Windows Programming CSA2040

Kristian Guillaumier
http://www.cs.um.edu.mt/~kguil
kguil@cs.um.edu.mt

Getting Started

- Examples are in C and/or PowerBASIC. It will be trivial to port from and to different languages.
- All the examples in the “Petzold” book are available ported to PowerBASIC. You can get them from: http://www.powerbasic.com/files/pub/pbwin/Petzold.zip
- You can download the Borland C/C++ Compiler Version 5.5 for free from: http://www.borland.com/products/downloads/download_cbuilder.html
- There is a good WIN32 tutorial at: http://www.winprog.org/tutorial/ - a number of examples here are borrowed from this site.
- www.allapi.net is cool.
- msdn.microsoft.com is the definitive resource.

Recommended Books

- **Programming Windows, The Definitive Guide to the Win32 API** by Charles Petzold, 5th edition, Microsoft Press, ISBN: 157231995X.
- **Windows Programming with C++** by Henning Hansen, Addison Wesley Professional, ISBN: 0201758814.

Getting Started

- Programming Windows, requires you to understand the services offered by the WIN32 Application Programming Interface (API).
- The API consists of a number of DLLs containing common Windows functions.
  - Kernel32.dll
  - GDI32.dll
  - User32.dll
  - …
- To access the API functions, constant declarations and types you will need to “wrap” them in your code. In C this is already available in “windows.h” and in PowerBASIC, this is available in “WIN32API.INC”.

WIN32 Hello World

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpCmdLine,
                   int       nCmdShow)
{
  MessageBox(NULL, "Hello World!",
                   "My Caption", MB_OK);
  return 0;
}
```


What's Going On? (1)

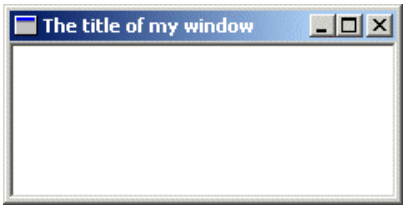
- **windows.h** contains the declarations of functions and constants such as *MessageBox*, *MB_OK*, *HINSTANCE* and *LPSTR*.
- **WinMain** is the equivalent of the main() functions in C – it is the starting point of a *Windows* application:
  - **hInstance**: Handle/pointer to the EXE in memory.
  - **hPrevInstance**: Always NULL – Never used.
  - **lpCmdLine**: Pointer to the command line string.
  - **nCmdShow**: an integer determining whether the window will be visible/hidden/…

What's Going On? (2)

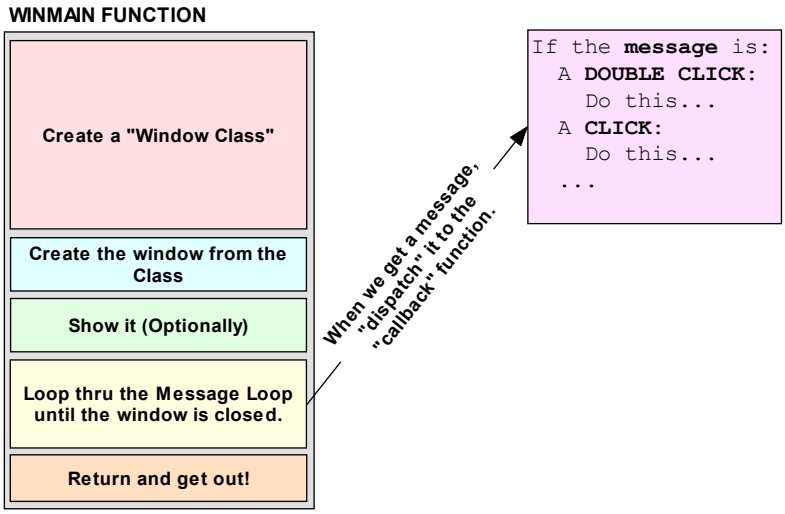
- The *hInstance* handle is used as the pointer to the EXE in memory so it is useful to locate resources such as images and icons in the program.
- In C, the **WINAPI** calling convention before the *WinMain* function is equivalent to *_stdcall*. In some languages such as PB it is not necessary.
- The WIN32 header file defines a number of types such as **LPSTR** (this is exactly equivalent to char*).

More Windows

- We will now see how to create a simple window like:



General Structure (1)



The Window Class

- **A Window Class is NOT related to Object Oriented Software.**
- In Windows everything is more or less a Window (including a button, combo box, …). The type of window is determined by it's class.
- A Window Class is a special structure (**WNDCLASS**) that is populated to specify the general properties of the window (e.g. it's icon, background colour, cursor, …).
- One of the most important properties is the definition of the Callback function (more on this later).
- Once a class has been created it is "Registered".

Creating the Window

- Once a class structure has been populated and registered, a window is created based on it.
- To create the window, the **CreateWindow** function is used (There is a variant called CreateWindowEx).
- Some arguments, CreateWindow takes are:
  - The class name to base this window on.
  - The text in the title bar.
  - The type of window (e.g. borderless, tool window, …)
  - The x,y coordinate of the top-left corner – in pixels.
  - The width and height of the window – in pixels.
  - …

Showing the Window (1)

- The CreateWindow function returns a handle/pointer to the window just created. The window is not yet visible.
- The window will be referred to it using its handle.
- To show the window, the **ShowWindow** API call is used. ShowWindow takes 2 arguments:
  - The handle of the window to show/hide.
  - An integer constant determining whether to show/hide the window.

Showing the Window (2)

- The show command can be:
  - **SW_HIDE** - Hides the window and activates another window.
  - **SW_MAXIMIZE** - Maximizes the specified window.
  - **SW_MINIMIZE** - Minimizes the specified window and activates the next top-level window in the Z order.
  - **SW_RESTORE** - Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized window.
  - **SW_SHOW** - Activates the window and displays it in its current size and position.
  - …
- Usually after a call to ShowWindow, another call to **UpdateWindow** is made. This call basically makes sure the window is displayed correctly.

The Message Loop (1)

- A windows program has a special "**message queue**".
- Whenever something happens to the window a message is placed in its queue. For example, if the window is clicked a "click" message is placed on its queue.
- We will use a while loop to retrieve messages from this queue and send them to the callback function for processing.
- Each message is defined by an integer constant. For example the WM_CREATE message is sent to the window when it is created. It is more-or-less equivalent to the Form_Load event in Visual Basic.

The Message Loop (2)

- Each message can be accompanied by some parameters. For example a mouse move message would be accompanied by the corresponding x and y mouse coordinates.
- These parameters are sent together with the message to the callback function.
- You can have a maximum of 2 parameters. These are called wParam and lParam. They are both long integers (signed 32-bit).
- Note that wParam and lParam being long integers may be pointers to whole data structures.

The Real Thing (1)

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

    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    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 = g_szClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
    {
      MessageBox(NULL, "Window Registration Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
      return 0;
    }
```

Prepare the Window Class

Tell the Class which function will act as the Callback

Register the Class

The Real Thing (2)

```
    // Step 2: Creating the Window
    hwnd = CreateWindowEx(WS_EX_CLIENTEDGE,
                          g_szClassName,
                          "The title of my window",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          240, 120, NULL, NULL,
                          hInstance, NULL);

    if (hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
                          MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }
    ShowWindow(hwnd, nCmdShow);

    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

Create the Window

Show It

Loop Thru the Messages the Window Receives and send them to the callback function,

The Real Thing (3)

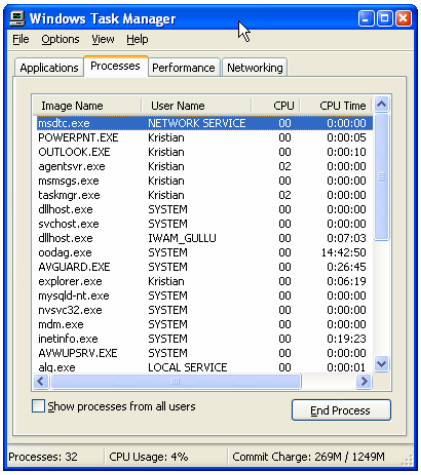
```
// Step 4: the Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_CLOSE:
            DestroyWindow(hwnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

Select which Message
WE CHOOSE
to Handle

Those we do not handle ourselves, we'll ask Windows to process!

Processes (1)

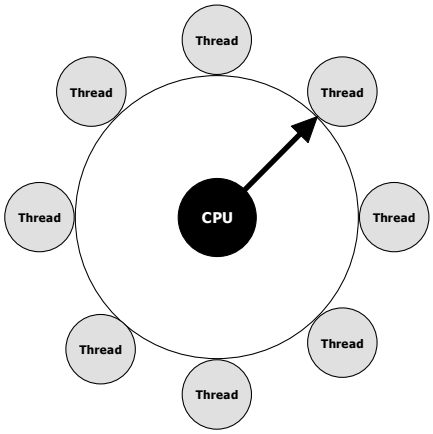
- Sometimes defined as an instance of a running program.
- You can check the processes running on your machine in Task Manager:



Processes (2)

- In Win32 each process owns a 4-GB address space.
- IMPORTANT: a process on its own does not execute anything. For execution a process requires at least on **thread**.
- A process without threads is automatically destroyed.
- When a Win32 process is created, a **Primary Thread** is automatically created for you.
- The primary thread can then create others.

Running Multiple Threads



1. Windows will allocate timeslices (quantums) of CPU time for each thread to execute.
2. Round-Robin scheduling is used (note that threads can have different and changing priorities).
3. This gives the *illusion* that things are executing concurrently.

Environment Variables (1)

- Each process is assigned an **Environment Block**.
- An environment block is simply a portion of allocated memory (owned by the process), containing strings like:

```
VarName1=VarValue1\0
VarName2=VarValue2\0
...
VarNameN=VarValueN\0
\0
```

Environment Variables (2)

- In special cases, environment variables may be used to pass special global parameters to an application.
- In Windows 9X, these variables are set in a special file called autoexec.bat which is parsed when Windows is started. In Windows NT/2000/XP these values can be set from My Computer→Properties.
- Values may be read and written using:
  - `GetEnvironmentVariable`
  - `SetEnvironmentVariable`

Current Drive and Directory

- The **Current Drive** of the **Current Directory** is the default path where Windows looks for files when you try to access them without supplying a fully qualified filename.
- For example if you call the **CreateFile** API call to create a file, if you do not specify the full path, the file will be created in the current directory.
- The current drive/directory is maintained on a per process basis. So all threads in the process will use the same values.
- You can obtain or change the current directories using the following API calls:
  - `GetCurrentDirectory`
  - `SetCurrentDirectory`

Creating Processes (1)

- A process is created when your application is started.
- Note that WinMain isn't really called by the operating system. Instead, it is "expanded" by the compiler.
- Processes are created using the CreateProcess API call (see msdn.microsoft.com for details regarding the function parameters)
- Then the function is called, a process is not actually created. Instead,
  - A small data structure is initialised containing statistical info regarding the process.
  - 4-GB of virtual address space is created.
  - The code and data for the process and associated DLLs are loaded in the address space.

Creating Processes (2)

- Next, a thread is created for the process. This will be the primary thread.
- This thread will eventually run your WinMain function.
- A process can terminate in the following ways:
  - A thread calls the **ExitProcess** API call.
  - A thread in *another process* calles **TerminateProcess** (not very nice).
  - All threads in the process complete.

Process Termination (1)

- A process will terminate when a thread in the process calls ExitProcess.
- ExitProcess is usually automatically called by the primary thread immediately after your WinMain function has completed.
- A separate process can terminate another one by calling TerminateProcess.
- Except in special cases this is discouraged because:
  - When a process terminates properly, all attached DLLs are notified.
  - Using TerminateProcess, this does not happen.

Threads

- Threads must “live” within the context of a process.
- A thread is basically a unit of execution within a process.
- Example: Background printing in Word for Windows.
- On a single processor, threads give the *illusion* that things are happening concurrently.
- Although threads are “cool” and very useful, there are a number of problems associated with them.

Thread Issues (1)

- Consider the following scenario:
  - A user clicks the print button on a Word Processor.
  - The print thread started executing – repaginating, rendering the page, sending to printer, etc…
  - The user can start editing the document when the above is happening.
  - Global Variables:

Thread Issues (2)

```
Global Counter As Long
```

```
Function Calc
  ... {b is 16, x is 1}
  Counter = Sqrt(b)
  Counter = Counter + x*2
  Return Counter
End Function
```

Context Switch

```
Function Calc
  ... {b is 4, x is 3}
  Counter = Sqrt(b)
  Counter = Counter + x*2
  Return Counter
End Function
```

Thread Properties

- Each thread has it's own stack for local variables, etc…
- This stack is allocated from the address space of the main process.
- Static and global variables are shared by all threads in the process.
- Each thread has it's set of CPU registers. A special **Context** structure holds the state of these registers when the thread was last executing.
- This structure is probably the only CPU-specific structure in the API.

Thread Termination

- A thread can terminate in 3 ways:
  - The thread calls the ExitThread API call to terminate itself.
  - Another thread within the same process calls TerminateThread (passing the handle to the thread to terminate) – Webserver monitor thread example.
  - The process "owning" all the threads exits.

Thread Scheduling (1)

- A preemptive operating system must have some defined algorithm for determining when a thread runs and for how long.
- Each thread has a priority ranging from 0 to 31. A thread with priority zero is a special thread used for “memory cleanup”. One system thread has this priority level and no other thread can be assigned this priority.
- The scheduler assigns each priority 31 thread to a CPU to execute.
- Once all priority 31 threads are given a timeslice, another timeslice to each of the priority 31 threads is given.
- This continues until there are no remaining priority 31 threads. Then all priority 30 threads are processed in the same way… and so on.

Thread Scheduling (2)

- Using this technique low priority threads may suffer from a condition known as **Starvation**.
- Also, if a priority 5 thread is running and there is a thread with a higher priority waiting to be serviced, the priority 5 thread is *immediately* suspended for the system to service the higher priority one (even if it is in the middle of a timeslice).

Assigning Priorities (1)

- When a process is created, it is assigned one of 4 priority levels:
  - Idle: Level 4
  - Normal: Level 8
  - High: Level 13
  - Real time: Level 24
- Any thread created in the process will be given that priority as a default.
- The Normal priority level is the one most commonly used.
- The normal priority class is special – it can be “boosted” depending on whether it is a foreground window or not.

Assigning Priorities (2)

- In Windows NT a “boosted” priority is given a bigger timeslice.
- In Windows 9X a “boosted” priority increases the thread priority value by 1 – A “boosted” normal thread has a priority of 8+1=9.
- Real time priority should almost never be used. Even the processes/threads handling the CTRL+ALT+DEL buttons, background disk flushing, mouse and keyboard get a lower priority. This may cause system instability.
- Process “base” priorities can be changed at runtime using the `(Get/Set)PriorityClass` API functions.

Assigning Priorities (3)

- When a thread is created, it is given the priority of the process (base priority).
- You can change the thread's priority relative to the base priority using the `SetThreadPriority` API call.
  - Lowest = Base - 2
  - Below Normal = Base - 1
  - Normal = Base
  - Above Normal = Base + 1
  - Highest = Base + 2
  - Critical = 15, except if the process is real time. Then the priority becomes 31.

Dynamic Link Libraries

- You can think of a DLL as a module containing a set of autonomous functions.
- Each process has and address space (portion of memory allocated to it).
- For functions in a DLL to be used in a program (process), the contents of the DLL must be loaded in the caller's address space:
  - Load time linking.
  - Run-time linking.
- When linked in the address space of the process, all DLL global variables and functions become part of the process code (a global variable in a DLL is global to the calling process). HeapCreate, HeapAlloc, HeapFree example.

Implicit Linking

- When a program is compiled you specify a LIB file to the linker (or it is automatically done for you).
- The LIB file contains a list of functions in the DLL.
- The linker will then embed information in the EXE to indicate the names of the DLLs required by your program.
- When Windows loads the EXE file, it will search for the DLLs required. Windows looks for DLLs in order:
  - The folder where the EXE lives.
  - The current directory of the process.
  - The windows system or system32 folder.
  - The windows folder.
  - The folders listed in your PATH environment variable.

Explicit Linking

- A process can explicitly link to a DLL using the **LoadLibrary(dll_file_name)** API call.
- This function locate the DLL, map it into the process address space and return the virtual memory address where the DLL was mapped (HINST).
- See **LoadLibraryEx** for a variation of the above.

Usage Counts

- When your process loads a DLL for the first time, it actually loads it and sets its **usage count** by 1.
- If your process loads a DLL for the second time, it is NOT reloaded but the usage count is incremented again.
- A FreeLibrary call decrements this usage count. When the usage count reaches 0 it is unmapped from the process space.
- DLL usage counts are maintained on a per-process basis.

DLL Entry Points

- Just like an application has a main function, a DLL has it's equivalent. It is called DLLMain.
- In C:

```
BOOL WINAPI DllMain(HINSTANCE hInst, DWORD dReason, DWORD
                                                    dReserved)
{
}
```

- In PB:

```
FUCNTION DLLMAIN(BYVAL hInstance&, BYVAL Reason&, _
                                 BYVAL Reserved&)  _
                                           EXPORT AS LONG
```

Anatomy of DLLMain (1)

```
FUNCTION DllMain(BYVAL hInstance AS LONG, BYVAL Reason AS LONG, BYVAL Reserved AS LONG)
                                                                    EXPORT AS LONG

  SELECT CASE Reason
    CASE %DLL_PROCESS_ATTACH
      MSGBOX "%DLL_PROCESS_ATTACH"
      DllMain= 1
      EXIT FUNCTION

    CASE %DLL_PROCESS_DETACH
      MSGBOX "%DLL_PROCESS_DETACH"
      EXIT FUNCTION

    CASE %DLL_THREAD_ATTACH
      MSGBOX "%DLL_THREAD_ATTACH"
      EXIT FUNCTION

    CASE %DLL_THREAD_DETACH
      MSGBOX "%DLL_THREAD_DETACH"
      EXIT FUNCTION

  END SELECT
END FUNCTION
```


Anatomy of DLLMain (2)

- The main purpose of the select/switch statement in DllMain is to provide a place for per-process or per-thread initialisation or clean up.
- When a DLL is mapped into a process address space the **Reason** is DLL_PROCESS_ATTACH.
- When a DLL is unmapped from a process address space we have DLL_PROCESS_DETACH.
- Note: The **TerminateProcess** API call, will NOT call the DllMain function resulting in the detach block never executing.

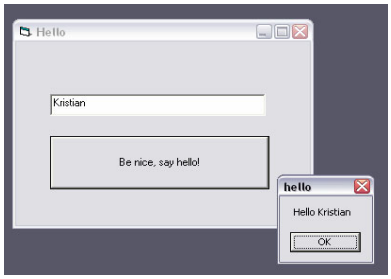
Anatomy of DLLMain (3)

- When a thread is created in a process, Windows examines all the mapped DLLs and calls their DllMain function with a DLL_THREAD_ATTACH reason. **After all the DllMains have been called, the thread function will execute**. This **Reason** is not executed if the thread is the primary thread of the process.
- When a thread terminates, the DllMains of the mapped DLLs are called with the DLL_THREAD_DETACH reason.

Reality Check (1) - Overview

- Creating a DLL in PowerBASIC or C and running it from Visual Basic.
- Problem:
  - Create a DLL with one function called **Hello**.
  - The function takes 1 string argument called **name**. The argument is taken by reference (not by value).
  - The function returns after changing the value of name to **"Hello " & name**.

Reality Check (2) - VB



Note:
VB requires ByRef string arguments as they are universally understood to be declared as ByVal. This is an issue related to type conversion.

Be careful.

```
Private Declare Function Hello Lib "hello.dll" (ByVal Name_ As String) As Long

Private Sub Command1_Click()
  Dim a As String
  a = Text1.Text & Space(255) ' Alloc enough space to the string.
  Hello a

  MsgBox Trim(a)
End Sub
```


Reality Check (3) - PB

```
#Compile Dll
#Include "win32api.inc"

Function DllMain(ByVal hInstance As Long, _
                 ByVal Reason    As Long, _
                 ByVal Reserved  As Long) _
                 Export As Long

  Select Case Reason
    Case %DLL_PROCESS_ATTACH
      DllMain = 1
      Exit Function
    Case %DLL_PROCESS_DETACH
      Exit Function
    Case %DLL_THREAD_ATTACH
      Exit Function
    Case %DLL_THREAD_DETACH
      Exit Function
  End Select

End Function

Function Hello Alias "Hello" (ByRef Name_ As Asciiz) Export As Long
  Name_ = "Hello " & Name_
  Function = 1
End Function
```

Dynamic Memory

- In general, to dynamically allocate memory under Win32, you should use:
  - HeapCreate
  - HeapAlloc
  - HeapFree
  - HeapDestroy
- Reserves space in the virtual address space of the **process**.
- If created in a DLL, the spaces is still in the process.
- Best used when around 3Mb to 4Mb of memory are required.

Linked List Example (1)

```
#Include "win32api.inc"

Type TItem
  Value    As Long
  NextItem As TItem Ptr
  PrevItem As TItem Ptr
End Type

Global hHeap As Long
Global Head  As TItem Ptr
Global Tail  As TItem Ptr
```

Linked List Example (2)

```
Function PbMain
  hHeap = HeapCreate(%NULL, 1000000, 0)

  If hHeap = %NULL Then
    Print "HeapCreate Failed."
    Exit Function
  End If

  Head = %NULL
  Tail = %NULL

  Enqueue 1
  Enqueue 2
  Enqueue 3
  Enqueue 4
  Dequeue
  Enqueue 5
  Enqueue 6
  Dequeue

  PrintAll

  HeapDestroy hHeap

  WaitKey$
End Function
```


Linked List Example (3)

```
Function Enqueue(ByVal Value As Long) As Long
  Local NewItem As TItem Ptr
  NewItem = HeapAlloc(hHeap, %NULL, SizeOf(TItem))

  If NewItem = %NULL Then
    Print "HeapAlloc Failed."
    Function = 0
    Exit Function
  End If

  @NewItem.Value    = Value
  @NewItem.NextItem = %NULL
  @NewItem.PrevItem = Tail

  If Head = %NULL Then
    Print "Enqueuing first item at: ", NewItem
    Head = NewItem
    Tail = NewItem
  Else
    Print "Enqueuing item at: ", NewItem
    @Tail.NextItem = NewItem
    Tail = NewItem
  End If

  Function = 1
End Function
```

Linked List Example (4)

```
Function Dequeue As Long
  If Tail = %NULL Then
    ? "List is empty"
    Function = 0
    Exit Function
  End If

  Dim Result As Long, Temp As Long
  Result = @Tail.Value
  Temp   = @Tail.PrevItem


  Print "Dequeuing item at:", Tail
  HeapFree hHeap, %NULL, Tail

  Tail = Temp
  If Tail = %NULL Then
    Head = %NULL
  Else
    @Tail.NextItem = %NULL
  End If

  Print "New tail at:", Tail

  Function = Result
End Function
```


Linked List Example (5)

```
Sub PrintAll
  Print
  Print "List Items:"

  Dim Current As TItem Ptr
  Current = Head

  While Current <> %NULL
    Print @Current.Value
    Current = @Current.NextItem
  Wend
End Sub
```

Other Memory Allocation Techniques

- GlobalAlloc, GlobalFree
  - Slower than Heap equivalents. Provided for compatibility.
- VirtualAlloc, VirtualFree
  - Memory in the process virtual address space.

GDI

- Graphics Device Interface.
- Device Contexts
  - You are not allows to ‘touch’ physical video memory.
  - A DC is a memory structure associated with a device (e.g. the screen or printer).
  - For screens a DC is associated with the display area of a window.
  - All drawing functions (lince, circles, etc…) are invoked on a DC.

WM_Paint

- WM_Paint is a special message sent to your callback instructing you that a portion (or all) of a window (the DC actually) needs to be repainted.
  - A hidden portion if the window is made visible.
  - Resizing.
  - Scrolling.
  - Programmatically invalidating a portion of the screen (e.g. InvalidateRect).
- The default window proc will just paint the basic window background, border, etc…

More WM_Paint

- Your program must know (and be able) to draw all it needs on the screen.
- However, sometimes only a small portion of the screen would require a repaint (e.g. closing a message box).
- The portion of the window that needs to be repainted is called an invalid area.

```
Case WM_PAINT
  hDC = BeginPaint(hWnd, PS)
  ...
  EndPaint(hWnd, PS)
  Return 0
```

BeginPaint/EndPaint (1)

- BeginPaint is usually the first API call in a WM_Paint message.
- BeginPaint also populates a tagPAINTSTRUCT structure with details of the invalid area that needs repainting.
- It returns the device context of the window that needs repainting.
- A BeginPaint is always accompanied by a call to EndPaint.
- EndPaint (amongst other things) will tell windows that the invalid area has been handled.

BeginPaint/EndPaint (2)

- You should never do this:

```
Case WM_Paint
  Return 0
```

- This would never validate the area and you program will continue sending WM_Paints forever.

```
Case WM_Paint
  hDC = BeginPaint(hWnd, PS)
  MoveToEx hDC, 10, 10, oldPoint
  LineTo hDC, 100, 100
  EndPaint hWnd, PS
```

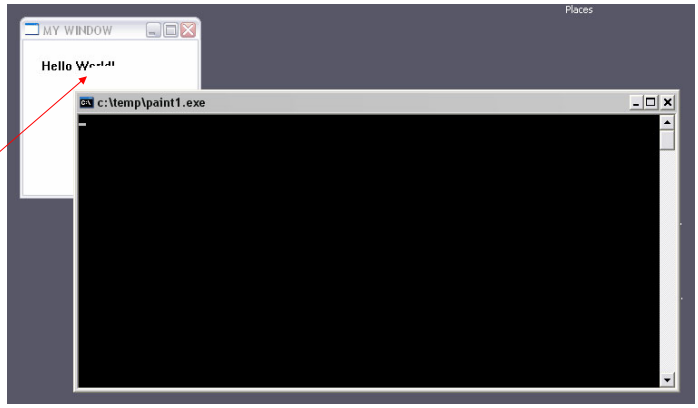
The Paint Structure

- Windows maintains a paint info structure for each window, populated and passed to you following a call to BeginPaint.
- The important fields in the structure are:
  - BOOL fErase: Tells you whether or not windows has erased the background (with the default windows background) or whether you need to handle the background manually.
  - RECT rcPaint: The coordinates of the invalud rectangle/area.
- In some cases you can completely ignore the rcPaint values and just redraw the whole client area. However if you are concerned about performance you should use it to avoid unnecessary drawing.

Drawing Example 1

```
Function MyProc (ByVal hWnd As Long, ByVal wMsg As Long, _
                 ByVal wParam As Long, ByVal lParam As Long)
                                                  As Long

    Dim hDC As Long

    Select Case wMsg
    Case %WM_LBUTTONDOWN

      hDC = GetDC(hWnd)
      TextOut hDC, 20, 20, "Hello World!", 12

    Case Else
      Function = DefWindowProc(hWnd, wMsg, wParam, lParam)
    End Select

End Function
```

Drawing Example 1



Drawing Example 2

```
Function MyProc (ByVal hWnd As Long, ByVal wMsg As Long, _
                 ByVal wParam As Long, ByVal lParam As Long)  As Long

    Dim hDC As Long
    Dim PS  As PAINTSTRUCT

    Select Case wMsg
    Case %WM_PAINT
      hDC = BeginPaint(hWnd, PS)
      TextOut hDC, 20, 20, "Hello World!", 12
      EndPaint hWnd, PS

    Case Else
      Function = DefWindowProc(hWnd, wMsg, wParam, lParam)
    End Select

End Function
```

Drawing Example 3

```
Case %WM_PAINT
     hDC = BeginPaint(hWnd, PS)

     Dim r As RECT
     GetClientRect hWnd, r
     r.nLeft   = r.nLeft   + 10
     r.nTop    = r.nTop    + 10
     r.nRight  = r.nRight  - 10
     r.nBottom = r.nBottom - 10

     Dim hBrush As Long
     hBrush = CreateSolidBrush(RGB(255,0,0))

     FillRect hDC, r, hBrush

     DeleteObject hBrush

     EndPaint hWnd, PS
```


Mouse Messages

- Click messages:
  - WM_LBUTTONDOWN
  - WM_LBUTTONUP
  - WM_LBUTTONDBLCLK
  - WM_RBUTTONDOWN
  - WM_RBUTTONUP
  - WM_RBUTTONDBLCLK
- Tracking mouse movement:
  - WM_MOUSEMOVE

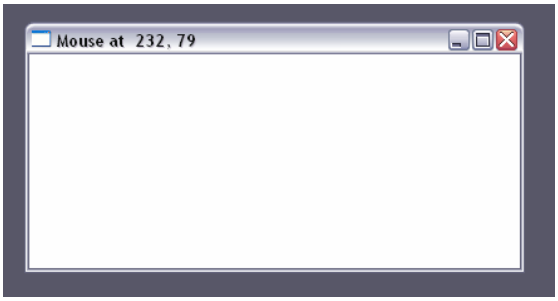
WM_MOUSEMOVE

- You can mask the wParam against these values to get extra info:
  - MK_CONTROL: CTRL key is pressed.
  - MK_LBUTTON: Left button is down.
  - MK_MBUTTON: Middle button is down.
  - MK_RBUTTON: Right button is down.
  - MK_SHIFT: Shift key is down.
- The coordinates are in the low and high words of the lParam:
  - X = LoWord(lParam)
  - Y = HiWord(lParam)

Mouse Movement Example

```
Case %WM_MOUSEMOVE

      Dim x As Long, y As Long
      Dim NewCaption As Asciiz*64

      x = LoWrd(lParam)
      y = HiWrd(lParam)

      NewCaption = "Mouse at " & _
                   Str$(x) & "," & Str$(y)

      SetWindowText hWnd, NewCaption
```

Mouse Movement Example



Timers

- You can associate a number of timers with a window.
- These timers will fire a WM_TIMER message or call a function each time a number of milliseconds have elapsed.
- API Calls
  - SetTimer
  - KillTimer

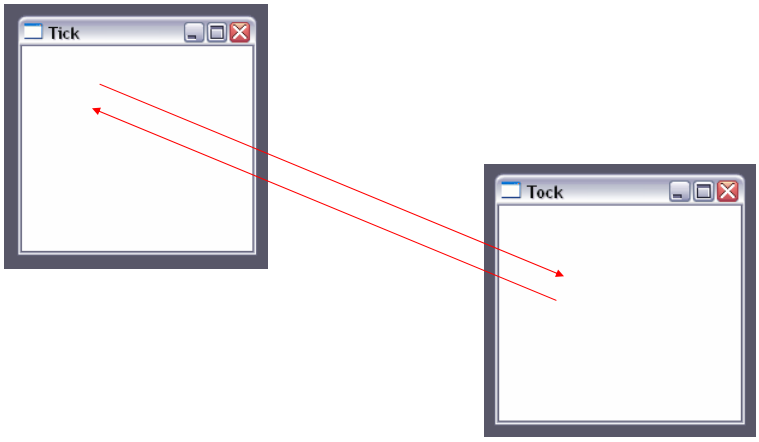
SetTimer/KillTimer

- SetTimer takes the following arguments:
  - hWnd: The handle of the window to associate the timer with.
  - nIDEvent: A long integer unique to each timer (so you can distinguish which timer fired).
  - uElapse: The elapse time of the timer.
  - lpTimerFunc: A pointer the function that will be called then the timer fires. If this value is NULL, the system will post a WM_TIMER message to your callback instead.
- KillTimer arguments:
  - hWnd: same as above.
  - uIDEvent: same as above.

Timer Example 1

```
Case %WM_CREATE
     SetTimer hWnd, 101, 500, 0
     TmrToggle = 0

   Case %WM_TIMER

     If wParam = 101 Then
       If TmrToggle = 0 Then
         SetWindowText hWnd, "Tick"
       Else
         SetWindowText hWnd, "Tock"
       End If

       TmrToggle = Not TmrToggle
     End If

   Case %WM_CLOSE
     KillTimer hWnd, 101
     PostQuitMessage 0
```

Timer Example 1



Timer Example 2

```
Function TimerProc (ByVal hWnd As Long, ByVal wMsg As Long, _
                    ByVal wParam As Long, ByVal lParam As Long)  As Long

  If TmrToggle = 0 Then
    SetWindowText hWnd, "Tick"
  Else
    SetWindowText hWnd, "Tock"
  End If

  TmrToggle = Not TmrToggle
End Function

...

 Case %WM_CREATE
      SetTimer hWnd, 101, 500, CodePtr(TimerProc)
      TmrToggle = 0

    Case %WM_CLOSE
      KillTimer hWnd, 101
      PostQuitMessage 0
```

Notes on Winsock (1)

- An API used for TCP/IP to communicate with Windows applications.
- winsock.dll, wsock32.dll

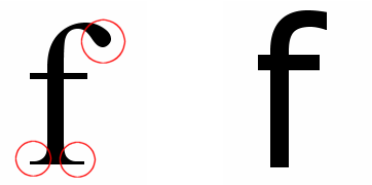
Application using Winsock
→
winsock.dll/wsock32.dll
→
TCP/IP
→
Modem
→
Network

Notes on Winsock (2)

- A socket is a network connection between two computers.
- To create a socket, you will need:
  - The **IP address** of the computer you are connecting to.
  - The **Port** you want to connect to.
- You can check out:
  - http://world.std.com/~jimf/papers/sockets/winsock.html
  - http://www.district86.k12.il.us/central/activities/computerclub/Tutorials/Winsock/Index.htm

Font Lingo (1)

- Type face: a font design.
- Type family: a group of fonts belonging to the same type face but include italics, bold, etc… variants.
- Serif:
- Sans-Serif:



Font Lingo (2)

- Fixed-width font (mono spaced):
  courier new
  wwwwwww
  iiiiiii
- Proportional-width font:
  times new roman
  wwwwwww
  iiiiiii
- Point – a unit to measure the size of a font. a point is approximately 1/72 th of an inch.
- Point size – the height of a font measured in points.
- Weight – the darkness of a font (how bold it is).

Font Families in Windows

Decorative	Novelty fonts. E.g. Old English.
Modern	Mono spaced with or without serifs. E.g. Courier New.
Roman	Proportional font with serifs. E.g. Times New Roman.
Swiss	Proportional font without serifs. E.g. Arial.
Script	Looks like handwriting. E.g. Script.
Dontcare	No font family info is known or does not matter when creating the font.

Font Technologies

- Raster: Bitmapped fonts, designed for a specific resolution of a device.
- Vector: Uses scalable lines and curves. This makes them resolution independent.
- TrueType: Similar to Vector fonts but optimised for fast drawing speed.
- OpenType: Similar to TrueType but shape definitions may contain PostScript data too.

Character Sets

- Each font has a specific character set.
- A character set defines which shapes/printable characters (letters, punctuation, symbols, etc…) are defined in the font.
- Each character is identified by a number.
- Most character sets are supersets of ASCII (retaining the meanings for the fonts numbered from 32 to 127).

Common Character Sets (1)

- The **Windows character set** – very similar to the ASCII set from 32 up to 255. The first character is conveniently the space/blank.
- The above character sets uses 1 byte to represent a character (i.e. 2^8=255 chars). This is enough for most western languages including diacriticals. Eastern languages are not catered for so the **Unicode** standard was adopted that uses 16 bits to identify a character.

Common Character Sets (2)

- The **OEM character set** is the font used in full screen text mode/console applications. Characters 32 to 255 are similar to ASCII and the Windows character set but the lower 32 characters are used for custom symbols (see http://www.ascii-table.org/).
- The **Symbol character set** is a font containing symbols (e.g. math symbols).
- **Vendor Specific**.

Mapping Modes

- In Windows, mapping modes define how values describing the sizes of objects are interpreted. For example:

Mapping Modes (not all)	Each unit is equivalent to…
MM_HIMETRIC	0.001 millimeters
MM_HIENGLISH	0.001 inches
MM_TEXT	1 pixel
MM_TWIPS	1/1440 of an inch

- To select a mapping mode for a device context, you use the **SetMapMode** API call.

CreateFont

- A font instance may be created using the **CreateFont** GDI function.
- Arguments to the function include:
  - The desired height of the font,
  - The average width of the font,
  - Weight,
  - …
  - Face name.
- Usually the height parameter is passed in Point Sizes and the width is set to 0 for the font engine to automatically choose the best value.
- The function however does not expect the height value in point sizes but in **logical units** depending on the current mapping mode.
- To use point sizes, IF you are in the MM_TEXT mapping mode you can use the following expression to convert the desired point size to logical units:

  MulDiv(**PointSize**, GetDeviceCaps(hDC, LOGPIXELSY), 72) * -1

- Muldiv divides two 32-bit numbers and divides the resulting 64-bit number by another 32-bit number.

Printing Text

- Once the font has been created, it is selected as the default font of the device context using the **SelectObject** API call.
- Now a function like **TextOut** can be called to draw text in that font on the specified DC.
- Once we are ready from the font, it's memory is freed using **DeleteObject**.

Using Controls

- As mentioned earlier on, a lot of basic Windows controls are windows themselves. It is no surprise that these controls are created using the **CreateWindow** API call.
- However, in this case we would use predefined windows class names. Some include:
  - BUTTON
  - COMBOBOX
  - EDIT
  - LISTBOX
  - STATIC (a label)

Creating a Button

```
hwndButton = CreateWindow("BUTTON",
                          "OK",
                          <button styles>,
                          ...
                          hWnd, // parent window
                          101,  // Control ID
                          ...
```

- Button styles include:
  - BS_PUSHBUTTON (normal button)
  - BS_DEFPUSHBUTTON (like above but has a heavy border and responds to the enter key even if not focused)
  - WS_CHILD is used to tell Windows that the button is the child of a container window.

Handling Controls in the Callback

- When a button event occurs (like a click) the parent window receives a WM_COMMAND message.
- The LoWord of the wParam tells us the ID of the control that generated the command message.
- The HiWord of the wParam tells us the notification message for the control (e.g. BN_CLICKED if a button is clicked).
- The lParam contains the window handle of the control.

Windows Resources

- An executable file, may use a number of resources such as:
  - Icons,
  - Bitmaps,
  - Strings,
  - Any other binary data.
- The resources are embedded (linked) in the executable file.
- The resources are defined using a resource script file.
- The script file is compiled using the Windows resource compiler tool rc.exe.

The Resource Script File

- A resource script file is a text file with the RC extension.
- For example:

```
MyIcon      ICON   my.ico
MyCursor    CURSOR my.cur
MyBitmap    BITMAP my.bmp
```

- When the file is compiled, the result is emmitted to a file with the RES extension that may be linked to your application.

Example: Loading an Icon

- When creating a window class we had:

```
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

- IDI_APPLICATION is a constant telling Windows to use the default Windows icon for windows derived from this class.
- If we want to use an icon specified in a resource file, we use:

```
wc.hIcon = LoadIcon(NULL, "MyIcon");
```

Loading Bitmaps and Cursors

```
HBITMAP LoadBitmap(HINSTANCE hInstance,
                   LPCTSTR lpBitmapName);

HCURSOR LoadCursor(HINSTANCE hInstance,
                   LPCTSTR lpCursorName);
```


Creating Menus (Resource File)

```
#include "Resource.h"

MyMenu MENU
{
  POPUP "&File"
  {
    MENUITEM "&New",      IDM_FILENEW
    MENUITEM "&Open...", IDM_FILEOPEN
    MENUITEM SEPARATOR
    MENUITEM "E&xit",     IDM_FILEEXIT
  }
  POPUP "&Help"
  {
    MENUITEM "&About",    IDM_ABOUT
  }
}
```

Creating Menus (Resource.h)

- Here we define the constants used by our resource file:

```
#define IDM_FILENEW  1000
#define IDM_FILEOPEN 1001
#define IDM_FILEEXIT 1002
#define IDM_ABOUT    1003
```

- By conventions menu item constants start with IDM_ (For icons it is IDI_, etc…)

Using the Menu

- Now, when creating the window class, we specify the resource name for the menu:

```
wc.lpszMenuName = "MyMenu";
```

- The menu events are handled in the WM_COMMAND message send to the callback.
- The wParam value of the message will tell us, the ID of the menu item clicked (i.e. match against the IDM_XXX constants).
- Actually the LoWord of wParam tells us the ID, by the HiWord is zero if the message comes from a menu.

Using Accelerators (1)

- An accelerator table must be defined in the resource file. Like:

```
MyAccelTable ACCELERATORS
BEGIN
   "^C",  IDDCLEAR         ; control C
   "K",   IDDCLEAR         ; shift k
   "k",   IDDELLIPSE, ALT  ; alt k
   98,    IDDRECT, ASCII   ; b
   66,    IDDSTAR, ASCII   ; B (shift b)
   "g",   IDDRECT          ; g
   "G",   IDDSTAR          ; G (shift G)
   VK_F1, IDDCLEAR, VIRTKEY                ; F1
   VK_F1, IDDSTAR, CONTROL, VIRTKEY        ; control F1
   VK_F1, IDDELLIPSE, SHIFT, VIRTKEY       ; shift F1
   VK_F1, IDDRECT, ALT, VIRTKEY            ; alt F1
   VK_F2, IDDCLEAR, ALT, SHIFT, VIRTKEY    ; alt shift F2
   VK_F2, IDDSTAR, CONTROL, SHIFT, VIRTKEY ; ctrl shift F2
END
```


Using Accelerators (2)

- To load the accelerator table from the resource file, use:

```
HACCEL LoadAccelerators(HINSTANCE hInstance,
                        LPCTSTR   lpTableName);
```

- The normal window message loop was:

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
  TranslateMessage(&Msg);
  DispatchMessage(&Msg);
}
```

Using Accelerators (3)

- To handle accelerators, it must be modified to:

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
  if (TranslateAccelerator(hWnd, hAccel, Msg) == 0)
   {
     TranslateMessage(&Msg);
      DispatchMessage(&Msg);
    }
}
```

- In other words, the normal Translate/DispatchMessage calls must not be mage if a accelerator was translated (see TranslateAccelerator on MSDN).

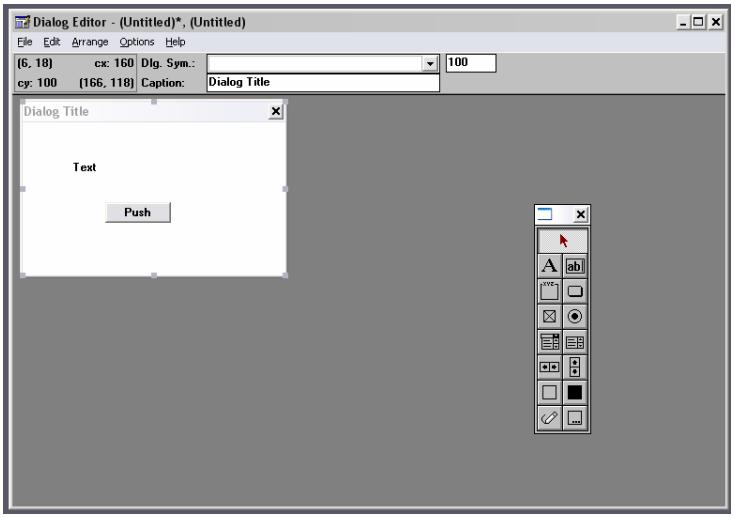
Using Accelerators (4)

- Again, the fact that an accelerator was pressed, is detected in the WM_COMMAND part of the callback.
- As usual, the LoWord of wParam will tell us the ID of the accelerator pressed, and the HiWord will be 1 so we can distinguish it from a menu item click.
- Note:
  - This means that the constant used to identify a menu and the constant to identify an accelerator may be the same. We can have a menu item “Save” and the combination “CTRL+S” handled by the same piece of code!

Dialogs

- Another resource can be a windows dialog.
- It is likely that you will use the dialog editor to create the dialogs rather than write the script manually.
- To create the dialog in your application you will call the CreateDialog API call.
- This function
  - Returns the hWnd of the dialog window.
  - You also specify (apart from others) the pointer to the callback function of the dialog.

The Dialog Editor



The Dialog Callback

- The prototype of the dialog callback is the same as the prototype of a “normal” windows callback.
- The messages are different.
- For example, when the dialog is created, we use the WM_INITDIALOG message rather than WM_CREATE.
- You should check out the MSDN entry for CreateDialog for more details.

What is COM?

- COM – **Component Object Model**.
- Platform Independent.
- Object Oriented.
- Binary Component Format.
- COM is a STANDARD specifying how objects are accessed and how they interact with other objects.
- The only requirements for a language to support com is the ability to create and manipulate pointers and calling functions through pointers.

More COM

- A software component is made up of:
  - **Object Data** (Variables/Properties).
  - Data manipulated by a number of **functions**.
- The set (list) of functions is called the **interface**.
- These functions are implemented as **methods**.
- COM specifies that the only way to call a function is through a pointer to the interface.
- In COM there are a number of interfaces that must be implemented by all components.

Interfaces

- An interface is usually thought of as a **contract** specifying a group of related function prototypes:
  - Their name, return types and arguments.
- No implementation is associated with an interface.
- For example if we have an IQueue interface (by convention interface names start with the letter I) that defines the functions:
  - Enqueue
  - Dequeue

Accessing the Component

- An instance of an interface is a pointer to an array of function pointers.
- Each interface is assigned a Globally Unique Identifier (GUID).
- Note that one object may have multiple interfaces.
- COM interfaces are **immutable** – you cannot change the interface once it is assigned a GUID.

Registering COM Components

- The Windows Registry is a global system database for the OS.
- When a component is registered the GUID and the actual COM EXE or DLL are saved in the registry.
- Whenever we need to access a component (which we identify by the GUID), the registry is consulted to resolve the GUID into the COM EXE or DLL required.
- The component will be loaded in-process or out-of-process as required.

COM Clients and Servers

- A **COM Client** is the actual software that gets the interface pointer to an object and calls the methods.
- A **COM Server** where the interface implementations actually exist. The client gets a pointer to the interface to call the functions.
- There are 2 types of servers:
  - In-process: Implemented as DLLs.
  - Out-of-process: Implemented as EXEs. The process can be run on a different machine.