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.

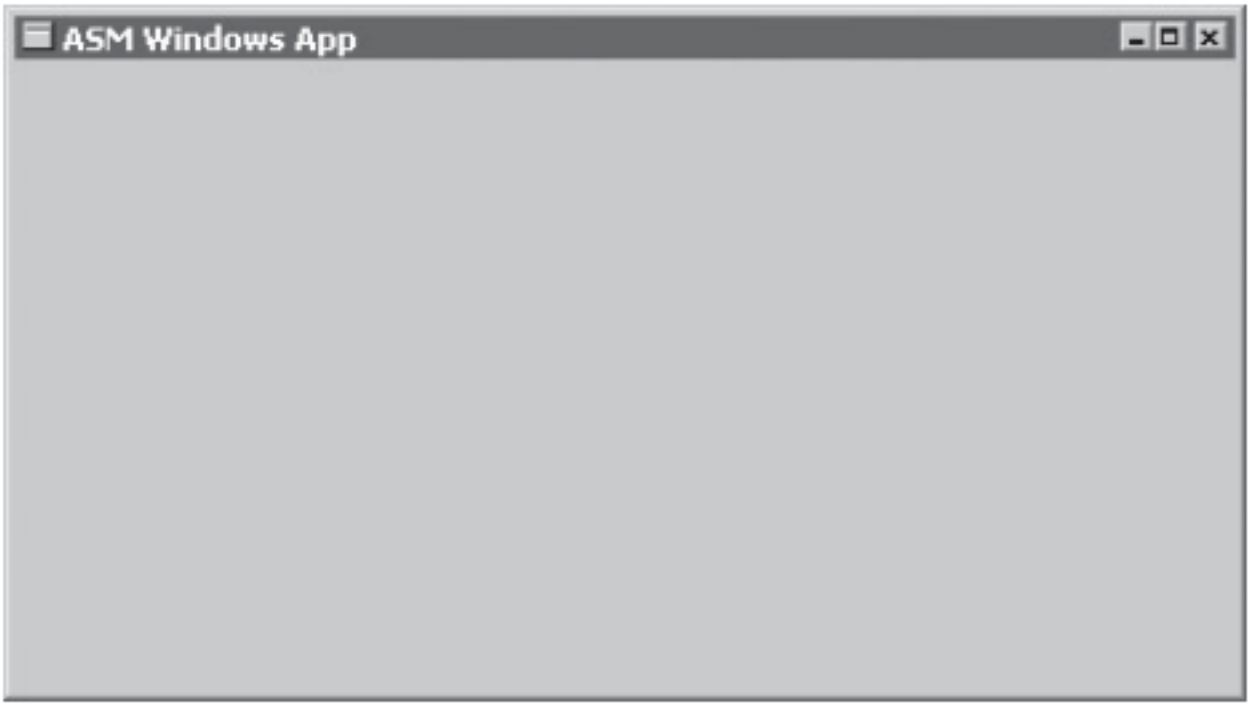
Once you have called the WriteConsoleA function, you can check the value of the bytesWritten variable to see how many bytes were written.

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.

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

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

The MessageBox Function

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

The WinMain Procedure

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

- Getting a handle to the current program.
- Loading the program's icon and mouse cursor.
- Registering the program's main window class and identifying the

procedure that will process event messages for the window.
- Creating the main window.
- Showing and updating the main window.
- Beginning a loop that receives and dispatches messages.
- The loop continues until the user closes the application window.

The WinProc Procedure

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

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

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

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

```
#include <windows.h>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

; ==================== DATA =======================
.data
AppLoadMsgTitle BYTE "Application Loaded",0
AppLoadMsgText BYTE "This window displays when the WM_CREATE "
               BYTE "message is received",0
PopupTitle BYTE "Popup Window",0
```

```
PopupText BYTE "This window was activated by a "
          BYTE "WM_LBUTTONDOWN message",0
GreetTitle BYTE "Main Window Active",0
GreetText BYTE "This window is shown immediately after "
          BYTE "CreateWindow and UpdateWindow are called.",0
CloseMsg BYTE "WM_CLOSE message received",0
ErrorTitle BYTE "Error",0
WindowName BYTE "ASM Windows App",0
className BYTE "ASMWin",0

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

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

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

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

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

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

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

```
  INVOKE UpdateWindow, hMainWnd

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

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

Exit_Program:
  INVOKE ExitProcess,0

WinMain ENDP

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

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

WinMain Procedure:

WinMain is the entry point of the application, where the program

execution begins.

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

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

The window class is registered using RegisterClass.

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

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

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

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

A greeting message is displayed in a message box.

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

Exit_Program Label:

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

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

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

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

```
;-----------------------------------------------------
WinProc PROC,
hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD
;
; The application's message handler, which handles
; application-specific messages. All other messages
; are forwarded to the default Windows message
; handler.
;-----------------------------------------------------
    mov eax, localMsg
    .IF eax == WM_LBUTTONDOWN
        ; Mouse button?
        INVOKE MessageBox, hWnd, ADDR PopupText,
            ADDR PopupTitle, MB_OK
        jmp WinProcExit
    .ELSEIF eax == WM_CREATE
        ; Create window?
        INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
            ADDR AppLoadMsgTitle, MB_OK
        jmp WinProcExit
    .ELSEIF eax == WM_CLOSE
        ; Close window?
        INVOKE MessageBox, hWnd, ADDR CloseMsg,
            ADDR WindowName, MB_OK
        INVOKE PostQuitMessage,0
        jmp WinProcExit
    .ELSE
        ; Other message?
        INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
        jmp WinProcExit
    .ENDIF
WinProcExit:
    ret
WinProc ENDP

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

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

    ; Get the corresponding message string.
```

```
    INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
        ADDR pErrorMsg, NULL, NULL

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

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

END WinMain
```

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

WinProc Procedure:

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

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

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

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

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

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

ErrorHandler Procedure:

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

It first declares data and code sections for its implementation.

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

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

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

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

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

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

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