

# CM1205

## INPUT & OUTPUT VIA API CALLS

# INTRODUCTION

- ▶ So far we have looked at the basic assembler instructions for manipulating data.
- ▶ We now need to think about how we get values into and out of the computer via keyboard and screen.
- ▶ In the 16-bit architecture we were able to call the various routines held in the BIOS of the machine (see next couple of slides) through software interrupt calls.
- ▶ With 32bit and 64-bit systems the concept of these low level calls has been removed.
- ▶ In a Windows environment we can call the Windows API

# THE WINDOWS API

- ▶ The API (Application Programming Interface) is a collection of types, constants and functions that provide a way to directly manipulate objects through programming.
- ▶ [https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx)

# CONSOLE PROGRAMS

- ▶ Our programs are currently being run in Visual Studio with the LINK option set to **/SUBSYSTEM:CONSOLE**
- ▶ The console program looks and behaves like an MS-DOS window, with some enhancements.
- ▶ The console has a single input buffer and one or more screen buffers.
  - ▶ The *input buffer* contains a queue of input records, each containing data about an input event, eg. Keyboard, mouse etc.
  - ▶ The *screen buffer* is a two dimensional array of character and colour data that affects the appearance of text in the console window.

# CONSOLE HANDLES

- ▶ Nearly all Win32 console functions require you to pass a 'handle' as the first argument. A 'handle' is a 32-bit unsigned integer that uniquely identifies an object such as a bitmap, drawing pen, or any I/O device.
- ▶ These are the main handles we will be using:
  - ▶ `STD_INPUT_HANDLE` -10      standard input
  - ▶ `STD_OUTPUT_HANDLE` -11      standard output.

In order for us to get going we first need to get the **handle** from the system and storing it for future use in our program.

This is achieved using **GetStdHandle** function. See example next slide.

.586

.model flat, stdcall

GetStdHandle proto :dword

WriteConsoleA proto :dword, :dword, :dword, :dword, :dword

ExitProcess proto :dword

STD\_OUTPUT\_HANDLE equ -11

.data

sum\_string db "Test Console output",0

outputHandle DWORD ?

buffer db 64 dup(?)

bytes\_written dd ?

.code

main proc

invoke GetStdHandle, STD\_OUTPUT\_HANDLE

mov outputHandle, eax

invoke WriteConsoleA, outputHandle, addr sum\_string, eax, addr bytes\_written, 0

mov eax,0

mov eax,bytes\_written

invoke ExitProcess, 0

main endp

end

# PROTO

- ▶ When using the API a number of parameters need to be passed to the invoked function.
- ▶ We can either define them as STRUCT or by defining a Prototype using the directive PROTO.
- ▶ PROTO declares a Procedure name and a parameter list.
- ▶ MASM requires a prototype for each procedure called by the **INVOKE** statement.
- ▶ **PROTO** must appear first before **INVOKE**.

# LET'S NOW READ FROM THE CONSOLE

- ▶ Using the **ReadConsoleA** function built into the windows API:
- ▶ Windows definition of the function shows it requires the following information:

handle:DWORD	;input handle
pBuffer:DWORD	;pointer to buffer
maxBytes:DWORD	;number of characters to read
pBytesRead:PTR DWORD	;pointer to number of bytes read
notUsed:DWORD	;(not used)



► Now look at the sheet provided for the syntax of the other main API calls we will use.

► **Ref: Win32APIfunction.doc**

# LAB EXERCISE COMBINE THE TWO.

Assembler  
Directive

Value obtained  
from GetHandle  
API call

Maximum No.  
characters to  
Read in

Not USED  
so set to 0

**invoke ReadConsoleA, inputHandle, addr buffer, bufSize, addr bytes\_read,0**

Windows  
API call

Memory  
Location  
to write to

Actual No. Characters  
Read is returned by the  
function and stored in  
this memory location

# LAB EXERCISE COMBINE THE TWO.

Assembler  
Directive

Value obtained  
from GetHandle  
API call

Maximum No.  
characters to  
Read in

Not USED  
so set to 0

**invoke WriteConsoleA, outputHandle, addr sum\_string, eax, addr bytes\_written, 0**

Windows  
API call

Memory  
Location  
to read from

Actual No. Characters  
written is returned by the  
function and stored in  
this memory location

# PART B OF LAB EXERCISE 2

► See `demo.asm`

Several thin, parallel white lines of varying lengths and orientations are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.


# NUMBERS IN ASSEMBLER.

- ▶ When we enter numbers using ReadConsole remember the values are ascii so 1234 actually comes in as 31h,32h, 33h, 34h. You need to convert them before using them.
- ▶ Easy to do you need to subtract 30h from each value read in.
- ▶ Similarly, if you want to output values to WriteConsole you first need to get the individual values and add 30h to them see examples.

# CONVERTING A ASCII NUMERIC STRING INTO A NUMBER

1. SET A MEMORY LOCATION TO HOLD A 16-bit WORD VALUE
2. READ A VALUE FROM KEYBOARD.
3. SUBTRACT 30H to get actual number.
4. STORE IT IN YOUR MEMORY LOCATION.
5. IF ANOTHER NUMBER ENTERED STORE IT TEMPORARILY IN BL register
6. GET PREVIOUS VALUE FROM STORE, MULTIPLE BY 10
7. ADD contents of BL
8. STORE BACK IN MEMORY
9. GET ANOTHER VALUE REPEAT STEP FROM 5

# LETS DO THIS AGAIN

- ▶ Characters are read in one at a time, so whilst we think of typing in the number 123 the computer actually gets a stream of single digits 1 then 2 then 3.
  - ▶ What we have to do is organize the conversion of this stream of digits into a number.
- 
- Several white lines of varying lengths and orientations are positioned on the right side of the slide, extending from the middle to the bottom right corner.

- ▶ As mentioned before each number you type actually enters the computer as an ascii character value 30H to 39H. So we first have to strip the 30H from the input to get the real digit between 1 and 9 and then if the value input is more then one digit we have to do a bit of arithmetic.



# EXAMPLE OF NUMBER MANIPULATION

- ▶ First initialise register BX to 0
- ▶ Read first character from buffer and subtract 30H from it.
- ▶ Add the value to the current contents of BX.
- ▶ If there is another character we multiply BX by 10 and add in the numeric value of the next digit at this stage BX would contain the value 12.
- ▶ Again if there is another number we again multiple BX by 10 and add the new digit at this point BX would contain 123

# PTR OPERATOR

- ▶ As you saw in the demo we needed to override the default size of an operand.
- ▶ This is only necessary when you are trying to access the variable using a size attribute that is from the one defined in the variable.

▶ E.g.

.data

```
myDouble    DWORD    12345678h
```

.code

```
mov    ax,mDouble
```

This will give an error.

- ▶ Using the WORD PTR makes it possible to move the low-order word (5678h) to AX

```
mov    ax, WORD PTR myDouble
```

# PTR OPERATOR CON'T

- ▶ Similarly,

```
mov    bl, BYTE PTR myDouble
```

Will move bl with the value 78h

- ▶ PTR can be used in conjunction with: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD or TBYTE
  - ▶ NOTE: TBYTE (define tenbyte) directive creates storage for 80-bit integers. This data type is primarily used for the storage of binary-coded decimal numbers. Manipulating these values requires special instructions in the floating-point instruction set.
  - ▶ FWORD 48-bit integer, QWORD is 64-bit integer, SBYTE, SWORD signed.

# ▶ The following few slides are for information ONLY.

- ▶ The int functions DO NOT work in 32bit assembler.
- ▶ The slides are included for completeness as you might come across the code on the web, and in most of the books that are around on assembler.

**WE WILL BE USING  
WINDOWS API CALLS**

# FIRST ASSEMBLER PROGRAM THE OLD WAY

► Lets write the traditional HELLOWORLD program in assembler.

; First define a few useful constants

CR equ 13 ; RETURN is 13D ASCII

LF equ 10 ; LINE FEED is 10D ASCII

JMP START ; Skip over the message

► ; The message and the string terminator

MESSAGE:

DB "Hello, world!", CR, LF, "\$"

START:

MOV DX, MESSAGE ; put address of the message in DX

MOV AH,9 ; MS-DOS Print string function

INT 21h ; Call DOS

MOV AH, 4Ch ; MS-DOS function-number for successful exit

INT 21h ; go back to the operating system

# INTERRUPT VECTOR TABLE

- ▶ This is a table Stored in RAM when the machine startups.
- ▶ It contains the starting addresses of all the routines held in ROM that enable the machine to work.
- ▶ We can now take advantage of these routines to make life a little easier.
- ▶ There are 256 interrupts defined. Many with sub functions.
- ▶ The following slides give some of the more common routines that we can use.
- ▶ We will see others latter in the course.

# INTERRUPT 21H

- ▶ The interrupt that is most commonly used is interrupt 21h.
- ▶ Interrupt 21h gives access to a whole range of input and output functions.
- ▶ Specifying which function you want involves putting a code number in register AH.
- ▶ Some of the more common functions follow on the next slides.

# SUB FUNCTIONS OF INT 21H

Function No In AH	Description	What it does
1	Keyboard input	Wait for a character to be typed. Ascii code is returned in AL and printed to the screen.
2	Output on the display screen	Display the character in DL on the screen.
3	Asynchronous input	Read from serial port place value received into register AL.
4	Asynchronous output	Output the character in register DL to the serial port
5	Print Character	Send character in DL register to the printer port



# Sub functions of INT 21h

6	Keyboard input/character display	<p>If DL = 0FFh, the zero flag is set to 0 if a keyboard character is ready. The character is placed in AL</p> <p>If no character available the zero flag is set to 1.</p> <p>No waiting takes place.If DL &lt;&gt; 0FFh the contents are assumed to be printable and are sent to the display</p>
7	Keyboard input/no display	<p>Waits for input from keyboard but does not display the character when it arrives in AL.</p>

# Sub functions of INT 21h

Function No In AH	Description	What it does
8	Keyboard input/no display	Same as function 7 except CTRL-BREAK service enabled
9	Display STRING	Prints a complete string of characters on the screen. The address of the string is given in DX relative to DS. It stops printing when it reaches a \$ sign. \$ not printed.
0Ah	Read keyboard String	Read from keyboard place value received into register DS:DX. First byte typed is the maximum length of the string, the second byte is the actual length. Then the rest of the string until enter pressed.

# Sub functions of INT 21h

Function No In AH	Description	What it does
8	Keyboard input/no display	Same as function 7 except CTRL-BREAK service enabled
9	Display STRING	Prints a complete string of characters on the screen. The address of the string is given in DX relative to DS. It stops printing when it reaches a \$ sign. \$ not printed.
0Ah	Read keyboard String	Read from keyboard place value received into register DS:DX. First byte typed is the maximum length of the string, the second byte is the actual length. Then the rest of the string until enter pressed.

# Sub functions of INT 21h

Function No In AH	Description	What it does
0Bh	Keyboard status	AL is set to 0FFh if a character is available from the keyboard. Else AL set to 0 A check for CTRL-BREAK is made.
0Ch	Keyboard buffer Clear	The keyboard buffer within the keyboard is cleared and INT 21h (1,6,7,8 and 0ah are allowed.
4ch	Return to DOS	Terminates program execution and returns control to DOS. An error level may be sedt in AL. Normally AL contain should be set to 0

# USING INT 21H

- ▶ In order to use the functions, it is simply a matter of loading the AH register with the function number and then calling INT 21h.

# MODERN WAY USING WINDOWS APPLICATION PROGRAMMING INTERFACE (API)

