Irvine32 Library

ReadInt: Reads a 32-bit signed decimal integer from the keyboard, terminated by the Enter key.

READINT

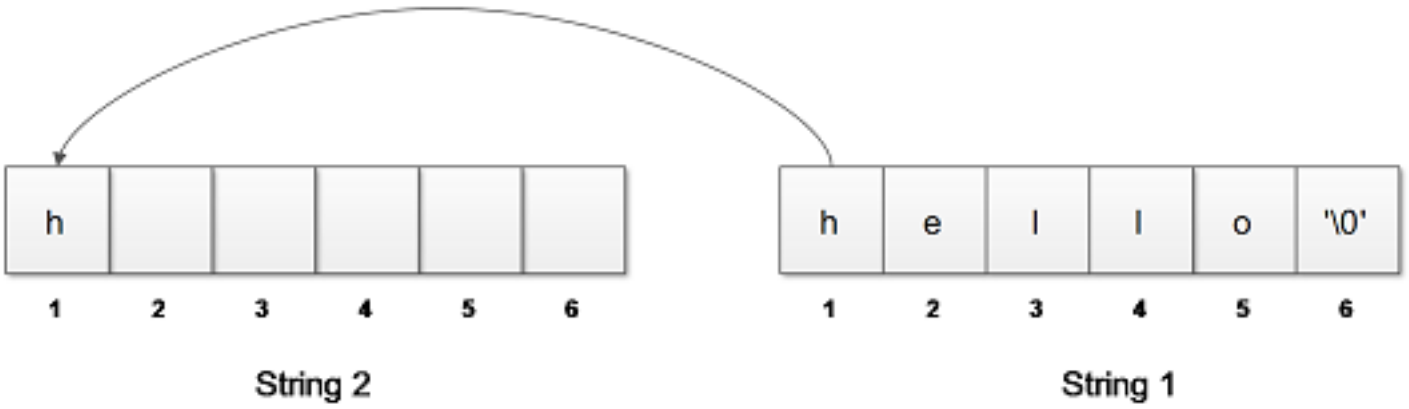
ReadKey: Reads a single character from the keyboard's input buffer without waiting for input. Useful for detecting key presses.

ReadString: Reads a string from the keyboard, terminated by the Enter key.

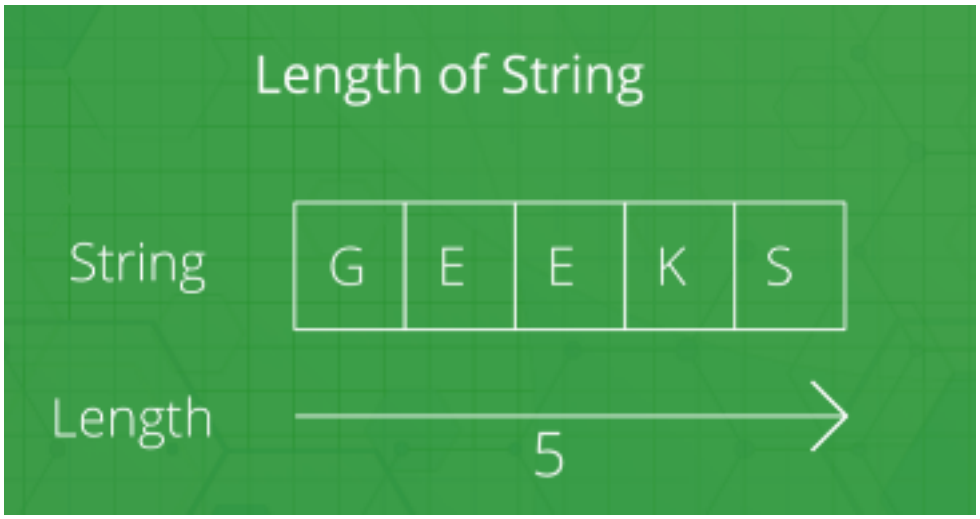
SetTextColor: Sets the foreground and background colors for all subsequent text output to the console.

Str_compare: Compares two strings and returns a result indicating their relationship.

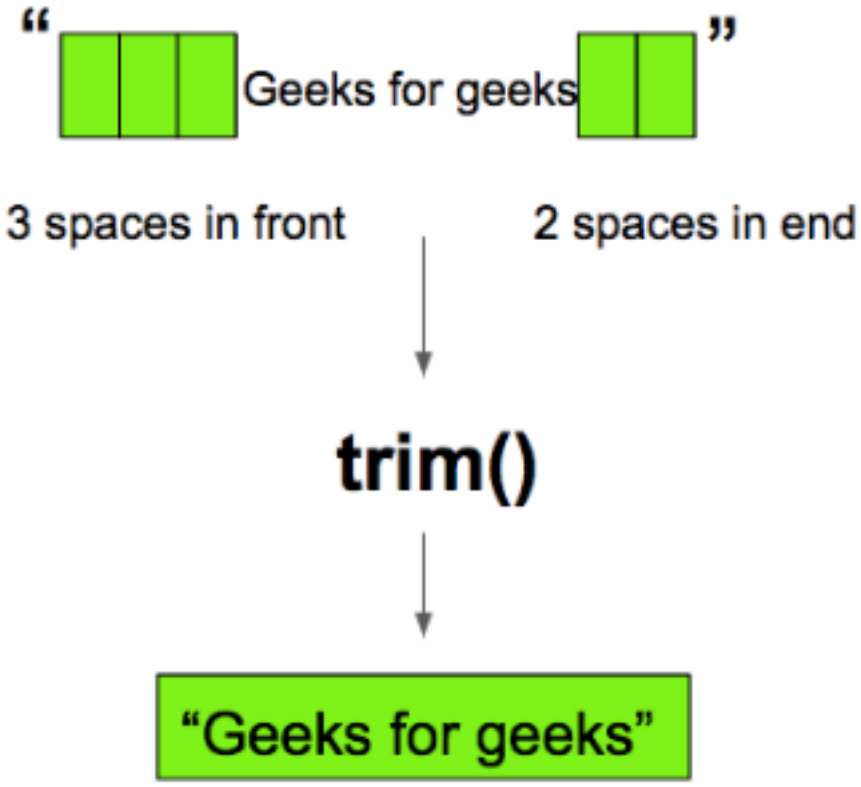
Str_copy: Copies a source string to a destination string.



Str_length: Returns the length of a string in EAX (in characters).

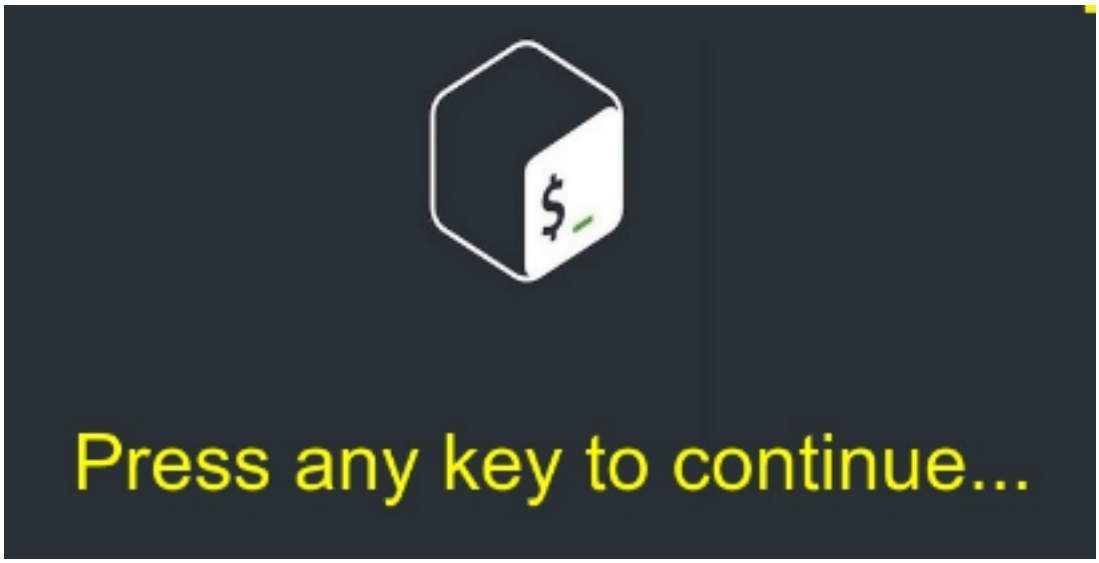


Str_trim: Removes unwanted characters (specified by a given character set) from a string.

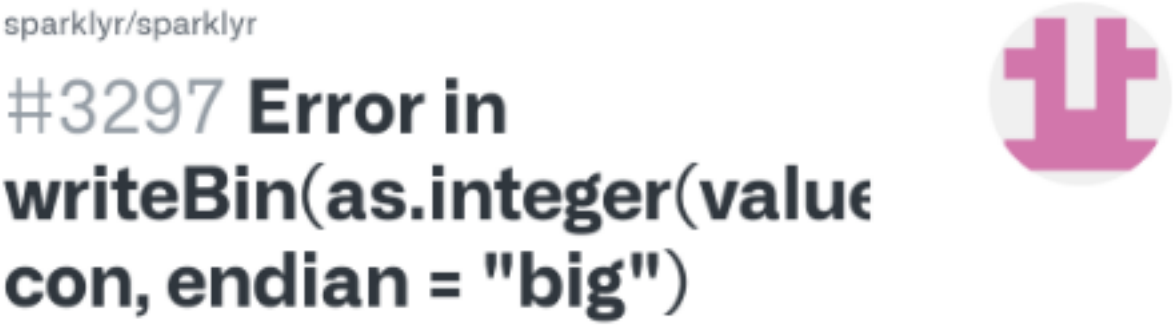


Str_ucase: Converts a string to uppercase letters.

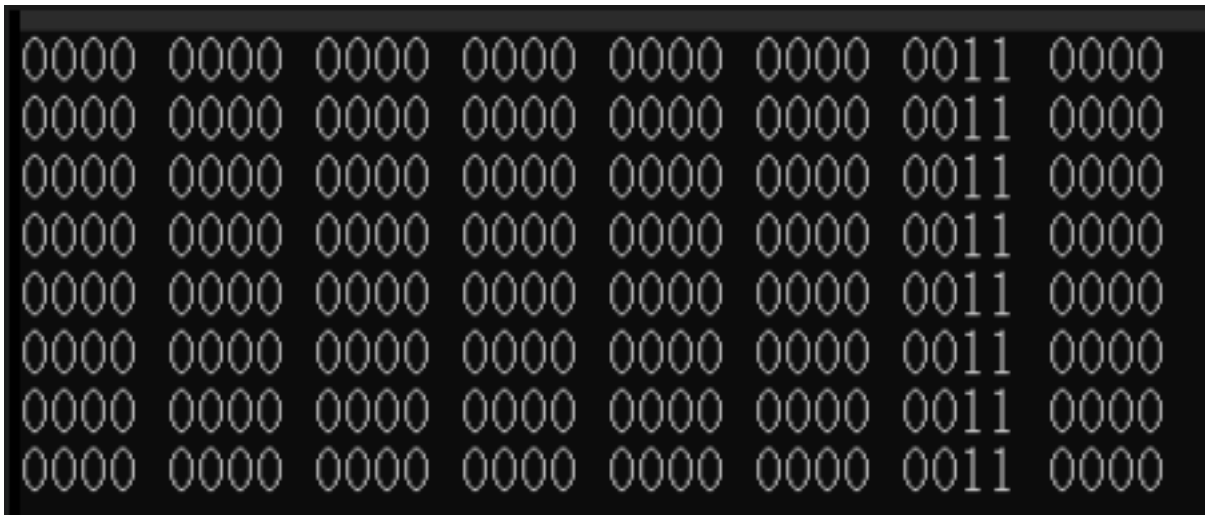
WaitMsg: Displays a message and waits for a key to be pressed before continuing.



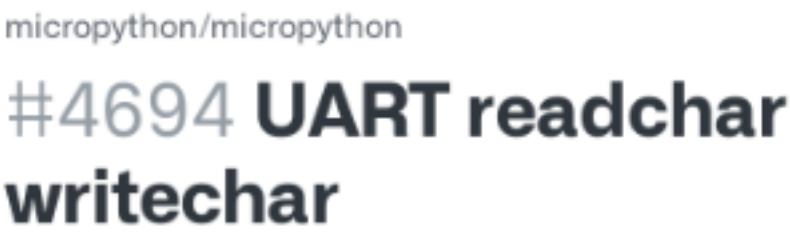
WriteBin: Writes an unsigned 32-bit integer to the console window in ASCII binary format.



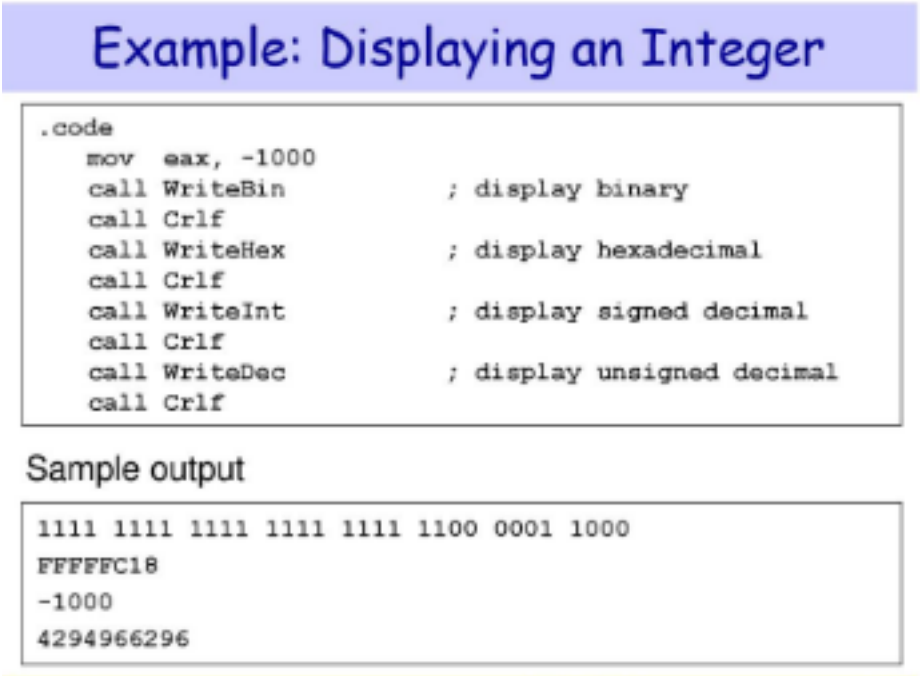
WriteBinB: Writes a binary integer to the console window in byte, word, or doubleword format.



WriteChar: Writes a single character to the console window.



WriteDec: Writes an unsigned 32-bit integer to the console window in decimal format.



WriteHex: Writes a 32-bit integer to the console window in hexadecimal format.

WriteHexB: Writes a byte, word, or doubleword integer to the console window in hexadecimal format.

WriteInt: Writes a signed 32-bit integer to the console window in decimal format.

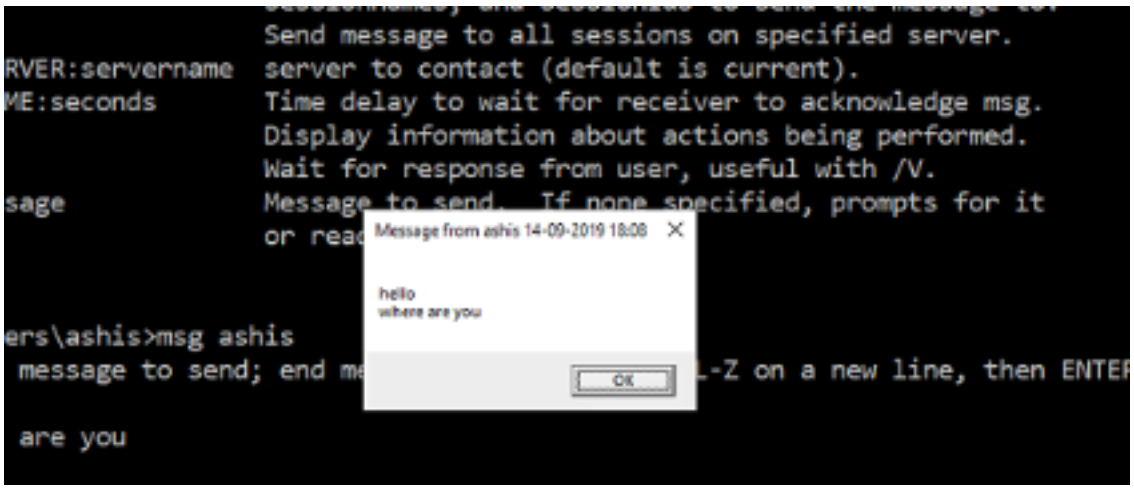
WriteStackFrame: Writes the current procedure's stack frame information to the console.

WriteStackFrameName: Writes the name of the current procedure and its stack frame information to the console.

WriteString: Writes a null-terminated string to the console window.

WriteToFile: Writes a buffer of data to an output file.

WriteWindowsMsg: Displays a string containing the most recent error message generated by MS-Windows.



Each of these procedures serves specific purposes, such as input/output operations, string manipulation, and text formatting, and can be extremely helpful for assembly language programmers to simplify common tasks.

Overview of Console Window The console window is a text-only window in MS-Windows created when a command prompt is displayed. You can customize its size, font size, and colors.

• Mode Command: You can use the "mode" command in the command prompt to change the number of columns and lines in the console window. For example:

```
mode con cols=40 lines=30
```

==========

Individual Procedure Descriptions

==========

Here are descriptions of some of the procedures in the Irvine32 library:

1. CloseFile

• Closes a file that was previously created or opened using a file handle. Pass the file handle in EAX.

```
mov eax, fileHandle call CloseFile
```

2. Clrscr

• Clears the console window. Typically called at the beginning and end of a program to clear the screen. If called at other times, you may need to pause using WaitMsg to allow the user to view existing screen contents.

```
call WaitMsg call Clrscr
```

3. CreateOutputFile

• Creates a new disk file and opens it for writing. Provide the filename offset in EDX. Returns a valid file handle (32-bit integer) in EAX if successful, otherwise INVALID_HANDLE_VALUE.

```
mov edx, offset filename call CreateOutputFile
```

4. Crlf

• The Crlf procedure advances the cursor to the beginning of the next line in the console window. It essentially moves to the next line. Sample Call:

```
call Crlf
```

This instruction calls the Crlf procedure. The Crlf procedure stands for "Carriage Return" and "Line Feed." It is used to advance the cursor to the beginning of the next line in the console window. Essentially, it simulates pressing the Enter key, which moves the cursor to the next line as in a typical text editor.

5. Delay

• The Delay procedure pauses the program for a specified number of milliseconds. You need to set EAX to the desired interval before calling it. Sample Call:

```
mov eax, 1000 ; 1 second call Delay
```

This instruction calls the Delay procedure. The Delay procedure is used to pause the program for a specified number of milliseconds. In this case, it's designed to create a 1-second delay since EAX is set to 1000 (milliseconds).

6. DumpMem

• The DumpMem procedure writes a range of memory to the console window in hexadecimal format. It requires you to pass the starting address in ESI, the number of units in ECX, and the unit size in EBX.

```
mov esi, OFFSET array ; starting OFFSET
mov ecx, LENGTHOF array ; number of units
mov ebx, TYPE array ; doubleword format
call DumpMem
```

This call displays the content of the array in hexadecimal.

Here's an explanation of the provided assembly code:

mov esi, OFFSET array: This instruction loads the memory address (offset) of the array into the ESI register. It's setting up ESI to point to the starting address of the array in memory.

mov ecx, LENGTHOF array: This instruction loads the length of the array in terms of the number of elements into the ECX register. It determines how many units of data will be displayed when calling DumpMem.

mov ebx, TYPE array: This instruction loads the data type of the array elements into the EBX register. It specifies the format of the data when calling DumpMem. In this case, it indicates that the elements are in doubleword format.

call DumpMem: This instruction calls the DumpMem procedure. The DumpMem procedure is responsible for displaying a range of memory in hexadecimal format. It takes the starting address in ESI, the number of units in ECX, and the unit size in EBX (in this case, doubleword format). It will display the contents of the array specified by ESI, ECX, and EBX in hexadecimal format.

In summary, these instructions set up the parameters for the DumpMem procedure to display the contents of the array in doubleword format from its starting address with the specified length.

7. DumpRegs

• The DumpRegs procedure displays the values of various registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFL), as well as the Carry, Sign, Zero, Overflow, Auxiliary Carry, and Parity flags in hexadecimal.

```
call DumpRegs
```

This instruction calls the DumpRegs procedure. The DumpRegs procedure displays the values of various CPU registers and flags in hexadecimal format. It provides a snapshot of the CPU's current state, which can be helpful for debugging and understanding the program's execution.

8. GetCommandTail

• GetCommandTail copies the program's command line into a null-terminated string and checks if the command line is empty.
• To use it, you must provide the offset of a buffer in EDX where the command line will be stored. Sample Call:

```
.data
    cmdTail BYTE 129 DUP(0) ; empty buffer
.code
    mov edx, OFFSET cmdTail
    call GetCommandTail ; fills the buffer
```

9. GetMaxXY

• The GetMaxXY procedure retrieves the size of the console window's buffer. The number of buffer columns is stored in DX, and the number of buffer rows is stored in AX. Sample Call:

```
.data
    rows BYTE ?
    cols BYTE ?
.code
    call GetMaxXY
    mov rows, al
    mov cols, dl
```

10. GetMseconds

• GetMseconds returns the number of milliseconds elapsed since midnight on the host computer in the EAX register. It's useful for measuring time between events. Sample Call:

```
.data
    startTime DWORD ?
.code
    call GetMseconds
    mov startTime, eax
    L1:
    ; (loop body)
    loop L1
    call GetMseconds
    sub eax, startTime ; EAX = loop time, in milliseconds
```

This example measures the execution time of a loop.

11. GetTextColor

• The GetTextColor procedure retrieves the current foreground and background colors of the console window. It has no input parameters. The background color is stored in the upper four bits of AL, and the foreground color is stored in the lower four bits. Sample Call:

```
.data
    color BYTE ?
.code
    call GetTextColor
    mov color, al
```

Here's a breakdown of what this code does:

1. .data color BYTE ?: This declares a byte-sized variable named color in the data section. This variable will be used to store the color information retrieved from the console.

2. .code call GetTextColor: This line calls the GetTextColor procedure, which retrieves the current text color attributes of the console window. The result is returned in the AL register.

3. mov color, al: This instruction moves the value in the AL register (which contains the retrieved color information) into the color variable declared earlier. It stores the current text color in the color variable.

So, after executing these instructions, the color variable will contain the current text color attribute of the console window, allowing you to use or manipulate it in your program as needed.

12. Gotoxy

• The Gotoxy procedure positions the cursor at a specific row and column in the console window. You should pass the row (Y-coordinate) in DH and the column (X-coordinate) in DL. Sample Call:

```
mov dh, 10 ; row 10
mov dl, 20 ; column 20
call Gotoxy ; position cursor
```

Here's a breakdown of what this code does:

mov dh, 10: This instruction moves the value 10 into the DH (Destination High) register, which represents the row number where you want to position the cursor. In this case, it sets the row to 10.

mov dl, 20: This instruction moves the value 20 into the DL (Destination Low) register, which represents the column number where you want to position the cursor. It sets the column to 20.

call Gotoxy: This is a procedure call to the Gotoxy procedure. When called, it takes the values in DH (row) and DL (column) and uses them to position the cursor in the console window.

So, after executing these instructions, the cursor will be moved to

row 10 and column 20 in the console window. This can be useful for controlling the cursor's position when you want to display text or interact with the user at specific locations on the screen.

13. IsDigit

• IsDigit checks whether the value in AL is the ASCII code for a valid decimal digit. If AL contains a valid decimal digit, it sets the Zero flag (ZF); otherwise, it clears ZF. Sample Call:

```
mov AL, somechar
call IsDigit
```

You are trying to determine whether the value in the AL register represents a valid decimal digit by calling the IsDigit procedure.

You move the value of somechar into the AL register using mov AL, somechar. somechar presumably contains an ASCII character.

You call the IsDigit procedure with the current value in the AL register using call IsDigit.

Inside the IsDigit procedure, it checks if the value in AL is a valid ASCII code for a decimal digit. If it is, the procedure sets the Zero flag (ZF) to 1. If it's not a valid digit, the ZF is cleared (set to 0).

After calling IsDigit, you can check the state of the ZF to determine whether the character in AL is a valid decimal digit or not.

If ZF is set (ZF = 1), then somechar represents a valid decimal digit. If ZF is cleared (ZF = 0), then somechar does not represent a valid decimal digit.

This code is useful if you want to validate whether a character is a decimal digit in your program.

14. MsgBox

The MsgBox procedure displays a popup message box with an optional caption. Pass it the offset of a string in EDX to appear inside the

box, and optionally, pass the offset of a string for the box's title in EBX. To leave the title blank, set EBX to zero.

```
.data
    caption BYTE "Dialog Title", 0
    HelloMsg BYTE "This is a pop-up message box.", 0dh,0ah, "Click OK to continue...", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET HelloMsg
    call MsgBox
```

You are setting up a message box with a title ("Dialog Title") and a message ("This is a pop-up message box. Click OK to continue..."). Then, you are calling the MsgBox procedure to display this message box.

You define two null-terminated strings in the .data section. caption contains the title of the message box, and HelloMsg contains the message along with a line break (0dh,0ah) and the instruction "Click OK to continue..."

You load the address of the caption string into the EBX register using mov ebx, OFFSET caption. This prepares the address of the title string to be passed as a parameter to the MsgBox procedure.

You load the address of the HelloMsg string into the EDX register using mov edx, OFFSET HelloMsg. This prepares the address of the message string to be passed as a parameter to the MsgBox procedure.

You call the MsgBox procedure using call MsgBox. The procedure displays a graphical popup message box with an OK button. It expects the address of the title in EBX and the address of the message in EDX.

After the user clicks the OK button in the message box, the MsgBox procedure returns, and your program continues executing any subsequent code.

This code demonstrates how to display a simple informational message box with a title and a message, allowing the user to acknowledge the message by clicking the OK button.

15. MsgBoxAsk

The MsgBoxAsk procedure displays a popup message box with Yes and No buttons. It returns an integer in EAX that indicates which button was selected by the user (IDYES or IDNO).

```
.data
    caption BYTE "Survey Completed", 0
    question BYTE "Thank you for completing the test.", 0dh,0ah, "Receive results?", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET question
    call MsgBoxAsk ; check return value in EAX
```

You are setting up a message box with a title ("Survey Completed") and a question ("Thank you for completing the test. Receive results?"). Then, you are calling the MsgBoxAsk procedure to display this message box.

Here's what's happening step by step:

1. You define two null-terminated strings in the .data section. caption contains the title of the message box, and question contains the message along with a line break (0dh,0ah) and the question "Receive results?"

2. You load the address of the caption string into the EBX register using mov ebx, OFFSET caption. This prepares the address of the title string to be passed as a parameter to the MsgBoxAsk procedure.

3. You load the address of the question string into the EDX register using mov edx, OFFSET question. This prepares the address of the message/question string to be passed as a parameter to the MsgBoxAsk procedure.

4. You call the MsgBoxAsk procedure using call MsgBoxAsk. The procedure displays a graphical popup message box with a Yes and No button. It expects the address of the title in EBX and the address of the message/question in EDX.

After the user interacts with the message box (by clicking either Yes or No), the MsgBoxAsk procedure returns an integer value in EAX, which tells you which button was selected. The value will be either IDYES (equal to 6) if the user clicked Yes or IDNO (equal to 7) if the user clicked No.

To check the return value and take further actions based on the user's choice, you can use conditional statements or other logic in your code, depending on whether EAX contains IDYES or IDNO.

16. OpenInputFile

The OpenInputFile procedure opens an existing file for input. Pass it the offset of the filename in EDX. If the file is opened successfully, EAX contains a valid file handle; otherwise, EAX equals INVALID_HANDLE_VALUE.

```
.data
    filename BYTE "myfile.txt",0
.code
    mov edx, OFFSET filename
    call OpenInputFile
```

After the call, you can check the value of EAX to determine if the file was opened successfully. If EAX equals INVALID_HANDLE_VALUE, the file was not opened successfully.

You are defining a null-terminated string "myfile.txt" in the .data section and then using it as the filename to open an input file using the OpenInputFile procedure.

Here's what's happening step by step:

You define a null-terminated string filename with the content "myfile.txt" in the .data section.

You load the address of the filename string into the EDX register using mov edx, OFFSET filename. This prepares the address of the filename to be passed as a parameter to the OpenInputFile procedure.

You call the OpenInputFile procedure using call OpenInputFile. The procedure expects the filename (the address of the null-terminated string) in the EDX register. It will attempt to open the file with the given name for input.

After calling OpenInputFile, if the file was successfully opened, the EAX register will contain a valid file handle. If the file could not

be opened, EAX will be set to INVALID_HANDLE_VALUE (a predefined constant).

In your code, you have not shown how you are handling the result of opening the file (the value in EAX). Depending on whether the file was opened successfully or not, you would typically check the value in EAX and take appropriate actions, such as reading from the file or displaying an error message.

17. ReadString:

Use ReadString to read a string from the keyboard until the Enter key is pressed. Pass the offset of a buffer where the input will be stored in EDX and specify the maximum number of characters the user can enter in ECX. It returns the count of the number of characters typed by the user in EAX. Example:

```
.data
    buffer BYTE 21 DUP(0) ; input buffer
    byteCount DWORD ? ; holds counter
.code
    mov edx, OFFSET buffer ; point to the buffer
    mov ecx, SIZEOF buffer ; specify max characters
    call ReadString ; input the string
    mov byteCount, eax ; number of characters
```

The provided code segment is using the ReadString procedure to read a string from the keyboard and save it in a buffer. Let's break down what each part of the code does:

.data section:

buffer BYTE 21 DUP(0): This defines a buffer named buffer that can hold up to 21 bytes (including the null terminator). It's initialized with zeros to ensure it's an empty string.

byteCount DWORD ?: This declares a DWORD variable named byteCount to hold the count of characters entered by the user.

.code section:

mov edx, OFFSET buffer: This line sets the EDX register to point to the buffer variable, effectively providing the address of the buffer to the ReadString procedure.

mov ecx, SIZEOF buffer: Here, the ECX register is loaded with the size of the buffer variable, which is the maximum number of characters that can be read. In this case, it's set to 21.

call ReadString: This line calls the ReadString procedure, which reads a string from the keyboard. It stops reading when the Enter key is pressed. The string entered by the user is stored in the buffer, and the number of characters read is returned in EAX.

mov byteCount, eax: Finally, this line stores the value of EAX (the number of characters read) into the byteCount variable, so you can later access and use this count in your program.

Where did eax come from?

The **ReadString procedure** reads a string from the keyboard and returns the number of characters read in the **EAX register.** After calling ReadString, the value in EAX represents the number of characters read, and it's stored in the byteCount variable for later use in your program.

In summary, this code segment initializes a buffer, reads a string from the keyboard into the buffer using ReadString, and stores the count of characters read in the byteCount variable.

18. SetTextColor:

Use SetTextColor to set the foreground and background colors for text output. Set the desired color attribute in EAX, combining foreground and background colors. Example (setting text to white on a blue background).

```
mov eax, white + (blue SHL 4) ; white on blue
call SetTextColor
```

The provided code segment is setting the text color for text output in a console window using the SetTextColor procedure. Let's break down what each part of the code does:

mov eax, white + (blue SHL 4): This line sets up the eax register to specify the desired text color attribute. It combines two color constants, white and blue, to achieve the desired color. Here's the breakdown:

white and blue are color constants defined in the Irvine32 library. white represents white text, and blue represents blue as the background color.

(blue SHL 4) shifts the value of blue four bits to the left. This bitwise shift operation is
used to specify the background color. In this case, it's indicating a blue background.

Adding white and the shifted blue value together combines the foreground and background colors. So, the result is a combination of white text on a blue background.

call SetTextColor: This line calls the SetTextColor procedure with the color attribute specified in eax. The procedure then sets the text color for subsequent text output to the console using the specified color combination.

In summary, this code segment sets the text color to white on a blue background using the SetTextColor procedure, so any text output that follows will be displayed with this color combination in the console window.

19. Str_length:

Use Str_length to find the length of a null-terminated string. Pass the offset of the string in EDX. It returns the string's length in EAX.

```
.data
    buffer BYTE "abcde", 0
    bufLength DWORD ?
.code
    mov edx, OFFSET buffer ; point to string
    call Str_length ; EAX = 5
    mov bufLength, eax ; save length
```

The provided code snippet calculates the length of a null-terminated string stored in memory using the Str_length procedure and then saves the length in a DWORD variable named bufLength. Here's a breakdown of what each part of the code does:

.data: This section defines a data segment where you declare data variables.

buffer BYTE "abcde", 0: This line declares a null-terminated string named buffer with the value "abcde". The null terminator (0) signifies the end of the string.

bufLength DWORD ?: This line declares an uninitialized DWORD variable named bufLength to store the length of the string.

.code: This section is the code segment where you write the program's instructions.

mov edx, OFFSET buffer: This instruction loads the offset of the buffer string into the edx register. It sets edx to point to the beginning of the string.

call Str_length: This line calls the Str_length procedure, passing the address of the buffer string as a parameter in edx. The Str_length procedure calculates the length of the string and stores it in the eax register.

mov bufLength, eax: This instruction copies the value of eax (which now contains the length of the string) into the bufLength variable. This variable will now hold the length of the "abcde" string, which is 5 characters.

In summary, this code segment calculates and stores the length of the

"abcde" string in the bufLength variable, resulting in bufLength being set to 5.

20. WaitMsg:

Use WaitMsg to display the message "Press any key to continue..." and wait for the user to press a key. Example:

```
call WaitMsg
```

The code snippet call WaitMsg is a function call to the WaitMsg procedure. Here's what it does:

call WaitMsg: This line of code calls the WaitMsg procedure, which is part of the Irvine32 library. When you call WaitMsg, it displays the message "Press any key to continue..." on the console window and waits for the user to press any key.

The **purpose of using WaitMsg** is to pause the program's execution temporarily and provide a message to the user, prompting them to press a key to continue. This can be useful when you want to give the user time to read the output on the console before it disappears.

So, when this line of code is executed, it will display the "Press any key to continue..." message on the console, and the program will wait until the user presses a key before proceeding further.

21. WriteInt:

Use WriteInt to write a 32-bit signed integer to the console window in decimal format. Pass the integer in EAX.

```
mov eax, -12345
call WriteInt ; displays: "-12345"
```

In this code, you are performing the following steps:

mov eax, -12345: This instruction moves the signed 32-bit integer value -12345 into the EAX register. It's a negative integer

represented in two's complement form.

call WriteInt: Here, you are calling the WriteInt procedure. This procedure takes a 32-bit signed integer in the EAX register and writes it to the console window in decimal format with a leading sign if it's negative. In this case, since you loaded -12345 into EAX, it will display "-12345" on the console.

So, when you run this code, it will display the signed integer -12345 on the console window.

22. WriteString:

Use WriteString to write a null-terminated string to the console window. Pass the offset of the string in EDX.

```
.data
    greeting BYTE "Hello, Assembly Programmer!", 0
.code
    mov edx, OFFSET greeting
    call WriteString
```

In this part of the code:

mov edx, OFFSET greeting: This line sets the EDX register to the offset of the greeting string. In other words, it tells the program where the string is located in memory.

call WriteString: Here, you are calling the WriteString procedure. This procedure takes the offset of a string (in this case, the offset of the greeting string) and writes the content of the string to the console window.

So, the code you provided is essentially displaying the "Hello, Assembly Programmer!" message on the console window using the WriteString procedure.

23. WriteToFile:

This procedure is used to write the contents of a buffer to an output file.

• Parameters: EAX (file handle), EDX (buffer address), ECX (number of bytes to write).
• Returns the number of bytes written if successful; otherwise, an error code.

```
.data
    fileHandle DWORD ?
    buffer BYTE "This is a test.", 0

.code
    mov eax, fileHandle
    mov edx, OFFSET buffer
    mov ecx, LENGTHOF buffer - 1 ; Length excluding the null terminator
    call WriteToFile
```

24. WriteWindowsMsg:

This procedure writes a string containing the most recent error generated by your application to the Console window when executing a system function. It is often used to display error messages to the user.

```
.code
    ; Some code that might generate an error
    ; ...

    ; After an error occurs, call WriteWindowsMsg to display the error message
    call WriteWindowsMsg
```

==========

Implementing the procedures

==========

```
; Library Test #1: Integer I/O (InputLoop.asm)
; Tests the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor,
; WaitMsg, WriteBin, WriteHex, and WriteString procedures.

include Irvine32.inc

.data
COUNT = 4
BlueTextOnGray = blue + (lightGray * 16)
DefaultColor = lightGray + (black * 16)
arrayD SDWORD 12345678h, 1A4B2000h, 3434h, 7AB9h
prompt BYTE "Enter a 32-bit signed integer: ", 0

.code

main PROC
    ; Select blue text on a light gray background
    mov eax, BlueTextOnGray
    call SetTextColor

    ; Clear the screen
    call Clrscr

    ; Display an array using DumpMem.
    mov esi, OFFSET arrayD      ; starting OFFSET
    mov ebx, TYPE arrayD        ; doubleword = 4 bytes
    mov ecx, LENGTHOF arrayD  ; number of units in arrayD
    call DumpMem                ; display memory
```

```
    ; Ask the user to input a sequence of signed integers
    call Crlf                   ; new line
    mov ecx, COUNT

L1:
    mov edx, OFFSET prompt
    call WriteString            ; prompt the user
    call ReadInt                ; input integer into EAX
    call Crlf                   ; new line

    ; Display the integer in decimal, hexadecimal, and binary
    call WriteInt               ; display in signed decimal
    call Crlf
    call WriteHex               ; display in hexadecimal
    call Crlf
    call WriteBin               ; display in binary
    call Crlf

    Loop L1                     ; repeat the loop

    ; Return the console window to default colors
    call WaitMsg                ; "Press any key..."
    mov eax, DefaultColor
    call SetTextColor
    call Clrscr

    exit
main ENDP
END main
```

The program you've provided is a demonstration of using various procedures from the Irvine32 library to perform tasks like clearing the screen, displaying memory content, reading and displaying integers, and generating random numbers. Here's a breakdown of the program:

Setup and Definitions: The program starts with comments indicating its purpose and includes the Irvine32.inc library.

It defines constants COUNT, BlueTextOnGray, and DefaultColor.

An array named arrayD is declared with four signed doubleword integers. There's also a prompt message.

Main Procedure (main PROC):

• It sets the text color to blue on a light gray background using SetTextColor.
• Clears the screen using Clrscr.
• Displays the content of the arrayD array using the DumpMem procedure.
• It then enters a loop (L1) to ask the user to input signed integers COUNT times.
• For each input, it:
• Prompts the user to enter an integer using WriteString. Reads the integer input into EAX using ReadInt. Displays the entered integer in decimal, hexadecimal, and binary formats using WriteInt, WriteHex, and WriteBin. After the loop, it returns the console window to default colors using WaitMsg, and clears the screen again.
• Random Number Generation Procedures (Rand1 and Rand2):
• Rand1 generates ten pseudo-random unsigned integers in the range 0 to 4,294,967,294 using Random32.
• Rand2 generates ten pseudo-random signed integers in the range -50 to +49 using RandomRange.
• Main Procedure (continued):
• After defining the random number generation procedures, the program calls Randomize to initialize the random number generator.
• It then calls both Rand1 and Rand2 procedures to generate and display random integers.

Program End:

• The program ends with END main.
• This program is a comprehensive example of how to use various Irvine32 library procedures to perform tasks related to console input and output, as well as random number generation. It also demonstrates how to format and display data in different formats (decimal, hexadecimal, and binary).

```
; Library Test #3: Performance Timing (TestLib3.asm)
; Calculate the elapsed execution time of a nested loop.

include Irvine32.inc

.data
OUTER_LOOP_COUNT = 3
startTime DWORD ?
msg1 BYTE "Please wait...", 0dh, 0ah, 0
msg2 BYTE "Elapsed milliseconds: ", 0

.code

main PROC
    mov edx, OFFSET msg1  ; "Please wait..."
    call WriteString

    ; Save the starting time
    call GetMSeconds
    mov startTime, eax

    ; Start the outer loop
    mov ecx, OUTER_LOOP_COUNT
L1:
    call innerLoop
    loop L1

    ; Calculate the elapsed time
    call GetMSeconds
    sub eax, startTime
```

```
    ; Calculate the elapsed time
    call GetMSeconds
    sub eax, startTime

    ; Display the elapsed time
    mov edx, OFFSET msg2  ; "Elapsed milliseconds: "
    call WriteString
    call WriteDec          ; Write the milliseconds
    call Crlf

    exit
main ENDP

innerLoop PROC
    push ecx              ; Save current ECX value
    mov ecx, 0FFFFFFFh    ; Set the loop counter
L1:
    mul eax               ; Use up some cycles
    mul eax
    mul eax
    loop L1              ; Repeat the inner loop
    pop ecx               ; Restore ECX's saved value
    ret
innerLoop ENDP
END main
```

Certainly, let's delve into the additional information you provided regarding the Irvine64 library and some of its key procedures:

WriteHex64: This procedure is used to display the contents of the RAX register as a 64-bit hexadecimal integer. It's a convenient way to view the value of a 64-bit integer in hexadecimal format. Unlike some other procedures, it doesn't return any value; its purpose is to output the hexadecimal representation to the console.

WriteHexB: WriteHexB is a versatile procedure that displays the contents of the RAX register as a hexadecimal integer in different formats, depending on the display size specified in the RBX register. You can pass 1, 2, 4, or 8 in RBX to indicate whether you want to display the value as a 1-byte, 2-byte, 4-byte, or 8-byte hexadecimal

integer, respectively. Like WriteHex64, it doesn't return a value but is used for outputting hexadecimal representations.

WriteString: This procedure is used to display a null-terminated ASCII string. You pass the 64-bit offset of the string in RDX, and it will output the string to the console. Like the previous procedures, it also doesn't return a value.

The Irvine64 library you mentioned is designed for 64-bit assembly programming. It's worth noting that in 64-bit assembly, the calling conventions and register usage can be different from 32-bit assembly.

For example, some registers like RAX, RCX, RDX, R8, R9, R10, and R11 may not be preserved across function calls, while others like RBX, RBP, RDI, RSI, R12, R13, R14, and R15 are usually preserved. Understanding these conventions is important when writing 64-bit assembly code.

This library, while smaller than its 32-bit counterpart, still provides essential tools for interactive programming in a 64-bit environment.

As you progress in your 64-bit assembly programming journey, you may find it beneficial to expand this library with your own code to suit your specific needs and applications.

Didn't we deal with those earlier on?

You are correct; some of the procedures mentioned, such as WriteHex64, WriteHexB, and WriteString, are similar to their 32-bit counterparts that we discussed earlier.

However, the key difference here is that these procedures are designed for 64-bit assembly programming, as indicated by the use of the RAX, RBX, RDX, and other 64-bit registers.

In the previous discussions, we primarily focused on 32-bit assembly and the Irvine32 library.

The additional information you provided pertains to the Irvine64 library, which is tailored for 64-bit assembly programming.

While the basic concepts and functionality of these procedures may be similar, the implementation and usage may differ due to the nuances of 64-bit assembly, including register usage and calling conventions.