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.

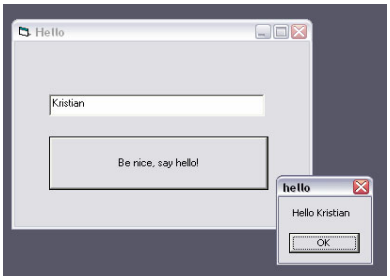
Anatomy of DLLMain (3)

- 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
```

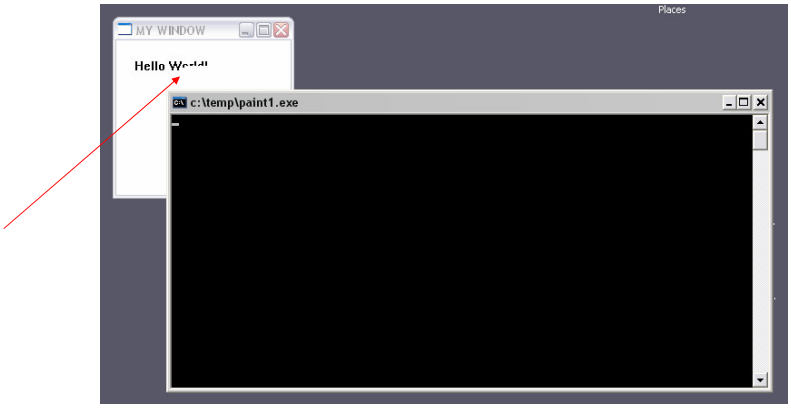

The Paint Structure

- 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

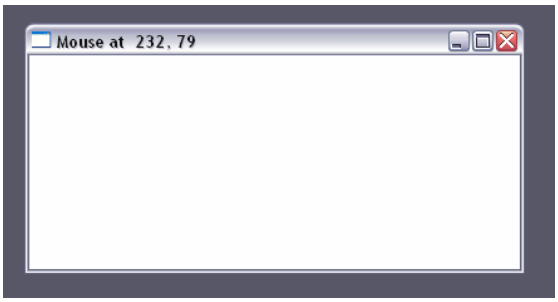
WM_MOUSEMOVE

- 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

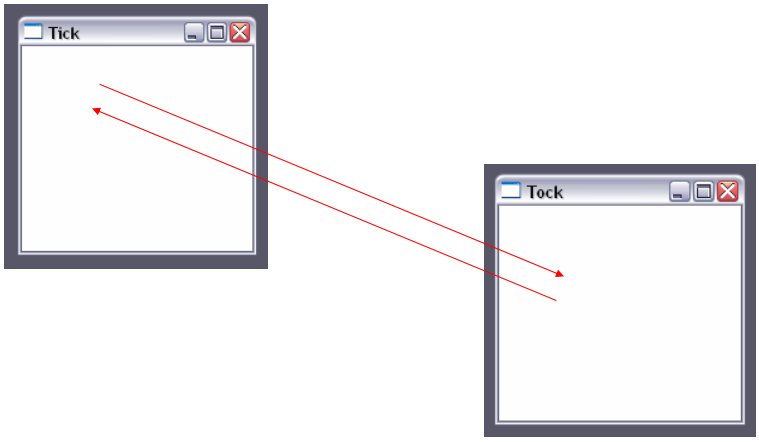
SetTimer/KillTimer

- 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
```