBuffer

This function flushes the console input buffer. The console input buffer is a memory area that stores keyboard input until it is read by an application. Flushing the console input buffer removes all unread input from the buffer.

FreeConsole

This function detaches the calling process from its console. This is useful for applications that need to run without a console, such as services or background tasks.

GenerateConsoleCtrlEvent

This function sends a specified signal to a console process group that shares the console associated with the calling process. This can be used to notify other applications that the calling process is terminating or that an event has occurred.

GetConsoleCP

This function retrieves the input code page used by the console associated with the calling process. The input code page is a table that maps character codes to characters.

GetConsoleCursorInfo

This function retrieves information about the size and visibility of the cursor for the specified console screen buffer.

GetConsoleMode

This function retrieves the current input mode of a console input buffer or the current output mode of a console screen buffer. The input and output modes control the behavior of the console input and output, respectively.

GetConsoleOutputCP

This function retrieves the output code page used by the console associated with the calling process. The output code page is a table that maps characters to character codes.

GetConsoleScreenBufferInfo

This function retrieves information about the specified console screen buffer.

GetConsoleTitle

This function retrieves the title bar string for the current console window.

GetConsoleWindow

This function retrieves the window handle used by the console associated with the calling process.

GetLargestConsoleWindowSize

This function retrieves the size of the largest possible console window.

GetNumberOfConsoleInputEvents

This function retrieves the number of unread input records in the console's input buffer.

GetNumberOfConsoleMouseButtons

This function retrieves the number of buttons on the mouse used by the current console.

GetStdHandle

This function retrieves a handle for the standard input, standard output, or standard error device. These handles are typically used by console applications to read keyboard input, write to the console display, and write error messages, respectively.

HandlerRoutine

This is an application-defined function that is used with the SetConsoleCtrlHandler function. The SetConsoleCtrlHandler function allows an application to specify a function to be called when the console receives certain signals, such as a close signal or a termination signal.

PeekConsoleInput

This function reads data from the specified console input buffer without removing it from the buffer. This can be used to check for keyboard input without actually reading it.

ReadConsole

This function reads character input from the console input buffer and removes it from the buffer. This is the most common way to read keyboard input in a console application.

ReadConsoleInput

This function reads data from a console input buffer and removes it from the buffer. This function is similar to the ReadConsole function, but it can also read mouse input and other types of input.

ReadConsoleOutput

This function reads character and color attribute data from a rectangular block of character cells in a console screen buffer. This can be used to read the text and appearance of a rectangular area of the console display.

ReadConsoleOutputAttribute

This function copies a specified number of foreground and background color attributes from consecutive cells of a console screen buffer. This can be used to read the color attributes of a rectangular area of the console display.

ReadConsoleOutputCharacter

This function copies a number of characters from consecutive cells of a console screen buffer. This can be used to read the text of a rectangular area of the console display.

ScrollConsoleScreenBuffer

This function moves a block of data in a screen buffer. This can be used to scroll the console display, or to move text or other data within the screen buffer.

SetConsoleActiveScreenBuffer

This function sets the specified screen buffer to be the currently displayed console screen buffer. This can be used to switch between different screen buffers, or to display a different screen buffer in a different console window.

SetConsoleCP

This function sets the input code page used by the console associated with the calling process. The input code page is a table that maps character codes to characters. This function can be used to change the language of the console input, or to support different character sets.

SetConsoleCtrlHandler

This function adds or removes an application-defined HandlerRoutine from the list of handler functions for the calling process. A HandlerRoutine is a function that is called when the console receives certain signals, such as a close signal or a termination signal. This function can be used to implement custom behavior when the console receives these signals.

SetConsoleCursorInfo

This function sets the size and visibility of the cursor for the specified console screen buffer. This function can be used to change the appearance of the cursor, or to hide the cursor altogether.

SetConsoleCursorPosition

This function sets the cursor position in the specified console screen buffer. This function can be used to move the cursor to a specific location on the console display.

SetConsoleMode

This function sets the input mode of a console's input buffer or the output mode of a console screen buffer. The input and output modes control the behavior of the console input and output, respectively. This function can be used to change the behavior of the console keyboard, mouse, and other input devices, or to change the appearance of the console display.

SetConsoleOutputCP

This function sets the output code page used by the console associated with the calling process. The output code page is a table that maps characters to character codes. This function can be used to change the language of the console output, or to support different character sets.

SetConsoleScreenBufferSize

This function changes the size of the specified console screen buffer. This function can be used to increase or decrease the size of the console display, or to accommodate different screen sizes.

SetConsoleTextAttribute

This function sets the foreground (text) and background color attributes of characters written to the screen buffer. This function can be used to change the appearance of text on the console display.

SetConsoleTitle

This function sets the title bar string for the current console window. This can be used to change the title of the console window, or to identify the console window in a list of windows.

SetConsoleWindowInfo

This function sets the current size and position of a console screen buffer's window. This function can be used to resize or move the console window, or to fit the console window to a specific screen area.

SetStdHandle

This function sets the handle for the standard input, standard output, or standard error device. These handles are typically used by console applications to read keyboard input, write to the console display, and write error messages, respectively.

WriteConsole

This function writes a character string to a console screen buffer beginning at the current cursor location. This is the most common way to write text to the console display.

WriteConsoleInput

This function writes data directly to the console input buffer. This function can be used to simulate keyboard input, or to send other types of input to the console.

WriteConsoleOutput

This function writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer. This function can be used to write text and color attribute data to a specific area of the console display.

WriteConsoleOutputAttribute

This function copies a number of foreground and background color attributes to consecutive cells of a console screen buffer. This function can be used to change the color attribute of a specific area of the console display.

WriteConsoleOutputCharacter

This function copies a number of characters to consecutive cells of a console screen buffer. This function can be used to write text to a specific area of the console display.

DISPLAYING A MESSAGEBOX

In Win32 console applications, you can use the MessageBoxA function to display a message box to the user. The MessageBoxA function takes four parameters:

hWnd: The handle to the window that owns the message box. If this parameter is NULL, the message box will be created as a top-level window.

lpText: A pointer to the text to display in the message box.

lpCaption: A pointer to the caption of the message box.

uType: A bit-mapped integer that specifies the type of message box to display. The uType parameter can be used to specify the buttons to display, the icon to display, and the default button.

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

Value	Integer Value	Description
MB_OK	0	Display a message box with an OK button.
MB_OKCANCEL	1	Display a message box with OK and Cancel buttons.
MB_YESNO	4	Display a message box with Yes and No buttons.
MB_YESNOCANCEL	3	Display a message box with Yes, No, and Cancel buttons.
MB_RETRYCANCEL	5	Display a message box with Retry and Cancel buttons.
MB_ABORTRETRYIGNORE	2	Display a message box with Abort, Retry, and Ignore buttons.
MB_CANCELTRYCONTINUE	6	Display a message box with Cancel, Try Again, and Continue buttons.
MB_ICONSTOP	16	Display a message box with a stop-sign icon.
MB_ICONQUESTION	32	Display a message box with a question-mark icon.
MB_ICONINFORMATION	64	Display a message box with an information-symbol icon.
MB_ICONEXCLAMATION	48	Display a message box with an exclamation-point icon.

The table describes various message box constants and their corresponding descriptions.

These constants are often used in programming to customize the appearance and behavior of message boxes, which are dialog boxes used to display information or request user input in applications.

Clearer Table:

Value	Description
MB_OK	Displays message box with OK button
MB_OKCANCEL	Displays message box with OK and Cancel buttons
MB_YESNO	Displays message box with Yes and No buttons
MB_YESNOCANCEL	Displays message box with Yes, No, and Cancel buttons
MB_RETRYCANCEL	Displays message box with Retry and Cancel buttons
MB_ABORTRETRYIGNORE	Displays message box with Abort, Retry, and Ignore buttons
MB_CANCELTRYCONTINUE	Displays message box with Cancel, Try Again, and Continue buttons
MB_ICONSTOP	Displays message box with stop-sign icon
MB_ICONQUESTION	Displays message box with question-mark icon
MB_ICONINFORMATION	Displays message box with information-symbol icon
MB_ICONEXCLAMATION	Displays message box with exclamation-point icon

Yes, the values listed in the table are often used as integer constants to specify the "uType" parameter when creating message boxes in programming.

The "uType" parameter is an integer value that determines the type and behavior of the message box.

You can use the uTypes in your programming language like:

```

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

```

Or

```

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

```

In this C program, we use different message box types, including MB_OK, MB_YESNO, and MB_ICONEXCLAMATION.

You can specify the desired message box type by passing the corresponding integer value as the third parameter to the MessageBoxW function.

The program also checks the user's response to the "Yes" and "No" buttons by examining the result returned by the function.

Or

```

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

```

In this code, we're using the integer values directly to specify the message box types.

For example, 0 corresponds to MB_OK, 4 corresponds to MB_YESNO, and 48 corresponds to MB_ICONEXCLAMATION.

You can use these values to create message boxes with the desired type and behavior in your C program.

Demonstration Program

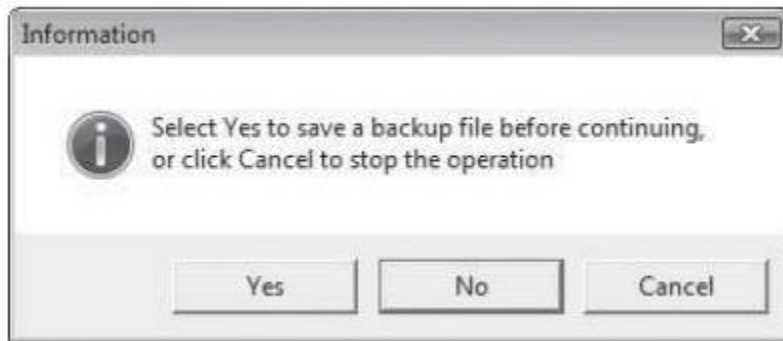
We will demonstrate a short program that demonstrates some capabilities of the MessageBoxA function. The first function call displays a warning message:



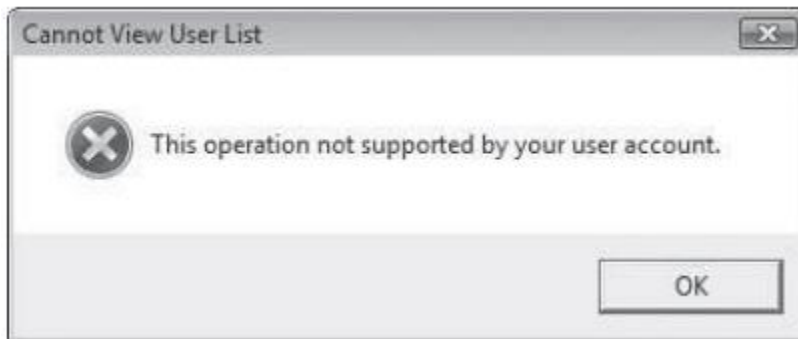
The second function call displays a question icon and Yes/No buttons. If the user selects the Yes button, the program could use the return value to select a course of action:



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)
INCLUDE Irvine32.inc
```

```
.data
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
```

```
; Display Exclamation icon with OK button
```

```
INVOKE MessageBox, NULL, ADDR warningMsg, ADDR captionW, MB_OK +
MB_ICONEXCLAMATION
```

```
; Display Question icon with Yes/No buttons
```

```
INVOKE MessageBox, NULL, ADDR questionMsg, ADDR captionQ, MB_YESNO +
MB_ICONQUESTION
```

```
; interpret the button clicked by the user
```

```
cmp eax,IDYES ; YES button clicked?
```

```
; Display Information icon with Yes/No/Cancel buttons
```

```
INVOKE MessageBox, NULL, ADDR infoMsg, ADDR captionC, MB_YESNOCANCEL +
MB_ICONINFORMATION + MB_DEFBUTTON2
```

```
; Display stop icon with OK button
```

```
INVOKE MessageBox, NULL, ADDR haltMsg, ADDR captionH, MB_OK + MB_ICONSTOP
```

```
exit
```

```
main ENDP
```

```
END main
```

The provided code is a demonstration program in assembly language for the Windows API. It showcases the use of the MessageBox function to create message boxes with different icons and button options. Here's a clear and concise explanation of the code:

The program starts with including the Irvine32 library, which provides various macros and functions for 32-bit assembly programming.

In the .data section, several strings are defined for different message boxes, including captions and messages.

These strings are null-terminated with a '0' byte at the end. The .code section contains the main procedure (main PROC) where the demonstration begins.

Four message boxes are created using the INVOKE macro, each with different options: A warning message box with an exclamation icon and an OK button.

A question message box with a question icon and Yes/No buttons. An information message box with an information icon and Yes/No/Cancel buttons, with the default button set to No.

A stop message box with a stop icon and an OK button. After displaying each message box, the program checks the result (in the eax register) to determine which button was clicked by the user.

It compares the result with predefined constants like IDYES to check if the "Yes" button was clicked.

Finally, the program exits after displaying all the message boxes.

In summary, this program demonstrates how to create message boxes with different icons and button options using the MessageBox function from the Windows API.

It also shows how to interpret the user's response by comparing the result with predefined constants for button IDs.

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
```

Console Input Buffer: In Win32 console programming, there exists an input buffer that stores input event records. These input events include keystrokes, mouse movements, and mouse-button clicks.

High-level input functions like `ReadConsole` process this input data and return a stream of characters to the program.

ReadConsole Function: The `ReadConsole` function is a Win32 API function used to read text input from the console and store it in a buffer.

It takes several parameters, including the console input handle, a pointer to a character buffer, the number of characters to read, a pointer to store the count of characters read, and a reserved parameter.

The provided code is an example program that demonstrates how to use the `ReadConsole` function to read characters entered by the user in a console application.

It starts by defining the size of the buffer (BufSize) and declaring the necessary data variables, including a buffer for storing the input, a handle for standard input (stdinHandle), and a variable for the number of bytes read (bytesRead).

In the main procedure, it retrieves the standard input handle using the GetStdHandle function, which returns a handle to the standard input.

It then calls the ReadConsole function to read input from the user.

The function parameters include the standard input handle, the buffer for storing input, the maximum number of characters to read, a pointer to store the count of characters read, and a value of 0 for the reserved parameter.

After reading the input, it displays the content of the buffer using the DumpMem function, which is part of the Irvine32 library.

The DumpMem function is used to display the buffer's content in both hexadecimal and ASCII representations.

The program can read and display user input, including any end-of-line characters (0Dh and 0Ah) inserted when the user presses the Enter key.

In summary, this code demonstrates how to read and display user input from the console using the ReadConsole function in a Win32 console application. It provides an example of handling console input in assembly language.

CHECKING FOR ERRORS

In Windows API programming, it's crucial to check for errors when using various API functions. If an API function returns an error value (typically NULL), you can call the GetLastError API function to obtain more information about the error. GetLastError returns a 32-bit integer error code in the EAX register.

GetLastError Function:

GetLastError is used to retrieve error information after a Windows API function call. It returns a 32-bit integer error code in the EAX register, which can be used to identify the specific error.

If a function returns NULL or an error code, you can call GetLastError to get more details about what went wrong.

FormatMessage Function:

After obtaining the error code from GetLastError, you might want to retrieve a human-readable error message to better understand the error.

FormatMessage is used for this purpose. It formats a message based on the error code. Its parameters are:

dwFlags: Formatting options, specifying how to interpret the lpSource parameter. Recommended values include FORMAT_MESSAGE_ALLOCATE_BUFFER and FORMAT_MESSAGE_FROM_SYSTEM.

lpSource: Location of the message definition. For system error messages, set it to NULL (0).

dwMsgID: The error code obtained from GetLastError.

dwLanguageID: Language identifier. Set to zero for a language-neutral or user's default locale message.

lpBuffer (output parameter): Pointer to a buffer that receives the null-terminated message string. If FORMAT_MESSAGE_ALLOCATE_BUFFER is used, the buffer is allocated automatically.

nSize: Buffer size, which can be set to 0 if using the recommended dwFlags options. va_list: Pointer to an array of values that can be inserted in a formatted message. Not used for error messages.

Sample Use of GetLastError and FormatMessage:

First, you call GetLastError to obtain the error code. Then, you invoke FormatMessage to retrieve the corresponding error message based on the error code. Finally, you can display or handle the error message as needed.

LocalFree Function:

After using FormatMessage to obtain the error message, it's important to release the storage allocated by FormatMessage.

You can use LocalFree for this purpose. The code provided demonstrates the use of GetLastError and FormatMessage to obtain and display error messages in a Windows API program.

It checks for errors, retrieves the error message, and frees allocated memory after use.

Note: Error codes and messages are essential for debugging and providing meaningful feedback to users when errors occur in Windows API applications.

```
.data
messageId DWORD ?
pErrorMsg DWORD ? ; points to error message
```

```
.code
call GetLastError
```

mov messageId, eax

INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER +
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, 0, ADDR pErrorMsg, 0, NULL

; After calling FormatMessage, you can use the error message in pErrorMsg

; ... (perform error handling or display the error message as needed)

; Don't forget to free the allocated memory when done

INVOKE LocalFree, pErrorMsg

This code checks for errors by calling GetLastError, retrieves the error message using FormatMessage, and then handles or displays the error message. Finally, it frees the allocated memory using LocalFree.

.data

messageId DWORD ?

pErrorMsg DWORD ? ; points to error message

.code

call GetLastError

mov messageId, eax

INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER +
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, 0, ADDR pErrorMsg, 0, NULL

; After calling FormatMessage, you can use the error message in pErrorMsg

; ... (perform error handling or display the error message as needed)

; Don't forget to free the allocated memory when done

INVOKE LocalFree, pErrorMsg

WriteWindowsMsg PROC USES eax edx

; Displays a string containing the most recent error

; generated by MS-Windows.

; Receives: nothing

; Returns: nothing

.data

WriteWindowsMsg_1 BYTE "Error ",0

WriteWindowsMsg_2 BYTE ": ",0

pErrorMsg DWORD ?

; points to the error message

messageId DWORD ?

```

.code
; Get the most recent error code
call GetLastError
mov messageId, eax
; Display the error number as "Error X: "
mov edx, OFFSET WriteWindowsMsg_1
call WriteString
call WriteDec
mov edx, OFFSET WriteWindowsMsg_2
call WriteString
; Get the corresponding error message string
invoke FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, 0, ADDR pErrorMsg, 0, 0
; Display the error message generated by MS-Windows
mov edx, pErrorMsg
call WriteString
; Free the error message string
invoke LocalFree, pErrorMsg
ret
WriteWindowsMsg ENDP

```

Explanation:

The WriteWindowsMsg procedure is designed to display error messages generated by the MS-Windows operating system.

It starts by calling GetLastError to obtain the most recent error code and stores it in the messageId variable.

The procedure then displays the error number in the format "Error X: " where X is the error code.

It uses the WriteString procedure to display the "Error " and the WriteDec procedure to display the error code.

After that, it retrieves the corresponding error message string using FormatMessage. The FORMAT_MESSAGE_ALLOCATE_BUFFER and FORMAT_MESSAGE_FROM_SYSTEM flags are used to allocate memory for the message and obtain it from the system.

The obtained error message is displayed using WriteString.

Finally, it frees the memory allocated for the error message using LocalFree.

This procedure is a convenient way to retrieve and display error messages when working with Windows API functions, making it easier to diagnose issues in your applications.

SINGLE CHARACTER INPUT

Single-Character Input and Irvine32 Keyboard Procedures

In console mode on MS-Windows, handling single-character input involves dealing with the keyboard device driver, scan codes, virtual-key codes, and the message queue. Here's an explanation of the process and the relevant Irvine32 keyboard procedures:

Keyboard Input Process:

MS-Windows provides a device driver for the installed keyboard. When a key is pressed, it sends an 8-bit scan code to the computer's keyboard port.

Upon releasing the key, a second scan code is transmitted. MS-Windows translates these scan codes into 16-bit virtual-key codes, which are device-independent values that identify the key's purpose.

A message containing the scan code, virtual-key code, and related information is created by MS-Windows and placed in the message queue.

The message eventually reaches the currently executing program thread, identified by the console input handle.

Irvine32 Keyboard Procedures:

The Irvine32 library provides two related procedures for handling keyboard input: `ReadChar` and `ReadKey`.

`ReadChar` waits for an ASCII character to be typed at the keyboard and returns the character in the AL register.

`ReadKey` performs a no-wait keyboard check. If no key is waiting in the console input buffer, it sets the Zero flag.

If a key is found, the Zero flag is clear, and AL contains either zero or an ASCII code. The upper halves of EAX and EDX are overwritten.

Using ReadKey and Control Key State:

In `ReadKey`, if AL contains zero, the user may have pressed a special key (e.g., function key, cursor arrow).

AH register contains the keyboard scan code, which can be matched to a list of keyboard keys. DX contains the virtual-key code.

EBX contains state information about the states of the keyboard control keys.

Control Key State Values:

- **`CAPSLOCK_ON`**: The CAPS LOCK light is on.

- **ENHANCED_KEY:** The key is enhanced.
- **LEFT_ALT_PRESSED:** The left ALT key is pressed. **LEFT_CTRL_PRESSED:** The left CTRL key is pressed.
- **NUMLOCK_ON:** The NUM LOCK light is on.
- **RIGHT_ALT_PRESSED:** The right ALT key is pressed.
- **RIGHT_CTRL_PRESSED:** The right CTRL key is pressed.
- **SCROLLLOCK_ON:** The SCROLL LOCK light is on.
- **SHIFT_PRESSED:** The SHIFT key is pressed.

You can use these control key state values to determine the state of control keys while processing keyboard input.

The **ReadChar** and **ReadKey** procedures in the Irvine32 library simplify handling keyboard input in your assembly programs, making it easier to respond to user interactions.

Testing Keyboard Input with ReadKey and GetKeyState

This section covers testing keyboard input using ReadKey and GetKeyState, including a program that reports the state of the CapsLock key and another program that checks the state of the NumLock and Left Shift keys.

Testing Keyboard Input with ReadKey:

The program tests ReadKey by waiting for a keypress and then reporting the state of the CapsLock key.

A **delay factor** is included when calling ReadKey to allow MS-Windows to process its message loop.

If ReadKey returns a non-zero value (a keypress has occurred), the program tests the value of EBX using the **CAPSLOCK_ON constant** to check the state of the CapsLock key.

It then displays a message indicating whether CapsLock is ON or OFF.

GetKeyState for Keyboard State Testing:

The GetKeyState API function allows you to test the state of individual keyboard keys. You pass it a virtual key value, like those identified in Table 11-4.

Table 11-4 Testing Keys with GetKeyState.

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	VK_LMENU	15
Right Menu	VK_RMENU	15

It returns a value in EAX, and you need to test the value to determine the state of the key.

The program demonstrates using GetKeyState to check the state of the NumLock and Left Shift keys:

It calls GetKeyState with VK_NUMLOCK and checks if the lowest bit (bit 0) of AL is set.

If it is set, it indicates that NumLock is ON.

It then calls GetKeyState with VK_LSHIFT and checks the high bit (bit 31) of EAX to determine if the Left Shift key is currently pressed.

Depending on the test results, it displays appropriate messages to report the state of the keys.

```

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

```

Program 2:

```

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

```

CONSOLE OUTPUT

It appears you're looking for an explanation of the WriteConsole function and some associated data structures. Below, I'll provide explanations for WriteConsole and the COORD and SMALL_RECT structures.

WriteConsole Function:

WriteConsole is a Win32 function used to write a string to the console window at the current cursor position. It is used for console output. Here's a breakdown of its parameters:

- **hConsoleOutput:** This is the handle to the console output stream.
- **lpBuffer:** It's a pointer to the array of characters you want to write.
nNumberOfCharsToWrite: This parameter holds the length of the array you want to write.
- **lpNumberOfCharsWritten:** It's a pointer to an integer that will receive the number of characters written when the function returns.
- **lpReserved:** This parameter is not used, so you can set it to zero.

WriteConsole writes the string and advances the cursor just past the last character written. It can handle standard ASCII control characters like tabs, carriage returns, and line feeds, and the string doesn't need to be null-terminated.

COORD Structure:

The COORD structure is used in various Win32 console functions. It represents the coordinates of a character cell in the console screen buffer.

The origin of this coordinate system is at the top left cell of the console screen. The structure has two fields:

- **X:** This field is of type WORD and represents the X-coordinate.
- **Y:** This field is also of type WORD and represents the Y-coordinate.

SMALL_RECT Structure:

The SMALL_RECT structure is another data structure used in Win32 console functions. It specifies a rectangular region by defining the upper left and lower right corners of the rectangle within the console window.

It's useful for specifying character cells in the console window. The structure has the following fields:

- **Left:** This field is of type WORD and represents the left coordinate of the rectangle.
- **Top:** This field, also of type WORD, represents the top coordinate.
- **Right:** This field, again of type WORD, represents the right coordinate.
- **Bottom:** This field, once more of type WORD, represents the bottom coordinate.

These data structures are used in various console-related Win32 functions to manage and manipulate console windows. The WriteConsole function is particularly useful for writing content to the console.

Example 1:

WriteConsole function

```

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

```

The provided program, Console1.asm, is a simple example that demonstrates the use of the GetStdHandle, WriteConsole, and ExitProcess functions in a Win32 console application. Here's a breakdown of the code:

The .data section begins by defining an endl constant that represents the end-of-line sequence (carriage return and line feed).

A message is defined using the message label. It contains a multiline text string that the program will write to the console. The messageSize variable is used to store the size of the message string.

The consoleHandle variable is declared as a HANDLE and initialized to 0. This variable will hold the handle to the standard output device (console).

bytesWritten is declared as a DWORD and will be used to store the number of bytes written by the WriteConsole function.

The .code section contains the main procedure.

INVOKE GetStdHandle, STD_OUTPUT_HANDLE is used to obtain the handle to the standard output (the console). The obtained handle is stored in the consoleHandle variable.

The **INVOKE WriteConsole function** is called to write the message string to the console. It takes the following parameters:

- **consoleHandle:** The handle to the console output.
- **ADDR message:** A pointer to the message string.
- **messageSize:** The length of the message string.
- **ADDR bytesWritten:** A pointer to a variable that will receive the number of bytes written.
- **0:** An unused parameter.
- Finally, **INVOKE ExitProcess, 0** is called to exit the program.

When you run this program, it will write the message to the console, and the console window will display the content of the message string.

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

```

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

```

The WriteConsoleOutputCharacter function in Win32 allows you to copy an array of characters to consecutive cells in the console screen buffer at a specified location. Here's a breakdown of its parameters:

- **hConsoleOutput:** This is the handle to the console output. It specifies the console screen buffer where you want to write the characters.
- **lpCharacter:** A pointer to the buffer containing the characters you want to write.
- **nLength:** The size of the buffer, indicating the number of characters to write from the buffer.
- **dwWriteCoord:** This parameter specifies the coordinates of the first cell where you want to start writing. It is of type COORD, which holds X (column) and Y (row) coordinates.
- **lpNumberOfCharsWritten:** A pointer to a DWORD that will receive the count of characters written by the function.

This function writes the characters to the console screen buffer, and if the text reaches the end of a line, it wraps around to the next line.

It's important to note that this function doesn't change the attribute values in the screen buffer, and it ignores ASCII control codes such as tab, carriage return, and line feed.

If the function is successful, it returns a non-zero value, and the number of characters written is stored in lpNumberOfCharsWritten. If it fails, it returns zero.

You can use WriteConsoleOutputCharacter to write text directly to the console, which can be useful for more advanced console applications.

FILE HANDLING

CreateFile Function

The CreateFile function is used to create a new file or open an existing file. It returns a handle to the open file if successful, otherwise, it returns INVALID_HANDLE_VALUE.

```
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:

- • **lpFilename:** Points to the null-terminated string containing the filename.
- • **dwDesiredAccess:** Specifies the type of access (read, write, read/write, device query, etc.).
- • **dwShareMode:** Controls how multiple programs can access the file while it's open.
- • **lpSecurityAttributes:** Points to a security structure controlling security rights.
- • **dwCreationDisposition:** Specifies what to do when the file exists or doesn't exist.
- • **dwFlagsAndAttributes:** Contains bit flags specifying file attributes like archive, encrypted, hidden, etc.
- • **hTemplateFile:** An optional handle to a template file for attributes and extended attributes.

dwDesiredAccess Parameter Options:

The dwDesiredAccess parameter specifies the type of access to the file. You can choose from the following options or specific flag values:

- **0:** Device query access, to check device attributes or file existence.
- **GENERIC_READ:** Read access for reading from the file.
- **GENERIC_WRITE:** Write access for writing to the file.

dwCreationDisposition Parameter Options:

The `dwCreationDisposition` parameter specifies actions on existing and non-existing files.

Choose one of the following options:

- **CREATE_NEW:** Creates a new file, fails if it already exists.
- **CREATE_ALWAYS:** Creates a new file, overwrites if it exists.
- **OPEN_EXISTING:** Opens an existing file, fails if it doesn't exist.
- **OPEN_ALWAYS:** Opens the file if it exists, creates if it doesn't.
- **TRUNCATE_EXISTING:** Opens the file and truncates it to size zero, fails if it doesn't exist.

=====

Let's delve deeper into the `CreateFile` function and provide some code examples.

lpFilename: This is the path to the file you want to create or open. It can be a fully qualified filename (including the drive and path) or just the filename. For example, "C:\myfolder\myfile.txt" or "myfile.txt".

dwDesiredAccess: Specifies the type of access to the file. You can use a combination of these flags (bitwise OR) to specify the desired access:

- **GENERIC_READ:** Read access.
- **GENERIC_WRITE:** Write access.

0: Device query access (useful for checking device attributes or file existence).

dwShareMode: This parameter controls how other processes can access the file while it is open. It can take one or a combination of these flags (bitwise OR):

- **FILE_SHARE_READ:** Other processes can read the file.
- **FILE_SHARE_WRITE:** Other processes can write to the file.
- **0:** No sharing allowed.

lpSecurityAttributes: This parameter allows you to specify a security structure, but you can usually set it to `NULL` if you don't need to set specific security attributes.

dwCreationDisposition: Specifies what to do when the file exists or doesn't exist. You can choose from these options:

- **CREATE_NEW:** Creates a new file. If the file already exists, the function fails.
- **CREATE_ALWAYS:** Creates a new file. If it exists, it overwrites it.
- **OPEN_EXISTING:** Opens an existing file. If it doesn't exist, the function fails.
- **OPEN_ALWAYS:** Opens the file if it exists or creates it if it doesn't.

- **TRUNCATE_EXISTING:** Opens the file and truncates it to size zero. Fails if the file doesn't exist.

dwFlagsAndAttributes: This parameter allows you to set various file attributes. Common attributes include FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_ARCHIVE, FILE_ATTRIBUTE_HIDDEN, etc.

hTemplateFile: You can typically set this to NULL. It's an optional handle to a template file that can supply file attributes and extended attributes for the file being created.

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

```

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

```

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

Combined Code:

Here's a single code example that demonstrates the creation and opening of files using CreateFile, and reading from a file using ReadFile. It uses the file "mydata.txt" for illustration:

```

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

```

CreateFile Function:

The CreateFile function is used to either create a new file or open an existing file. It returns a handle to the open file if successful or INVALID_HANDLE_VALUE if it fails. Several parameters determine how the file is accessed and what happens in various scenarios.

lpFileName: This parameter points to a null-terminated string representing the file's name and location.

dwDesiredAccess: Specifies the type of access, such as read, write, or both. It uses flags like GENERIC_READ and GENERIC_WRITE.

dwShareMode: It controls the sharing of the file among multiple programs. You can specify sharing options using constants like FILE_SHARE_READ and FILE_SHARE_WRITE.

lpSecurityAttributes: This parameter can point to a security structure that controls security rights, but for most use cases, it's set to NULL.

dwCreationDisposition: Determines what happens when the file already exists or not. It uses options like CREATE_NEW, CREATE_ALWAYS, OPEN_EXISTING, etc.

dwFlagsAndAttributes: Contains attributes that define the file, like being hidden or read-only, and can be a combination of various attribute flags.

hTemplateFile: Optional and is used for specifying a template file that provides attributes and extended attributes for the new file.

File Access and Sharing:

dwDesiredAccess defines the type of access you want. GENERIC_READ allows reading, GENERIC_WRITE permits writing, and you can combine them for both read and write access.

dwShareMode controls how the file can be shared among different programs. It includes options like FILE_SHARE_READ and FILE_SHARE_WRITE, which determine whether other processes can read or write to the file simultaneously.

Creation and Opening Scenarios:

dwCreationDisposition specifies what should happen when opening a file: CREATE_NEW: Creates a new file, failing if it already exists. CREATE_ALWAYS: Creates a new file, overwriting an existing one. OPEN_EXISTING: Opens an existing file. OPEN_ALWAYS: Opens the file if it exists or creates a new one if it doesn't. TRUNCATE_EXISTING: Opens the file and truncates it to size zero. Attributes:

dwFlagsAndAttributes allows you to set file attributes like FILE_ATTRIBUTE_ARCHIVE, FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_NORMAL, and FILE_ATTRIBUTE_READONLY. CloseHandle Function:

After working with a file, it's crucial to close the handle using the CloseHandle function. This releases system resources and ensures data integrity.

ReadFile Function:

ReadFile is used to read data from a file. It requires the file handle, a buffer to store the data, the number of bytes to read, and a pointer to a variable that will hold the number of bytes actually read.

Synchronous reading is achieved by setting lpOverlapped to NULL. In practice, you would use these functions in a sequence, like opening a file, reading or writing data, and then closing the file handle to ensure proper file handling.

The code example provided earlier demonstrates a simple file creation, writing, and reading scenario, where you can see these functions in action.

WRITEFILE

WriteFile Function:

The WriteFile function is used to write data to a file or an output handle. The handle can represent a file or another output destination like the screen buffer.

The function writes data to the file starting at the position indicated by the file's internal position pointer.

After the write operation is completed, the file's position pointer is adjusted by the number of bytes actually written.

hFile: This is the handle to the file or output destination where the data should be written.

lpBuffer: It's a pointer to the buffer containing the data you want to write.

nNumberOfBytesToWrite: Specifies how many bytes should be written to the file.

lpNumberOfBytesWritten: A pointer to an integer that will hold the number of bytes actually written after the operation is completed.

lpOverlapped: This should be set to NULL for synchronous operation. It's used for asynchronous operations. The return value is zero if the function fails, and it's a non-zero value if the write operation is successful.

SetFilePointer Function:

The SetFilePointer function is used to move the position pointer of an open file. This function is handy for appending data to a file or for performing random-access record processing. It's often used to navigate within a file.

hFile: The file handle represents the file you want to move the pointer within.

lDistanceToMove: This is the number of bytes you want to move the pointer. It can be positive or negative, allowing you to move forward or backward within the file.

lpDistanceToMoveHigh: This is a pointer to a variable that contains the upper 32 bits of the distance. It's used for handling large file sizes, and if it's set to NULL, only the value in lDistanceToMove is considered.

dwMoveMethod: Specifies the starting point for moving the file pointer and can take one of three values: FILE_BEGIN (absolute file positioning), FILE_CURRENT (relative to the current file position), and FILE_END (relative to the end of the file).

For example, to prepare to append data to the end of a file, you can use FILE_END as the move method:

```

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

```

These functions are crucial for managing file access and data writing in Windows programming. They are often used in sequence, with SetFilePointer positioning the file pointer to the desired location, and WriteFile writing data to that location. Proper usage of these functions ensures efficient file manipulation in Windows applications.

```

417 ;-----
418 ; CreateOutputFile PROC
419 ;
420 ; Creates a new file and opens it in output mode.
421 ;
422 ; Receives: EDX points to the filename.
423 ;
424 ; Returns: If the file was created successfully, EAX
425 ; contains a valid file handle. Otherwise, EAX
426 ; equals INVALID_HANDLE_VALUE.
427 ;
428 ;-----
429 INVOKE CreateFile,
430     edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
431     CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
432 ret
433 CreateOutputFile ENDP

```

```
435 ;-----
436 ; OpenFile PROC
437 ;
438 ; Opens a new text file and opens for input.
439 ;
440 ; Receives: EDX points to the filename.
441 ;
442 ; Returns: If the file was opened successfully, EAX
443 ; contains a valid file handle. Otherwise, EAX equals
444 ; INVALID_HANDLE_VALUE.
445 ;
446 ;-----
447 INVOKE CreateFile,
448     edx, GENERIC_READ, DO_NOT_SHARE, NULL,
449     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
450 ret
451 OpenFile ENDP
```

```

453 ;-----
454 ; WriteToFile PROC
455 ;
456 ; Writes a buffer to an output file.
457 ;
458 ; Receives: EAX = file handle, EDX = buffer offset,
459 ; ECX = number of bytes to write
460 ;
461 ; Returns: EAX = number of bytes written to the file.
462 ; If the value returned in EAX is less than the
463 ; argument passed in ECX, an error likely occurred.
464 ;
465 ;-----
466 .data
467 WriteToFile_1 DWORD ?
468 ; number of bytes written
469 .code
470 INVOKE WriteFile,
471     eax,      ; file handle
472     edx,      ; buffer pointer
473     ecx,      ; number of bytes to write
474     ADDR WriteToFile_1, ; number of bytes written
475     0         ; overlapped execution flag
476 mov eax, WriteToFile_1 ; return value
477 ret
478 WriteToFile ENDP

```

```

480 ;-----
481 ; ReadFromFile PROC
482 ;
483 ; Reads an input file into a buffer.
484 ;
485 ; Receives: EAX = file handle, EDX = buffer offset,
486 ; ECX = number of bytes to read
487 ;
488 ; Returns: If CF = 0, EAX = number of bytes read; if
489 ; CF = 1, EAX contains the system error code returned
490 ; by the GetLastError Win32 API function.
491 ;
492 ;-----
493 .data
494 ReadFromFile_1 DWORD ?
495 ; number of bytes read
496 .code
497 INVOKE ReadFile,
498     eax,      ; file handle
499     edx,      ; buffer pointer
500     ecx,      ; max bytes to read
501     ADDR ReadFromFile_1, ; number of bytes read
502     0         ; overlapped execution flag
503 mov eax, ReadFromFile_1
504 ret
505 ReadFromFile ENDP

507 ;-----
508 ; CloseFile PROC
509 ;
510 ; Closes a file using its handle as an identifier.
511 ;
512 ; Receives: EAX = file handle
513 ;
514 ; Returns: EAX = nonzero if the file is successfully closed.
515 ;
516 ;-----
517 INVOKE CloseHandle, eax
518 ret
519 CloseFile ENDP

```

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

```

524 ; Creating a File (CreateFile.asm)
525 INCLUDE Irvine32.inc
526 BUFFER_SIZE = 501
527
528 .data
529 buffer BYTE BUFFER_SIZE DUP(?)
530 filename BYTE "output.txt",0
531 fileHandle HANDLE ?
532 stringLength DWORD ?
533 bytesWritten DWORD ?
534 str1 BYTE "Cannot create file",0dh,0ah,0
535 str2 BYTE "Bytes written to file [output.txt]:",0
536 str3 BYTE "Enter up to 500 characters and press [Enter]: ",0dh,0ah,0
537
538 .code
539 main PROC
540     ; Create a new text file.
541     mov edx, OFFSET filename      ; Load the address of the filename.
542     call CreateOutputFile        ; Call the CreateOutputFile procedure.
543     mov fileHandle, eax          ; Store the file handle in fileHandle.
544
545     ; Check for errors.
546     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
547     jne file_ok                  ; If not equal, jump to file_ok.
548
549     ; If there's an error, display the error message and exit.
550     mov edx, OFFSET str1         ; Load the address of the error message.
551     call WriteString              ; Call WriteString to display the error message.
552     jmp quit                     ; Jump to quit to exit.

```

```

554 file_ok:
555     ; Ask the user to input a string.
556     mov edx, OFFSET str3      ; Load the address of the input prompt.
557     call WriteString          ; Call WriteString to display the input prompt.
558
559     mov ecx, BUFFER_SIZE      ; Load the maximum buffer size.
560
561     ; Input a string.
562     mov edx, OFFSET buffer    ; Load the address of the buffer.
563     call ReadString           ; Call ReadString to get user input.
564     mov stringLength, eax      ; Store the length of the entered string.
565     ; Write the buffer to the output file.
566     mov eax, fileHandle       ; Load the file handle.
567     mov edx, OFFSET buffer    ; Load the address of the buffer.
568     mov ecx, stringLength     ; Load the length of the string.
569     call WriteToFile          ; Call WriteToFile to write to the file.
570     mov bytesWritten, eax     ; Store the number of bytes written.
571     ; Close the file.
572     call CloseFile            ; Call CloseFile to close the file.
573     ; Display the return value.
574     mov edx, OFFSET str2      ; Load the address of the output message.
575     call WriteString          ; Call WriteString to display the message.
576     mov eax, bytesWritten     ; Load the number of bytes written.
577     call WriteDec             ; Call WriteDec to display the value.
578     call Crlf                 ; Call Crlf to add a new line.
579 quit:
580     exit
581 main ENDP
582 END main

```

That's the second program.

Let's try another program:

```

587 ; Reading a File (ReadFile.asm)
588 ; Opens, reads, and displays a text file using
589 ; procedures from Irvine32.lib.
590 INCLUDE Irvine32.inc
591 INCLUDE macros.inc
592 BUFFER_SIZE = 5000
593
594 .data
595 buffer BYTE BUFFER_SIZE DUP(?)
596 filename BYTE 80 DUP(0)
597 fileHandle HANDLE ?
598
599 .code
600 main PROC
601     ; Let the user input a filename.
602     mWrite "Enter an input filename: " ; Display the input prompt.
603     mov edx, OFFSET filename ; Load the address of the filename.
604     mov ecx, SIZEOF filename ; Load the size of the filename.
605     call ReadString ; Call ReadString to get user input.
606
607     ; Open the file for input.
608     mov edx, OFFSET filename ; Load the address of the filename.
609     call OpenInputFile ; Call OpenInputFile to open the file.
610     mov fileHandle, eax ; Store the file handle in fileHandle.
611
612     ; Check for errors when opening the file.
613     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
614     jne file_ok ; If not equal, jump to file_ok.

```

```

616     ; If there's an error, display the error message and exit.
617     mWrite <"Cannot open file", 0dh, 0ah> ; Display the error message.
618     jmp quit ; Jump to quit to exit.
619
620 file_ok:
621     ; Read the file into a buffer.
622     mov edx, OFFSET buffer ; Load the address of the buffer.
623     mov ecx, BUFFER_SIZE ; Load the buffer size.
624     call ReadFromFile ; Call ReadFromFile to read the file.
625
626     jnc check_buffer_size ; If no error, jump to check_buffer_size.
627
628     ; If there's an error, display an error message.
629     mWrite "Error reading file. " ; Display the error message.
630     call WriteWindowsMsg ; Call WriteWindowsMsg to display the Windows error message.
631     jmp close_file ; Jump to close_file to close the file.
632
633 check_buffer_size:
634     cmp eax, BUFFER_SIZE ; Compare the result to BUFFER_SIZE.
635     jb buf_size_ok ; If less, jump to buf_size_ok.
636
637     ; If the buffer is too small for the file, display an error message and exit.
638     mWrite <"Error: Buffer too small for the file", 0dh, 0ah> ; Display the error message.
639     jmp quit ; Jump to quit to exit.
640
641 buf_size_ok:
642     mov buffer[eax], 0 ; Insert a null terminator.
643     mWrite "File size: " ; Display a message about the file size.
644     call WriteDec ; Call WriteDec to display the file size.
645     call CrLf ; Call CrLf to add a new line.
646
647     ; Display the buffer.
648     mWrite <"Buffer:", 0dh, 0ah, 0dh, 0ah> ; Display the buffer message.
649     mov edx, OFFSET buffer ; Load the address of the buffer.
650     call WriteString ; Call WriteString to display the buffer.
651     call CrLf ; Call CrLf to add a new line.
652
653 close_file:
654     mov eax, fileHandle ; Load the file handle.
655     call CloseFile ; Call CloseFile to close the file.
656
657 quit:
658     exit ; Exit the program.
659 main ENDP
660
661 END main

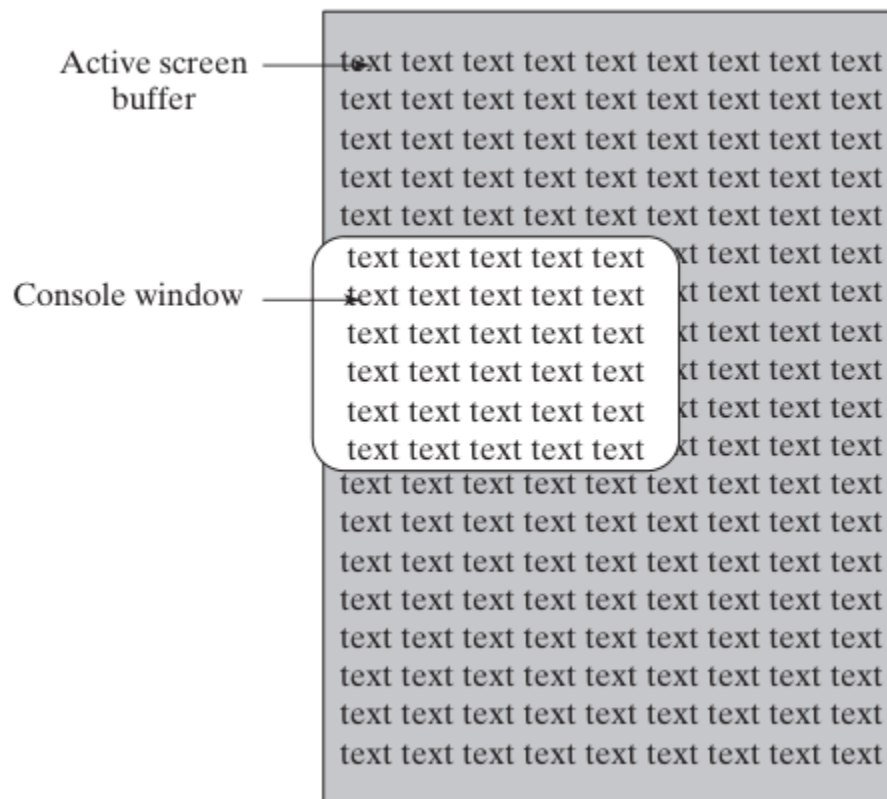
```

That's the 3rd program.

CONSOLE WINDOW MANIPULATION

I'll simplify the notes and provide commented code for each of the functions:

Screen buffer and console window.



The image you sent shows a screen buffer and console window. The screen buffer is a memory area that stores the text and color attributes for the console display. The console window is the window that displays the console buffer.

To manipulate the screen buffer in assembly WinAPI, you can use the following functions:

WriteConsoleOutput(): Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer.

ReadConsoleOutput(): Reads character and color attribute data from a specified rectangular block of character cells in a console screen buffer.

SetConsoleCursorPosition(): Sets the cursor position in the specified console screen buffer. The following code shows an example of how to use the WriteConsoleOutput function to write text to the screen buffer:

```

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

```

This code will write the character 'A' to the screen buffer at the top-left corner of the console window.

You can use the ReadConsoleOutput function to read text from the screen buffer. For example, the following code shows how to read a single character from the screen buffer:

```

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

```

This code will read a single character from the screen buffer at the top-left corner of the console window.

You can use the `SetConsoleCursorPosition` function to set the cursor position in the screen buffer. For example, the following code shows how to set the cursor position to the middle of the console window:

```

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

```

This code will set the cursor position to the middle of the console window.

SetConsoleTitle, GetConsoleScreenBufferInfo, and SetConsoleWindowInfo.

```

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

```

This code demonstrates how to change the console window's title using the SetConsoleTitle function.

```

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

```

This code demonstrates how to use the **SetConsoleWindowInfo** function to set the size and position of the console window relative to the screen buffer. The windowRect structure defines the new window dimensions and position.

CONSOLE_SCREEN_BUFFER_INFO structure.

Watch 1		
Name	Value	Type
consoleInfo	{dwSize={X=0x0078 Y=0x0032 } dwCursorPosition=	CONSOLE_SCREEN_BUFFER_INFO
dwSize	{X=0x0078 Y=0x0032 }	COORD
X	0x0078	unsigned short
Y	0x0032	unsigned short
dwCursorPosition	{X=0x0014 Y=0x0005 }	COORD
X	0x0014	unsigned short
Y	0x0005	unsigned short
wAttributes	0x0007	unsigned short
srWindow	{Left=0x0000 Top=0x0000 Right=0x004f ...}	SMALL_RECT
Left	0x0000	unsigned short
Top	0x0000	unsigned short
Right	0x004f	unsigned short
Bottom	0x0018	unsigned short
dwMaximumWindowSize	{X=0x0078 Y=0x0032 }	COORD
X	0x0078	unsigned short
Y	0x0032	unsigned short

I'll provide a simplified and commented version of the Scroll.asm program:

```

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

```

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

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

Another example:

```
819 INCLUDE Irvine32.inc
820
821 .data
822 consoleInfo CONSOLE_CURSOR_INFO <25, 1> ; Default cursor info
823 outHandle HANDLE 0
824 coord COORD <10, 10> ; New cursor position
825
826 .code
827 main PROC
828     ; Get the standard output handle
829     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
830     mov outHandle, eax
831
832     ; Get the current cursor information
833     INVOKE GetConsoleCursorInfo, outHandle, ADDR consoleInfo
834
835     ; Display the current cursor size and visibility
836     mov eax, consoleInfo.dwSize
837     call WriteDec
838     call WriteString, ADDR " - Cursor Size, Visible: "
839     mov eax, consoleInfo.bVisible
840     call WriteDec
841     call Crlf
842
843     ; Set a new cursor size and visibility
844     mov consoleInfo.dwSize, 50
845     mov consoleInfo.bVisible, TRUE
846     INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
847
```

```
848 ; Move the cursor to a new position
849 INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
850
851 ; Display a message at the new cursor position
852 call WriteString, ADDR "New Cursor Position"
853
854 ; Wait for a key press
855 call ReadChar
856
857 ; Reset cursor info to the default values
858 mov consoleInfo.dwSize, 25
859 mov consoleInfo.bVisible, TRUE
860 INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
861
862 ; Move the cursor back to the original position
863 mov coord.X, 0
864 mov coord.Y, 0
865 INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
866
867 ; Display a message at the original cursor position
868 call WriteString, ADDR "Original Cursor Position"
869
870 ; Wait for a key press to exit
871 call ReadChar
872
873 INVOKE ExitProcess, 0
874 main ENDP
875
876 END main
```

This program demonstrates the usage of cursor control functions. It first retrieves the current cursor info, changes the cursor size and visibility, and moves the cursor to a new position. After displaying a message, it resets the cursor to its original state and waits for a key press before exiting.

SETTING TEXT COLOR

Controlling Text Color in a Console Window

In a console window, you can control text color using two main methods:

SetConsoleTextAttribute Function: This function allows you to set the foreground and background colors for all subsequent text output in the console window. It takes the console output handle and a color attribute as parameters. The color attribute specifies both foreground and background colors and is stored in the low-order byte of the `wAttributes` parameter.

WriteConsoleOutputAttribute Function: This function enables you to set the attributes (including text color) for specific cells in the console screen buffer. You provide an array of attributes, a length, starting coordinates, and a count of the number of cells affected.

Example Program: Let's create a simple program that demonstrates how to use these functions to set text colors. In this example, we'll display characters with different colors in a console window:

```
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
```

This program sets different text colors for characters 1 to 20 and displays them in a console window. The text colors are specified in the attributes array. The characters and their associated colors are written to the console screen buffer, resulting in colorful text output.

This program is a simple example to get you started with text color manipulation in a console window. You can modify the attributes array to set different colors for your text as needed.

TIME, WINAPI AND ASSEMBLY

Here's a brief description of each of the time and date-related functions in the Win32 API:

CompareFileTime:

Compares two 64-bit file times to determine their order.

DosDateTimeToFileTime:

Converts MS-DOS date and time values to a 64-bit file time, allowing easy compatibility with older date and time representations.

FileTimeToDosDateTime:

Performs the reverse operation, converting a 64-bit file time to MS-DOS date and time values.

FileTimeToLocalFileTime:

Converts a UTC (universal coordinated time) file time to a local file time, making it suitable for use in the local time zone.

FileTimeToSystemTime:

Converts a 64-bit file time to a SYSTEMTIME structure, providing detailed date and time information.

GetFileTime:

Retrieves the date and time when a file was created, last accessed, and last modified.

GetLocalTime:

Retrieves the current local date and time, useful for obtaining the current local system time.

GetSystemTime:

Retrieves the current system date and time in UTC format, allowing for consistent time information across different time zones.

GetSystemTimeAdjustment:

Determines whether the system is applying periodic time adjustments to its time-of-day clock, important for handling time adjustments like daylight saving time.

GetSystemTimeAsFileTime:

Retrieves the current system date and time in UTC format, providing a 64-bit file time.

GetTickCount:

Retrieves the number of milliseconds that have elapsed since the system was started, useful for measuring time intervals or system uptime.

GetTimeZoneInformation:

Retrieves the current time-zone parameters, allowing you to obtain information about the system's time zone.

LocalFileTimeToFileTime:

Converts a local file time to a file time based on UTC, enabling the conversion of local time to a more universal format.

SetFileTime:

Sets the date and time that a file was created, last accessed, or last modified, allowing for the modification of file timestamps.

SetLocalTime:

Sets the current local time and date on the system, making it useful for adjusting the system's local time settings.

SetSystemTime:

Sets the current system time and date, enabling adjustments to the system's time settings.

SetSystemTimeAdjustment:

Allows you to enable or disable periodic time adjustments to the system's time-of-day clock.

SetTimeZoneInformation:

Sets the current time-zone parameters, providing control over the system's time zone settings.

SystemTimeToFileTime:

Converts a SYSTEMTIME structure to a 64-bit file time, allowing for the transformation of detailed date and time information into a universal format.

SystemTimeToTzSpecificLocalTime:

Converts a UTC time to a specified time zone's corresponding local time, useful when you need to adjust time information to a specific time zone.

Here's the combined MASM program that includes the SYSTEMTIME structure, GetLocalTime, SetLocalTime, and GetTickCount functions, as well as a stopwatch timer:

```
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 function** 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:

```

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

```

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:

```

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

```

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.

```

1133 RECT STRUCT
1134     left DWORD ?
1135     top  DWORD ?
1136     right DWORD ?
1137     bottom DWORD ?
1138 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.

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

lpzClassName: 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, \
                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, lParam, 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), lParam (the message ID), wParam, and lParam (message-specific data).

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

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

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

If lParam 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.

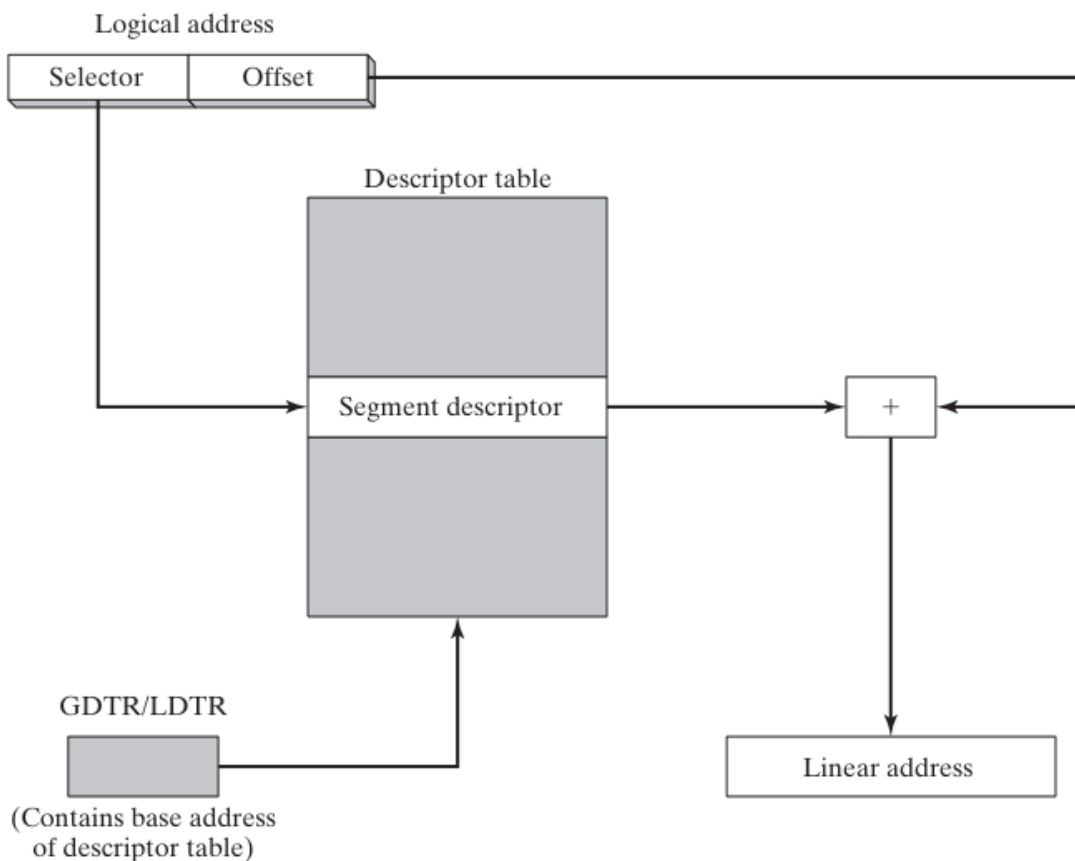
x86 MEMORY MANAGEMENT

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:

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

Once the linear address 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.

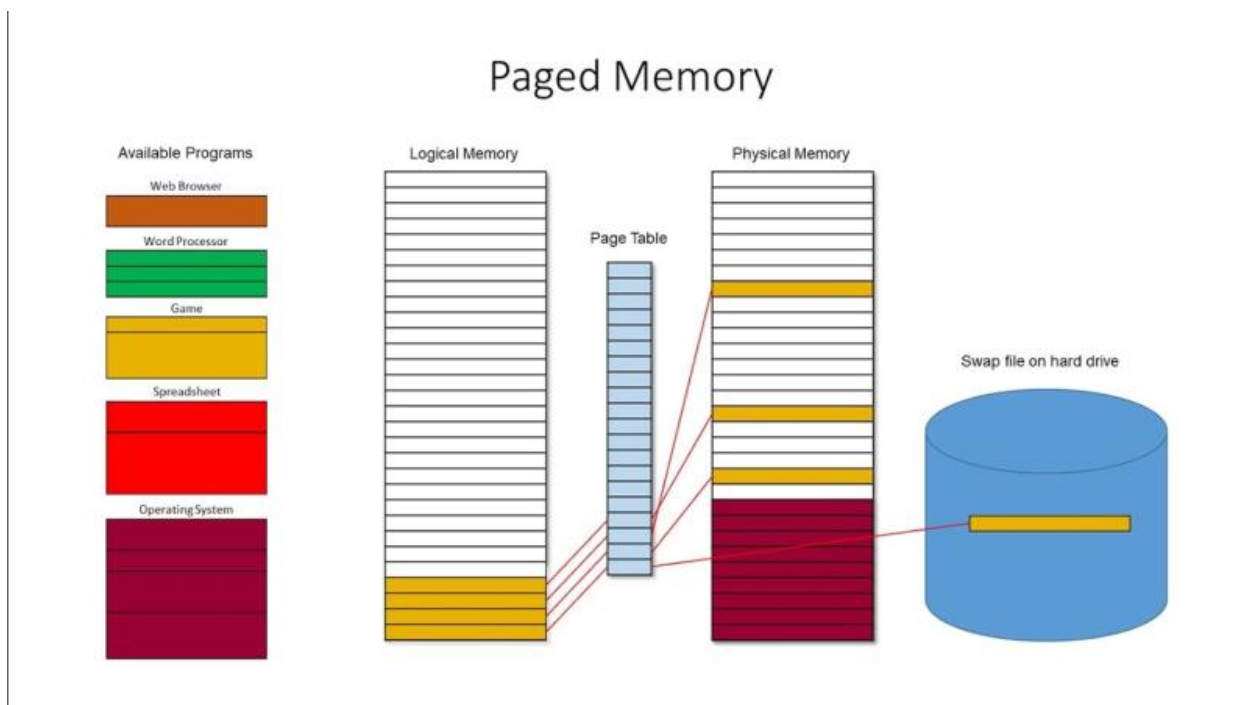
To access this variable, the processor would first calculate the linear address:

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

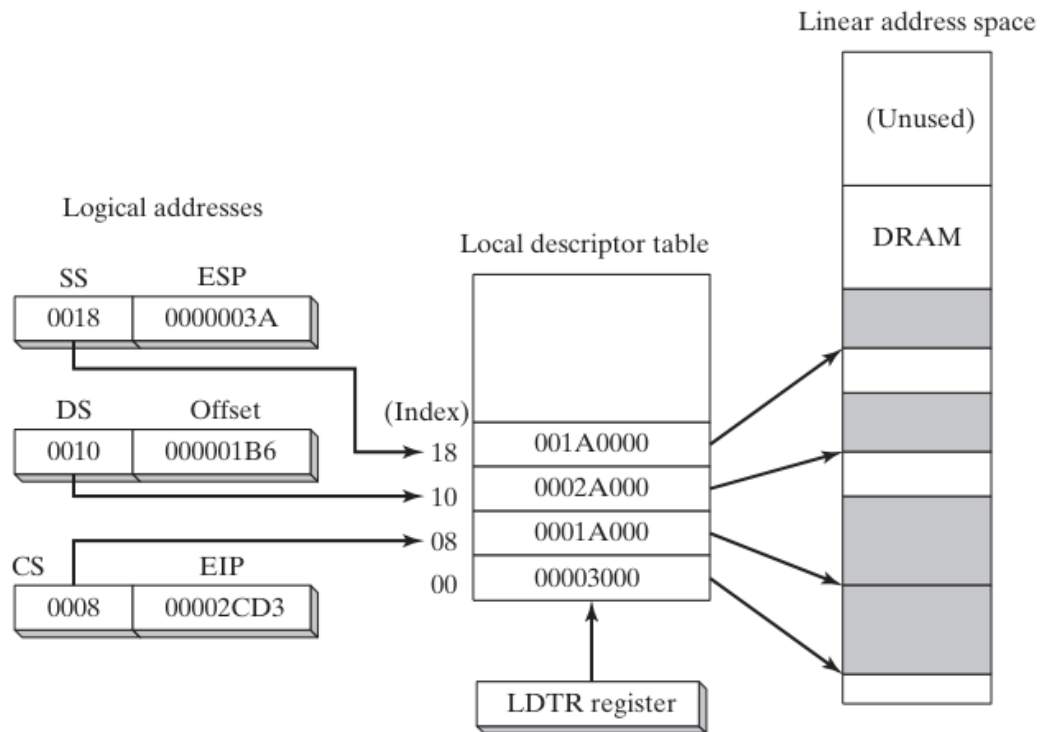
The processor translates logical addresses to linear addresses using segment descriptors. Linear addresses can then be translated to physical addresses using a page table.

=====

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.

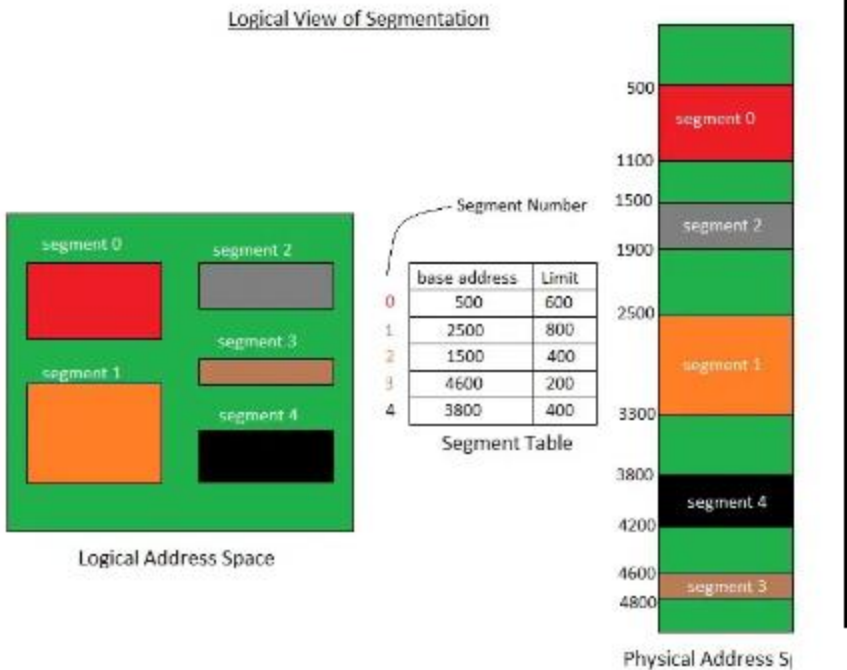
Segment descriptor details

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.

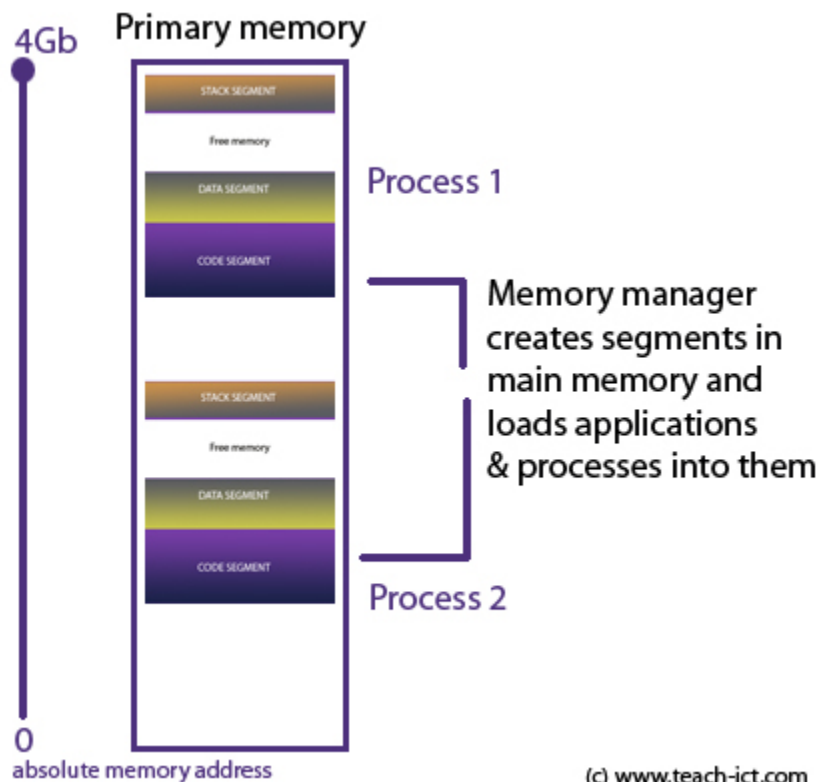
Segment limit: The maximum size of the segment.

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.



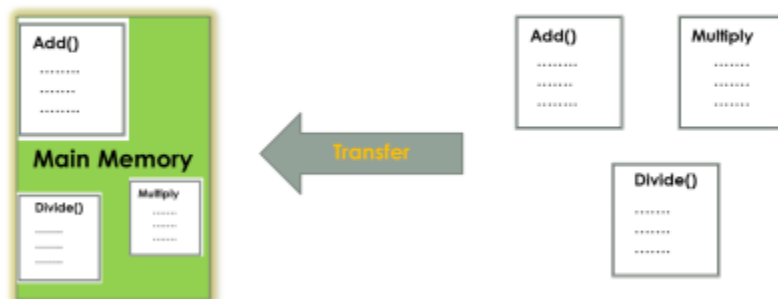
So,

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.

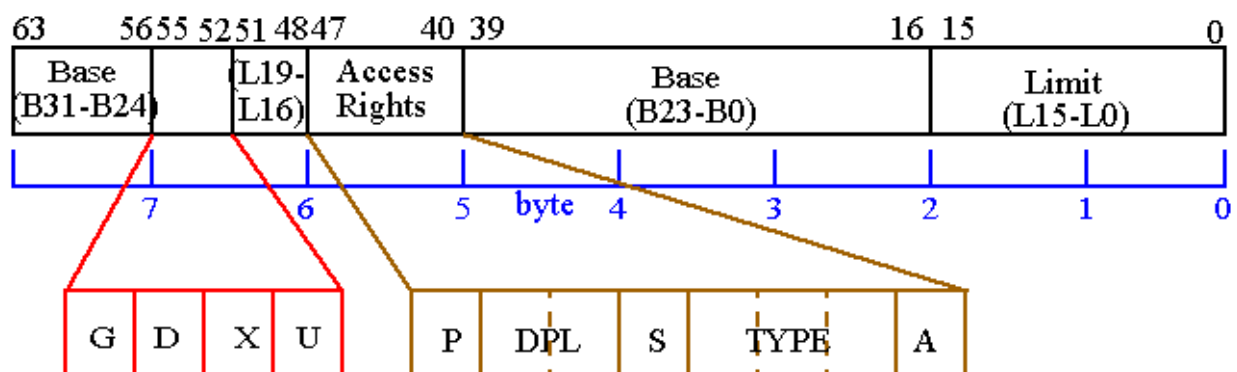
The **granularity flag** is used to determine how the segment limit field is interpreted. If the flag is set, the segment limit is interpreted in 4096-byte units. If the flag is not set, the segment limit is interpreted in bytes.

SEGMENTATION CONCEPT



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

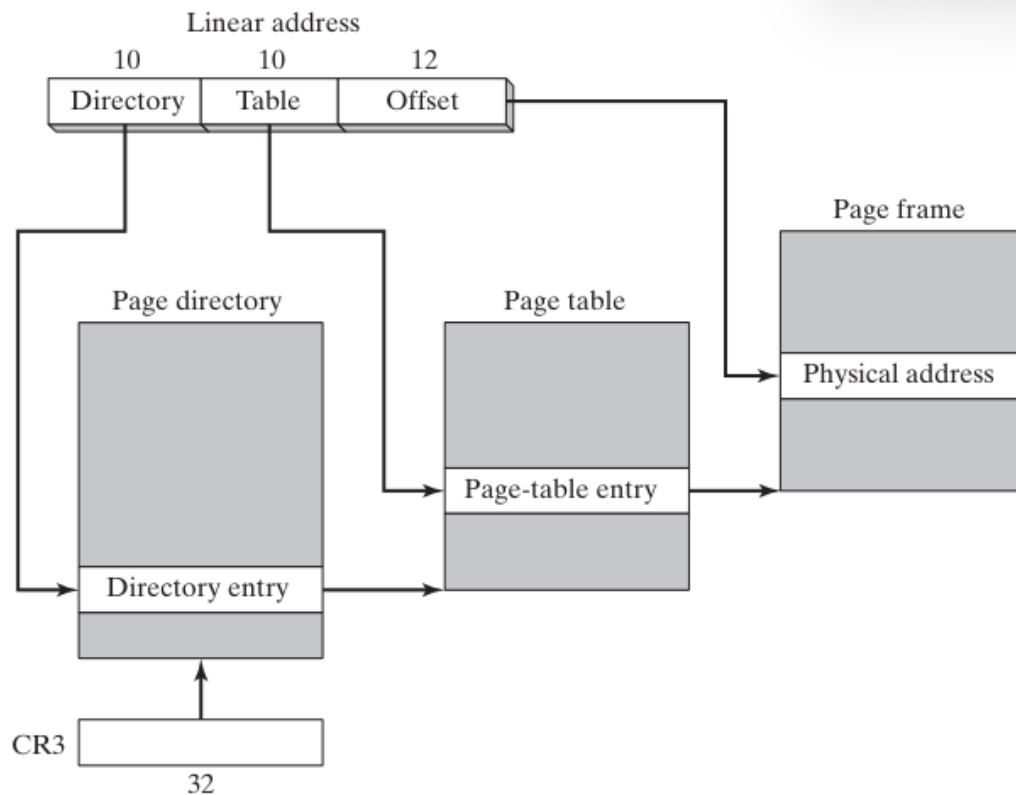
Let's describe the page table image first:

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.

Translating linear address to physical address.



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

The processor splits the linear address into three fields:

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

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

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

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

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

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

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

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

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

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

Conclusion

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

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

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

=====

Windows Virtual Machine Manager

=====

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



Microsoft®
System Center
Virtual Machine Manager

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

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

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

How VMM handles memory management

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

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

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

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

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

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

Benefits of using VMM

VMM offers a number of benefits, including:

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



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



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



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



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