CHAPTER 11: MS-WINDOWS PROGRAMMING

Chapter Overview

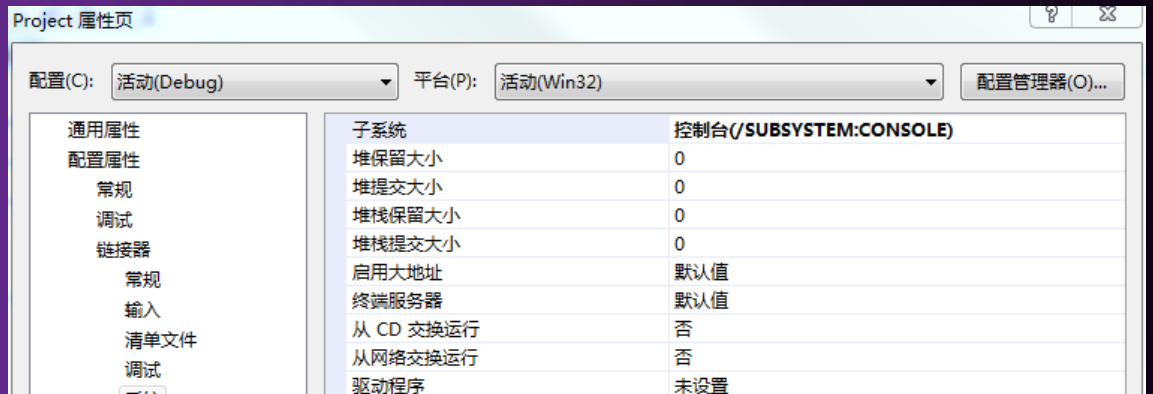
- **Win32 Console Programming**
- Writing a Graphical Windows Application
- Dynamic Memory Allocation
- x86 Memory Management

Win32 Console Programming

- Background Information
  - Win32 Console Programs
  - API and SDK
  - Windows Data Types
  - Standard Console Handles
- Console Input
- Console Output
- Reading and Writing Files
- Console Window Manipulation
- Controlling the Cursor
- Controlling the Text Color
- Time and Date Functions

Win32 Console Programs

- Run in Protected mode
- Emulate MS-DOS
- Standard text-based input and output
- Linker option : /SUBSYSTEM:CONSOLE
- The console input buffer contains a queue of input records, each containing data about an input event.
- A console screen buffer is a two-dimensional array of character and color data that affects the appearance of text in the console window.



Classifying Console Functions

- Text-oriented (high-level) console functions
  - Read character streams from input buffer
  - Write character streams to screen buffer
  - Redirect input and output
- Event-oriented (low-level) console functions
  - Retrieve keyboard and mouse events
  - Detect user interactions with the console window
  - Control window size & position, text colors

Translating Windows Data Types

Windows Type(s)	MASM Type
BOOL	DWORD
LONG	SDWORD
COLORREF, HANDLE, LPARAM, LPCTSTR, LPTSTR, LPVOID, LRESULT, UINT, WNDPROC, WPARAM	DWORD
BSTR, LPCSTR, LPSTR	PTR BYTE
WORD	WORD
LPCRECT	PTR RECT

Standard Console Handles

A handle is an unsigned 32-bit integer. The following MS-Windows constants are predefined to specify the type of handle requested:

- STD_INPUT_HANDLE
  - standard input
- STD_OUTPUT_HANDLE
  - standard output
- STD_ERROR_HANDLE
  - standard error output

GetStdHandle

- GetStdHandle returns a handle to a console stream
- Specify the type of handle (see previous slide)
- The handle is returned in EAX
- Prototype:

```
GetStdHandle PROTO,
  nStdHandle:DWORD          ; handle type
```

- Sample call:

```
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov myHandle, eax
```

Console Input

- The ReadConsole function provides a convenient way to read text input and put it in a buffer.
- Prototype:

```
ReadConsole PROTO,
    handle:DWORD,              ; input handle
    pBuffer:PTR BYTE,          ; pointer to buffer
    maxBytes:DWORD,            ; number of chars to read
    pBytesRead:PTR DWORD,      ; ptr to num bytes read
    notUsed:DWORD              ; (not used)
```

Single-Character Input

Here's how to input single characters:

- Get a copy of the current console flags by calling GetConsoleMode. Save the flags in a variable.
- Change the console flags by calling SetConsoleMode.
- Input a character by calling ReadConsole.
- Restore the previous values of the console flags by calling SetConsoleMode.

COORD and SMALL_RECT

- The COORD structure specifies X and Y screen coordinates in character measurements, which default to 0-79 and 0-24.
- The SMALL_RECT structure specifies a window's location in character measurements.

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

```
SMALL_RECT STRUCT
  Left   WORD ?
  Top    WORD ?
  Right  WORD ?
  Bottom WORD ?
SMALL_RECT ENDS
```

WriteConsole

- The WriteConsole function writes a string to the screen, using the console output handle. It acts upon standard ASCII control characters such as tab, carriage return, and line feed.
- Prototype:

```
WriteConsole PROTO,
    handle:DWORD,              ; output handle
    pBuffer:PTR BYTE,          ; pointer to buffer
    bufsize:DWORD,             ; size of buffer
    pCount:PTR DWORD,          ; output count
    lpReserved:DWORD           ; (not used)
```

Example: Console1.asm

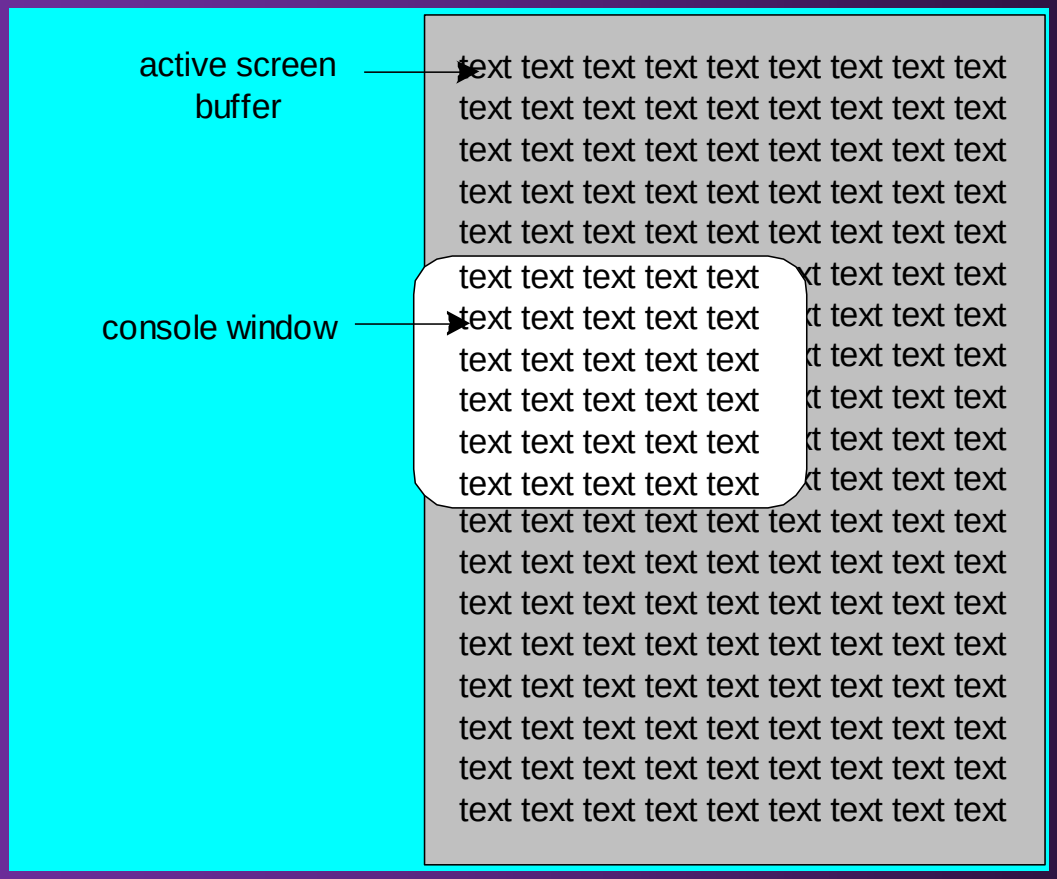
```
mainc1 PROC
; Get the console output handle:
      INVOKE GetStdHandle, STD_OUTPUT_HANDLE
      mov consoleHandle,eax
      mov ebx, messageSize

  ; Write a string to the console:
      INVOKE WriteConsole,
        consoleHandle,          ; console output handle
        ADDR message,           ; string pointer
        ebx,                    ; string length
        ADDR bytesWritten,      ; ret num bytes written
        0                       ; not used

      INVOKE ExitProcess,0
mainc1 ENDP
```

Console Window Manipulation

- Screen buffer
- Console window
- Controlling the cursor
- Controlling the text color

Screen Buffer and Console Window

- The active screen buffer (屏幕缓冲区) includes data displayed by the console window (控制台窗口).



SetConsoleTitle

SetConsoleTitle changes the console window's title. Pass it a null-terminated string:

```
.data
titleStr BYTE "Console title",0
.code
INVOKE SetConsoleTitle, ADDR titleStr
```

GetConsoleScreenBufferInfo

GetConsoleScreenBufferInfo returns information about the current state of the console window. It has two parameters: a handle to the console screen, and a pointer to a structure that is filled in by the function:

```
.data
outHandle DWORD ?
consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
.code
    INVOKE GetConsoleScreenBufferInfo,
        outHandle,
        ADDR consoleInfo
```

CONSOLE_SCREEN_BUFFER_INFO

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT
  dwSize              COORD <>
  dwCursorPos         COORD <>
  wAttributes         WORD ?
  srWindow            SMALL_RECT <>
  maxWinSize          COORD <>
CONSOLE_SCREEN_BUFFER_INFO ENDS
```

- dwSize - size of the screen buffer (char columns and rows)
- dwCursorPos - cursor location
- wAttributes - colors of characters in console buffer
- srWindow - coords of console window relative to screen buffer
- maxWinSize - maximum size of the console window

SetConsoleWindowInfo

- SetConsoleWindowInfo lets you set the size and position of the console window relative to its screen buffer.
- Prototype:

```
SetConsoleWindowInfo PROTO,
   nStdHandle:DWORD,              ; screen buffer handle
   bAbsolute:DWORD,               ; coordinate type.绝/相对
   pConsoleRect:PTR SMALL_RECT    ; window rectangle
```

SetConsoleScreenBufferSize

- SetConsoleScreenBufferSize lets you set the screen buffer size to X columns by Y rows.
- Prototype:

```
SetConsoleScreenBufferSize PROTO,
  outHandle:DWORD,          ; handle to screen buffer
  dwSize:COORD              ; new screen buffer size
```

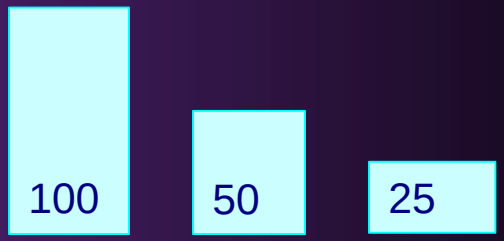
Controlling the Cursor

- GetConsoleCursorInfo
  - returns the size and visibility of the console cursor
- SetConsoleCursorInfo
  - sets the size and visibility of the cursor
- SetConsoleCursorPosition
  - sets the X, Y position of the cursor

CONSOLE_CURSOR_INFO

- Structure containing information about the console's cursor size and visibility
  - dwSize: Percentage 1 to 100 of the character cell
  - bVisible: TRUE(1) or FALSE(0)

```
CONSOLE_CURSOR_INFO STRUCT
  dwSize   DWORD ?
  bVisible DWORD ?
CONSOLE_CURSOR_INFO ENDS
```



SetConsoleTextAttribute

- Sets the foreground and background colors of all subsequent text written to the console.
- Prototype:

```
SetConsoleTextAttribute PROTO,
    hConsoleOutput:HANDLE,  ; console output handle
    wAttributes:WORD        ; color attribute
```

WriteConsoleOutputAttribute

- Copies an array of attribute values to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputAttribute PROTO,
    hConsoleOutput:DWORD,       ; output handle
    lpAttribute:PTR WORD,       ; write attributes颜色属性数组
    nLength:DWORD,              ; number of cells 颜色属性数量
    dwWriteCoord:COORD,         ; first cell coordinates
    lpNumberOfAttrsWritten:PTR DWORD     ; output count
```

WriteConsoleOutputCharacter

- The WriteConsoleOutputCharacter function copies an array of characters to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputCharacter PROTO,
  hConsoleOutput:HANDLE,    ; console output handle
  lpCharacter:PTR BYTE,     ; pointer to buffer
  nLength:DWORD,            ; size of buffer
  dwWriteCoord:COORD,       ; first cell coordinates
  lpNumberOfCharsWritten:PTR DWORD   ; output count
```

WriteColors Program

- Creates an array of characters and an array of attributes, one for each character
- Copies the attributes to the screen buffer
- Copies the characters to the same screen buffer cells as the attributes
- Sample output:

(starts in row 2, column 10)

View the source code

File Manipulation

- Win32 API Functions that create, read, and write to files:
  - CreateFile
  - ReadFile
  - WriteFile
  - SetFilePointer

CreateFile

- CreateFile either creates a new file or opens an existing file. If successful, it returns a handle to the open file; otherwise, it returns a special constant named INVALID_HANDLE_VALUE.
- Prototype:

```
CreateFile PROTO,
  pFilename:PTR BYTE,          ; ptr to filename
  desiredAccess:DWORD,         ; access mode
  shareMode:DWORD,             ; share mode
  lpSecurity:DWORD,            ; ptr to security attribs
  creationDisposition:DWORD,   ; file creation options
  flagsAndAttributes:DWORD,    ; file attributes
  htemplate:DWORD              ; handle to template file
```

CreateFile Examples (1 of 3)

Opens an existing file for reading:

```
INVOKE CreateFile,
    ADDR filename,          ; ptr to filename
    GENERIC_READ,           ; access mode
    DO_NOT_SHARE,           ; share mode
    NULL,                   ; ptr to security attributes
    OPEN_EXISTING,          ; file creation options
    FILE_ATTRIBUTE_NORMAL,  ; file attributes
    0                       ; handle to template file
```

CreateFile Examples (2 of 3)

Opens an existing file for writing:

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,          ; access mode
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0
```

CreateFile Examples (3 of 3)

Creates a new file with normal attributes, erasing any existing file by the same name:

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,
    DO_NOT_SHARE,
    NULL,
    CREATE_ALWAYS,          ; overwrite existing file
    FILE_ATTRIBUTE_NORMAL,
    0
```

ReadFile

- ReadFile reads text from an input file
- Prototype:

```
ReadFile PROTO,
    handle:DWORD,              ; handle to file
    pBuffer:PTR BYTE,          ; ptr to buffer
    nBufsize:DWORD,            ; num bytes to read
    pBytesRead:PTR DWORD,      ; bytes actually read
    pOverlapped:PTR DWORD      ; ptr to asynch info
```

WriteFile

- WriteFile writes data to a file, using an output handle. The handle can be the screen buffer handle, or it can be one assigned to a text file.
- Prototype:

```
WriteFile PROTO,
    fileHandle:DWORD,              ; output handle
    pBuffer:PTR BYTE,              ; pointer to buffer
    nBufsize:DWORD,                ; size of buffer
    pBytesWritten:PTR DWORD,       ; num bytes written
    pOverlapped:PTR DWORD          ; ptr to asynch info
```

SetFilePointer

SetFilePointer moves the position pointer of an open file. You can use it to append data to a file, and to perform random-access record processing:

```
SetFilePointer PROTO,
  handle:DWORD,                ; file handle
  nDistanceLo:SDWORD,          ; bytes to move pointer
  pDistanceHi:PTR SDWORD,      ; ptr to bytes to move
  moveMethod:DWORD             ; starting point
```

Example:

```
; Move to end of file:

INVOKE SetFilePointer,
  fileHandle,0,0,FILE_END
```

Time and Date Functions

- GetLocalTime, SetLocalTime
- GetTickCount, Sleep
- GetDateTime
- SYSTEMTIME Structure
- Creating a Stopwatch Timer

GetLocalTime, SetLocalTime

- GetLocalTime returns the date and current time of day, according to the system clock.
- SetLocalTime sets the system's local date and time.

```
GetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
```

```
SetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
```

SYSTEMTIME Structure

- SYSTEMTIME is used by date and time-related Windows API functions:

```
SYSTEMTIME STRUCT
    wYear WORD ?              ; year (4 digits)
    wMonth WORD ?             ; month (1-12)
    wDayOfWeek WORD ?         ; day of week (0-6)
    wDay WORD ?               ; day (1-31)
    wHour WORD ?              ; hours (0-23)
    wMinute WORD ?            ; minutes (0-59)
    wSecond WORD ?            ; seconds (0-59)
    wMilliseconds WORD ?      ; milliseconds (0-999)
SYSTEMTIME ENDS
```

GetTickCount, Sleep

- GetTickCount function returns the number of milliseconds that have elapsed since the system was started.
- Sleep pauses the current program for a specified number of milliseconds.

```
GetTickCount PROTO     ; return value in EAX
```

```
Sleep PROTO,
   dwMilliseconds:DWORD
```

What's Next

- Win32 Console Programming
- **Writing a Graphical Windows Application**
- Dynamic Memory Allocation
- x86 Memory Management

Writing a Graphical Windows Application

- Required Files
- POINT, RECT Structures
- MSGStruct, WNDCLASS Structures
- MessageBox Function
- WinMain, WinProc Procedures
- ErrorHandler Procedure
- Message Loop & Processing Messages
- Program Listing

MessageBox Function

Displays text in a box that pops up and waits for the user to click on a button:

```
MessageBox PROTO,
    hWnd:DWORD,
    lpText:PTR BYTE,
    lpCaption:PTR BYTE,
    uTyle:DWORD
```

```
int WINAPI MessageBox(
  _In_opt_  HWND hWnd,
  _In_opt_  LPCTSTR lpText,
  _In_opt_  LPCTSTR lpCaption,
  _In_      UINT uType
);
```

hWnd is a handle to the current window.
pText points to a null-terminated string that will appear inside the box.
pCaption points to a null-terminated string that will appear in the box's caption bar.
style is an integer that describes both the dialog box's icon (optional) and the buttons (required).

MessageBox Example

Displays a message box that shows a question, including an OK button and a question-mark icon:

```
.data
hMainWnd      DWORD ?
QuestionText  BYTE "Register this program now?",0
QuestionTitle BYTE "Trial Period Has Expired",0

.code
INVOKE MessageBox,
    hMainWnd,
    ADDR QuestionText,
    ADDR QuestionTitle,
    MB_OK + MB_ICONQUESTION
```

Required Files

- make32.bat - Batch file specifically for building this program
- WinApp.asm - Program source code
- GraphWin.inc - Include file containing structures, constants, and function prototypes used by the program
- kernel32.lib - Same MS-Windows API library used earlier in this chapter
- user32.lib - Additional MS-Windows API functions

When linking the program, use /SUBSYSTEM:WINDOWS
Not /SUBSYSTEM:CONSOLE

POINT and RECT Structures

- POINT - X, Y screen coordinates
- RECT - Holds the graphical coordinates of two opposing corners of a rectangle

```
POINT STRUCT
  ptX  DWORD ?
  ptY  DWORD ?
POINT ENDS
```

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

MSGStruct Structure

MSGStruct - holds data for MS-Windows messages (usually passed by the system and received by your application):

```
MSGStruct STRUCT
  msgWnd                DWORD ?
  msgMessage            DWORD ?
  msgWparam             DWORD ?
  msgLparam             DWORD ?
  msgTime               DWORD ?
  msgPt                 POINT <>
MSGStruct ENDS
```

WNDCLASS Structure (1 of 2)

Each window in a program belongs to a class, and each program defines a window class for its main window:

```
WNDCLASS STRUC
  style         DWORD ?     ; window style options
  lpfnWndProc   DWORD ?     ; WinProc function pointer
  cbClsExtra    DWORD ?     ; shared memory
  cbWndExtra    DWORD ?     ; number of extra bytes
  hInstance     DWORD ?     ; handle to current program
  hIcon         DWORD ?     ; handle to icon
  hCursor       DWORD ?     ; handle to cursor
  hbrBackground DWORD ?     ; handle to background brush
  lpszMenuName  DWORD ?     ; pointer to menu name
  lpszClassName DWORD ?     ; pointer to WinClass name
WNDCLASS ENDS
```

WNDCLASS Structure (2 of 2)

- style is a conglomerate of different style options, such as WS_CAPTION and WS_BORDER, that control the window's appearance and behavior.
- lpfnWndProc is a pointer to a function (in our program) that receives and processes event messages triggered by the user.
- cbClsExtra refers to shared memory used by all windows belonging to the class. Can be null.
- cbWndExtra specifies the number of extra bytes to allocate following the window instance.
- hInstance holds a handle to the current program instance.
- hIcon and hCursor hold handles to icon and cursor resources for the current program.
- hbrBackground holds a background (color) brush.
- lpszMenuName points to a menu string.
- lpszClassName points to a null-terminated string containing the window's class name.

WinMain Procedure

Every Windows application needs a startup procedure, usually named WinMain, which is responsible for the following tasks:

- Get a handle to the current program
- Load the program's icon and mouse cursor
- Register the program's main window class and identify the procedure that will process event messages for the window
- Create the main window
- Show and update the main window
- Begin a loop that receives and dispatches messages

WinProc Procedure

- WinProc receives and processes all event messages relating to a window
  - Some events are initiated by clicking and dragging the mouse, pressing keyboard keys, and so on
- WinProc decodes each message, carries out application-oriented tasks related to the message

```
WinProc PROC,
    hWnd:DWORD,        ; handle to the window
    localMsg:DWORD,    ; message ID
    wParam:DWORD,      ; parameter 1 (varies)
    lParam:DWORD       ; parameter 2 (varies)
```

(Contents of wParam and lParam vary, depending on the message.)

Sample WinProc Messages

- In the example program from this chapter, the WinProc procedure handles three specific messages:
  - WM_LBUTTONDOWN, generated when the user presses the left mouse button
  - WM_CREATE, indicates that the main window was just created
  - WM_CLOSE, indicates that the application's main window is about to close

(many other messages are possible)

ErrorHandler Procedure

- The ErrorHandler procedure has several important tasks to perform:
  - Call GetLastError to retrieve the system error number
  - Call FormatMessage to retrieve the appropriate system-formatted error message string
  - Call MessageBox to display a popup message box containing the error message string
  - Call LocalFree to free the memory used by the error message string

(sample)

ErrorHandler Sample

```
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
```

Message Loop

In WinMain, the message loop receives and dispatches (relays) messages:

```
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
```

Processing Messages

WinProc receives each message and decides what to do with it:

```
WinProc PROC, hWnd:DWORD, localMsg:DWORD,
    wParam:DWORD, lParam:DWORD
    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
    (etc.)
```

- (WinApp.asm)

Program Listing

- View the program listing (WinApp.asm)
- Run the program

When linking the program, remember to replace

/SUBSYSTEM:CONSOLE

with: /SUBSYSTEM:WINDOWS

What's Next

- Win32 Console Programming
- Writing a Graphical Windows Application
- **Dynamic Memory Allocation**
- IA-32 Memory Management

Dynamic Memory Allocation

- Reserving memory at runtime for objects
  - aka *heap allocation*
  - standard in high-level languages (C++, Java)
- Heap manager
  - allocates large blocks of memory
  - maintains free list of pointers to smaller blocks
  - manages requests by programs for storage

Windows Heap-Related Functions

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.)

Sample Code

- Get a handle to the program's existing heap
  or Create a handle to a private heap.

```
HEAP_START = 2000000          ;   2 MB
HEAP_MAX = 400000000          ; 400 MB
.data
hHeap HANDLE ?

.code
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
.IF eax == NULL               ; cannot get handle
   jmp quit
.ELSE
   mov hHeap,eax              ; handle is OK
.ENDIF
```

Sample Code

- Allocate block of memory from existing heap:

```
.data
hHeap HANDLE ?       ; heap handle
pArray DWORD ?       ; pointer to array

.code
INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 1000
.IF eax == NULL
   mWrite "HeapAlloc failed"
   jmp quit
.ELSE
   mov pArray,eax
.ENDIF
```

Sample Code

- Free a block of memory previously created by calling HeapAlloc:

```
.data
hHeap HANDLE ?                  ; heap handle
pArray DWORD ?                  ; pointer to array

.code
INVOKE HeapFree,
   hHeap,                       ; handle to heap
   0,                           ; flags
   pArray                       ; pointer to array
```

CHAPTER 14:
16-BIT MS-DOS PROGRAMMING

Chapter Overview

- **MS-DOS and the IBM-PC**
- MS-DOS Function Calls (INT 21h)
- Standard MS-DOS File I/O Services

MS-DOS and the IBM-PC

- Real-Address Mode
- MS-DOS Memory Organization
- MS-DOS Memory Map
- Redirecting Input-Output
- Software Interrupts
- INT Instruction
- Interrupt Vectoring Process
- Common Interrupts

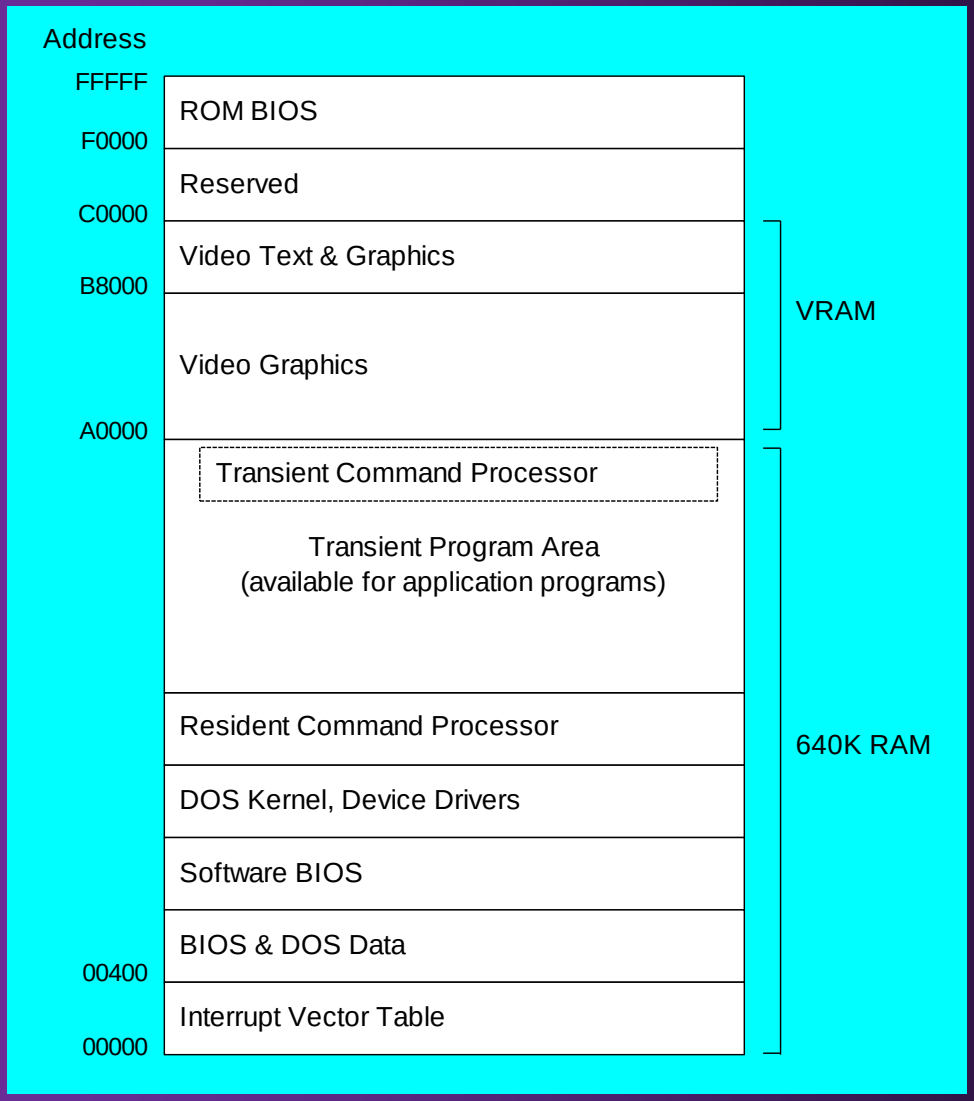
Real-Address Mode

- Real-address mode (16-bit mode) programs have the following characteristics:
  - Max 1 megabyte addressable RAM
  - Offsets are 16 bits
  - No memory boundary protection
  - Single tasking
- IBM PC-DOS: first Real-address OS for IBM-PC
  - Has roots in Gary Kildall's highly successful Digital Research CP/M
  - Later renamed to MS-DOS, owned by Microsoft

MS-DOS Memory Organization

- Interrupt Vector Table
- BIOS & DOS data
- Software BIOS
- MS-DOS kernel
- Resident command processor
- Transient programs
- Video graphics & text
- Reserved (device controllers)
- ROM BIOS

MS-DOS Memory Map



Redirecting Input-Output (1 of 2)

- Input-output devices and files are interchangeable
- Three primary types of I/O:
  - Standard input (console, keyboard)
  - Standard output (console, display)
  - Standard error (console, display)
- Symbols borrowed from Unix:
  - < symbol: *get input from*
  - > symbol: *send output to*
  - | symbol: pipe output from one process to another
- Predefined device names:
  - PRN, CON, LPT1, LPT2, NUL, COM1, COM2

Redirecting Input-Output (2 of 2)

- Standard input, standard output can both be redirected
- Standard error cannot be redirected
- Suppose we have created a program named myprog.exe that reads from standard input and writes to standard output. Following are MS-DOS commands that demonstrate various types of redirection:

```
myprog < infile.txt

myprog > outfile.txt

myprog < infile.txt > outfile.txt
```

INT Instruction

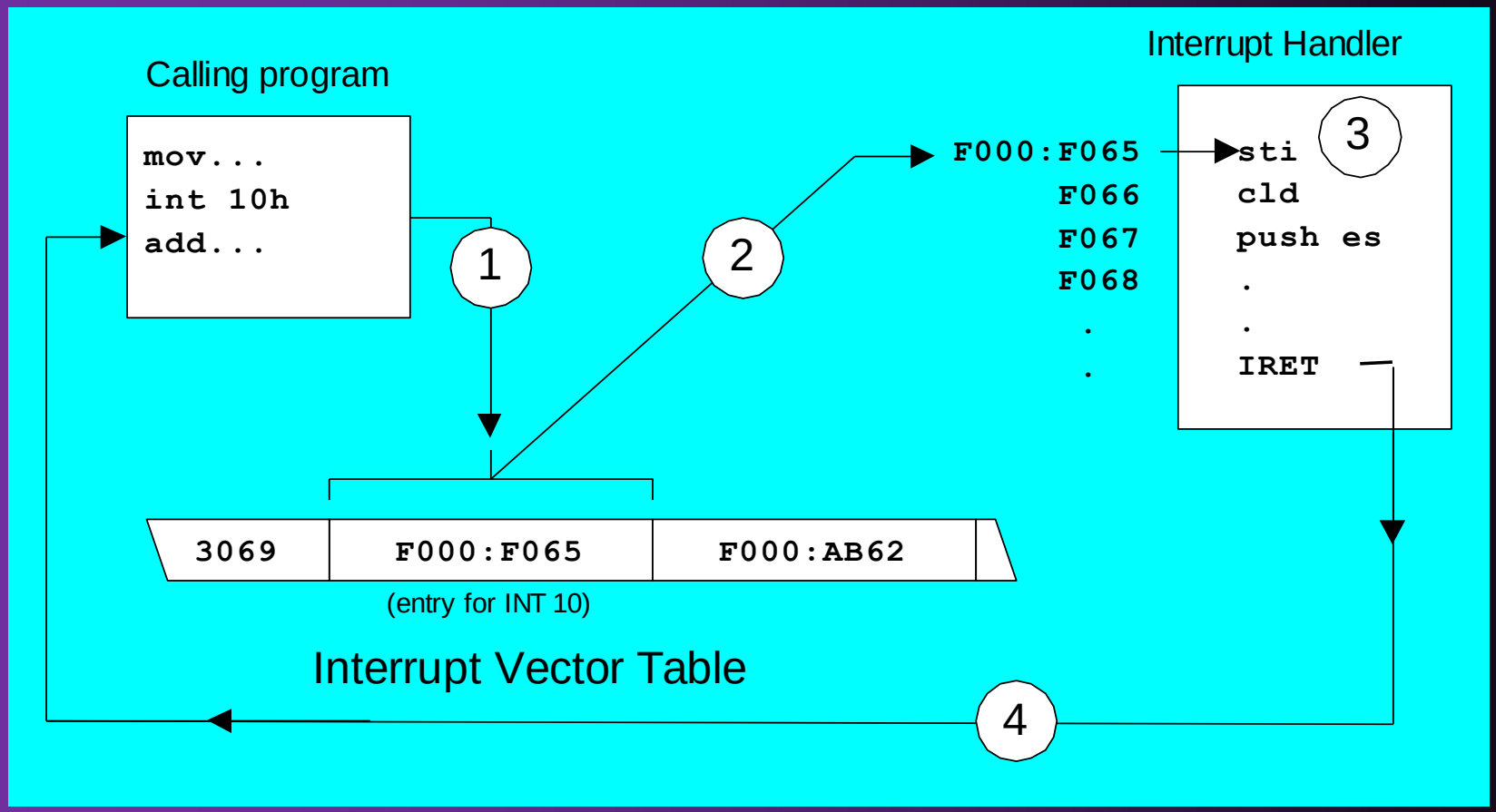
- The INT instruction executes a software interrupt.
- The code that handles the interrupt is called an interrupt handler.
- Syntax:

```
INT number
(number = 0..FFh)
```

The Interrupt Vector Table (IVT) holds a 32-bit segment-offset address for each possible interrupt handler.

Interrupt Service Routine (ISR) is another name for interrupt handler.

Interrupt Vectoring Process



Common Interrupts

- INT 10h Video Services
- INT 16h Keyboard Services
- INT 17h Printer Services
- INT 1Ah Time of Day
- INT 1Ch User Timer Interrupt
- INT 21h MS-DOS Services

What's Next

- MS-DOS and the IBM-PC
- **MS-DOS Function Calls (INT 21h)**
- Standard MS-DOS File I/O Services

MS-DOS Function Calls (INT 21h)

- ASCII Control Characters
- Selected Output Functions
- Selected Input Functions
- Example: String Encryption
- Date/Time Functions

INT 21h Function 4Ch: Terminate Process

- Ends the current process (program), returns an optional 8-bit return code to the calling process.
- A return code of 0 usually indicates successful completion.

```
mov ah,4Ch              ; terminate process
mov al,0                ; return code
int 21h

; Same as:

.EXIT 0
```

Selected Output Functions

- ASCII control characters
- 02h, 06h - Write character to standard output
- 05h - Write character to default printer
- 09h - Write string to standard output
- 40h - Write string to file or device

ASCII Control Characters

Many INT 21h functions act upon the following control characters:

- 08h - Backspace (moves one column to the left)
- 09h - Horizontal tab (skips forward n columns)
- 0Ah - Line feed (moves to next output line)
- 0Ch - Form feed (moves to next printer page)
- 0Dh - Carriage return (moves to leftmost output column)
- 1Bh - Escape character

INT 21h Functions 02h and 06h:
Write Character to Standard Output

Write the letter 'A' to standard output:

```
mov ah,02h
mov dl,'A'
int 21h
```

or: mov ah,2

Write a backspace to standard output:

```
mov ah,06h
mov dl,08h
int 21h
```

INT 21h Function 05h:
Write Character to Default Printer

Write the letter 'A':

```
mov ah,05h
mov dl,65
int 21h
```

Write a horizontal tab:

```
mov ah,05h
mov dl,09h
int 21h
```

INT 21h Function 09h:
Write String to Standard Output

- The string must be terminated by a '$' character.
- DS must point to the string's segment, and DX must contain the string's offset:

```
.data
string BYTE "This is a string$"

.code
mov  ah,9
mov  dx,OFFSET string
int  21h
```

INT 21h Function 40h:
Write String to File or Device

Input: BX = file or device handle (console = 1), CX = number of bytes to write, DS:DX = address of array

```
.data
message BYTE "Writing a string to the console"
bytesWritten WORD ?

.code
    mov ah,40h
    mov bx,1
    mov cx,LENGTHOF message
    mov dx,OFFSET message
    int 21h
    mov bytesWritten,ax
```

Selected Input Functions

- 01h, 06h - Read character from standard input
- 0Ah - Read array of buffered characters from standard input
- 0Bh - Get status of the standard input buffer
- 3Fh - Read from file or device

INT 21h Function 01h:
Read single character from standard input

- Echoes the input character
- Waits for input if the buffer is empty
- Checks for Ctrl-Break (^C)
- Acts on control codes such as horizontal Tab

```
.data
char BYTE ?
.code
mov ah,01h
int 21h
mov char,al
```

INT 21h Function 06h:

Read character from standard input without waiting

- Does not echo the input character
- Does not wait for input (use the Zero flag to check for an input character)
- Example: repeats loop until a character is pressed.

```
.data
char BYTE ?
.code
L1: mov  ah,06h          ; keyboard input
    mov  dl,0FFh         ; don't wait for input
    int  21h
    jz   L1              ; no character? repeat loop
    mov  char,al         ; character pressed: save it
    call DumpRegs        ; display registers
```

INT 21h Function 0Ah:
Read buffered array from standard input (1 of 2)

- Requires a predefined structure to be set up that describes the maximum input size and holds the input characters.
- Example:

```
count = 80

KEYBOARD STRUCT
    maxInput BYTE count             ; max chars to input
    inputCount BYTE ?               ; actual input count
    buffer BYTE count DUP(?)        ; holds input chars
KEYBOARD ENDS
```

INT 21h Function 0Ah (2 of 2)

Executing the interrupt:

```
.data
kybdData KEYBOARD <>

.code
    mov ah,0Ah
    mov dx,OFFSET kybdData
    int 21h
```

INT 21h Function 0Bh:
Get status of standard input buffer

- Can be interrupted by Ctrl-Break (^C)
- Example: loop until a key is pressed. Save the key in a variable:

```
L1: mov ah,0Bh        ; get buffer status
    int 21h
    cmp al,0          ; buffer empty?
    je  L1            ; yes: loop again
    mov ah,1          ; no: input the key
    int 21h
    mov char,al       ; and save it
```

Example: String Encryption

Reads from standard input, encrypts each byte, writes to standard output.

```
XORVAL = 239                ; any value between 0-255
.code
main PROC
     mov  ax,@data
     mov  ds,ax
L1:  mov  ah,6              ; direct console input
     mov  dl,0FFh           ; don't wait for character
     int  21h               ; AL = character
     jz   L2                ; quit if ZF = 1 (EOF)
     xor  al,XORVAL
     mov  ah,6              ; write to output
     mov  dl,al
     int  21h
     jmp  L1                ; repeat the loop
L2:  exit
```

INT 21h Function 3Fh:
Read from file or device

- Reads a block of bytes.
- Can be interrupted by Ctrl-Break (^C)
- Example: Read string from keyboard:

```
.data
inputBuffer BYTE 127 dup(0)
bytesRead WORD ?
.code
mov  ah,3Fh
mov  bx,0                          ; keyboard handle
mov  cx,127                        ; max bytes to read
mov  dx,OFFSET inputBuffer         ; target location
int  21h
mov  bytesRead,ax                  ; save character count
```

Date/Time Functions

- 2Ah - Get system date
- 2Bh - Set system date *
- 2Ch - Get system time
- 2Dh - Set system time *

* may be restricted by your user profile if running a console window under Windows NT, 2000, and XP.

INT 21h Function 2Ah:
Get system date

- Returns year in CX, month in DH, day in DL, and day of week in AL

```
mov   ah,2Ah
int   21h
mov   year,cx
mov   month,dh
mov   day,dl
mov   dayOfWeek,al
```

INT 21h Function 2Bh:
Set system date

- Sets the system date. AL = 0 if the function was successful in modifying the date.

```
mov   ah,2Bh
mov   cx,year
mov   dh,month
mov   dl,day
int   21h
cmp   al,0
jne   failed
```

INT 21h Function 2Ch:
Get system time

- Returns hours (0-23) in CH, minutes (0-59) in CL, and seconds (0-59) in DH, and hundredths (0-99) in DL.

```
mov   ah,2Ch
int   21h
mov   hours,ch
mov   minutes,cl
mov   seconds,dh
```

INT 21h Function 2Dh:
Set system time

- Sets the system date. AL = 0 if the function was successful in modifying the time.

```
mov   ah,2Dh
mov   ch,hours
mov   cl,minutes
mov   dh,seconds
int   21h
cmp   al,0
jne   failed
```

Example: Displaying the Date and Time

- Displays the system date and time, using INT 21h Functions 2Ah and 2Ch.
- Demonstrates simple date formatting
- View the source code
- Sample output:

```
Date: 12-8-2001,   Time: 23:01:23
```

ToDo: write a procedure named ShowDate that displays any date in mm-dd-yyyy format.

What's Next

- MS-DOS and the IBM-PC
- MS-DOS Function Calls (INT 21h)
- **Standard MS-DOS File I/O Services**

Standard MS-DOS File I/O Services

- 716Ch - Create or open file
- 3Eh - Close file handle
- 42h - Move file pointer
- 5706h - Get file creation date and time
- Selected Irvine16 Library Procedures
- Example: Read and Copy a Text File
- Reading the MS-DOS Command Tail
- Example: Creating a Binary File

INT 21h Function 716Ch:
Create or open file

- AX = 716Ch
- BX = access mode (0 = read, 1 = write, 2 = read/write)
- CX = attributes (0 = normal, 1 = read only, 2 = hidden, 3 = system, 8 = volume ID, 20h = archive)
- DX = action (1 = open, 2 = truncate, 10h = create)
- DS:SI = segment/offset of filename
- DI = alias hint (optional)

Example: Create a New File

```
mov   ax,716Ch                  ; extended open/create
mov   bx,2                      ; read-write
mov   cx,0                      ; normal attribute
mov   dx,10h + 02h              ; action: create + truncate
mov   si,OFFSET Filename
int   21h
jc    failed
mov   handle,ax                 ; file handle
mov   actionTaken,cx            ; action taken to open file
```

Example: Open an Existing File

```
mov   ax,716Ch                    ; extended open/create
mov   bx,0                        ; read-only
mov   cx,0                        ; normal attribute
mov   dx,1                        ; open existing file
mov   si,OFFSET Filename
int   21h
jc    failed
mov   handle,ax                   ; file handle
mov   actionTaken,cx              ; action taken to open file
```

INT 21h Function 3Eh:
Close file handle

- Use the same file handle that was returned by INT 21h when the file was opened.
- Example:

```
.data
filehandle WORD ?
.code
    mov  ah,3Eh
    mov  bx,filehandle
    int  21h
    jc   failed
```

INT 21h Function 42h:
Move file pointer

Permits random access to a file (text or binary).

```
mov   ah,42h
mov   al,0              ; offset from beginning
mov   bx,handle
mov   cx,offsetHi
mov   dx,offsetLo
int   21h
```

AL indicates how the pointer's offset is calculated:

- 0: Offset from the beginning of the file
- 1: Offset from the current pointer location
- 2: Offset from the end of the file

INT 21h Function 5706h:
Get file creation date and time

- Obtains the date and time when a file was created (not necessarily the same date and time when the file was last modified or accessed.)

```
mov ax,5706h
mov bx,handle          ; handle of open file
int 21h
jc  error
mov date,dx
mov time,cx
mov milliseconds,si
```

Selected Irvine16 Library Procedures

- 16-Bit ReadString procedure
- 16-Bit WriteString procedure

Summary (Chap 11)

- 32-bit console programs
  - read from the keyboard and write plain text to the console window using Win32 API functions
- Important functions
  - ReadConsole, WriteConsole, GetStdHandle, ReadFile, WriteFile, CreateFile, CloseHandle, SetFilePointer
- Dynamic memory allocation
  - HeapAlloc, HeapFree

Summary (Chap 14)

- MS-DOS applications
  - 16-bit segments, segmented addressing, running in real-address mode
  - complete access to memory and hardware
- Software interrupts
  - processed by interrupt handlers
- INT (call to interrupt procedure) instruction
  - pushes flags & return address on the stack
  - uses interrupt vector table to find handler
- BIOS Services (INT 10h, INT 16h, INT 17h, ...)
- MS-DOS Services (INT 21h)

Homework

- Reading Chap 11, 14
- Exercises
- Thanks!